



Pivotal Certified Professional Spring Developer Exam

A Study Guide

Examination preparation on core Spring
concepts and principles

Iuliana Cosmina

Apress®

www.allitebooks.com

Pivotal Certified Professional Spring Developer Exam

A Study Guide



Iuliana Cosmina

Apress®

Pivotal Certified Spring Web Application Developer Exam

Iuliana Cosmina
Sibiu, Romania

ISBN-13 (pbk): 978-1-4842-0812-0
DOI 10.1007/978-1-4842-0811-3

ISBN-13 (electronic): 978-1-4842-0811-3

Copyright © 2017 by Iuliana Cosmina

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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

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.

Managing Director: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Manuel Jordan
Coordinating Editor: Mark Powers
Copy Editor: David Kramer
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Image: Designed by Freepik

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 <http://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 <http://www.apress.com/us/services/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers via the book's product page, located at www.apress.com/9781484208120. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*To all passionate Java developers, never stop learning and
never stop improving your skills.*

*To all my friends for supporting me to make this book happen;
you have no idea how dear you are to me.*

Contents at a Glance

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
■ Chapter 1: Book Overview	1
■ Chapter 2: Spring Bean LifeCycle and Configuration.....	17
■ Chapter 3: Testing Spring Applications	115
■ Chapter 4: Aspect Oriented Programming with Spring	153
■ Chapter 5: Data Access	185
■ Chapter 6: Spring Web.....	271
■ Chapter 7: Spring Advanced Topics.....	349
■ Chapter 8: Spring Microservices with Spring Cloud	435
Index.....	461

Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
■ Chapter 1: Book Overview	1
What Is Spring and Why Should You Be Interested in It?.....	1
What Is the Focus of This Book?	3
Who Should Read This Book?.....	3
About the Certification Exam.....	3
How to Use This Book as a Study Guide.....	5
How Is This Book Structured?	5
How Each Chapter Is Structured.....	6
Recommended Development Environment.....	7
Recommended JVM.....	8
Recommended Project Build Tool	8
Recommended IDE	10
The Project Sample	11
■ Chapter 2: Spring Bean LifeCycle and Configuration	17
Old Style Application Development	17
Spring IoC and Dependency Injection.....	24
Spring Configuration	29
Providing Configuration via XML.....	29
Spicing Up XML Configuration	53

Application Context and Bean Lifecycle	64
Providing Configuration Using Java Configuration and Annotations.....	85
Summary	110
Quick quiz.....	111
■ Chapter 3: Testing Spring Applications	115
A Few Types of Testing	115
Test-Driven Development	115
Unit and Integration Testing	116
Testing with Stubs	117
Testing with Mocks.....	124
Testing with Spring	134
Using Profiles.....	144
Summary	146
Quick Quiz	146
Practical Exercise.....	148
■ Chapter 4: Aspect Oriented Programming with Spring	153
Problems Solved by AOP	154
Spring AOP	157
AOP Terminology.....	158
Quick Start.....	159
Aspect Support Configuration using XML.....	165
Defining Pointcuts	165
Implementing Advice	172
Conclusions	178
Summary	181
Quick Quiz	181
Practical Exercise.....	183

■ Chapter 5: Data Access	185
Basic Data Access Using JDBC	187
Spring Data Access	189
Introducing JdbcTemplate	190
Spring Data Access Exceptions	207
Data Access Configuration In a Transactional Environment	209
How Transaction Management Works in Spring.....	212
Configure Transactions Support.....	214
Introducing Hibernate and ORM	235
Session and Hibernate Configuration	235
Session and Hibernate Querying	240
Exception Mapping	243
Object Relational Mapping.....	245
Java Persistence API	247
Spring Data JPA.....	256
**Spring and MongoDB.....	260
Summary.....	265
Quiz	265
■ Chapter 6: Spring Web	271
Spring Web App Configuration.....	274
Quickstart	276
XML.....	281
@MVC	285
Java Configuration for Spring MVC.....	286
Getting Rid of web.xml	288
Running a Spring Web Application.....	291
Running with Jetty.....	292
Running with Tomcat	294

Spring Security	298
Spring Security Configuration.....	301
XML Configuration	301
Spring XML Configuration without web.xml	313
Java Configuration.....	313
Security Tag Library.....	317
Method Security	321
Spring Boot	326
Configuration	327
Configuration Using YAML.....	338
Logging.....	341
Testing with Spring Boot.....	341
Summary	344
Quiz	345
■ Chapter 7: Spring Advanced Topics	349
Spring Remoting	350
Spring Remote Configuration	353
Spring JMS	362
JMS Connections and Sessions.....	363
JMS Messages	364
JMS Destinations.....	365
Apache ActiveMQ.....	367
Spring JmsTemplate.....	370
JMS with Spring Boot.....	378
Spring Web Services	382
SOAP Messages.....	384
Generating Java Code with XJC.....	386
Spring Boot WS Application	387
Publishing WSDL.....	391
Testing Web Services applications	392

Spring REST	395
Spring Support for REST.....	397
Exception Handling.....	402
HTTP Message Converters.....	404
Spring MVC Configuration for RESTful Applications	405
Using RestTemplate to Test RESTful Applications.....	407
Advantages of REST	416
Spring JMX.....	421
JMX Architecture	421
Plain JMX.....	423
Spring JMX	424
Summary.....	432
Quick Quiz	433
■ Chapter 8: Spring Microservices with Spring Cloud	435
Microservices with Spring.....	436
Registration and Discovery Server	439
Microservices Development	442
Microservices Communication	451
More Novelties	456
Practice Section.....	457
Summary.....	458
Quick Quiz	458
Index.....	461

About the Author



Iuliana Cosmina is a software architect and passionate developer. She has been programming in Java for more than 10 years. She also taught Java at the Gheorge Asachi Technical University in Iasi, Romania. She has a bachelor's degree in computer science and a master's degree in distributed systems from the same university.

She discovered Spring in June 2012 and loved it so much that she trained for and passed the exam to become a Certified Spring Professional in November 2012. She trained for and passed the exam to become a Certified Web Application Developer in May 2014.

Her plan is to become a Spring Enterprise Integration Specialist in the near future.

She has contributed to the development of different types of enterprise applications such as search engines, ERPs, track and trace, and banking. During her career in outsourcing she has been a team leader,

acting software architect, and DevOps professional. She likes to share her knowledge and expertise via tutoring, teaching, and mentoring, but in the summer of 2014, everything changed because of Steve Anglin, who proposed that she write a Spring Web Study Guide. A short time after the first book was released, she was given the chance to write the Spring Core Study Guide as well, and she seized the opportunity. She currently lives in Sibiu, Romania, and works as a software architect for BearingPoint, a multinational management and technology consulting company.

When she is not programming, she spends her time reading, traveling, hiking, or biking.

- You can find some of her personal work on her GitHub account: <https://github.com/iuliana>.
- You can find her complete CV on her LinkedIn account: <https://ro.linkedin.com/in/iulianacosmina>.
- You can contact her at Iuliana.Cosmina@gmail.com.

About the Technical Reviewer

Manuel Jordan is a self-taught developer and researcher who enjoys learning new technologies for his own experiments in creating new integrations among them.

Manuel won the 2010 Springy Award, Community Champion and Spring Champion 2013. In his little free time, he reads the Bible and composes music on his bass and guitar.

Acknowledgments

Creating this guide involved a lot of teamwork. It is the second time I've written a technical book, and I wouldn't have made it without all the help and advice I received from Mark Powers and Manuel Jordan. Mark has been very supportive, sharing with me his experience in book writing and encouraging me when I was ready to give up because I thought my work was not good enough. He was also very understanding and forgiving when deadlines were missed because of writer's block or personal problems.

Manuel has been a great collaborator; I loved our exchanges of technical ideas, for which I am very thankful, because working with them has helped me grow professionally. Many thanks to the team that helped turn my technical verbiage into human-readable literature.

Most of all, I want to thank Steve Anglin for trusting me to get this book done.

Apress has published many of the books I have read and used to improve myself professionally during my studies and beyond. It is a great honor for me to write a book and publish it with Apress, and it gives me enormous satisfaction to be able 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 sleep loss, having too much work to do, and writer's block. Thank you all for being supportive and making sure I still had some fun while writing this book.

And I would also like to add a very special thank you to Marian Lopatnic, Cristina Lutai, and Andreea Jugarean. These three special persons ensured that my determination to finish this book never flagged, by continually reminding me that I am a badass in my profession, and as long as I do my best, the outcome will be great.

Introduction

More than four years have passed since I wrote my first Spring project, and since then, the Spring Framework has turned into a full-blown technology that provides everything needed to build complex and reliable Java Enterprise Applications.

Four major versions of Spring have been released so far, and the fifth is right around the corner. And except for the official study guide required for passing the certification exam, before the conception of this book there was no additional resource like this.

This study guide provides a complete overview of all the technologies involved in creating a Spring core application from scratch. It guides you step by step into the Spring world, covering Spring 3 and Spring 4. More advanced topics such as RMI and JMS have been covered as well, because there are still companies that prefer to use them, and developers might encounter them while in the field.

There is a multimodule project associated with this book named Pet Sitter, covering every example presented in the book. As the book was written, new versions of Spring were released, a new version of IntelliJ IDEA was released, and new versions of Gradle were released as well. I upgraded to the new versions in order to provide the most recent information and keep this book synchronized with the official documentation. A group of reviewers has gone over the book, but if you notice any inconsistencies, please send an email to editorial@apress.com, and a correction will be made.

The example source code for this book can be found on GitHub via the **Download Source Code** button on the book's product page, located at www.apress.com/9781484208120. 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.

The code for the Pet Sitter project will likewise be made available on a public GitHub repository.

An appendix with answers to the questions at the end of every chapter and additional details related to development tools that can be used to develop and run the code samples of the book will also be available as part of the source code package hosted at Github. A sample practice exam will also be published on the Pet Sitter repository.

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

CHAPTER 1



Book Overview

Spring is currently one of the most influential and rapidly growing Java frameworks. Every time a new startup idea is born, if the development language is Java, Spring will be taken into consideration. Spring will be fourteen years old on the first of October 2016, and it has grown into a full-fledged software technology over the years.¹

This book covers much of the core functionality from multiple projects. The topics that are required for the official certification exam are covered deeply, and all extras are covered succinctly enough to give you a taste and make you curious to learn more.

What Is Spring and Why Should You Be Interested in It?

When a project is being built using Java, a great deal of functionality needs to be constructed from scratch. Yet many useful functionalities have already been built and are freely available because of the open source world we are living in. A long time ago, when the Java world was still quite small, when you were using open source code in your project developed by somebody else and shipped as a **.jar*, you would say that you were using a **library**. But as time passed, the software-development world evolved, and the libraries grew too. They became **frameworks**. Because they were no longer a single **.jar* file that you could import, they became a collection of more-or-less decoupled libraries, with different responsibilities, and you had the possibility of importing only what you needed. As frameworks grew, tools to build projects and add frameworks as dependencies evolved. One of the currently most widely used tools to build projects is Maven, but new build tools are now stealing the scene. One of these new-age build tools will be used to build the projects for this book. It will be introduced later.

The Spring Framework was released in October 2002 as an open source framework and inversion of a control container developed using Java. As Java evolved, Spring did too. Spring version 4.3, the one covered in this book, is fully compatible with Java 8, and Spring 5 is planned to be released in the fourth quarter of 2016.² The intention is to make it compatible with Java 9, which is planned for release in September 2016.³ But since the release of Java 8 was delayed over six months, nothing is certain at the moment.

Spring comes with a great deal of default behavior already implemented. Components called “infrastructure beans” have a default configuration that can be used or easily customized to fit the project’s requirements. Having been built to respect the “Convention over Configuration” principle, it reduces the number of decisions a developer has to make when writing code, since the infrastructure beans can be used to create functional basic applications with minimum customization (or none at all) required.

¹Just as a coincidence and a fun fact, in Romania you obtain your first identity card when you are fourteen years old, and it is considered the age of intellectual maturity that allows you to differentiate good from evil.

²Information can be found on the official Spring blog at <https://spring.io/blog/2015/08/03/coming-up-in-2016-spring-framework-4-3-5-0>.

³Information at <https://jaxenter.com/java-9-release-date-announced-116945.html>.

Spring is open source, which means that many talented developers have contributed to it, but the last word on analyzing the quality of the components being developed belongs to the Pivotal Spring Development Team, previously known as the SpringSource team, before Pivotal and VMware merged. The full code of the Spring Framework is available to the public on Github,⁴ and any developer that uses Spring can fork the repositories and propose changes.

A Java application is essentially composed of objects talking to each other. The reason why Spring has gained so much praise in Java application development is that it makes connecting and disconnecting objects easy by providing a comprehensive infrastructure support for assembling objects. Using Spring to develop a Java application is like building a lightly connected Lego castle; each object is a Lego piece that you can easily remove and replace with a different one. Spring is currently the VIP of Java Frameworks, and if all you have read so far has not managed to make you at least a bit interested in it, then I am doing a really bad job at writing this book, and you should write an email and tell me so.

Before going further about what this book will provide you, let’s have an overview of the Spring projects. Figure 1-1 depicts all the Spring projects. For 2016, there are nineteen main projects, two community projects, and three projects that are “in the attic,” so to speak, because there will be no further contributions to them, since they are going to be dropped in the future.

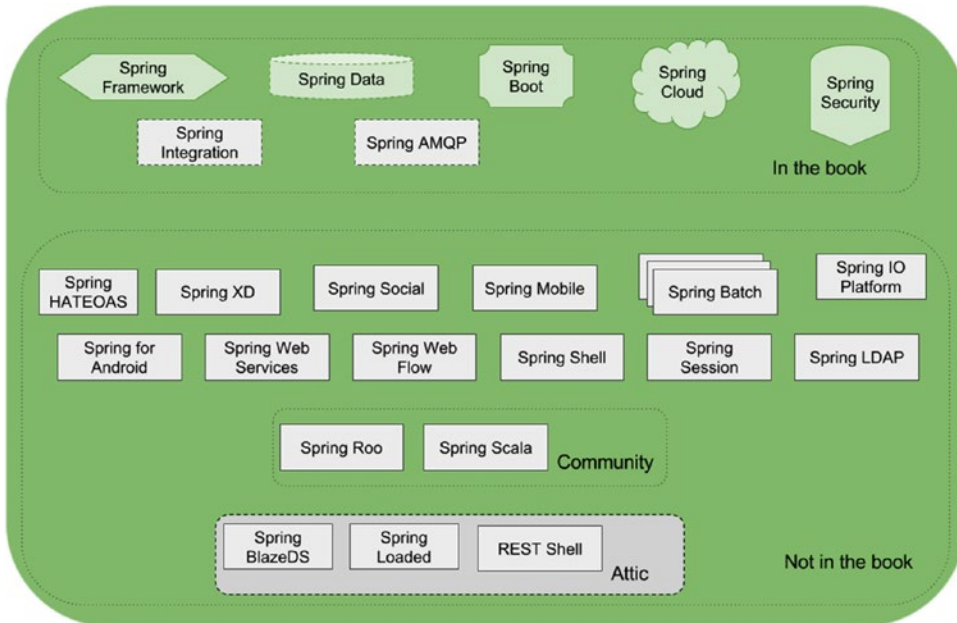


Figure 1-1. Spring Web Stack. (The projects drawn with dotted contours will be covered only partially in this book.)

⁴Github Spring Framework sources: <https://github.com/spring-projects/spring-framework>.

What Is the Focus of This Book?

The topics covered in this book are mostly Spring Framework's support components for the backend tier. We only scratch the surface for Spring Boot, Spring Data JPA, REST, MVC, and Microservices. This book aims to provide a natural path in the development of a complete Spring application. With each chapter, the application will become more complex, until its final form is reached, which will also have a security setup, a simple web application, and will support REST requests.

This book is focused on helping developers understand how Spring's infrastructure was designed and how to write Spring applications in a few easy steps using the maximum of Spring's potential. Its objectives are as follows:

- Use Spring to develop applications
- Use Spring Security to secure resources
- Use Spring Test and other test frameworks (JUnit, JsMock) to test applications
- Create Spring applications using Gradle⁵

Who Should Read This Book?

This book was written to provide clear insight into creating applications using Spring core components. It can also be a big help to a developer who wants to become a **Certified Spring Professional**.⁶ That is why every topic that is found in the official Pivotal Spring Core study guide is given the attention it deserves.

You just need a minimal knowledge of Java in order to make good use of this book, but online documentation for Java⁷ and Spring⁸ should be consulted every time something is not fully covered in the book.

In a nutshell, this book was written to be used by the following audiences:

- Java developers who want a taste of Spring
- Spring developers who are interested in learning to use Spring proficiently, but not interested in official certification
- Spring and Java developers who want to become certified and want all the help they can get.

About the Certification Exam

If you are interested in becoming a **Certified Spring Professional**, the first step you have to take is to go to the Pivotal official learning site <http://pivotal.io/training> and search for the Spring Certification section. There you will find all the details you need regarding the official trainings, including where are they

⁵Gradle is an automated build tool that is easy to configure and use for any type of application. Its build files are written using Groovy. Gradle combines the power and flexibility of Ant with the dependency management and conventions of Maven into a more effective way to build. Read more about it at <https://www.gradle.org/>.

⁶Keep in mind that attending a Spring Web training from Pivotal or a VMware Authorized Training Center is a prerequisite to becoming a Certified Spring Professional, as stated on the official site: <http://pivotal.io/academy#certification>.

⁷JSE8 official reference: <http://docs.oracle.com/javase/8/docs/>; JEE7 official documentation: <http://docs.oracle.com/javaee/7/>.

⁸Spring official Javadoc: <http://docs.spring.io/spring/docs/current/javadoc-api/>; Spring Reference: <http://docs.spring.io/spring/docs/current/spring-framework-reference/>.

taking place and when. The training is four days long. There are online trainings available as well. After creating an account on the Pivotal site, you can select the desired training. After you make the payment, if you choose an online training, about a month later, you will receive through the mail an official training kit consisting of the following:

- A pair of conference headphones (usually Logitech) for use during the training to hear your trainer talk and so you can ask questions.⁹
- A professional webcam (usually Logitech) for use during the training, so that your trainer and colleagues can see you, thus simulating the classroom experience.¹⁰
- A Spring study guide containing the printed version of the slides your tutor will be using during the training. (This might also be in electronic form, from consideration of the environment.)
- A Spring Study Lab book containing explanations and instructions for the practical exercises you will do during the training. (This might also be in electronic form, from consideration of the environment.)
- A Pivotal official flash drive containing the following:
 - Jdk installer
 - Sources necessary during the training. Each study lab has a small Spring application attached to it with missing configuration and code, and the student’s task is to complete it in order to have a working application. The same model is used in the code associated with this book.
 - An installer of the most recent stable version of the Spring Tool Suite. The version on the flash drive is mandatory for the course, because the installer sets up a local Maven repository with all the needed dependencies and a full eclipse project configuration with the lab sources. The STS also has an internal tcServer to run the web lab applications.
 - An html or PDF version of the Spring Study Lab.

If you decide against an online training course, you will not receive the headphones and the webcam. The training kit and the rest of the materials will be given to you when you arrive at the location where the training is taking place. After the training, you will receive a free voucher that is required to schedule the certification exam at an approved exam center near you. Basically, the voucher or voucher code being given to you is the proof that you have attended the official Spring Web Training.

! The exam duration is ninety minutes and consists of fifty questions. There are both single and multiple-answer questions. The latter are quite explicit, telling you how many correct answers you are expected to select. The questions in the book are actually more difficult, because you will not be told the number of correct options you must select. But you will be given a complete explanation of the answers in the appendix.

⁹Depending on the area and the training center, this item is optional.

¹⁰Depending on the area and the training center, this item, too, is optional.

The questions will cover (approximately) the following topics:

- Spring overview container, IoC and dependency injection
- SpEL and Spring AOP
- Spring JDBC, Transactions, ORM
- Spring MVC and the web layer
- Spring Security
- Spring Messaging and REST
- Spring Testing

The passing score for the exam is **76%**. This means that **38** correct answers are needed in order to pass. Most of the questions will present you a piece of Java code or configuration and ask you what it does, so make sure you understand the code attached to this book and write your own beans and configurations in order to understand the framework better. The good news is that all the code in the exam can be found in the sources you are given when you attend the official training. Other questions will present you with assertions about Spring Web and will require you to select the correct or the invalid statement.

If you read this book, understand all the examples, solve the practice exercises, and then attend the official training, my recommendation is that you take the certification exam as soon as possible. Do not allow for too much time to pass between finishing the training and taking the exam, because we are all human after all and information can be forgotten. Also, **the certification voucher is valid for only a year**. You can retake the exam again if you fail the first time, but it will cost you about \$150.

How to Use This Book as a Study Guide

This book was written in such a way as to guide you step by step through the wonderful technology that is Spring. It follows the same learning curve as the official training and focuses on the same topics that are required for the certification exam, since those are also the most needed in real production applications. The topics that are not needed for the certification exam are marked, so you know that you can skip them, although if you are truly interested in Spring, you will definitely not do so.

The main differences are in the tools used for the practical examples, which will be covered shortly.

How Is This Book Structured?

This book has eight chapters and an appendix. The official Spring guide has sixteen chapters, but for the purposes of the book, related topics are wrapped up together. For example, the official study guide has four separate chapters that cover dependency injection, Spring Core fundamentals, and configuration. This book has only a single big chapter about these topics: **Chapter 2—Bean LifeCycle and Configuration**.

The list of chapters and a short description of each are presented in Table 1-1.

Table 1-1. List of topics by chapter

Chapter	Topic	Details
1	Book overview	Introduction to Spring history, technologies and tools used for practice
2	Bean LifeCycle and Configuration	Basic Spring core concepts, components, and configuration
3	Testing Spring Applications	How Spring applications can be tested, most used testing libraries and test principles
4	Aspect Oriented Programming	AOP concept, problems that it solves and how it is supported in Spring
5	Data Access	Advanced Spring Data access using JDBC, Hibernate and Spring Data JPA
6	Spring Web	Basic introduction of Spring MVC
7	Spring Advanced Topics	Remoting, Messaging, Web Services with REST
8	Spring Microservices with Spring Cloud	Introduction to Spring Microservices and what they can be used for
A	Appendix	Two mock exams, answers to review questions, and other comments

How Each Chapter Is Structured

The introductory chapter, the one you are reading now, covers the basics of Spring that every developer using this book should know: what Spring is, how it has evolved, how many official Spring projects there are, the technologies used to build and run the practical exercises, how you can register for the exam to become a Certified Spring Professional, and so on. This chapter is the exception. It is structured differently from the others, because it was designed to prepare you for what it will be coming next.

The remaining chapters are designed to cover a Spring Module and associated technologies that will help you build a specific type of Spring application. Each chapter is split into a few sections, but in a nutshell, a chapter is organized as follows:

- Basics
- Configuration
- Components
- Summary
- Quick quiz
- Practical exercise

The longer chapters deviate from this structure, introducing small practice exercises after key sections, since solving these exercises will help you to check your understanding and solidify your knowledge of the presented components.

Code that is irrelevant to Spring understanding will not always be quoted in this book, but it is available to you in the book's practice project.

Conventions

! This symbol appears in front of paragraphs to which you should pay particular attention.

** This symbol appears in front of a paragraph that is an observation or an execution step that you can skip.

? This symbol appears in front of a question for the user.

... This symbol represents missing code that is not relevant for the example.

CC This symbol appears in front of a paragraph that describes a **Convention over Configuration** practice in Spring, a default behavior that helps the developer reduce his or her work.

[random text here] When you have text surrounded by square brackets, this means that the text between the brackets should be replaced by a context-related notion.

Downloading the Code

This book has code examples and practical exercises associated with it. There will be missing pieces of code that you will have to fill in to make applications work and test your understanding of Spring Web. I recommend that you go over the code samples and do the exercises, since similar pieces of code and configurations will appear in the certification exam.

The following downloads are available:

- Source code for the programming examples in the practice section using XML configuration
- Source code for the programming examples in the practice section using Java configuration

You can download these items from the Source Code area of the Apress website <http://www.apress.com>.

Contacting the Author

More information about Iuliana Cosmina can be found at <http://ro.linkedin.com/in/iulianacosmina>. She can be reached at <mailto:iuliana.cosmina@gmail.com>.

Follow her personal coding activity on <https://github.com/iuliana>.

Recommended Development Environment

If you decide to attend the official course, you will notice that the development environment recommended in this book differs considerably from the one used at the course. A different editor was recommended, a different application server, and even a different build tool. The reason for this was to improve and expand your experience as a developer and to offer a practical development infrastructure. Motivation for each choice will be mentioned on the corresponding sections.

Recommended JVM



Java 8, the official JVM from Oracle. Download the JDK matching your operating system from

<http://www.oracle.com> and install it.

! It is recommended that you set the `JAVA_HOME` environment variable to point to the directory where Java 8 was installed (the directory in which the JDK was unpacked) and add `%JAVA_HOME%\bin` for Windows, `$JAVA_HOME/bin` for Unix-based operating systems, to the general path of the system. The reason behind this is to ensure that other development applications written in Java will use this version of Java and prevent strange incompatibility errors during development.


! Verify that the version of Java the operating system sees is the one you just installed by opening a terminal (Command Prompt in Windows, and any type of terminal you have installed on MacOs and Linux) and typing:

```
java -version
```

You should see something similar to this:

```
java version "1.8.0_74"
Java(TM) SE Runtime Environment (build 1.8.0_74-b02)
Java HotSpot(TM) 64-Bit Server VM (build 25.74-b02, mixed mode)
```

Recommended Project Build Tool

 **Gradle 2.x** ** The sources attached to this book can be compiled and executed using the Gradle wrapper, which is a batch script on Windows and a shell script for other operating systems. When you start a Gradle build via the wrapper, Gradle will be automatically downloaded and used to run the build; thus you do not need to install Gradle as stated previously. Instructions on how to do this can be found by reading the public documentation at http://www.gradle.org/docs/current/userguide/gradle_wrapper.html.

A good practice is to keep code and build tools separate, but for this study guide, it was chosen to use the wrapper to make setting up the practice environment easy by skipping the Gradle installation step and also because the recommended source code editor uses the wrapper internally.

If you decide to use Gradle outside the editor, you can download the binaries only (or if you are curious, you can download the full package, which contains binaries, sources, and documentation) from their official site <https://www.gradle.org/>, unpack them, and copy the contents somewhere on the hard drive. Create a `GRADLE_HOME` environment variable and point it to the location where you have unpacked Gradle. Also add `%GRADLE_HOME%\bin` for Windows, `$GRADLE_HOME/bin` for Unix-based operating systems, to the general path of the system.

Gradle was chosen as a build tool for the sources of this book because of the easy setup, small configuration files, flexibility in defining execution tasks, and the fact that the Pivotal Spring team currently uses it to build all Spring projects.

! Verify that the version of Gradle the operating system sees is the one you just installed by opening a terminal (Command Prompt in Windows, and any type of terminal you have installed on MacOS and Linux) and typing

```
gradle -version
```

You should see something similar to this:

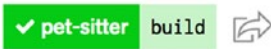
```
-----  
Gradle 2.11  
-----
```

```
Build time:    2016-02-08 07:59:16 UTC  
Build number: none  
Revision:     584db1c7c90bdd1de1d1c4c51271c665bfcb978  
  
Groovy:       2.4.4  
Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013  
JVM:         1.8.0_74 (Oracle Corporation 25.74-b02)  
OS:          -- whatever operating system you have --
```

The text above being displayed is confirmation that Gradle commands can be executed in your terminal; thus Gradle was installed successfully.

The reason Gradle was used to build the projects for this book is its simplicity. Gradle was presented recently as the modern open source polyglot build automation system, and the Gradle team now also helps you analyze your builds in order to prove it. Gradle now offers the possibility of registering a receipt on their site that will be used to connect to the site and generate the build statistics. The project of this book will use this new service provided by the Gradle team to keep you informed on how the project grows with each chapter.

If you enter <https://gradle.com/demo/> and follow the instructions in the demo, you can build your project and get a set of statistics about how healthy your project and team are. In the first step of conception, the project is quite simple, so the initial automatic build does not provide much information. In Figure 1-2, you can see the statistics for the project in the initial phase. As you can see, the build takes one second, no tests are run, and the JVM needed a maximum of 910 MB of memory to run this build.



Ran for 1 sec
 Started today at 11:43:59 PM +02:00
 Gradle 2.11

1 sec

+02:00 Started on **Feb 17, 2016** at **11:43:59 PM** Finished on **Feb 17, 2016** at **11:44:00 PM**

EET Started on **Feb 17, 2016** at **11:43:59 PM** Finished on **Feb 17, 2016** at **11:44:00 PM**

:00-ps-core:compileJava UP-TO-DATE	226 ms	91%
:00-ps-core:processResources UP-TO-DATE	5 ms	2%
:00-ps-core:jar UP-TO-DATE	4 ms	1%
:01-ps-start-practice:processResources UP-TO-DATE	3 ms	1%
:01-ps-start-practice:jar UP-TO-DATE	3 ms	1%
:00-ps-core:processTestResources UP-TO-DATE	1 ms	
:00-ps-core:test UP-TO-DATE	1 ms	
:01-ps-start-practice:classes UP-TO-DATE	1 ms	

See all tasks

Daemon ⓘ	On
Parallel ⓘ	Off
Refresh dependencies ⓘ	Off
Re-run tasks ⓘ	Off
Continuous ⓘ	Off

See 4 more inactive switches

 Java 1.8	 Mac OS X 10.10.5	 8 cores
 JVM max mem 910 MB	 8 max workers	 English (United States)

Figure 1-2. Gradle.com build statistics

Recommended IDE



The recommended IDE to use in this study guide is IntelliJ IDEA. The reason for this is that it is the most intelligent Java IDE. IntelliJ IDEA offers outstanding framework-specific coding assistance and productivity-boosting features for Java EE, and Spring also includes support for Maven and Gradle. It is the

perfect choice to help you focus on learning Spring, and not how to learn to use an IDE. It can be downloaded from the JetBrains official site <https://www.jetbrains.com/idea/>. It is also light on your operating system and quite easy to use.

Since Spring Boot will be used to run the web applications in the project attached to the book, you can use the community edition in order to build and solve the TODOs in the project. But if you are looking for a professional experience in development with Java and Spring, you can try working with the Ultimate Edition, which has a trial period of thirty days. The figures with code being run, launchers being created, and other IDE-related details in this book are made using an IntelliJ IDEA Ultimate version.

I believe that an IDE should be so easy to use and so intuitive that you can focus on what really matters: the solution you are implementing. But in case you are already familiar with a different Java editor, you can go ahead and use it as long as it supports Gradle.

The Project Sample

The project attached to this book is called **Pet Sitter**. As you have probably figured out, it is a proof of concept for an application designed to help pet owners find people to take care of their pets while they are on vacation or are forced to leave the pet alone for some reason. Here is what this project should provide:

- A user should have a secured account to access the application. The type of the account can be:
 - OWNER = user that is only looking for a pet sitter for its pet(s)
 - SITTER = user that is only looking to provide pet sitter services
 - BOTH = both of the above
 - ADMIN = account with special privileges that can manage other users' activities on the site.
- A user account of type OWNER can have one or more pet instances associated with it.
- Each pet must have an RFID¹¹ microchip implanted, and the barcode should be provided to the application.
- A user account of type OWNER is able to create a request for a pet sitter for more than one interval.
- A user account of type SITTER can reply to requests by creating response objects that will be approved or rejected by the owner of that request.
- A user account of type BOTH can act as an OWNER and as a SITTER.
- Admin accounts can deactivate other types of accounts for inactivity.
- Pet sitters and owners can rate each other by writing a review of their experience. The results are stored in a rating field attached to the user account.

The project is a multimodule Gradle project. Every module is a project that covers a specific Spring Topic. The projects suffixed with `practice` are missing pieces of code and configuration and are the ones that need to be solved by you to test your understanding of Spring Web. The projects suffixed with `solution` are a proposal resolution for the tasks. Some projects are suffixed with `sample` to tell you that they contain a sample of code or configuration that you are to analyze and pay special attention to.

In Figure 1-3, the structure of the Pet Sitter project as it is viewed in IntelliJ IDEA is depicted. Each module name is prefixed with a number, so no matter what IDE you use, you will always have the modules in the exact order in which they were intended to be used.

¹¹Radio-frequency identification (RFID) uses electromagnetic fields to automatically identify and track tags attached to an object, or pet in this case.

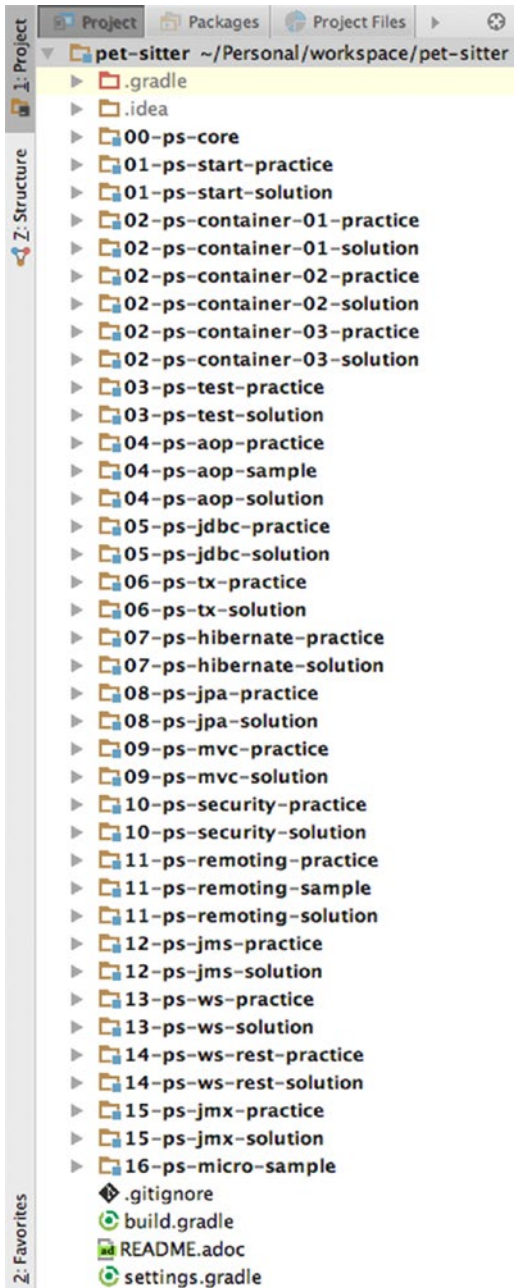


Figure 1-3. Pet Sitter modules

The **00-ps-core** contains the entity classes that map on database tables, enumerations, and other utility classes that are referenced from other modules. As the name of the project implies, this is the core project, the base tier. The other projects are implementation of service tiers that are built upon it. The Pet Sitter was designed with the multitier architecture in mind, and the abstract internal layer structure is depicted in Figure 1-4.

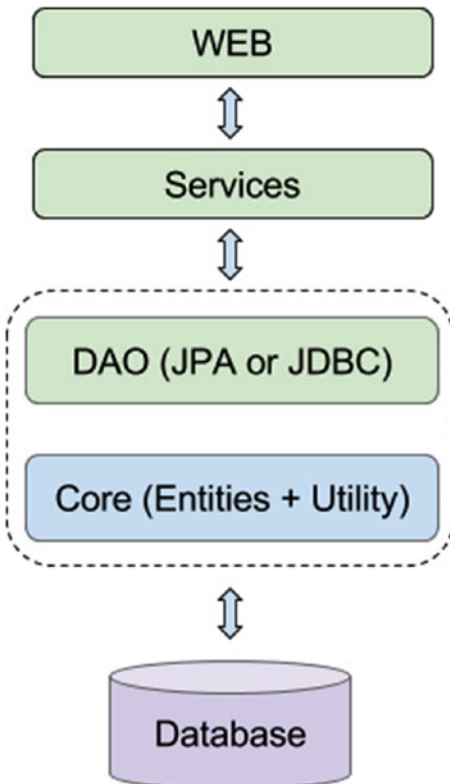


Figure 1-4. *Pet Sitter application layers*

The entities have common fields used by hibernate to identify uniquely each entity instance (`id`) and fields used to audit each entity instance (`createdAt` and `modifiedAt`) and keep track of how many times an entity was modified (`version`). These fields have been grouped in the `AbstractEntity` class to avoid having duplicated code. Other classes are enumerations used to define different types of objects and other utility classes (for conversion and serialization). The contents of the **00-ps-core** project are depicted in Figure 1-5.

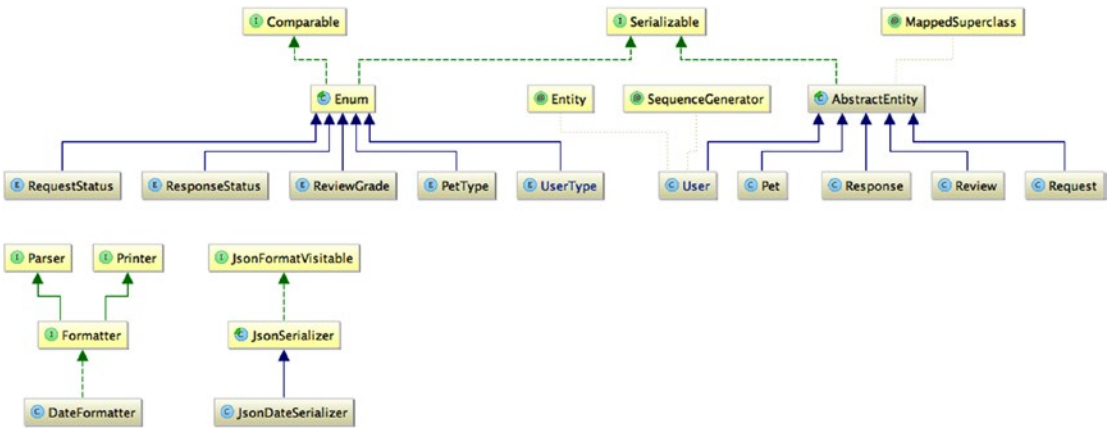


Figure 1-5. Pet Sitter 00-ps-core project contents

The class hierarchy, class members, and relationships between classes can be analyzed in Figure 1-6.

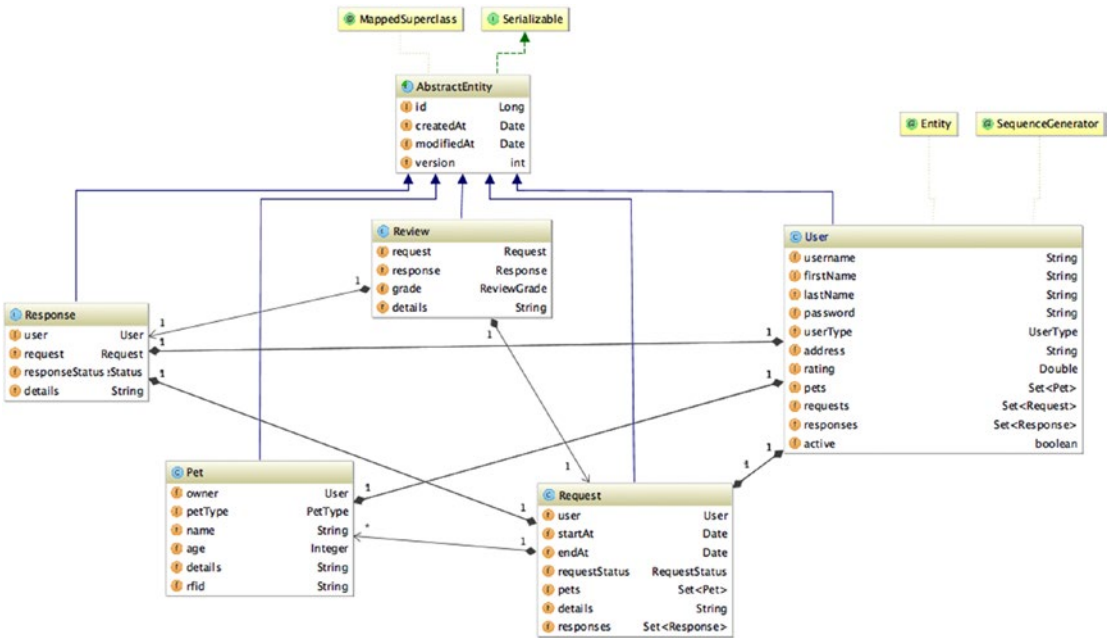


Figure 1-6. Pet Sitter entity class hierarchy

The UML diagram in Figure 1-7 describes the general functionality of the application. The RequestDispatcher and Controller are part of the web tier and are included here because the **09-ps-mvc-*** and **10-ps-security-*** projects also have a simple web tier in place, and basic notions of Spring Web are part of the certification exam.

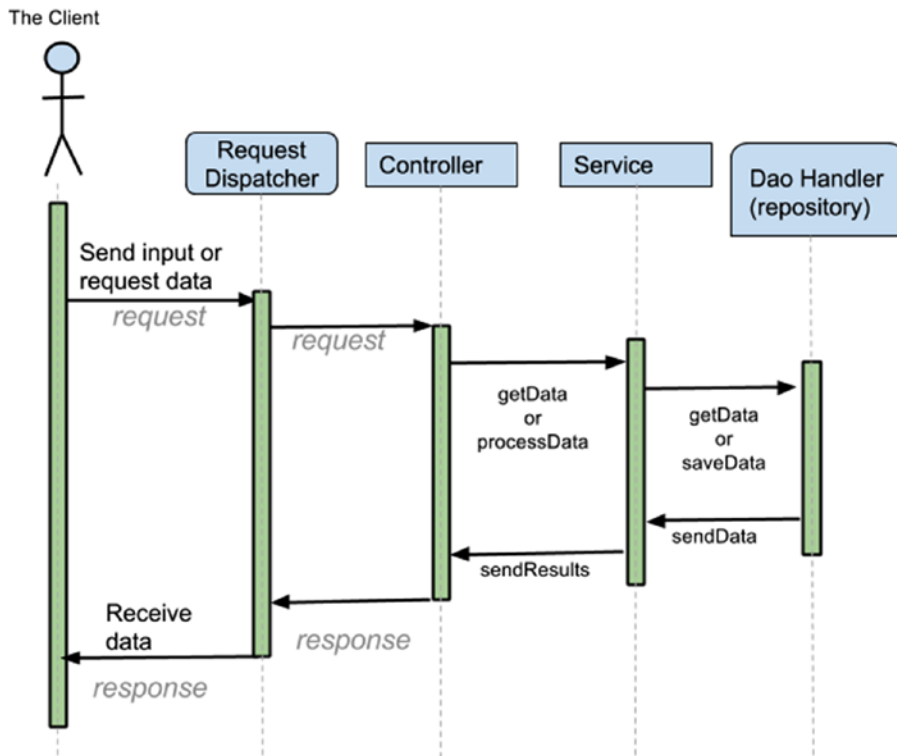


Figure 1-7. UML diagram describing the general behavior of the Pet Sitter application

This chapter does not have any practice and sample code attached to it, so more information regarding the setup of the project and how it is built and executed will be provided in the following chapters.

CHAPTER 2



Spring Bean LifeCycle and Configuration

The Spring Framework provides an easy way to create, initialize, and connect objects into competent, decoupled, easy to test enterprise-ready applications. Every software application consists of software components that interact, that collaborate and depend on other components to successfully execute a set of tasks. Each software component provides a service to other components, and linking the customer and the provider component is the process known as *Dependency Injection*. Spring provides a very simplistic way to define the connections between them in order to create an application.

Before Spring entered the picture, defining connections between classes and composing them required for development to be done following different design patterns, such as *Factory*, *Abstract Factory*, *Singleton*, *Builder*, *Decorator*, *Proxy*, *Service Locator*, and even reflection.¹ Spring was built in order to make *Dependency Injection* easy. This software design pattern implies that clients delegate the dependency resolution to an external service. The client is not allowed to call the injector service, which is why this software pattern is characterized by *Inversion of Control* behavior, also known as the *Don't call us, we'll call you!* principle.

The software components that Spring uses to build applications are called *beans* and are nothing more than *Plain Old Java Objects* (POJOs) that are being created, initialized, assembled, and managed by the Spring *Inversion of Control* container. The order of these operations and the relationships between objects are provided to the Spring IoC container using XML configuration files prior to Spring version 2.5. Starting with 2.5, a small set of annotations was added for configuring beans, and with Spring 3, Java configuration using annotations was introduced.

This chapter covers everything a developer needs to know in order to configure a basic Spring application using XML and Java Configuration. The Java annotations that represent the intermediate step between configurations using XML and full Java Configuration will also be covered.

Old Style Application Development

In the most competent development style, a Java application should be composed of POJOs, simple Java objects each with a single responsibility. In the previous chapter, the entity classes that will be used throughout the book were introduced along with the relationships between them. In order to manage this type of object at the lowest level, the *dao (repository) layer* of the application, classes named repositories will be used. The purpose of these classes is to retrieve, update, create, and delete entities from the storage support, which usually is some type of database.

¹If you are interested in more books about Java Design Patterns, you can check out this book from Apress:
<http://www.apress.com/9781484218013?gtmf=s>.

These classes are called repositories, and in the code for this book, their name is created by concatenating a short denomination for the type of entity management, the name of entity object being managed, and the *Repo* postfix. Each class implements a simple interface that defines the methods to be implemented to provide the desired behavior for that entity type. All interfaces extend a common interface declaring the common methods that should be implemented for any type of entity. For example: saving, searching by id, and deleting should be supported for every type of entity. This method of development is used because Java is a very object-oriented programming language, and in this case, the inheritance principle is very well respected. Also, generic types make possible such a level of inheritance.

In Figure 2-1, the *AbstractRepo* interface and the child interfaces for each entity type are depicted. You can notice how the *AbstractRepo* interface defines typical method skeletons for every entity type, and the child interface defines method skeletons designed to work with only a specific type of entity.

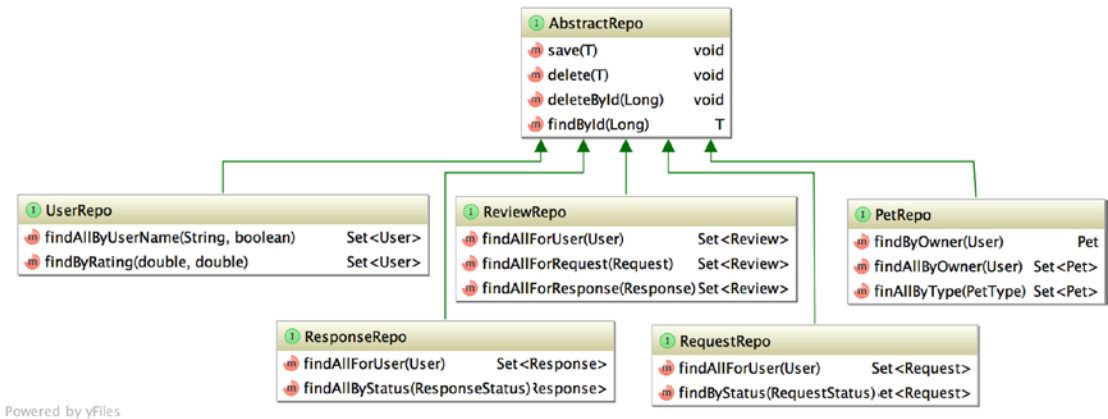


Figure 2-1. Repository interfaces hierarchy

At the end of this section you can take a break from reading in order to get comfortable with this implementation. The project specifically designed for this is named `01-ps-start-practice`. It contains stub implementations for the repository classes, which can be found in the test sources under the package `com.ps.repo.stub`. In Figure 2-2, the stub classes are depicted. Again, inheritance was used in order to reduce the amount of code to be written and avoid writing duplicate code.

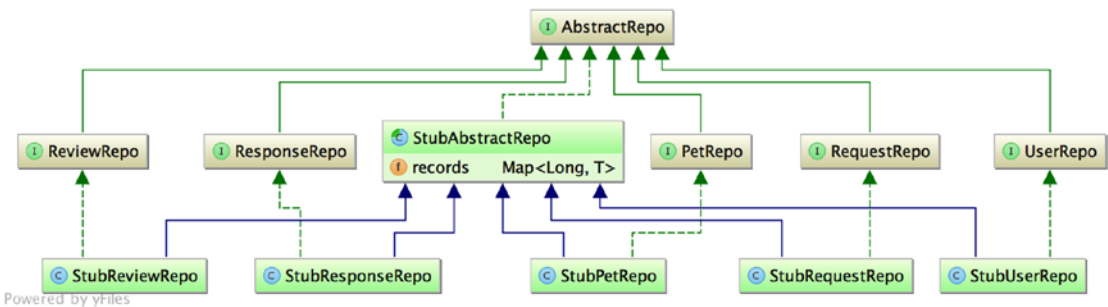


Figure 2-2. Repository stub implementations

The stub repositories store all the data created by the user in a map data structure named records. The unique id for each record is generated based on the size of this map. The implementation is in the `StubAbstractRepo` class and is depicted below, together with all basic repository operations:

```
package com.ps.repo.stub;

import com.ps.base.AbstractEntity;
import com.ps.repos.AbstractRepo;
import java.util.HashMap;
import java.util.Map;

public abstract class StubAbstractRepo
    <T extends AbstractEntity> implements AbstractRepo<T> {
    protected Map<Long, T> records = new HashMap<>();
    @Override
    public void save(T entity) {
        if (entity.getId() == null) {
            Long id = (long) records.size() + 1;
            entity.setId(id);
        }
        records.put(entity.getId(), entity);
    }

    @Override
    public void delete(T entity) {
        records.remove(entity.getId());
    }

    @Override
    public void deleteById(Long entityId) {
        records.remove(entityId);
    }

    @Override
    public T findById(Long entityId) {
        return records.get(entityId);
    }
}
```

The next layer after the dao (repository) layer is the *service layer*. This layer is composed of classes doing modifications to the entity objects before being passed on to the repositories for persisting the changes to the storage support (database). The service layer is the bridge between the web layer and the dao layer and will be the main focus of the book. It is composed of specialized classes that work together in order to implement behavior that is not specific to web or data access. It is also called *the business layer*, because most of the application business logic is implemented here. Each service class implements an interface that defines the methods that it must implement to provide the desired behavior. Each service class uses one or more repository fields to manage entity objects. Typically, for each entity type, a service class also exists, but also more complex services can be defined that can use multiple entity types in order to perform complex tasks. In the code for this book, such a class is the `SimpleOperationsService` service, which contains methods useful for executing common Pet Sitter operations such as create a pet sitter request, accept a response, close a request, and rate the pet owner.

The most complex service class is called `SimpleOperationsService`, and this service class can be used to create a response, accept a response, or close a request. In Figure 2-3 all the service classes and interfaces are depicted.

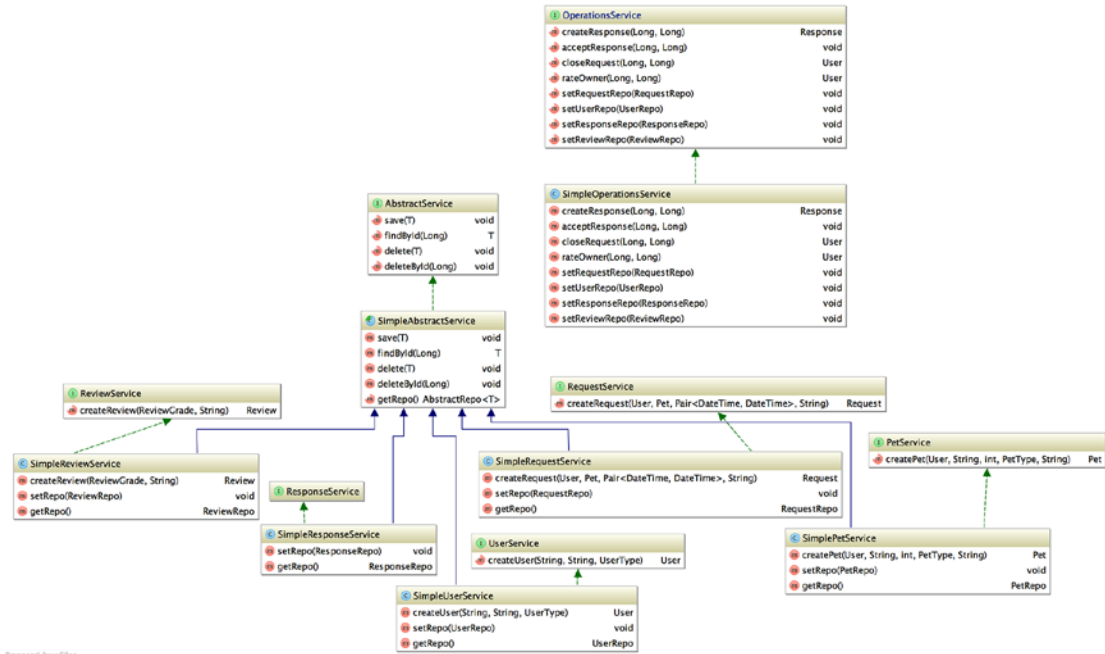


Figure 2-3. Service interfaces and implementations

All the classes presented here are parts that will be assembled to create an application that will manage user data. In a production environment, a service class needs to be instantiated, and a repository instance must be set for it so data can be managed properly. Applications running in production support complex operations such as transactions, security, messaging, remote access, and caching. They are connected to other applications and software components. In order to test them, pieces of them have to be isolated, and some of them that are not the object of testing are replaced with simplified implementations. In a test environment, stub or mock implementations can be used to replace implementations that are not meant to be covered by the testing process. In Figure 2-4, you can see a service class and a dependency needed for it in a production and test environment side by side.

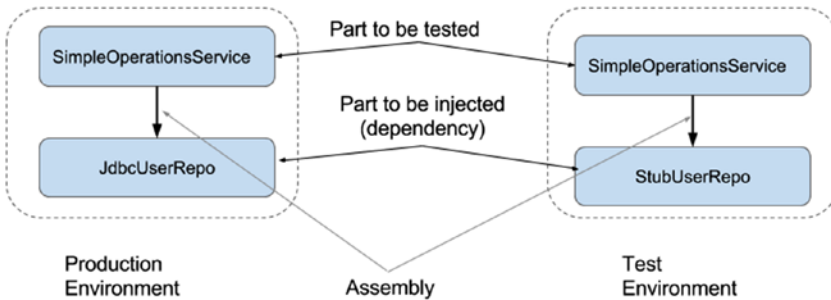


Figure 2-4. Service class and its dependency in different running environments

The code snippet relevant to the previous image is depicted below and is a piece of the `SimpleOperationsService` class definition.

```
public class SimpleOperationsService
    implements OperationsService {

    private UserRepo userRepo;
    ...

    public void setUserRepo(UserRepo userRepo) {
        this.userRepo = userRepo;
    }
}
```

As you can see, the dependency is defined using an abstract type, in this case, the `UserRepo` interface, so any implementation can be provided. So in a production environment, an instance of type `JdbcUserRepo` will be provided, and that type will be defined to implement the `UserRepo` interface.

```
public class JdbcUserRepo extends JdbcAbstractRepo<User>
    implements UserRepo {
    //implementation not relevant at this point
    ...
}
```

The creation of an instance of type `SimpleOperationsService` will require the following steps:

1. instantiate and initialize the repository instance


```
JdbcUserRepo userRepo = new JdbcUserRepo(...);
```
2. instantiate the service class


```
OperationsService service = new SimpleOperationsService();
```
3. inject the dependency


```
service.setUserRepo(repo);
```

In a test environment, a mock or a stub will do, as long as it implements the same interface:

```
public class StubUserRepo extends StubAbstractRepo<User>
    implements UserRepo {
    //implementation not relevant at this point
    ...
}
```

For the test environment, the assembling steps are the same:

1. `StubUserRepo userRepo = new StubUserRepo(...);`
2. `OperationsService service = new SimpleOperationsService();`
3. `service.setUserRepo(repo);`

Spring will help assembling the components a very pleasant job. Swapping them depending on the environment is also possible in a practical manner with Spring. Because connecting components is so easy, writing tests becomes a breeze also, since each part can be isolated from the others and tested without any unknown influence. Spring provides support for writing tests via the `spring-test.jar` library.

! And now that you know what Spring can help you with, you are invited to have a taste of how things are done without it. Take a look at the `01-ps-start-practice` project. In the `SimpleOperationsService` class there is a method named `createResponse` that needs an implementation. The steps to create a `Response` instance are:

1. retrieve the sitter `User` instance using `userRepo`
2. retrieve the `Request` instance using `requestRepo`
3. instantiate a `Response` instance
4. populate the `Response` instance:
 - set the response status to `ResponseStatus.PROPOSED`
 - set the user to the sitter instance
 - set details with a sample text
 - add the response to the request object
 - save the response object using the `verb|responseRepo`

Figure 2-5 depicts the sequence of operations.

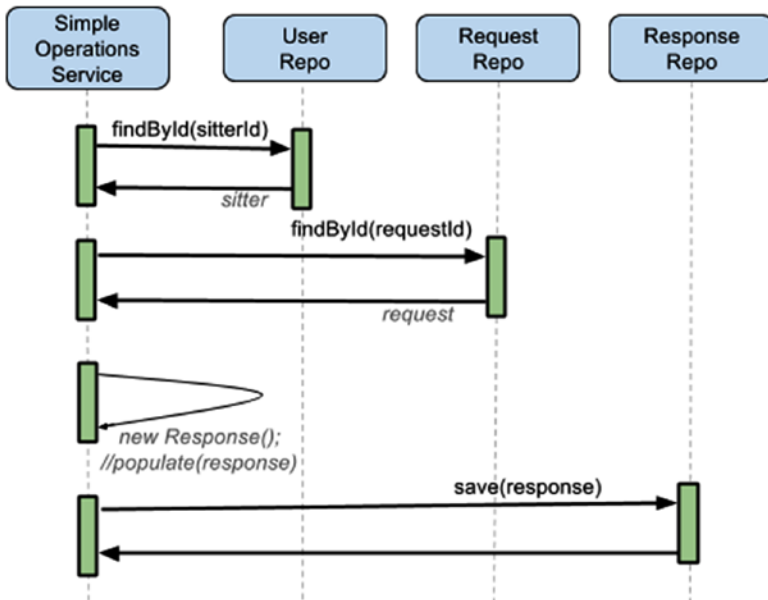


Figure 2-5. Service class and its dependency

To run the implementation, search for the class `com.ps.repo.services.SimpleOperationsServiceTest` under the test directory. Inside this class there is a method annotated with `@Test`. This is a JUnit annotation. More details about testing tools are covered in Chapter 3, *Testing Spring Applications*. In order to run a unit test in IntelliJ IDEA, just click right on the method name, and a menu like the one in Figure 2-6 will be displayed. Select the Run option to run the test. Select Debug if you want to run the test in debug mode and check field values.



Figure 2-6. JUnit test contextual menu in IntelliJ IDEA

If the implementation is not correct, the test will fail, and in the IntelliJ IDEA console you should see something similar to what is depicted in Figure 2-7. And yes, a red progress bar is a clear sign that the test failed.

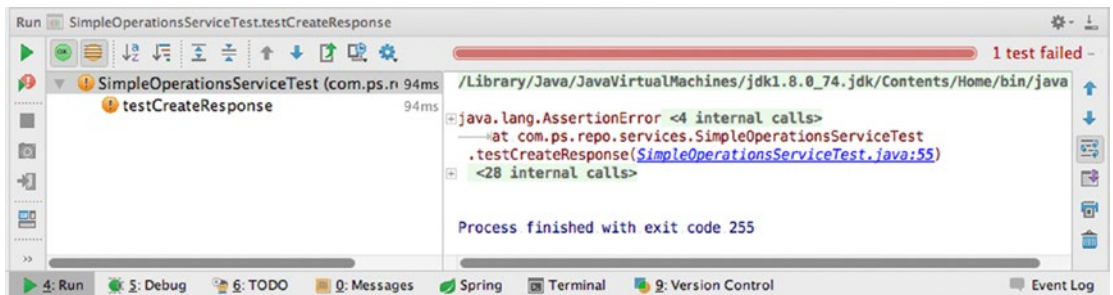


Figure 2-7. JUnit test failure in IntelliJ IDEA

If the implementation is correct, the test will pass, and a lot of green in the IntelliJ IDEA console, as in Figure 2-8, is a sign that everything went ok.

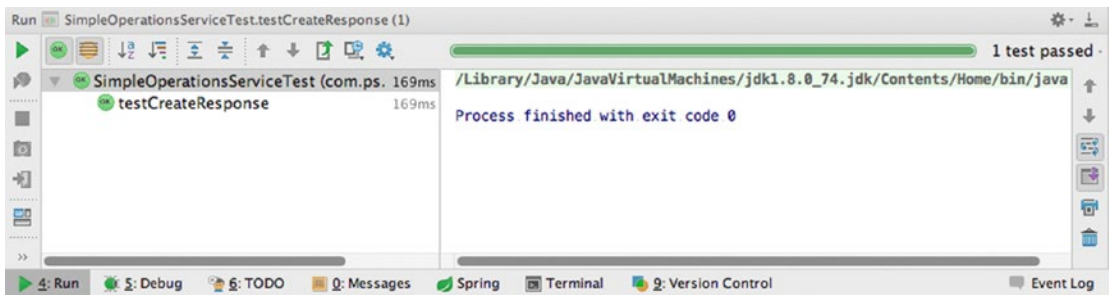


Figure 2-8. JUnit test passed in IntelliJ IDEA

You can check the solution by comparing it with the proposed code from the 01-ps-start-solution.

Spring IoC and Dependency Injection

At the beginning of this chapter, it was mentioned that creating objects and assembling them is what Spring does. The Spring Framework Inversion of Control (IoC) component is the nucleus of the framework. It uses dependency injection to assemble Spring-provided (also called infrastructure components) and development-provided components in order to rapidly wrap up an application. Figure 2-9 depicts where the Spring IoC fits in the application development process.

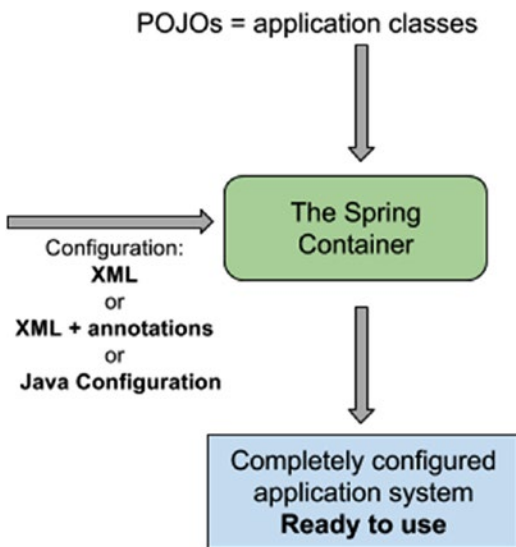


Figure 2-9. Spring IoC purpose

The application being developed over the course of this book includes service classes that are built using repository instances. For example, this is how a `UserService` implementation can be defined:

```
public class SimpleUserService extends SimpleAbstractService<User>
    implements UserService {

    private UserRepo repo;

    public SimpleUserService(UserRepo userRepo) {
        this.repo = userRepo;
    }
    ...
}
```

As you can see, the dependency is injected using a constructor, so creating the service instance requires a repository instance to be provided as a parameter. The repository instance is needed in order to retrieve and persist `User` objects in the database. The repository can be defined like this:

```
import javax.sql.DataSource;

public class JdbcUserRepo extends JdbcAbstractRepo<User> {

    public JdbcUserRepo(DataSource dataSource) {
        super(dataSource);
    }
    ...
}
```

//JdbcAbstractRepo.java contains common
//implementation for all repository classes
import javax.sql.DataSource;

```
public class JdbcAbstractRepo<T> extends AbstractEntity>
    implements AbstractRepo<T> {
    protected DataSource dataSource;

    public JdbcAbstractRepo(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public void save(T entity) {...}

    @Override
    public void delete(T entity) {... }

    @Override
    public void deleteById(Long entityId) {...}

    @Override
    public T findById(Long entityId) {...}
}
```

In order to create a repository instance, a `DataSource` instance is needed, in order to connect to the database.

In order to tell the Spring Container to create these objects and how to link them together, a configuration must be provided. This configuration can be provided using XML files or XML + annotations or Java Configuration classes.

The following configuration snippet depicts the contents and template of an XML Spring Configuration file, used to define the components that will make up the application.²

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="simpleUserService" class="com.ps.services.impl.SimpleUserService">
    <property name="userRepo" ref="userRepo"/>
  </bean>

  <!-- Loads users from the data source -->
  <bean id="userRepo" class="com.ps.repos.impl.JdbcUserRepo">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource">
    <property name="url" value="jdbc:oracle:thin:@sample:1521:PET"/>
    <property name="username" value="sample"/>
    <property name="password" value="sample"/>
  </bean>
</beans>
```

The XML file is usually placed in the project directory under `src/main/resources/spring`. The `spring` directory tells you that files under it are Spring configuration files, because an application can have more XML configuration files for other purposes (configuring other infrastructure components like Hibernate, configuring caching with Ehcache, logging, etc.). An application configured in this way will run in an application context created by the Spring container.

The application context will manage all objects instantiated and initialized by the Spring IoC, which from now on we will refer to as beans in order to get you accustomed with the Spring terminology. The relationship among these objects and the application context is depicted in Figure 2-10 along with their unique identifier.

²Oracle was used for data storage in this example because most production applications use Oracle for storage, and this book aims to provide real configurations such as you will probably encounter and need while working in software development.

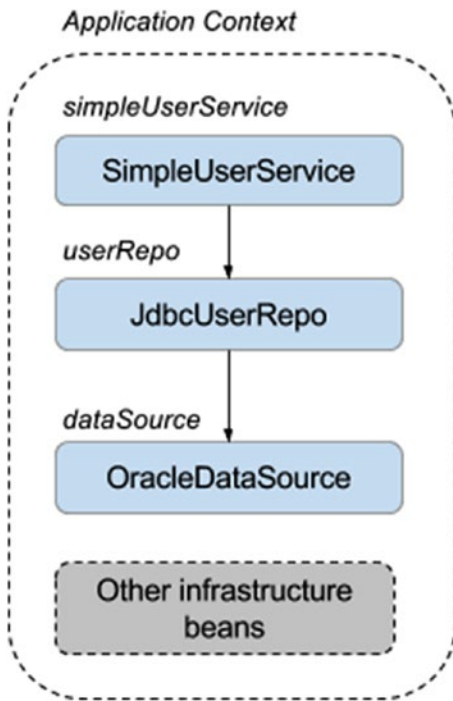


Figure 2-10. Application Context and the beans managed by it

The application context is nothing else than a class implementing the Spring interface `org.springframework.context.ApplicationContext` that needs to be instantiated and the configuration file given as an argument. There are more implementations for the application context provided by Spring, and the one to use depends on the location and the resources containing the configuration. For XML, the class `org.springframework.context.support.ClassPathXmlApplicationContext` is used.

```

//creating the context
(1)ApplicationContext context = new ClassPathXmlApplicationContext
    ("classpath:spring/application-config.xml");

// Get the bean to use to invoke the service
(2)UserService userService = (UserService)context.get("simpleUserService");

// create user entity
(3)User user = new User();
// populate user

// invoking the save method of the bean
(4)userService.save(user);
  
```

In the previous example, a lot of information is new and will be introduced gradually in the next sections. The `classpath` is a common prefix used in Spring applications configured using XML files, to tell the Spring IoC where the configuration is located. The bean unique identifier in the application context, the bean id was evidenced by underlining its value, in order to give you a hint as to how the identification of a certain instance is done. More on this topic will be covered in the following sections.

Configuration using Java Configurations can be done as in the following code snippet, but what everything means will be covered in more detail in the following sections. The method names used to create the beans were evidenced by underlining them, in order to give you a hint as to how the bean id is determined.

```
@Configuration
@PropertySource("classpath:db/datasource.properties")
public class ApplicationConfig {

    @Value("${url}")
    private String url;
    @Value("${username}")
    private String username;
    @Value("${password}")
    private String password;

    @Bean
    public UserService simpleUserService() throws SQLException {
        return new SimpleUserService(userRepo());
    }

    @Bean
    public UserRepo userRepo() throws SQLException {
        return new JdbcUserRepo(dataSource());
    }

    @Bean
    public DataSource dataSource() throws SQLException {
        OracleDataSource ds = new OracleDataSource();
        ds.setURL(url);
        ds.setUser(username);
        ds.setPassword(password);
        return ds;
    }
}
```

The `datasource.properties` file contains the database access credentials, which are set for the fields annotated with `@Value`, but how this works will be explained in greater detail in the Java Configuration section. For Java Configuration classes, the class `org.springframework.context.annotation.AnnotationConfigApplicationContext` is used to create an application context.

```
ApplicationContext ctx = new AnnotationConfigApplicationContext
    (ApplicationConfig.class);
// everything wires up across configuration classes...
SimpleUserService simpleUserService =
    (SimpleUserService) ctx.getBean("simpleUserService");
```

And now that the surface has been scratched, you are ready to dive into advanced Spring configuration.

Spring Configuration

The beans definitions that make up a Spring application are provided using XML files, annotations, Java-based Configuration annotations, or all of them together. XML was the first method for storing bean definitions that Spring used, probably because annotations had not yet been invented when Spring was. A form of annotation-based configuration was introduced in Spring 2.5, when annotations such as `@Component`, `@Service`, `@Repository`, also called stereotype annotations, were introduced to define common business beans. But the most common way to provide the configuration for a Spring application nowadays is Java-based Configuration, which introduces annotations such as `@Configuration`, `@Bean` for infrastructure beans beginning with Spring 3.0. The stereotype annotations and the Java Configuration annotations complement each other to provide a practical, non-XML way to define the configuration for a Spring application. XML configuration is still supported because of legacy code and to support diversity. Indeed, there are programmers who still prefer to completely decouple all configuration from code, and not writing extra code just for configuration and Java-based Configuration and/or annotations is not quite allowed.

In this section, all aspects of configuration and dependency injection types will be covered, so get yourself a big cup of coffee (or tea) and start reading.

Providing Configuration via XML

In order to provide an XML configuration for a Spring application that will define one or more bean definitions, the file must use certain namespaces and respect the structure defined by them. A bean definition specifies the bean id, name, alias, type, arguments used for its initialization, dependencies, and many more items. The main namespace is the beans namespace, which specifies what a bean definition should look like. The namespace has a matching XSD, which needs to be added to the `<beans/>` element so that the configuration file can be validated according to the rules defined by it. A Spring XML configuration file must have at least this namespace defined. The code snippet below depicts the standard XML-based configuration metadata:

```
<?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="..." class="...">
    <!-- props to set -->
  </bean>
</beans>
```

! In the code sample above, the Spring version is not used in the schema location declaration. If used, the schema location element would look like this:

```
...
xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd">
...
```

A recommended (best) practice is not to do this, since the version will be correctly identified from the Spring dependency version in the project. Also, the other advantage is that you can upgrade the Spring version you are using, and the new definition specifications will be automatically supported in your configuration files without your having to modify them.

For bean definitions to be read and for beans to be created and managed, many Spring-provided beans must be created too. That is why a few core Spring modules must be added as dependencies to your project:

- `spring-core`: fundamental parts of the Spring Framework
- `spring-beans`: together with `spring-core` provide the core components of the framework, including the Spring IoC and dependency Injection features.
- `spring-context`: expands the functionality of the previous two, and as the name says, it contains components that help build and use an application context. The `ApplicationContext` interface is part of this module, being the interface that every application context class implements.
- `spring-context-support`: provides support for integration with third-party libraries, for example Guava, Quartz, FreeMarker, and many more.
- `spring-expressions`: provides a powerful Expression Language (SpEL) used for querying and manipulating objects at runtime; for example, properties can be read from external sources decided at runtime and used to initialize beans. But this language is quite powerful, since it also supports logical and mathematical operations, accessing arrays, and manipulating collections.

And now that that you know where all the infrastructure components come from, we can return to the configuration details and show you what a bean definition looks like. The simplest bean definition 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"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

</beans>
```

In the configuration above we just defined a bean with the id `simpleBean` of type `com.ps.SimpleBeanImpl` that has no dependencies. In order to define a bean with dependencies, we have to decide how those dependencies are injected. In Spring, there are two types of dependency injection specific to XML: via constructor and via setters, and one that can be used only using annotations and Java Configuration: field injection, which will be covered in the next section.

Constructor Injection

Constructor injection can be used to define beans when the bean type is a class that has a constructor with arguments defined. If the bean definition looks like this:

```
<beans ...>

  <bean id="complexBean" class="com.ps.beans.ctr.ComplexBeanImpl">
    <constructor-arg ref="simpleBean" />
  </bean>

<bean id= "simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

</beans>
```

Then the class `ComplexBeanImpl` looks like this:

```
public class ComplexBeanImpl implements ComplexBean {

    private SimpleBean simpleBean;
    public ComplexBeanImpl(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }
    ...
}
```

The code and configuration snippets above provide all the necessary information so that the Spring container can create a bean of type `SimpleBeanImpl` named `simpleBean` and then inject it into a bean of type `ComplexBeanImpl` named `complexBean`. What is done behind the scenes is:

```
SimpleBean simpleBean = new SimpleBeanImpl();
ComplexBean complexBean = new ComplexBeanImpl(simpleBean);
```

Spring creates the beans in the order they are needed. The dependencies are first created and then injected into the beans that need them.

What you have to remember when it comes to constructor injection is that the template for a bean definition using it looks like this:

```
<bean id="..." class="...">
  <constructor-arg ref="..." />
</bean>
```

The `<constructor-arg />` element defines the constructor argument and does so using a number of attributes, but `ref` is the most common, and it is used to tell the container that the value of this attribute is a reference to another bean. The complete list of attributes and documentation for each one can be found in `spring-beans-4.3.xsd`, which can be accessed easily from IntelliJ IDEA by pressing CTRL (Command in MacOS systems) while clicking on `constructor-arg` in your configuration file.

Another attribute that is commonly used is `value`. This attribute is used when the value to inject is a scalar.³ So text values, numbers, booleans, all can be used as arguments for the constructor using the `value` attribute.

```
<beans ...>
  <bean id="complexBean" class="com.ps.beans.ctr.ComplexBeanImpl">
    <constructor-arg ref="simpleBean"/>
    <constructor-arg value="true"/>
  </bean>
...
</beans>
// ComplexBeanImpl.java constructor
public ComplexBeanImpl(SimpleBean simpleBean, boolean complex) {
    this.simpleBean = simpleBean;
    this.complex = isComplex;
}
```

So what happens here is equivalent to:

```
ComplexBean complexBean = new ComplexBeanImpl(simpleBean, true);
```

And also quite useful is the `index` attribute, which should be used when the constructor has more parameters of the same type.

```
<beans ...>
  <bean id="simpleBean0" class="com.ps.beans.SimpleBeanImpl"/>
  <bean id="simpleBean1" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="complexBean2" class="com.ps.beans.ctr.ComplexBean2Impl">
    <constructor-arg ref="simpleBean0" index="0"/>
    <constructor-arg ref="simpleBean1" index="1"/>
  </bean>
</beans>

// ComplexBean2Impl.java constructor
public ComplexBean2Impl(SimpleBean simpleBean1, SimpleBean simpleBean2) {
    this.simpleBean1 = simpleBean1;
    this.simpleBean2 = simpleBean2;
}
```

So what happens here is equivalent to:

```
ComplexBean complexBean2 = new ComplexBean2Impl(simpleBean0, simpleBean1);
```

Another way to handle constructors with more parameters of the same type is to use the `name` attribute, which receives as its value the name of the parameter of the constructor. And obviously, using it makes the configuration a lot clearer and easier to read. So for the `complexBean2` defined previously, there is another configuration that does the same job.

³The term “*scalar*” comes from linear algebra, where it is used to differentiate a number from a vector or matrix. In computing, the term has a similar meaning. It distinguishes a single value such as an integer or float from a data structure like an array. In Spring, *scalar* refers to any value that is not a bean and cannot be treated as such.

```

<beans ...>
  <bean id="simpleBean0" class="com.ps.beans.SimpleBeanImpl"/>
  <bean id="simpleBean1" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="complexBean2" class="com.ps.beans.ctr.ComplexBean2Impl">
    <constructor-arg ref="simpleBean0" name="simpleBean1"/>
    <constructor-arg ref="simpleBean1" name="simpleBean2"/>
  </bean>
</beans>

```

Developers choose to use constructor injection when it is mandatory for the dependencies to be provided, since the bean depending on them cannot be used properly without them. Dependency injection also is suitable when a bean needs to be immutable, by assigning the dependencies to final fields. The most common reason to use constructor injection is that sometimes, third-party dependencies are used in a project, and their classes were designed to support only this type of dependency injection.

In creating a bean, there are two steps that need to be executed one after the other. The bean first needs to be instantiated, and then the bean must be initialized. The constructor injection combines two steps into one, because injecting a dependency using a constructor means basically instantiating and initializing the object at the same time.

If `constructor-arg` seems to be a long name for a configuration element, do not worry. Spring has introduced a fix for that in version 3.1 the *c-namespace*: `xmlns:c="http://www.springframework.org/schema/c"`. This namespace allows usage of inline attributes for configuring the constructor arguments and does not have an XSD schema, since it does not introduce any new configuration elements. Just add it to the bean configuration file and your configuration can be simplified like this:

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:c="http://www.springframework.org/schema/c"
  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="simpleBean0" class="com.ps.beans.SimpleBeanImpl"/>
  <bean id="simpleBean1" class="com.ps.beans.SimpleBeanImpl"/>

  <!-- usage for reference to dependency -->
  <bean id="complexBean0" class="com.ps.beans.ctr.ComplexBeanImpl"
    c:simpleBean-ref="simpleBean0"/>

  <!-- usage for primitive type dependency -->
  <bean id="complexBean1" class="com.ps.beans.ctr.ComplexBeanImpl"
    c:simpleBean-ref="simpleBean0" c:complex="true"/>

  <!-- usage for index specified references -->
  <bean id="complexBean2" class="com.ps.beans.ctr.ComplexBean2Impl"
    c:_0-ref="simpleBean0" c:_1-ref="simpleBean1" />
</beans>

```

What you should remember when using the *c-namespace* is that when the dependency is another bean, the "-ref" postfix is added to the c: prefixed attributes.

! If you are using the name of the constructor parameter to inject the dependency, then the attribute definition with c: should match the pattern `c:nameConstructorParameter[-ref]`, while if you are using indexes, the attribute definition should match `c:_{index}[-ref]`.

! All these examples can be found in the `02-ps-container-01-practice` project. The package containing all beans and configurations for trying out constructor injections is named `com.ps.beans.ctr`. The configuration files are stored under `02-ps-container-01-practice/src/main/resources/spring/ctr`, and there are two configuration files: `sample-config-01.xml`, containing the bean definitions using the `constructor-arg` element, and `sample-config-02.xml`, containing the bean definitions using the *c-namespace*.

In order to test the validity of these files, run the test in the `com.ps.beans.ctr.CIBeansTest` class. There is also a TODO task in there, asking you to retrieve beans of types `ComplexBean` and make sure that their dependencies were correctly set. How you do this is up to you, since there are many ways to do it. A solution is provided for you in the `02-ps-container-01-solution`. If you want, you can take a break from the book and try to solve the task now.

Setter Injection

In order to use setter injection, the class type of the bean must have setter methods used to set the dependencies. A constructor is not mandatory. If no constructor is declared, Spring will use the default “no argument” constructor, which every class automatically inherits for the Java Object class to instantiate the object and the setter methods to inject dependencies. If a no argument constructor is explicitly defined, Spring will use it to instantiate the bean. If a constructor with parameters is defined, the dependencies declared in this way will be injected using `constructor-injection`, and the ones defined using setters will be injected via `setter injection`.

In conclusion, when creating a bean using setter injection, the bean is first instantiated by calling the constructor and then initialized by injecting the dependencies using setters. (Of course, you will learn later in this chapter that initialization can split into two steps too: setting dependencies and calling a special `init` method that processes those dependencies.) So if your bean definition looks like this:

```
<beans ...>
  <bean id="simpleBean0" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="complexBean" class="com.ps.beans.set.ComplexBeanImpl">
    <property name="simpleBean" ref="simpleBean"/>
  </bean>
</beans>
```

Then the class `com.ps.beans.set.ComplexBeanImpl` looks like this:

```
public class ComplexBeanImpl implements ComplexBean {
    private SimpleBean simpleBean;

    // no-argument empty constructor, not mandatory
    public ComplexBeanImpl() {}

    public void setSimpleBean(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }

    public SimpleBean getSimpleBean() {
        return simpleBean;
    }
}
```

The code and configuration snippets above provide all the necessary information so that the Spring container can create a bean of type `SimpleBeanImpl` and a bean named `simpleBean0` and then inject it into a bean of type `ComplexBeanImpl` named `complexBean`. The dependency is injected after the bean is instantiated by calling the `setSimpleBean` method and providing the *simpleBean* as argument. What is done behind the scenes is:

```
SimpleBean simpleBean = new SimpleBeanImpl();
ComplexBean complexBean = new ComplexBeanImpl();
complexBean.setSimpleBean(simpleBean);
```

What you have to remember when it comes to setter injection is that the template for a bean definition using it looks like this:

```
<bean id="..." class="...">
    <property name="..." ref="..." />
</bean>
```

The `<property />` element defines the property to be set and the value to be set with and does so using a pair of attributes: `[name, ref]` or `[name,value]`.

The `name` attribute is **mandatory**, because its value is the name of the bean property to be set.

The `ref` attribute is used to tell the container that the value of this attribute is a reference to another bean.

The `value`, as you probably suspect, is used to tell the container that the value is not a bean, but a scalar value.

```
<beans ...>

    <bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

    <bean id="complexBean" class="com.ps.beans.set.ComplexBeanImpl">
        <property name="simpleBean" ref="simpleBean"/>
        <property name="complex" value="true"/>
    </bean>
</beans>
```

```
// ComplexBeanImpl.java
public class ComplexBeanImpl implements ComplexBean {
    private SimpleBean simpleBean;
    private boolean complex;
    public ComplexBeanImpl() {}

    public void setSimpleBean(SimpleBean simpleBean) {
        this.simpleBean = simpleBean;
    }

    public void setComplex(boolean complex) {
        this.complex = complex;
    }
}
```

As with construction injection, Spring also has a namespace for simplifying XML definition when one is using setter injection. It is called the *p-namespace*: <http://www.springframework.org/schema/p>. This namespace allows the use of inlined attributes for configuring property arguments and does not have an XSD schema, since it does not introduce any new configuration elements. Just add it to the bean configuration file and your configuration can be simplified. The `-ref` postfix for the `"p:"` attributes has the role of telling the container that the value of the attribute is a reference to another bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       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="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

    <bean id="complexBean" class="com.ps.beans.set.ComplexBeanImpl"
          p:simpleBean-ref="simpleBean" p:complex="true"/>
</beans>
```

And this is all that can be said about the setter injection for now. Choosing between setter and constructor injection depends only on the needs of the application, the code that is already written and cannot be changed (legacy code) and third party libraries. The advantage of setter injection is that the dependencies are optional and defaults can be used, and also inheritance. When deciding on what to use, follow the best practices as much as possible:

- Use constructor injection when properties are required.
- Use setter injection when properties are not required.
- Use the strategy matching the implementation of bean type.
- Be consistent.

Constructor and setter injection can be used together in creating the same bean. Here is an example:

```
<!-- typical configuration -->
<beans .../>
  <bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="complexBean2" class="com.ps.beans.set.ComplexBean2Impl">
    <constructor-arg ref="simpleBean"/>
    <property name="complex" value="true"/>
  </bean>
</beans>

<!-- configuration optimized using p-namespace and c-namespace -->
<beans .../>
  <bean id="complexBean2" class="com.ps.beans.set.ComplexBean2Impl"
    c:simpleBean-ref="simpleBean" p:complex="true"/>
</beans>

//ComplexBean2Impl.java
public class ComplexBean2Impl implements ComplexBean {
  private SimpleBean simpleBean;
  private boolean complex;

  public ComplexBean2Impl(SimpleBean simpleBean)
  {
    this.simpleBean = simpleBean;
  }

  public void setComplex(boolean complex) {
    this.complex = complex;
  }
}
```

! All these examples can be found in the `02-ps-container-01-practice` project. The package containing all beans and configurations for trying out setter injection is named `com.ps.beans.set`. The configuration files are stored under `02-ps-container-01-practice/src/main/resources/spring/set`, and there are two configuration files: `sample-config-01.xml`, containing the bean definitions using the `property` element, and `sample-config-02.xml`, containing the bean definitions using the *p-namespace*.

In order to test the validity of these files, run the test in the `com.ps.beans.set.SIBeansTest` class. There is also a TODO task in there, asking you to retrieve beans of types `ComplexBean` and make sure that their dependencies were correctly set. How you do this is up to you, since there are many ways to do it. A solution is provided for you in the `02-ps-container-01-solution`. If you want, you can take a break from the book and try to execute this task now.

Injecting Dependencies That Are Not Beans

Dependency injection with Spring is a vast subject, and this book was written with the intention of covering all the cases you might need in enterprise development. It was mentioned that scalar values can be injected. This of course opens the discussion about type conversion. In the previous example, we injected a boolean value, and the Spring container knew to take the text value from the configuration file and convert it to the property type defined in the bean type definition. The Spring container knows how to do this for all primitive types and their reference wrapper types. *String* values, booleans, and numeric types are supported by default. Just keep in mind that for booleans and decimal types, however, the syntax typical for these types must be respected; otherwise, the default Spring conversion will not work.

Spring also knows to automatically convert *Date* values before injection if the syntax matches any date pattern with "/" separators (e.g., dd/MM/yyyy ,yyyy/MM/dd, dd/yyyy/MM) just take care, because even invalid numerical combinations are accepted and converted, e.g., 25/2433/23.

In order to support any other types, the Spring container must be told how to convert the value of the attribute to the type that the constructor or setter requires as an argument. The most common method is to use a property editor. This is a Spring concept that describes a component used to handle the transformation between any *Object* and *String*. In order to make such a conversion possible, you have to define a custom implementation for the `java.beans.PropertyEditor` and register it in the Spring context. Out of the box, Spring provides a number of *PropertyEditor* implementations for commonly used types.⁴ All the developer has to do is to customize them and register them.

Let's assume we want to define a *Person* bean like this:

```
<bean id="person" class="com.ps.beans.PersonBean">
  <property name="fullName" value="John Mayer"/>
  <property name="birthDay" value="1977-10-16"/>
</bean>
```

There is more than one way to tell the Spring IoC how the value 1977-10-16 can be converted to a *Date*. And since `java.beans.PropertyEditor` was mentioned, the first example will use the Spring `org.springframework.beans.propertyeditors.CustomDateEditor`. In order for this to work, we need to do the following:

- Define a class that implements `org.springframework.beans.PropertyEditorRegistrar` that registers the customized version of `org.springframework.beans.propertyeditors.CustomDateEditor` (the Spring built-in implementation for conversion between `java.util.Date` and *String*):

```
package com.ps.beans.others;

import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.propertyeditors.CustomDateEditor;

import java.text.SimpleDateFormat;
import java.util.Date;
```

⁴You can find them all listed in the Spring Reference <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/validation.html#beans-beans-conversion>.

```
public class DateConverter implements PropertyEditorRegistrar {
    @Override
    public void registerCustomEditors(PropertyEditorRegistry registry) {
        registry.registerCustomEditor(Date.class,
            new CustomDateEditor(new SimpleDateFormat("yyyy-MM-dd"), false));
    }
}
```

- Add a bean definition for the `org.springframework.beans.factory.config.CustomEditorConfigurer` and provide as parameter for its `propertyEditorRegistrars` property a list containing a bean of type `DataConverter`:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <bean class="com.ps.beans.others.DateConverter" />
        </list>
    </property>
</bean>
```

The second example shows you how to do this only from configuration and uses two attributes of the `<bean />` element that will be covered in a following section: `factory-bean` and `factory-method`. The `formatter` bean is used as a factory bean that creates date values from the *String* values passed as arguments to the `parse` method. The downside of this method is that the definition of the bean is polluted with the conversion definition.

```
<beans ...>

    <bean id="formatter" class="java.text.SimpleDateFormat">
        <constructor-arg value="yyyy-MM-dd" />
    </bean>

    <bean id="person" class="com.ps.beans.PersonBean">
        <property name="fullName" value="John Mayer"/>
        <property name="birthDay">
            <bean factory-bean="formatter" factory-method="parse">
                <constructor-arg value="1977-10-16" />
            </bean>
        </property>
    </bean>
</beans>
```

In a web application context, an implementation of the `org.springframework.format.Formatter` must be provided and registered.

! An example bean with fields of various types was provided in `o2-ps-container-01-practice`. The class is called `com.ps.beans.others.MultipleTypesBean`, and it can be tested by running the test class called `com.ps.beans.others.MtBeanTest`. The bean definition can be found in the `o2-ps-container-01-practice/src/main/resources/spring/others/sample-config-01.xml` file and is depicted in the following code snippet:

```
<bean id="mtBean" class="com.ps.beans.others.MultipleTypesBean">
  <property name="noOne" value="1"/>
  <property name="noTwo" value="2"/>
  <property name="longOne" value="3"/>
  <property name="longTwo" value="4"/>
  <property name="floatOne" value="5.0"/>
  <property name="floatTwo" value="6.0"/>
  <property name="doubleOne" value="7.0"/>
  <property name="doubleTwo" value="8.0"/>
  <property name="boolOne" value="true"/>
  <property name="boolTwo" value="false"/>
  <property name="charOne" value="1"/>
  <property name="charTwo" value="A"/>
  <property name="date" value="1977-10-16"/>
</bean>
```

Run the test in debug mode as depicted in Figure 2-11, and set a breakpoint on the `assertNotNull(mtBean);` line. In the Debug console, visible at the bottom of the figure, you can see the values for all the fields of the `mtBean`.

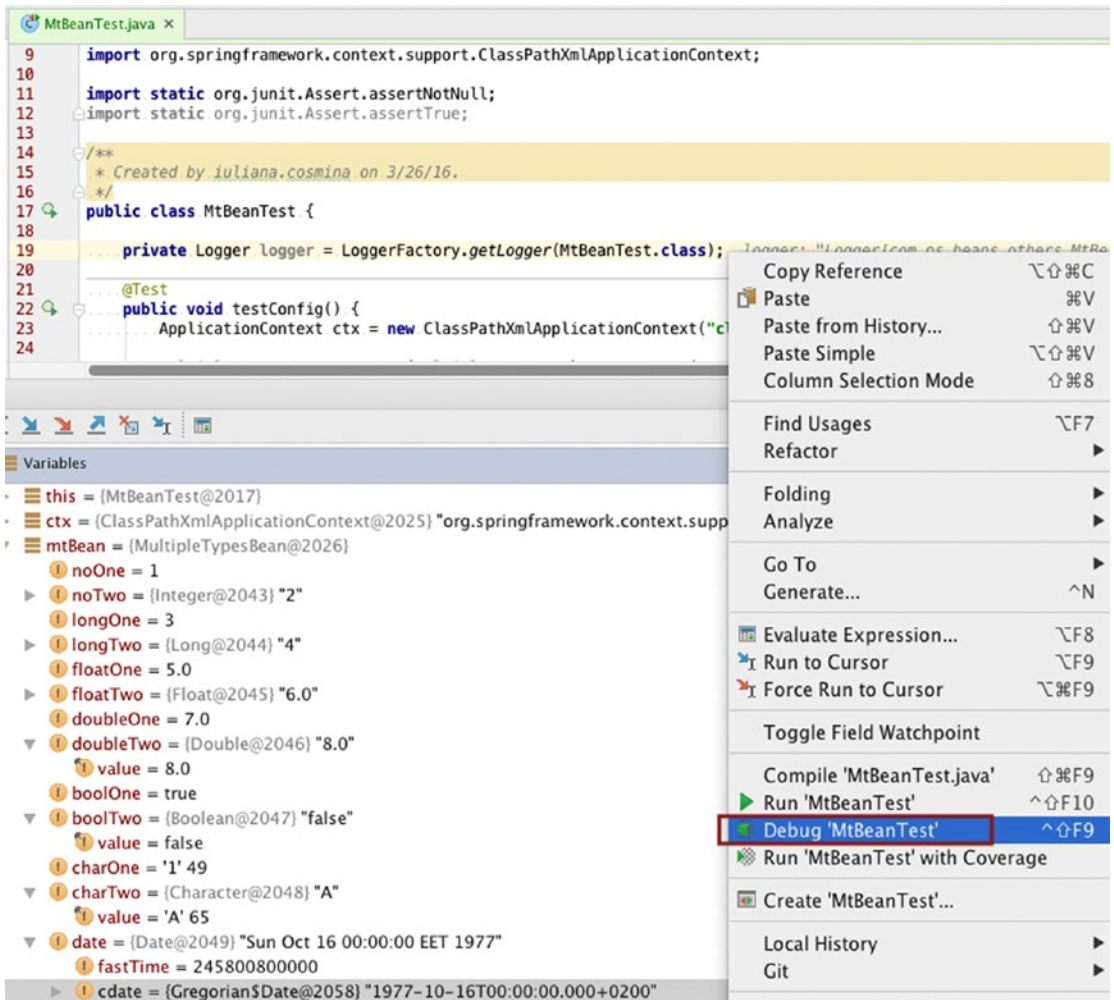


Figure 2-11. Running a test in debug mode to inspect the fields of a bean of type `MultipleTypesBean`

The fields postfix with `One` are primitives; the others are objects. Notice how they were all initialized correctly, and the values in the bean definition were converted to the appropriate types.

Another type of object commonly used is collections, and to make using them together with beans easier, the Spring development team has created the `util` namespace. In the previous code snippet you probably noticed the `<list />` element. That is how collections can be used without the `util` namespace, and another code snippet will be provided to make their usage obvious. Consider a class named `CollectionHolder` defined as in the following code snippet:

```

public class CollectionHolder {
    private List<SimpleBean> simpleBeanList;
    private Set<SimpleBean> simpleBeanSet;
    private Map<String, SimpleBean> simpleBeanMap;

```

```

public void setSimpleBeanList(List<SimpleBean> simpleBeanList) {
    this.simpleBeanList = simpleBeanList;
}

public void setSimpleBeanSet(Set<SimpleBean> simpleBeanSet) {
    this.simpleBeanSet = simpleBeanSet;
}

public void setSimpleBeanMap(Map<String, SimpleBean> simpleBeanMap) {
    this.simpleBeanMap = simpleBeanMap;
}

/**
 * This method was implemented just to verify the collections injected
 * into beans of this type
 */
@Override
public String toString() {
    return "CollectionHolder{" +
        "simpleBeanList=" + simpleBeanList +
        ", simpleBeanSet=" + simpleBeanSet +
        ", simpleBeanMap=" + simpleBeanMap +
        '}';
}
}

```

The state of the collectionHolder bean will be tested using the following test class:

```

// CollectionTest.java
public class MtBeanTest {
    private Logger logger = LoggerFactory.getLogger(SIBeansTest.class);
    @Test
    public void testConfig() {
        ApplicationContext ctx = new ClassPathXmlApplicationContext
            ("classpath:spring/others/sample-config-01.xml");

        CollectionHolder ctBean = (CollectionHolder) ctx.getBean("collectionHolder");
        assertNotNull(ctBean);
        (*) logger.info(ctBean.toString());
    }
}

```

Beans can be created, and values for their collection properties can be injected as in the following examples:

- inject empty collections

```

<bean id="collectionHolder" class="com.ps.beans.others.CollectionHolder">
    <property name="simpleBeanList">
        <list/>
    </property>

```

```

<property name="simpleBeanSet">
  <set/>
</property>

<property name="simpleBeanMap">
  <map/>
</property>
</bean>

```

The `logger.info` statement in the line marked with (*) in the test class will print the following: `CollectionHolder{simpleBeanList=[], simpleBeanSet=[], simpleBeanMap={}}`

- inject a property of type `List` with `SimpleBean` elements

```

<bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

<bean id="collectionHolder" class="com.ps.beans.others.CollectionHolder">
  <property name="simpleBeanList">
    <list>
      <ref bean="simpleBean"/>
      <bean class="com.ps.beans.SimpleBeanImpl"/>
      <null/>
    </list>
  </property>
</bean>

```

- The `logger.info` statement in the line marked with (*) in the test class will print the following:

```

CollectionHolder{
  simpleBeanList=[
    SimpleBeanImpl{ code: 454325163},
    SimpleBeanImpl{ code: 796667727},
    null
  ],
  simpleBeanSet=null, simpleBeanMap=null
}

```

- Here is a detailed explanation of the three injected elements:
 - `<ref bean="simpleBean"/>` is referencing an existing bean. At runtime, the bean named `simpleBean` will be added to the list. Equivalent to `list.add(simpleBean)`.
 - `<bean class="com.ps.beans.SimpleBeanImpl"/>` is used to define a bean that is created on the spot. Equivalent to `list.add(new SimpleBeanImpl)`. This bean definition declares a bean called **inner bean**, a concept that will be covered later.
 - `<null/>` is used to add a null element to the list. Equivalent to `list.add(null)`.
- what was done with the `list` element can be done with the `set` element as well.

- inject a property of type `Map<String, SimpleBean>`:

```
<bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

<bean id="collectionHolder" class="com.ps.beans.others.CollectionHolder">
  <property name="simpleBeanMap">
    <map>
      <entry key="one" value-ref="simpleBean"/>
    </map>
  </property>
</bean>
```

The `logger.info` statement in the line marked with (*) in the test class will print the following:

```
CollectionHolder{
  simpleBeanList=null, simpleBeanSet=null,
  simpleBeanMap={
    one=SimpleBeanImpl{ code: 454325163}
  }
}
```

In using the `<map />` element with values or keys of non-primitive types, the `"-ref"` postfix is used to reference an existing bean. Also, beans cannot be created on the spot, as in `<list />` and `<set />` elements. Nor can `<null />` elements be used.

Without using the `util` namespace, collection elements cannot be defined outside the definition of a bean in a configuration file. The `util` namespace was introduced to deal with common utility configuration issues such as defining and injecting collections, `java.util.Properties` objects, referencing constants. To use its elements, it has to be added to the list of namespaces in the `<beans />` element. Using this namespace, collections can be defined under the `<beans \>` element. They are assigned their own ids, which can be used to reference them for injection into multiple beans. They are basically used in the same ways that beans are.

```
<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"
  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="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>

  <util:list id="simpleList">
    <ref bean="simpleBean"/>
    <bean class="com.ps.beans.SimpleBeanImpl"/>
    <null/>
  </util:list>

  <util:set id="simpleSet">
    <ref bean="simpleBean"/>
  </util:set>
```

```

<util:map id="simpleMap">
  <entry key="one" value-ref="simpleBean"/>
</util:map>

<bean id="collectionHolder" class="com.ps.beans.others.CollectionHolder">
  <property name="simpleBeanList" ref="simpleList"/>
  <property name="simpleBeanSet" ref="simpleSet"/>
  <property name="simpleBeanMap" ref="simpleMap"/>
</bean>

<bean id="collectionHolder2" class="com.ps.beans.others.CollectionHolder">
  <property name="simpleBeanList" ref="simpleList"/>
</bean>

```

In the previous code snippet, you can notice how the `simpleList` component is injected into two beans: `collectionHolder` and `collectionHolder2`.

Another type of bean used quite often in Spring applications is the `java.util.Properties` type. It can be defined and used in a similar way as the other collections were used in the previous examples. This type was not covered in the previous examples, because it was saved for something special.

Using Bean Factories

The `<bean />` element has more attributes and can also be used to declare beans that are created by a factory method⁵ or singleton⁶ objects. To use a singleton class to create a bean, the `factory-method` attribute is used, and its value will be the **static** method name that returns the bean instance. Here in a simple example showing how a bean can be created using a singleton class and the bean configuration:

```

<beans .../>
  <bean id="simpleSingleton" class="com.ps.beans.others.SimpleSingleton"
    factory-method="getInstance" />
</beans>

```

```

// SimpleSingleton.java
public class SimpleSingleton {
  private static SimpleSingleton instance = new SimpleSingleton();

  private SimpleSingleton() { }

  public static SimpleSingleton getInstance(){
    return instance;
  }
}

```

⁵Factory Method is a design pattern that implies using factory methods to create objects without specifying the exact type of the object being created. More information about this pattern can easily be found on the Internet using a simple search on Google, but here is a quick good source: https://en.wikipedia.org/wiki/Factory_method_pattern.

⁶Singleton is a creation pattern that is characterized by the fact that a singleton class can be instantiated only once. More about it can be read here: https://en.wikipedia.org/wiki/Singleton_pattern.

To use a factory object to create a bean, the `factory-bean` and `factory-method` attributes are used. As their so obvious names say, the first one points to the object used to create the bean, and the other specifies the method name that returns the actual result (not static in this case). Here is a snippet of code depicting a bean being created by a factory bean:

```
<beans .../>
  <bean id="simpleBeanFactory" class="com.ps.beans.others.SimpleFactoryBean"/>
  <bean id="simpleFB" factory-bean="simpleBeanFactory"
        factory-method="getSimpleBean" />
</beans>
// SimpleSingleton.java
public class SimpleFactoryBean {

    public SimpleBean getSimpleBean() {
        return new SimpleBeanImpl();
    }
}
```

Although it seems somewhat redundant, creating beans in this way might be useful when one is using third-party libraries that only allow creating objects using a factory class. And this method can be used to connect existing components to provide a certain behavior, such as helping Spring to convert *String* values to *Date* objects using a format pattern different from the default one, as was presented in the section **Injecting Dependencies That Are Not Beans**.

Spring comes to the rescue in this case as well, by providing an interface named `org.springframework.beans.factory.FactoryBean<T>`. This is used by many Spring classes in order to simplify configuration. By implementing this interface, the factory beans will be automatically picked up by the Spring container, and the desired bean will be created by automatically calling the `getObject` method. Even if implementing your factories in this way ties your code to Spring components, which usually is recommended to be avoided, this method is practical and could be used in case of need.

```
<beans .../>
  <bean id="smartBean" class="com.ps.beans.others.SpringFactoryBean"/>
</beans>

//SpringFactoryBean.java
import org.springframework.beans.factory.FactoryBean;
...
public class SpringFactoryBean implements FactoryBean<SimpleBean> {
    private Logger logger = LoggerFactory.getLogger(SpringFactoryBean.class);

    private SimpleBean simpleBean = new SimpleBeanImpl();

    public SpringFactoryBean() {
        logger.info(">> Look ma, no definition!");
    }

    @Override
    public SimpleBean getObject() throws Exception {
        return this.simpleBean;
    }
}
```

```

@Override
public Class<?> getObjectType() {
    return SimpleBean.class;
}
@Override
public boolean isSingleton() {
    return true;
}
}

```

If you are curious about the Spring classes that implement `org.springframework.beans.factory.FactoryBean`, IntelliJ IDEA can help you there. The class depicted in the previous code snippet is a part of 02-ps-container-01-practice. Just open that class, click on the interface name, and press CTRL (Command in MacOS)+ALT+B, and a list of classes implementing it will be displayed, as depicted in Figure 2-12.

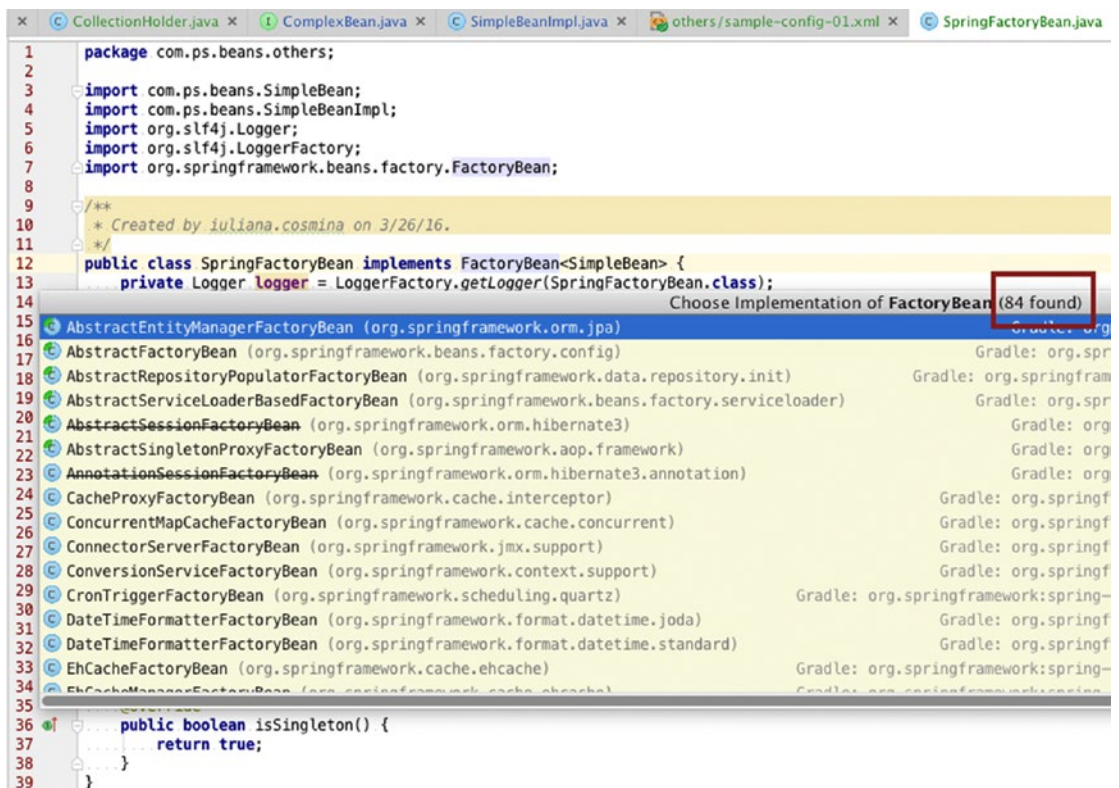


Figure 2-12. Classes implementing the `FactoryBean` interface

Spring factory bean classes provide assistance for configuring data access using Hibernate and JPA, for handling transactions, for configuring caching with EhCache, and so on. A few of them will be used in the code samples for this book, so you will have occasion to see them in action.

Creating an Application Context

In the examples provided up to now, the test classes contained a statement like this:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
...
ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath:spring/others/sample-config-02.xml");
```

The `org.springframework.context.ApplicationContext` is the interface implemented by classes that provide the configuration for an application. This interface is an extension of the interface `org.springframework.beans.factory.BeanFactory`, which is the root interface for accessing a Spring Bean container. Implementations of `ApplicationContext` manage a number of beans definition uniquely identified by their name. Multiple Spring application context implementations exist, each one specific to a type of application. An `ApplicationContext` implementation provides the following:

- access to beans using bean factory methods
- ability to load file resources in a generic way
- ability to publish events to registered listeners
- ability to resolve messages and support internationalization (most used in international web applications)

An application context is created by the Spring container and initialized with a configuration provided by a resource that can be an XML file (or more) or a configuration class (or more) or both. When the resource is provided as a String instance, the Spring container tries to load the resource based on the prefix of that String value. Also, based on the prefix, when instantiating an application context, different classes should be used. In Table 2-1 you can see the different prefixes that can be used when loading resources in Spring.

Table 2-1. Prefixes and corresponding paths

Prefix	Location	Comment
no prefix	In root directory where the class creating the context is executed	In the main or test directory. The Resource being loaded will have a type depending on the <code>ApplicationContext</code> instance being used. (A detailed example is presented after the table.)
classpath:	The resource should be obtained from the classpath	In the resources directory and the resource will be of type <code>ClassPathResource</code> . If the resource is used to create an application context, the <code>ClassPathXmlApplicationContext</code> class is suitable.
file:	In the absolute location following the prefix	Resource is loaded as an URL, from the filesystem and the resource will be of type <code>UrlResource</code> . If the resource is used to create an application context, the <code>FileSystemXmlApplicationContext</code> class is suitable.
http:	In the web location following the prefix	Resource is loaded as an URL and the resource will be of type <code>UrlResource</code> . If the resource is used to create an application context, the <code>WebApplicationContext</code> class is suitable.

In order to provide the functionality for loading resources, an application context must implement the `org.springframework.core.io.ResourceLoader` interface. Here is an example of resource loading without using a prefix:

```
Resource template = ctx.getResource("application-config.xml");
```

Depending on the context class used, the resource loaded can have one of the following types:

- If `ctx` is a `ClassPathXmlApplicationContext` instance resource type will be `ClassPathResource`
- If `ctx` is a `FileSystemXmlApplicationContext` instance resource type will be `FileSystemResource`
- If `ctx` is a `WebApplicationContext` instance resource type will be `ServletContextResource`

And here is where prefixes come in. If we want to force the resource type, no matter what context type is used, the resource must be specified using the desired prefix.

In the examples mentioned so far, all the bean definitions were in the same file, but for big multilayered applications, it is more appropriate to separate bean definitions depending on their purpose. Let's assume that for the Pet Sitter application there will only be one XML configuration file. That file would look like this, and the second... replaces other repository beans and services bean definitions, one for each type of object used in the application.

```
<beans ...>
  <bean id="simpleUserService" class="com.ps.services.impl.SimpleUserService">
    <property name="repo" ref="userRepo"/>
  </bean>
  <!-- Loads users from the data source -->
  <bean id="userRepo" class="com.ps.repos.impl.JdbcUserRepo">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource">
    ...
  </bean>

  ... // other repo beans
</beans>
```

Testing certain components in isolation would not be possible. Mixed configurations are not recommended for enterprise applications. They are more suitable for applications used to learn Spring.

An application context can be loaded from multiple XML files. This provides the opportunity to group bean definitions by their purpose, for example to separate infrastructure beans from application beans, because infrastructure changes between environments.⁷

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:spring/application-config.xml",
    "classpath:spring/db-config.xml");
```

⁷Most companies use three types of environments: development, testing, and production.

In the example above, the configuration for the storage layer is decoupled from the rest of the application configuration in `db-config.xml`. This makes the database configuration easily replaceable depending on the environment. The contents of `db-config.xml` for production could contain a `dataSource` bean definition that uses a professional database such as Oracle.

```
<beans ...>
<bean id="dataSource" class="oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@localhost:1521:PET"/>
    <property name="user" value="admin"/>
</bean>
</beans>
```

The `test-db-config.xml` file for a test environment could contain a `dataSource` bean definition that uses an in-memory database.

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseFactoryBean">
    <property name="databasePopulator" ref="populator"/>
</bean>

<bean id="populator"
    class="org.springframework.jdbc.datasource.init.ResourceDatabasePopulator">
    <property name="scripts">
        <list>
            <value>classpath:testdb/schema.sql</value>
            <value>classpath:testdb/test-data.sql</value>
        </list>
    </property>
</bean>
```

Using the `application-configuration.xml` file together with `test-db-config.xml`, a test environment application context can be created that can be used to run unit tests:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:spring/application-config.xml",
    "classpath:spring/test-db-config.xml");
```

But as we have seen in the initial configuration snippet, we can go even further, and split the repository from the service components, so the `application-configuration.xml` file will be split in two:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:spring/repo-config.xml",
    "classpath:spring/service-config.xml");
```

And because the file names are so similar, wildcards are supported and can be used to keep the code even simpler than this:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:spring/*-config.xml");
```

In working with multiple files, in order to group them, there is also the possibility of enclosing them one in another using the `<import />` element. The import element supports prefixes in order to properly locate files and should be used when beans defined in separate configuration files are part of the same logical group. The following code and configuration snippet depicts how configuration files can be imported. In Figure 2-13, the content of a resources directory is depicted.

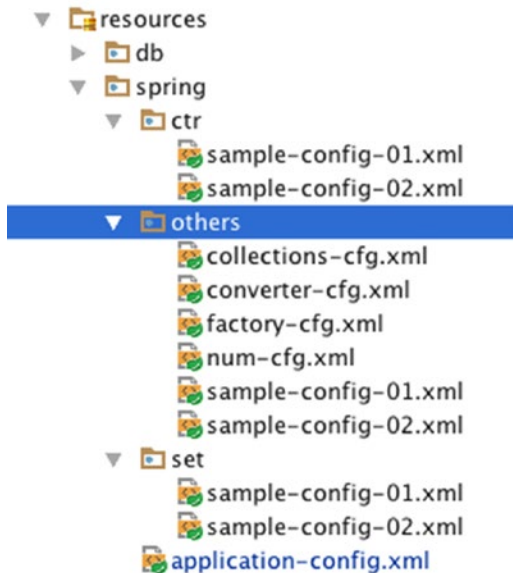


Figure 2-13. Spring configuration files in a resources directory

The file `application-config.xml` directory should enclose all the files named `sample-config-*` under directories `ctr`, `set`, and `others` so the application context can be created like this:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "classpath:spring/application-config.xml");
```

Using different prefixes, for learning purposes of course, here is how the files can be imported:

```
<beans ...>
  <!-- using relative path, no prefix-->
  <import resource="ctr/sample-config-01.xml"/>
  <import resource="ctr/sample-config-02.xml"/>

  <!-- using classpath-->
  <import resource="classpath: spring/ctr/sample-config-01.xml"/>
  <import resource="classpath: spring/ctr/sample-config-02.xml"/>

  <!-- using classpath and wildcards-->
  <import resource="classpath: spring/others/sample-config-*.xml"/>
</beans>
```

! Since this is the end of an important section, there is a task for you. In project `02-ps-container-01-practice` there is a configuration file that has gotten a little too big:

`02-ps-container-01-practice/src/main/resources/spring/others/sample-config-01.xml`.

Split this file into one or more configuration files, and modify the test class `com.ps.beans.others.MtBeanTest` to use those files to load the context. The test must complete successfully in order to validate your configuration. A solution is provided for you in the `02-ps-container-01-solution` project. You can compare your result with this and draw conclusions. As a bonus, try using wildcards.

Also, do not forget, after you have modified a class, you might want to build your project using `gradle build -x test` in the command line or run the `compileJava` task from the IntelliJ IDEA interface, in order to make sure you are running your test with the most recent sources. In Figure 2-14, the location of the `compileJava` task is evidenced for you.

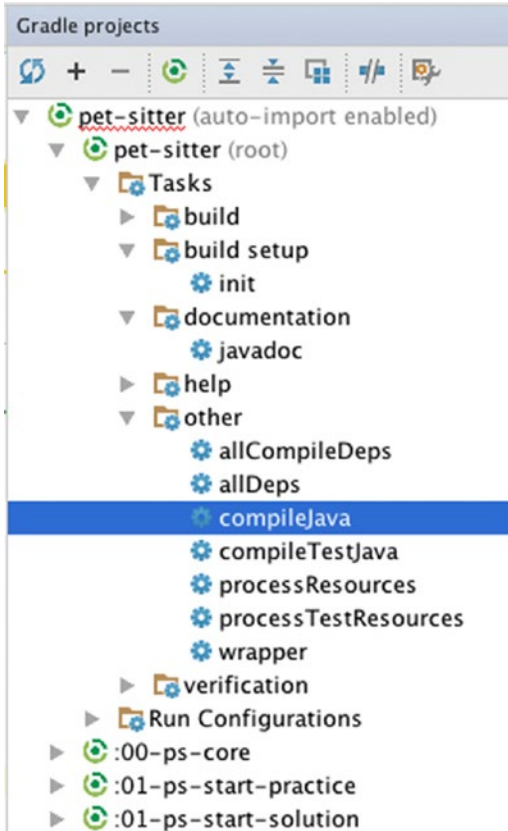


Figure 2-14. The Gradle view in IntelliJ IDEA

Spicing Up XML Configuration

XML is a very flexible text format that defines a set of rules for encoding text elements that is both human- and machine-readable. The purpose of XML configuration is to make configuring Spring applications easy. Even with XML configuring enterprise behavior like AOP, collections, and transactions, integration with third-party frameworks was cumbersome, so special namespaces that offer the possibility to define complex elements easily were introduced. This also helped with differentiating infrastructure beans from application beans.

Using XML is also practical because the configuration files can be externalized, which allows them to control the behavior of the application without recompiling it. So in the following sections, advanced XML configuration will be covered.

Spring Namespaces

The `util` namespace was introduced earlier in order to demonstrate how collections can be injected into beans easily and also how collections can be treated as beans. Below, you have a few things that you can do using the `util` namespace:

- access and use constants in configuration

```
<!-- without util namespace -->
<bean id="complexBean2" class="com.ps.sample.ComplexBean"
  p:simpleBean2-ref="simpleBean2">
  <constructor-arg>
    <bean id="com.ps.sample.SimpleBean.DEFAULT_SIMPLE_BEAN"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </constructor-arg>
</bean>
```

```
<!-- with util namespace -->
<bean id="complexBean2" class="com.ps.sample.ComplexBean"
  p:simpleBean2-ref="simpleBean2">
  <constructor-arg>
    <util:constant static-field="com.ps.sample.SimpleBean.DEFAULT_SIMPLE_BEAN"/>
  </constructor-arg>
</bean>
```

```
// SimpleBean.java, where the static field is defined
package com.ps.sample;
public class SimpleBean {
    public static final SimpleBean DEFAULT_SIMPLE_BEAN= new SimpleBean("DEFAULT");
}
```

`FieldRetrievingFactoryBean` is a `FactoryBean` that retrieves values of static or nonstatic fields. The name of the bean is the full path of the static object being retrieved. This approach works for enums too.

- Use typed collections. When you declare `<util:list />`, `<util:set />`, or `<util:map/>`, Spring decides what type of implementation to use, but you can change that:

```
<util:list id="beanList" list-class="java.util.LinkedList">
...
</util:list>
```

```

<util:set id="names" set-class="java.util.TreeSet">
...
</util:set>

<util:map id="simpleMap" map-class="java.util.TreeMap">
...
</util:map>

```

- Read properties from a file directly into a `java.util.Properties` that can be treated like a bean, assigned an id, and retrieved from the context.

```

<util:properties id="dbProp" location="classpath:db/datasource.properties"/>
...
Properties dbProp = ctx.getBean("dbProp", Properties.class);

```

Another useful namespace is the `jdbc` namespace. It provides configuration elements used to create in memory datasources for testing purposes and initializing databases.

```

<!-- without jdbc namespace -->
<beans ...>
  <bean id="dataSource" class="org.springframework.jdbc.datasource
    .embedded.EmbeddedDatabaseFactoryBean">
    <property name="databasePopulator" ref="populator"/>
  </bean>

  <bean id="populator" class="org.springframework.jdbc.datasource
    .init.ResourceDatabasePopulator">
    <property name="scripts">
      <list>
        <value>classpath:testdb/schema.sql</value>
        <value>classpath:testdb/test-data.sql</value>
      </list>
    </property>
  </bean>
</beans>

<!-- with jdbc namespace -->
<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/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">
    <jdbc:script location="classpath:testdb/schema.sql"/>
    <jdbc:script location="classpath:testdb/test-data.sql"/>
  </jdbc:embedded-database>
</beans>

```

```

<!-- initializing a database -->
<beans ...>
  <jdbc:embedded-database id="dataSource" type="HSQL"/>

  <jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="classpath:db/schema.sql"/>
    <jdbc:script location="classpath:db/test-data.sql"/>
  </jdbc:initialize-database>
</beans>

```

Some other important namespaces such context, aop, tx, and jms will be covered later in the book.

How to Read Property Files in Spring Evolution

As a example of how the namespaces help simplify configuration, let's take a simple operation: reading properties from a file. Here is how properties can be read from a file using the bean namespace:

```

<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">

  <!-- reading the properties, Spring infrastructure bean is exposed -->
  <bean
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:db/datasource.properties"/>
  </bean>

  <!-- here the values are injected into a datasource -->
  <bean id="dataSource1"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${driverClassName}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
  </bean>

</beans>

```

Here is how properties can be read from a file using the context namespace:

```

<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">

```



```

<!-- Spring infrastructure is not visible anymore -->
<context:property-placeholder location="classpath:db/datasource.properties" />

<!-- here the values are injected into a datasource -->
<bean id="dataSource2"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${driverClassName}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
</bean>

</beans>

```

Here is how properties can be read from a file using the util 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: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">

<!-- Spring infrastructure is not visible anymore -->
<util:properties id="dbProp" location="classpath:db/datasource.properties"/>

<bean id="dataSource3"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="#{dbProp.driverClassName}"/>
    <property name="url" value="#{dbProp.url}"/>
    <property name="username" value="#{dbProp.username}"/>
    <property name="password" value="#{dbProp.password}"/>
</bean>

</beans>

```

The advantage of using the util namespace is that the properties are loaded in a `java.util.Properties` object named `dbProp` that can be used in other beans' configurations. In order to access those values and inject them into other beans, SpEL is needed, which is why the values are injected using `#{}`, which is typical SpEL syntax to access a property of an object; basically, `#{dbProp.url}` is equivalent to `dbProp.getProperty("url")`.

Spring Expression Language

SpEL is the Spring Expression language. It is quite powerful, since it supports querying and manipulating an object graph at runtime. The SpEL is inspired from WebFlow EL,⁸ a superset of Unified EL,⁹ and it provides considerable functionality, such as:

- method invocation
- access to properties, indexed collections
- collection filtering
- boolean and relational operators
- and many more¹⁰

There are several extensions of SpEL (OGNL, MVEL, and JBoss EL), but the best part is that it is not directly tied to Spring and can be used independently.

Bean Definition Inheritance

Since classes are templates for creating objects and they can be extended, in order to extend an existing bean template, the same can be done for bean definitions. They can be inherited, and an existing definition can be enriched with extra details. Let us assume that the Pet Sitter application must run in production on a cluster of three servers. Each server has its own database, and the configuration must mention them all. The configuration file for connecting to the database would look like this:

```
<beans ...>
  <bean id="dataSource-1" class="oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
    <property name="user" value="admin"/>
    <property name="loginTimeout" value="300"/>
  </bean>

  <bean id="dataSource-2" class="oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
    <property name="user" value="admin"/>
    <property name="loginTimeout" value="300"/>
  </bean>

  <bean id="dataSource-3" class="oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
    <property name="user" value="admin"/>
    <property name="loginTimeout" value="300"/>
  </bean>
</beans>
```

⁸An expression language used to configure web flows: <http://docs.spring.io/spring-webflow/docs/current/reference/html/el.html>.

⁹Unified EL is the Java expression language used to add logic in JSP pages: <https://docs.oracle.com/javaee/5/tutorial/doc/bnahq.html>.

¹⁰The full list of capabilities is not in the scope of this book. If you are interested in SpEL, the official documentation is the best resource: <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/expressions.html>.

A lot of repetition in there, isn't there? Bean inheritance can help reduce the file in size and can separate common configuration elements in a template bean definition that will be defined as abstract. This type of bean cannot be created by the Spring IoC. The bean definition will declare the template as parent and will inherit all the details defined in it. For example, in the code snippet below, bean type, user, and loginTimeout property values are inherited. As in class inheritance, bean definitions can override inherited details. The dataSource3 bean definition overrides the loginTimeout, and it overrides the bean type, too.

```
<beans ...>
  <bean id="abstractDataSource" class="oracle.jdbc.pool.OracleDataSource"
        abstract="true">
    <property name="user" value="admin"/>
    <property name="loginTimeout" value="300"/>
  </bean>

  <bean id="dataSource-1" parent="abstractDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
  </bean>

  <bean id="dataSource-2" parent="abstractDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
  </bean>

  <bean id="dataSource3" parent="abstractDataSource"
        class="com.ps.CustomizedOracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@192.168.1.164:1521:PET"/>
    <property name="loginTimeout" value="100"/>
  </bean>
</beans>
```

The interesting part about bean inheritance is that the resulting beans do not have to have the same types as the parent. As long as the properties inherited are present in the bean definition that extends the parent template, the inheritance will work.

Inner beans

Inner beans are beans that are defined within the scope of another bean. They are the equivalent of private fields that cannot be accessed using getter methods and have no outside reference to them. This type of bean was mentioned briefly when collections were introduced, but it deserves its own section in order to make sure that it is clear when to use them and why. An inner bean is defined by a <bean/> element inside the <property/> or <constructor-arg/> elements.

```
<beans .../>
  <bean id="enclosingBean1" class="...">
    <property name="target">
      <bean class="..."/>
    </property>
  </bean>
```

```

<!-- or -->
<bean id="enclosingBean1" class="...">
  <constructor-arg index="0">
    <bean class="..." />
  </constructor-arg>
</bean>
</beans>

```

Inner beans could have the `id` or `name` attribute populated (both attributes will be explained in detail in the following section), but this would be useless, since these beans cannot be retrieved using them, because they are internal to the beans that enclose them. And of course, in case you were curious, beans can be nested as many times as you want, but avoid doing so unless you really really have to, since this will make your configuration look bloated, and it becomes hard to test. Nested beans are usually a choice for infrastructure beans.

Example: a `Musician` bean, that encloses a `Person` bean, that encloses an `Address` bean.

```

<beans .../>

<bean id="sampleMusician" class="com.ps.sample.Musician">
  <constructor-arg>
    <bean class="com.ps.sample.Person">
      <property name="fullName" value="John Mayer" />
      <property name="birthData" >
        <bean class="com.ps.sample.BirthData">
          <property name="city" value="Bridgeport" />
          <property name="state" value="Connecticut" />
          <property name="birthDate" value="1977-10-16"/>
        </bean>
      </property>
    </bean>
  </constructor-arg>
</bean>
</beans>

```

Bean Naming

Every bean hosted by the container has a unique identifier. It is the developer's responsibility to name beans accordingly using the `id` or `name` attributes where this is necessary, because as we've seen previously when we customized the `CustomDateEditor`, the configurer bean was defined without an `id`. A bean usually has at least one unique identifier, and in case of need, more of them can be defined by setting multiple values for the `name` attribute, by separating them with a comma (,), semicolon (;), or white space. Or by using the `<alias />` element.

Prior to Spring 3.1, the `id` attribute was of type `xsd:ID`, and as a consequence, characters such as comma, semicolon, white space, and others were not allowed. Starting with Spring 3.1, the type was changed to `xsd:string`, the same type used for the `name` attribute, which made these attributes interchangeable. So you can use one or the other, with the same syntax. You can use them both if you want, anything, just to make sure your bean is identified correctly.

Let's start at the beginning, and assume that you have a configuration file containing a single **bean definition with no id or name** defined:

```
<beans ...>
  <bean class="com.ps.beans.SimpleBeanImpl"/>
</beans>
```

An application context can be created from this file, and it will contain a single bean of type `SimpleBeanImpl` with a name that was set by the container. If we want to see the name that the container assigns to this bean, all we have to do is to print all the bean names in the context, and this can be done by calling `getBeanDefinitionNames()` provided by the application context and iterating the returned results:

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath:spring/application-config.xml");
for (String beanName : ctx.getBeanDefinitionNames()) {
    logger.info("Bean " + beanName + " of type "
        + ctx.getBean(beanName).getClass().getSimpleName());
}

// Result printed in the console:
Bean com.ps.beans.SimpleBeanImpl#0 of type SimpleBeanImpl
```

The fun part is that nothing stops you from duplicating the bean definition. The container will just assign a different name to the second bean of the same type, and the code above will print the following:

```
Bean com.ps.beans.SimpleBeanImpl#0 of type SimpleBeanImpl
Bean com.ps.beans.SimpleBeanImpl#1 of type SimpleBeanImpl
```

You could even retrieve those beans and their names by calling the `getBeansOfType(...)` method on the application context:

```
Map<String, SimpleBean> simpleBeans = ctx.getBeansOfType(SimpleBean.class);
```

So technically, in case you ever need to do something like this, you can. But keep in mind that all the beans with that type will be added to that map, whether they have a developer given id/name or not. When you have exactly one bean of a certain type, it can be retrieved from the context using the `getBean(...)` method, which has quite a few versions that you can use to retrieve beans, all of which will be covered in this book.

```
SimpleBean simpleBean = ctx.getBean(SimpleBean.class);
```

The `getBean` method, which requires only a `Class<T>` argument, returns the bean instance that uniquely matches the given object type, if any. The Spring IoC searches for any bean of that type, whether the bean type is an interface or superclass. In case more than one bean of that type is defined, the context will throw an exception, since it cannot know what bean you really want. The exception name is quite obvious, as you can see in the log snippet below.

```
org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type com.ps.beans.SimpleBean is defined:
expected single matching bean but found 2: sb01,com.ps.beans.SimpleBeanImpl#0
```

Although having beans with no explicit ids defined is possible, unless they are inner beans or some type of infrastructure beans (example: `org.springframework.beans.factory.config.CustomEditorConfigurer`), it rarely makes sense to define a bean with no id.

Beginning with Spring 3.1, the `id` and `name` attributes have the same type, so they accept special characters in their values. In versions of Spring previous to 3.1, the `name` attribute was mandatory for web controllers, because the syntax required their id to start with "/" (e.g., `name="/persons"`), and the `id` attribute did not allow special characters. Almost everything that will be covered from now on for the attribute `id` is valid for the `name` attribute too, except for a little detail that will be mentioned at the appropriate time.

! The Spring convention is to use the standard Java convention for instance field names when naming beans. This means that bean names start with a lowercase letter and are camel-cased from then on. The purpose of good bean naming is to make the configuration quick to read and understand. Also, when using more advanced features like AOP, it is useful to be able to apply advice to a set of beans related by name.

Let's cover the many ways in which a bean can be named using an id.

```
<beans ...>
  <bean id="sb01" class="com.ps.beans.SimpleBeanImpl"/>

  <!-- equivalent with -->
  <bean name="sb01" class="com.ps.beans.SimpleBeanImpl"/>
</beans>
```

If the `sb01` bean is named as in one of the examples before, there are two ways in which this bean can be retrieved from the application context:

```
SimpleBean sb01 = (SimpleBean)ctx.getBean("sb01");

SimpleBean sb01 = ctx.getBean("sb01", SimpleBean.class);
```

Since `id` and `name` have the same purpose, it is not allowed to have in the same application context bean definitions with the same value for `id`/`name`. Smart Java editors, like the one you were recommended to use, IntelliJ IDEA, even warns you against doing something like that by underlining the attribute value with a red line, as depicted in Figure 2-15.

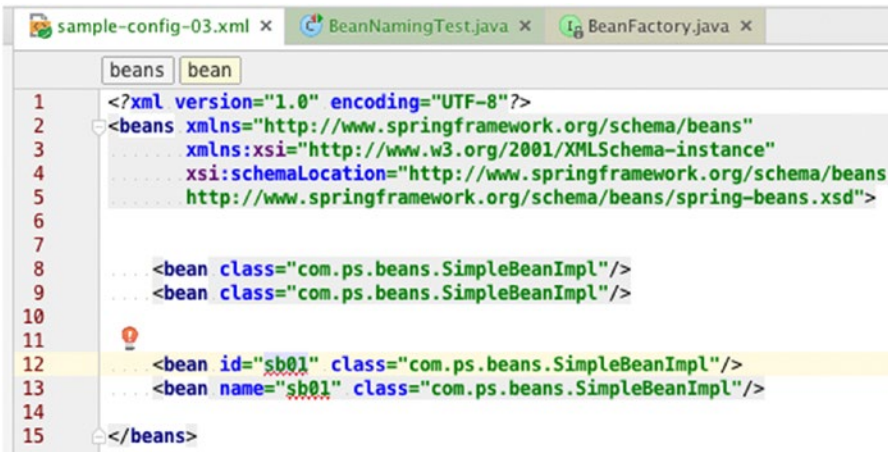


Figure 2-15. Bad bean naming is obvious in IntelliJ IDEA

? In the configuration below, a few bean-naming examples are depicted. Can you imagine what the code that logs all beans in the context will print for the following configuration file?

```
<beans ...>
  <bean id="sb01" name="sb01" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="sb02/sb03" name="mumu" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="sb04; sb05" class="com.ps.beans.SimpleBeanImpl"/>

  <bean name="sb04; sb05, sb004 sb005" class="com.ps.beans.SimpleBeanImpl"/>

  <bean id="$" class="com.ps.beans.SimpleBeanImpl"/>
</beans>
```

Well, the log will print this:

```
Bean sb01 of type SimpleBeanImpl
Bean sb02/sb03 of type SimpleBeanImpl
Bean sb04; sb05 of type SimpleBeanImpl
Bean sb04 of type SimpleBeanImpl
Bean $ of type SimpleBeanImp
```

So a detailed explanation for each case is appropriate.

- The attributes `id` and `name` can have the same value.
- `"sb02/sb03"` is an `id` containing a special character; `$` is an `id` consisting of one special character. Both are valid, and the beans can be successfully retrieved using:

```
SimpleBean sb1 = ctx.getBean("sb02/sb03", SimpleBean.class);
SimpleBean sb2 = ctx.getBean("$", SimpleBean.class);
```

But wait, what happened to the *"mumu"* value set with the name attribute? Can you guess? The application context can offer a reply to this too:

```
for (String name : ctx.getAliases("sb02/sb03") ){
    logger.info ("Alias for sb04 -> " + name);
}
//The result
Alias for sb02/sb03 -> mumu
```

- *"sb04; sb05"* is also a valid id containing a special character, which happens to be a separator, but the bean can be retrieved only using the full id value, because Spring treats the ids and names a little differently. **A bean can have many names, but only one id.**
- *"sb04; sb05, sb004 sb005"* is used as a value for the name attribute, so the separators are taken into consideration, and that's why in the log, only the *sb04* value is printed. But wait, what happened to the others? I think you already have a suspicion. If you run the code above, the one that prints the aliases, here is the output:

```
Alias for sb04 -> sb05
Alias for sb04 -> sb004
Alias for sb04 -> sb005
```

The conclusion is that the first name became the id, and the others became aliases.

And since we mentioned aliases, let's give an example for this too:

```
<beans ...>
  <bean id="$" class="com.ps.beans.SimpleBeanImpl"/>
  <alias name="$" alias="properName"/>
</beans>
```

So now the *\$* bean can be retrieved using its alias:

```
SimpleBean s = ctx.getBean("properName", SimpleBean.class);
```

Bean aliasing can be used to override already configured beans, thus providing a way to substitute them with a different bean definition. This is useful in testing, and the configuration is not decoupled enough, and it comes from an external source that cannot be modified. Using aliasing, real beans can be replaced with test beans with known behavior, making it possible to test a specific module in isolation. But more about testing can be read in **Chapter 3. Testing Spring Applications**.

All these examples can be found in *02-ps-container-01-practice*. The configuration file path is */02-ps-container-01-practice/src/main/resources/spring/others/sample-config-03.xml*, and the test class to run is *02-ps-container-01-practice/src/test/java/com/ps/beans/others/BeanNamingTest.java*. You are invited to add your own bean names and your own tests.

! The conclusions of this section are really important, so you might consider putting a bookmark right here, or make a note somewhere.

- If the bean has only the name attribute defined, then its value becomes the bean name used to uniquely identify this bean in the context.

- If the bean has `id` and `name` attributes defined, the `name` value is used as an alias, and the `id` value becomes the bean name used to uniquely identify this bean in the context.
- The `id` and `name` attribute values have the same type, but if the `id` value contains separator characters: comma (,), semicolon (;), or white space, they are not treated as such.
- If the bean has only the `name` attribute defined and the value contains multiple names, then the first value becomes the bean name used to uniquely identify this bean in the context; the others become aliases.

Application Context and Bean Lifecycle

Before explaining how everything presented so far can be done using Java Configurations and annotations, the application context and bean lifecycle must be covered, because some of the details are intertwined, so it is better to have an overall idea about this before going more deeply into this subject.

A Spring application has a lifecycle composed of three phases:

1. **Initialization:** In this phase, bean definitions are read, beans are created, dependencies are injected, and resources are allocated, also known as the bootstrap phase. After this phase is complete, the application can be used.
2. **Use:** In this phase, the application is up and running. It is used by clients, and beans are retrieved and used to provide responses for their requests. This is the main phase of the lifecycle and covers 99% of it.
3. **Destruction:** The context is being shut down, resources are released, and beans are handed over to the garbage collector.

These three phases are common to every type of application, whether it is a JUnit System test, a Spring or JEE web, or enterprise application. Look at the following code snippet (the sources can be found in the `/02-ps-container-02-solution` project); it was modified in order to make obvious where each phase ends.

```
import org.springframework.context.ConfigurableApplicationContext;

public class ApplicationContextTest {

    private Logger logger = LoggerFactory.getLogger(ApplicationContextTest.class);

    @Test
    public void testDataSource1() {
        ConfigurableApplicationContext ctx =
            new ClassPathXmlApplicationContext("classpath:spring/test-db01-config.xml");
        logger.info(" >> init done.");
        DataSource dataSource1 = ctx.getBean("dataSource1", DataSource.class);
        assertNotNull(dataSource1);
        logger.info(" >> usage done.");
        ctx.close();
    }

    <!-- test-db01-config.xml contents-->
    <?xml version="1.0" encoding="UTF-8"?>
    <beans ...>
```

```

<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations" value="classpath:db/datasource.properties"/>
</bean>

<bean id="dataSource1"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="${driverClassName}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</bean>
</beans>

```

The initialization phase of a Spring application ends when the application context initialization process ends. If the logger for the Spring framework is set to debug, before the `init done` message is printed, a lot of log entries show what the Spring container is doing. A simplified sample is depicted below:

```

DEBUG o.s.c.e.StandardEnvironment - Initialized StandardEnvironment
with PropertySources [systemProperties,systemEnvironment]
INFO o.s.c.s.ClassPathXmlApplicationContext - Refreshing
<<1>> o.s.context.support. ClassPathXmlApplicationContext@335eadca:
startup date; root of context hierarchy
...
INFO o.s.b.f.x.XmlBeanDefinitionReader -
<<2>> Loading XML bean definitions from class path resource
[spring/test-db01-config.xml]
...
DEBUG o.s.b.f.x.BeanDefinitionParserDelegate - Neither XML 'id' nor 'name' specified -
using generated bean name
<<3>> [org.springframework.beans.factory.config.PropertyPlaceholderConfigurer#0]
DEBUG o.s.b.f.x.XmlBeanDefinitionReader - Loaded 2 bean definitions from location
pattern [classpath:spring/test-db01-config.xml]
...
INFO o.s.b.f.c.PropertyPlaceholderConfigurer -
Loading properties file from class path resource [db/datasource.properties]
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Pre-instantiating singletons in
o.s.beans.factory.support.DefaultListableBeanFactory@6591f517:
defining beans [o.s.b.f.config.PropertyPlaceholderConfigurer#0,dataSource1];
root of factory hierarchy
...
<<4>> DEBUG o.s.b.f.s.DefaultListableBeanFactory -
Creating instance of bean 'dataSource1'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean 'dataSource1'
to allow for resolving
potential circular references
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
of bean 'dataSource1'
INFO c.p.ApplicationContextTest - >> init done.

```

```

<<7>> DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
of singleton bean 'dataSource1'
INFO c.p.ApplicationContextTest - >> usage done.

INFO c.p.ApplicationContextTest - >> bye bye.
INFO o.s.c.s.ClassPathXmlApplicationContext -
<<8>> Closing o.s.context.support.ClassPathXmlApplicationContext
@335eadca: startup date [Sun Mar 27 21:18:39 EEST 2016]; root of context hierarchy
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
of singleton bean 'lifecycleProcessor'
DEBUG o.s.b.f.s.DefaultListableBeanFactory -
<<9>>Destroying singletons in o.s.beans.factory.support.DefaultListableBeanFactory
@6591f517: defining beans
[o.s.beans.factory.config.PropertyPlaceholderConfigurer#0,dataSource1];
root of factory hierarchy...

```

Although the log is incomplete, all the important details were underlined, and by following them we can notice some of the following steps. To make it easier, the corresponding steps were marked in the log with the corresponding step number.

1. The application context is initialized.
2. The bean definitions are loaded (from the `spring/test-db01-config.xml` in this case).
3. The bean definitions are processed (in our case a bean of type `PropertyPlaceholderConfigurer` is created and used to read the properties from `datasource.properties`, which are then added to the `dataSource` bean definition).
4. Beans are instantiated.
5. Dependencies are injected (not visible from the log, since the `dataSource` bean does not require any dependencies).
6. Beans are processed (also not visible from the log, since the `dataSource` does not have any processing defined).
7. Beans are used.
8. The context starts the destruction process.
9. Beans are destroyed.

Figure 2-16 depicts the whole application context lifecycle and the bean lifecycle.

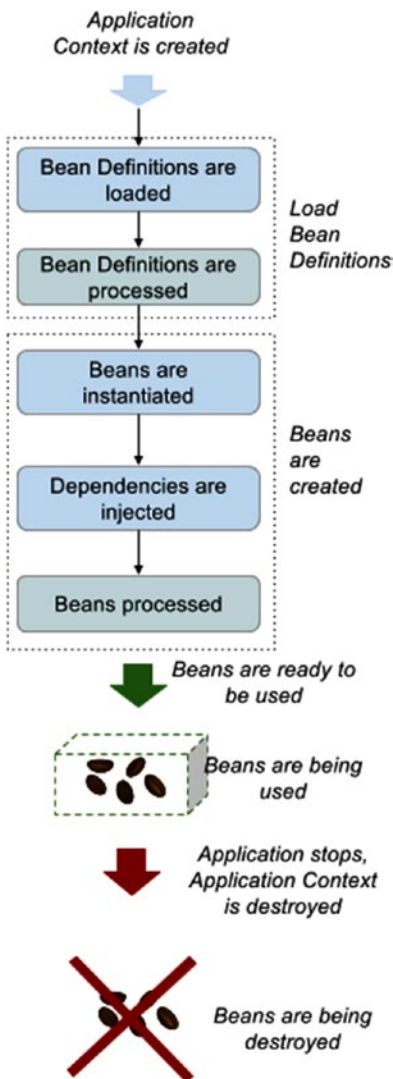


Figure 2-16. Application context and bean lifecycle

Bean Lifecycle Under the Hood

In the previous section it was mentioned that for a bean to *come to life* and become available to be used for a purpose, it has to go through a set of steps. Since in the previous section, the focus was on the application context, the references to bean creation steps were quite shallow. So we need to fix that. Let's start with a schema depicting the creation steps of a bean: Figure 2-17.

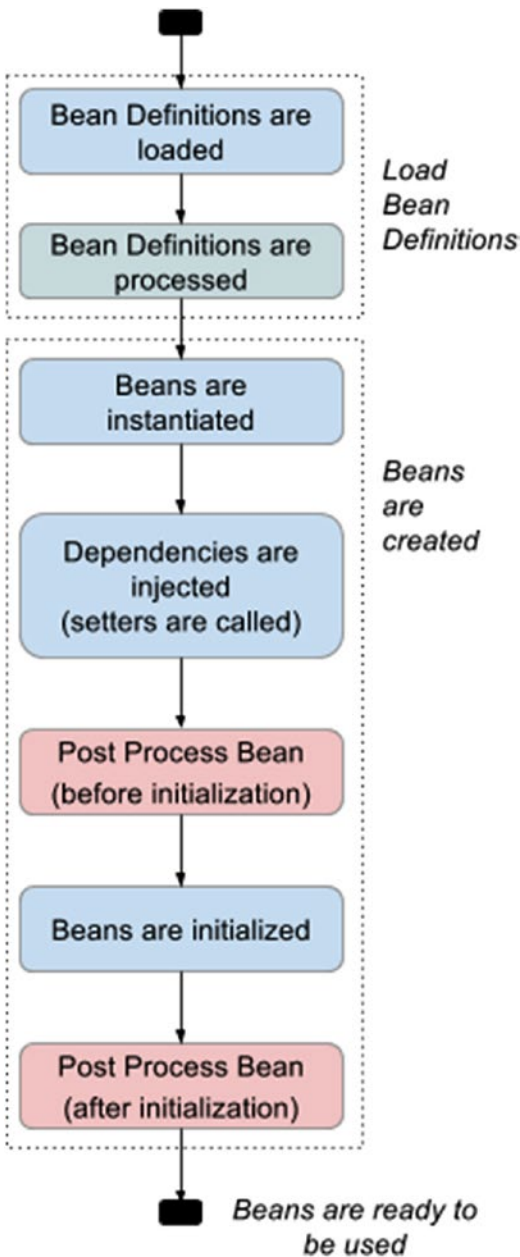


Figure 2-17. Complete list of bean creation steps

In the **Load Bean Definition** step, the XML files/Configuration classes are parsed, and bean definitions are loaded into the application context, indexed by ids. The bean definitions are then processed by beans called bean definition post processors that are automatically picked up by the application context, created, and applied before any other beans are created. These bean types implement the `org.springframework.beans.factory.config.BeanFactoryPostProcessor` interface, and that is how the application context

recognizes them. In the previous example, the `PropertyPlaceholderConfigurer` type was mentioned. This class is a bean definition post processor that resolves placeholders like `${propName}` with property values read from the Spring environment and its set of property sources.

The `BeanFactoryPostProcessor` contains a single method definition that must be implemented, `postprocess(BeanFactory)`. The parameter with which this method will be called is the factory bean used by the application context to create the beans. Developers can create their own bean definition postprocessors and define a bean of this type in their configuration, and the application context will make sure to invoke them.

In Figure 2-18, the effect of a `PropertyPlaceholderConfigurer` bean being invoked on the `dataSource1` bean definition is depicted.



Figure 2-18. Effect of a `PropertyPlaceholderConfigurer` on a bean definition using placeholders

These types of beans are useful, because they can process bean definitions at runtime and change them based on resources that are outside the application, so the application does not need to be recompiled in order to change a bean definition. In the above example, if instead of reading the properties of the `dataSource1` from an external file, the information were to be written directly in the configuration file, then if the `dataSource1` location and type changed, the XML configuration file would need to be changed, and the application would have to be recompiled and restarted. Not that practical, right?

The **Bean creation** step can be split into a small number of stages.

1. In the first stage, **the beans are instantiated**. This basically means that the bean factory is calling the constructor of each bean. If the bean is created using constructor dependency injection, the dependency bean is created first and then injected where needed. For beans that are defined in this way, the instantiation stage coincides with **the dependency injection** stage.
2. In the second stage, **dependencies are injected**. For beans that are defined having dependencies injected via setter, this stage is separate from the instantiation stage.
3. The next stage is the one in which **bean post process beans are invoked before initialization**.¹¹
4. In this stage, **beans are initialized**.
5. The next stage is the one in which **bean post process beans are invoked after initialization**.

As you can see, there are two stages involving bean post process beans being called. What is the difference between them? The stage between them, the initialization stage, is used to split them into bean post processors that are invoked before it and after it. Since the bean post processor subject is quite large, the initialization stage will be covered first. Long story short, a bean can be defined so that a certain method is called right after the bean is created and dependencies are injected in order to execute some code. This method is called an initialization method, and in using XML configuration, this method is set using the `init-method` attribute on a bean definition. The `init-method` attribute should have as value the name of a method defined in the bean type. The method must return void, have no arguments defined, and can have any access right, since Spring uses reflection to find and call it, and some developers actually recommend to make it private so it cannot be called from outside the bean, and also make sure that Spring has total control over it and that it calls it only one time during the bean lifecycle. In the code snippet below you can see the configuration of such a method for a bean of type `ComplexBean`. The bean is a very simple one with no methods to be picked up and executed by a bean post processor:

```
public class ComplexBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean1;

    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }
}
```

¹¹Unfortunately there is no other way to formulate this. We are talking about beans that have the ability to post process other beans. And they are called, confusingly: **bean post process beans**.

```

/**
 * The initialization method.
 * Just for fun: it instantiates the simpleBean2 only
 * if the current time is even.
 */
private void initMethod() {
    logger.info("Stage 4: Calling the initMethod.");
    long ct = System.currentTimeMillis();
    if (ct % 2 == 0) {
        simpleBean2 = new SimpleBean();
    }
}
}

```

The `ComplexBean` definition, contains a setter definition that is used to inject the `simpleBean2` dependency, but in the following configuration, the setter is not used, because the focus is on the initialization stage.

```

<!-- configuration file contents -->
<beans ...>

    <bean id="simpleBean" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:_o-ref="simpleBean" init-method="initMethod"/>
</beans>

```

In initializing an application context based on the configuration above, here is what could be seen in the log:

```

...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
INFO c.p.s.ComplexBean - Stage 1: Calling the constructor.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean 'complexBean' to allow for
resolving potential circular references
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Invoking init method 'initMethod' on bean with
name 'complexBean'
INFO c.p.s.ComplexBean - Stage 4: Calling the initMethod.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance of bean
'complexBean'
...

```

As you can see, the bean is created, and then the `initMethod` is called by the factory creating the bean. Keep this in mind, because it is important, as you will soon see. If an `init` method is not specified, there are other ways of initializing a bean available in Spring. Here is the complete list:

- Using the attribute `init-method` on a `<bean/>` XML definition to define a method to be called for initialization, covered previously.
- Implementing the `org.springframework.beans.factory.InitializingBean` interface and providing an implementation for the method `afterPropertiesSet` (not recommended, since it couples the application code with Spring infrastructure).

- Annotating with `@PostConstruct` the method that is called right after the bean is instantiated and dependencies injected.¹²
- The equivalent of the `init-method` attribute when using Java Configuration `@Bean(initMethod="...")`.

The `InitializingBean` interface is implemented by beans that need to react once all their dependencies have been injected by the `BeanFactory`. It defines a single method that is named accordingly, `afterPropertiesSet()`, which is called by the factory. In the code snippet below, the `ComplexBean` is modified to implement this interface.

```
public class ComplexBean implements InitializingBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        logger.info("Stage 4: Calling afterPropertiesSet.");
    }
}
<!-- configuration file contents -->
<beans ...>

    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:_0-ref="simpleBean1"
        p:simpleBean2-ref="simpleBean2"/>
</beans>
```

In this example, the setter of this bean was used, so that it will be obvious in the log that the `afterPropertiesSet()` method is called after injecting all the dependencies. If an application context is created using the previous configuration, this is what can be seen in the log:

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
INFO c.p.s.ComplexBean - Stage 1: Calling the constructor.
```

¹²Beginning with Spring 2.5, a few annotations are supported.

```

DEBUG o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean 'complexBean'
    to allow for resolving potential circular references
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
    of singleton bean 'simpleBean2'
INFO c.p.s.ComplexBean - Stage 2: Calling the setter.
(*)DEBUG o.s.b.f.s.DefaultListableBeanFactory - Invoking afterPropertiesSet()
    on bean with name 'complexBean'
INFO c.p.s.ComplexBean - Stage 4: Calling afterPropertiesSet.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
    of bean 'complexBean'
...

```

As you can notice from the log, the line marked with (*), calling the `afterProperties` method, is also a responsibility of the factory that creates the bean.

Annotations support was introduced in Spring 2.5 for a small set of annotations. Since then, Spring has evolved, and currently with Spring 4.0, an application can be configured using only annotations (stereotypes and the complete set of Java Configuration). The `@PostConstruct` annotation is part of the JSR 250¹³ and is used on a method that needs to be executed after dependency injection is done to perform initialization. The annotated method must be invoked before the bean is used, and, like any other initialization method chosen, may be called **only once** during a bean lifecycle. If there are no dependencies to be injected, the annotated method will be called after the bean is instantiated. Only one method should be annotated with `@PostConstruct`.

The method annotated with `@PostConstruct` is picked up and called by a pre-init bean of a type that implements the `org.springframework.beans.factory.config.BeanPostProcessor` interface. Classes implementing this interface are actually factory hooks that allow for modifications of bean instances. The application context auto-detects these types of beans and instantiates them before any other beans in the container, since after their instantiation they are used to manipulate other beans managed by the IoC container.

The `BeanPostProcessor` declares two methods to be implemented, `postProcessBeforeInitialization` and `postProcessAfterInitialization`, and their names depict clearly their purpose. In Figure 2-17, there are two pink rectangles depicting when the bean post processor is invoking methods on the bean. Typically, post processors that populate beans via marker interfaces (they pick up methods annotated with `@PostConstruct`) will implement `postProcessBeforeInitialization`, while post processors that wrap beans with proxies will normally implement `postProcessAfterInitialization`.

For `@PostConstruct`, annotated methods to be picked up and executed. A bean of this type must be present in the configuration. The bean that registers `@PostConstruct` (and other annotations from JSR 250, as you will see later) is `org.springframework.context.annotation.CommonAnnotationBeanPostProcessor`. Since this is a Spring infrastructure bean, it could be added to the configuration file like this:

```

<beans ...>

<bean
    class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>

<!-- other bean definitions here -->
</beans>

```

And this could work, but there will be some issues as configuring the bean like that, overrides the Spring defaults which might lead to unexpected behavior. Fortunately this bean configuration is one of those included in the following line, a Spring shortcut based on the context namespace:

¹³Java Request Specification 250 <https://jcp.org/en/jsr/detail?id=250>.

```

<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"
       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:annotation-config/>

  <!-- or this one, that is an extension of annotation-config-->
  <context:component-scan base-package="com.ps.sample"/>
</beans>

```

The `<context:annotation-config />` enables scanning of all the classes in the project for annotations, so using it on large applications might make them slow. The solution is to use `<context:component-scan />`, because the `base-package` attribute will reduce the number of classes to be scanned. Also, the good part is that you can set multiple base packages too, by separating them with a comma. The best part is that the classes in the packages can also be filtered by name, by what they are annotated with, or by other criteria.

```

<beans .../>
<context:component-scan base-package="com.ps.sample, com.ps.repos"/>

<!-- package filtering -->

<context:component-scan base-package="com.ps.sample, com.ps.all">
  <context:include-filter type="regex" expression="*Repo"/>

  <context:exclude-filter type="annotation"
    expression="org.springframework.stereotype.Service"/>
</context:component-scan>
</beans>

```

When the `<context:component-scan />` statement is present in an XML file, all supported annotations are detected in the class path, or only in the packages specified as values for the `base-package` attribute in the second case.

The `<context:annotation-config/>` activates detection for `@PostConstruct`, `@PreDestroy`, `@Resource`, `@Autowired`, and `@Required` and other JPA and EJB 3 annotations.

The `<context:component-scan/>` activates detection for all annotations mentioned previously, plus Spring stereotype annotations: `@Component` and extensions (e.g., `@Service`, `@Repository`).

Returning to our bean initialization, the code and configuration below depicts the `ComplexBean` class making use of the `@PostConstruct` annotation:

```

<beans ...>
  <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
  <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

  <bean id="complexBean" class="com.ps.sample.ComplexBean"
    c:_0-ref="simpleBean1"
    p:simpleBean2-ref="simpleBean2"/>

```

```

    <context:component-scan base-package="com.ps.sample"/>
</beans>

//ComplexBean.java
import javax.annotation.PostConstruct;

public class ComplexBean {
    private Logger logger = LoggerFactory.getLogger(ComplexBean.class);

    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }

    @PostConstruct
    private void initMethod() {
        logger.info("Stage 4: Calling the initMethod.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            simpleBean2 = new SimpleBean();
        }
    }
}

```

The methods that can be annotated with `@PostConstruct` must respect the same rules as every other init method: they must have no arguments, return void, and they can have any access right. And if an application context is created from this configuration, here is what the log might show:

```

...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
INFO c.p.s.ComplexBean - Stage 1: Calling the constructor.
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor -
    Found init method on class com.ps.sample.ComplexBean:
    private void com.ps.sample.ComplexBean.initMethod()
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor -
    Registered init method on class com.ps.sample.ComplexBean:
    o.s.b.f.a.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@fad21aed
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean 'complexBean'
    to allow for resolving potential circular references
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance of
    singleton bean 'simpleBean2'
INFO c.p.s.ComplexBean - Stage 2: Calling the setter.
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking init method

```

```

on bean 'complexBean': private void com.ps.sample.ComplexBean.initMethod()
INFO c.p.s.ComplexBean - Stage 4: Calling the initMethod.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
of bean 'complexBean'
...

```

As you can see from this log, although the same `initMethod` is called to initialize the bean, the caller is different from before, since the method is now called by the pre-init post processor bean, the `CommonAnnotationBeanPostProcessor` bean, not the factory bean creating the bean. And this is important, because now it makes sense that this type of bean can have an effect before and after the initialization of the bean, since obviously, while they can be used for initialization, they are intended to be used to do something before or after it.

? For example, how would the log above change if we added another method `initMethod2` to the class `ComplexBean` and configured the `complexBean` as in the code snippet below?

```

<beans ...>
  <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
  <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

  <bean id="complexBean" class="com.ps.sample.ComplexBean"
    c:_0-ref="simpleBean1"
    p:simpleBean2-ref="simpleBean2"
    init-method="initMethod2"/>

  <context:component-scan base-package="com.ps.sample"/>
</beans>

```

In case you haven't figured it out, this is how the log will change:

```

...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
INFO c.p.s.ComplexBean - Stage 1: Calling the constructor.
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor -
  Found init method on class com.ps.sample.ComplexBean:
  private void com.ps.sample.ComplexBean.initMethod()
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor -
  Registered init method on class com.ps.sample.ComplexBean:
  o.s.b.f.a.InitDestroyAnnotationBeanPostProcessor$LifecycleElement@fad21aed
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Eagerly caching bean 'complexBean'
to allow for resolving potential circular references
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance of
singleton bean 'simpleBean2'
INFO c.p.s.ComplexBean - Stage 2: Calling the setter.
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking init method
on bean 'complexBean': private void com.ps.sample.ComplexBean.initMethod()
INFO c.p.s.ComplexBean - Stage 3: Calling the initMethod.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Invoking init method

```

```
'initMethod2' on bean with name 'complexBean'
c.p.s.ComplexBean - Stage 4: Calling the initMethod2.
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
of bean 'complexBean'
...
```

The method `initMethod`, the one annotated with `@PostConstruct`, is invoked before the method `initMethod2`, making it cover the third stage in the lifecycle of a bean.

Bean post processors are quite powerful, since they can be used to enforce bean behavior (`@Required`) and even to add behavior to the application logic transparently (transactions, security). The `@Required` annotation is used to mark dependencies that are mandatory and is provided by Spring. At the beginning of the chapter, when it was shown how to configure setter dependency injection, it was mentioned that dependencies injected using setters were not required, but there is a way to enforce their injection on the developer. The `@Required` annotation is that way. In the code snippet below, the `ComplexBean` type was modified to make use of the `@Required` annotation:

```
import org.springframework.beans.factory.annotation.Required;

public class ComplexBean {
    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    @Required
    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }
    ...
}

<!-- configuration file -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:simpleBean1-ref="simpleBean1"
        init-method="initMethod2"/>
    <context:component-scan base-package="com.ps.sample"/>

</beans>
```

An application context cannot be created using the configuration above, since the property `simpleBean2` for `complexBean` is not set anywhere. In the console, an exception will be printed, and right under it, you can see the `BeanPostProcessor` type of the bean that enforces the mandatory behavior for the dependency:

```
org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor

org.springframework.beans.factory.BeanInitializationException:
  Property 'simpleBean2' is required for bean 'complexBean'
  at o.s.b.f.a.RequiredAnnotationBeanPostProcessor.postProcessPropertyValues
```

A Spring-capable IDE will recognize this annotation and will try to warn you about your mistake, underlining the bean definition with red, or showing you a popup with a comment, something similar to what you see in Figure 2-19.

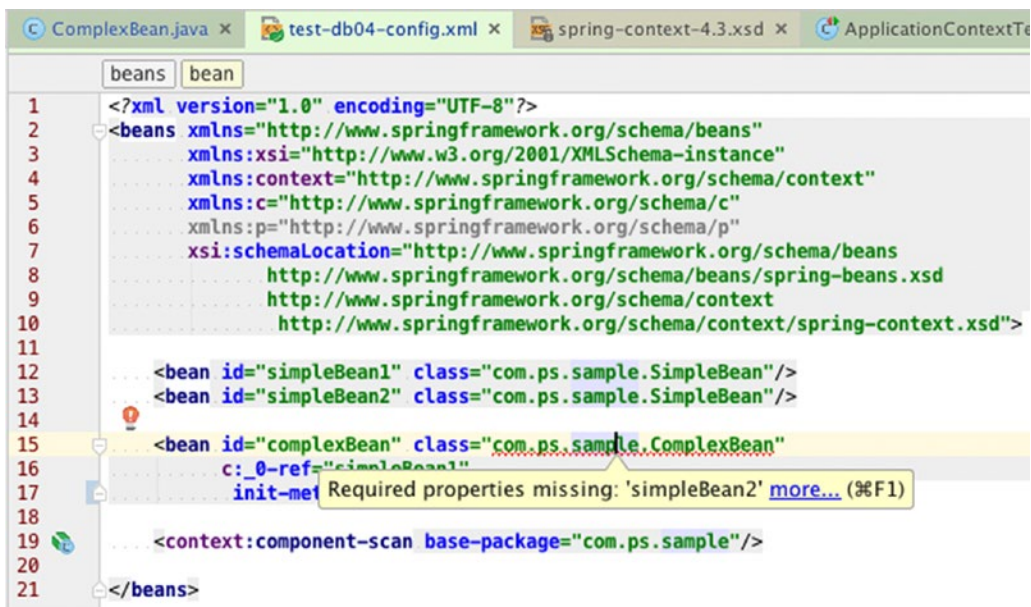


Figure 2-19. IntelliJ IDEA warning the developer about required dependencies

We are at the end of the section, so now we have to cover what happens to beans after they have been used and are no longer needed and the application shuts down, or only the context managing them. When a context is closed, it destroys all the beans; that is obvious. But some beans work with resources that might refuse to release them if they are not notified before destruction. In Spring, this can be done in three ways:

- Set a method to be called before destruction using the `destroy-method` attribute of the `<bean />` element.
- Modify the bean to implement the `org.springframework.beans.factory.DisposableBean` interface and provide an implementation for the `destroy()` method (not recommended, since it couples the application code with Spring infrastructure).

- Annotate a method with `@PreDestroy`, also part of JSR 250 and one of the first supported annotations in Spring.
- The equivalent of destroy-method for Java Configuration `@Bean(destroyMethod="...")`.

The destroy method for a bean has the same purpose as the finalize method for POJOs. The `ComplexBean` was modified to make use of the destroy method, and the code and configuration are depicted below.

```
public class ComplexBean {
    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    @Required
    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }

    private void destroyMethod(){
        logger.info(" --> Calling the destroyMethod.");
        simpleBean1 = null;
        simpleBean2 = null;
    }
    ...
}

<!-- configuration file -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:_o-ref="simpleBean1"
        p:simpleBean2-ref="simpleBean2"
        destroy-method="destroyMethod"/>

</beans>
```


In order to view something useful in the log, we need to close the application context gracefully, and this can be done by calling the close method.

```
@Test
public void testBeanCreation() {
    ConfigurableApplicationContext ctx =
        new ClassPathXmlApplicationContext("classpath:spring/test-db04-config.xml");

        ... // use the bean

    ctx.close();
}
```

And *voilà*: (Only the last part of the log is depicted, where the invocation of the destroy method is logged.)

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
of bean 'complexBean'
...
INFO o.s.c.s.ClassPathXmlApplicationContext - Closing
org.springframework.context.support.ClassPathXmlApplicationContext
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
of singleton bean 'lifecycleProcessor'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
org.springframework.beans.factory.support.DefaultListableBeanFactory...
DEBUG o.s.b.f.s.DisposableBeanAdapter - Invoking destroy method 'destroyMethod'
on bean with name 'complexBean'
INFO c.p.s.ComplexBean - --> Calling the destroyMethod.
</beans>
```

As you can see, the command to destroy the beans is given to the factory that created them, which delegates the dirty work to a bean of type `org.springframework.beans.factory.support.DisposableBeanAdapter`.¹⁴

! For the `destroy` method, the same rules and recommendations regarding signature and accessors apply as for the `init` method:

The `destroy` method may be called **only once** during the bean lifecycle.

The `destroy` method can have any accessor; some developers even recommend to make it private, so that only Spring can call it via reflection.

The `destroy` method must not have any parameters.

The `destroy` method must return void.

¹⁴`DisposableBeanAdapter` is an internal infrastructure bean type that performs various destruction steps on a given bean instance. Its code is available here: <https://github.com/spring-projects/spring-framework/blob/master/spring-beans/src/main/java/org/springframework/beans/factory/support/DisposableBeanAdapter.java>.

As you probably suspect by now, if `ComplexBean` implements `DisposableBean`, the dirty work will be done by the same bean that the factory delegated when the `destroy` method was configured via XML. Further below you can see the implementation and the configuration.

```
import org.springframework.beans.factory.DisposableBean;

public class ComplexBean implements DisposableBean {
    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    @Required
    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }

    @Override
    private void destroy(){
        logger.info(" --> Calling the destroy() method.");
        simpleBean1 = null;
        simpleBean2 = null;
    }
    ...
}

<!-- configuration file -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:_o-ref="simpleBean1"
        p:simpleBean2-ref="simpleBean2"/>
</beans>
```

The log is the same as when the `destroy` method is specified using the `destroy-method` attribute. The `DisposableBeanAdapter` will call the `destroy()` method, so it will not be depicted again.

Also, you probably also suspect that if `ComplexBean` makes use of the `@PreDestroy` annotation, the dirty work will be done by a bean post processor. Further below you can see the implementation, configuration, and log:

```
import javax.annotation.PreDestroy;

public class ComplexBean {
    private SimpleBean simpleBean1;
    private SimpleBean simpleBean2;

    public ComplexBean(SimpleBean simpleBean1) {
        logger.info("Stage 1: Calling the constructor.");
        this.simpleBean1 = simpleBean1;
    }

    @Required
    public void setSimpleBean2(SimpleBean simpleBean2) {
        logger.info("Stage 2: Calling the setter.");
        this.simpleBean2 = simpleBean2;
    }

    @PreDestroy
    private void destroyMethod(){
        logger.info(" --> Calling the destroyMethod.");
        simpleBean1 = null;
        simpleBean2 = null;
    }
    ...
}
<!-- configuration file -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...">

    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
        c:_0-ref="simpleBean1"
        p:simpleBean2-ref="simpleBean2"/>

    <context:component-scan base-package="com.ps.sample"/>

</beans>

// The log
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'complexBean'
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Finished creating instance
of bean 'complexBean'
```

```

...
INFO o.s.c.s.ClassPathXmlApplicationContext - Closing
    org.springframework.context.support.ClassPathXmlApplicationContext
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
    of singleton bean 'lifecycleProcessor'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in
    org.springframework.beans.factory.support.DefaultListableBeanFactory...
DEBUG o.s.c.a.CommonAnnotationBeanPostProcessor - Invoking destroy method
    on bean 'complexBean': private void com.ps.sample.ComplexBean.destroyMethod()
INFO c.p.s.ComplexBean ---> Calling the destroyMethod.
</beans>

```

Bean Scopes

And since the subject of destroying beans was covered, how much a bean can live will be covered, also known as *the bean scope*. As you probably noticed, when a context closes, in the log you always see this:

```

DEBUG o.s.b.f.s.DefaultListableBeanFactory - Destroying singletons in org.springframework.
beans.factory.support.DefaultListableBeanFactory@10e92f8f: defining beans simpleBean1,simple
Bean2,complexBean, o.s.c.a.internalConfigurationAnnotationProcessor...

```

So, Spring refers to the beans as singletons, because that is the default scope of a bean.¹⁵ When the Spring IoC instantiates beans, it creates a single instance for each bean, unless a property is set on the bean definition specifying otherwise. The property in question is called *scope*, and the default scope for a bean is singleton. The scopes are defined in Table 2-2.

Table 2-2. *Bean scopes*

Scope	Description
Singleton	The Spring IoC creates a single instance of this bean, and any request for beans with an id or ids matching this bean definition results in this instance being returned.
Prototype	Every time a request is made for this specific bean, the Spring IoC creates a new instance.
Request	The Spring IoC creates a bean instance for each HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
Session	The Spring IoC creates a bean instance for each HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	The Spring IoC creates a bean instance for each global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
Custom	Developers are provided the possibility to define their own scopes with their own rules.

¹⁵The Singleton design pattern is therefore used heavily in Spring.

So when a bean is created without a scope attribute:

```
<bean id="complexBean" class="com.ps.sample.ComplexBean"/>
```

The default scope is `singleton`. Otherwise, the scope of the bean is the one specified by the value of the scope attribute.

```
<bean id="complexBean" class="com.ps.sample.ComplexBean"
  scope="prototype"/>
```

Now that we know that more beans scopes are available, how do we solve dependencies between different scopes? When we have a `prototype` bean depending on a `singleton`, there is no problem. Every time the `prototype` bean is requested from the context, a new instance is created, and the `singleton` bean is injected into it. But in other cases, things get a little complicated.

The domain that is most sensitive when it comes to dependencies among beans with different scopes is the Web Applications domain. As you probably noticed in the bean scope table, there are three bean scopes designed to be used in web applications: `request`, `session`, and `global-session`. Let's assume that we have a service bean called `ThemeManager` that manages updates on an object of type `UserSettings` containing the settings that a `User` has for an interface in a web application. This means that the `ThemeManager` bean has to work with a different `UserSettings` bean for each HTTP session. Obviously, this means that the `UserSettings` bean should have the scope `session`:

```
<beans ...>
```

```
<bean id="userSettings" class="com.ps.sample.UserSettings"
  scope="session"/>
```

```
<bean id="themeManager" class="com.ps.sample.ThemeManager">
  <property name="userSettings" ref="userSettings"/>
</bean>
```

```
</beans>
```

But how can the problem with single instantiation mentioned earlier can be solved? On different HTTP sessions, methods like `themeManager.saveSettings(userSettings)` should be called with the `UserSettings` bean specific to that session. But the configuration above does not allow for this to happen.

Spring has a solution to this problem. The Spring IoC container not only handles bean instantiation, but also injecting dependencies. If only it could be instructed to reinject the proper dependency when needed! A behavior similar to this can be configured as follows:

```
<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"
  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">
```

```
<bean id="userSettings" class="com.ps.sample.UserSettings" scope="session"/>
  <aop:scoped-proxy/>
</bean>
```

```

<bean id="themeManager" class="com.ps.sample.ThemeManager">
  <property name="userSettings" ref="userSettings"/>
</bean>

</beans>

```

The `<aop:scoped-proxy/>` is part of the AOP¹⁶ namespace, and what it does is to tell the Spring IoC not to inject a `UserSettings` bean in there, but a replacement bean called a proxy¹⁷ with the same public interface as the intended object, which will be smart enough to fetch the `UserSettings` specific to the current session and delegate the calls from `ThemeManager` to it.

The AOP framework was introduced here because it was important to show how beans with different scopes can be used correctly. The AOP framework complements the Spring IoC container. Of course, the Spring container can be used without it in small applications that do not require the use of security or transactions, because these are the key crosscutting concerns for enterprise applications. The Spring AOP framework has the entire fourth chapter of this book dedicated to it.

! All the code snippets presented in this section can be found in `02-container-ps-02-practice`. The bean classes are under `com.ps.sample`, and in order to test your understanding so far, some pieces of the bean classes and configurations were removed, and TODO comments were added in their place. There are four tasks for you to complete, numbered 9 through 12. The TODOs cover bean lifecycle and scopes. The logging has been configured on DEBUG for the Spring framework, in the Logback configuration file located at `02-ps-container-02-practice/src/main/resources/logback.xml`, so you can study delayed logs and search for elements mentioned in this section. The core class to run in order to verify your solution is

```
com.ps.ApplicationContextTest.
```

A solution was provided for you in `02-container-ps-02-solution`. Use it for comparison, or if you have problems, use it for inspiration.

Providing Configuration Using Java Configuration and Annotations

The Spring namespaces provide a way to simplify the Spring configuration by hiding the framework details. They define a language to be used to create bean definitions. Because it is XML, validations are available, so any smart editor will be able to validate the configuration files before running the application. They also can be extended, so developers can create their own elements to configure beans. Besides the beans and the already mentioned `util` namespace, the Spring team has developed dedicated namespaces meant to simplify configuration and use of Spring infrastructure beans for a number of topics that will be covered in this book: `aop`, `context`, `jms`, `aop`, `jdbc`, `security`, `tx`, etc. And all of this is really good, but there is always room for something new.

¹⁶AOP is an acronym for Aspect Oriented Programming and is a programming paradigm aiming to increase modularity by allowing the separation of cross-cutting concerns. This is done by defining something called “pointcut,” which represents a point in the code where new behavior will be injected. This allows for business logic agnostic code to be separated from the code, avoiding code cluttering.

¹⁷The **Proxy** programming design pattern is characterized by the use of a surrogate or placeholder instead of the real intended object. This design template is heavily used in AOP and remoting.

In Section 2, **How to Read Property Files in Spring Evolution** you were taught three methods of reading properties from files using Spring. There is a fourth one. Using Java Configurations and annotations, the same thing looks like this:

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;
import java.sql.SQLException;

@Configuration
@PropertySource("classpath:db/datasource.properties")
public class DataSourceConfig {

    @Value("${driverClassName}")
    private String driverClassName;
    @Value("${url}")
    private String url;
    @Value("${username}")
    private String username;
    @Value("${password}")
    private String password;

    @Bean
    public DataSource dataSource() throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }

    // needed to resolve the properties injected with @Value
    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

Before starting to dig deeply into Java Configuration and Annotations usage for configuring Spring applications, it is proper to begin with a little history lesson.

When Spring 1.0 was released in 2004, it supported only XML as a method of configuration. The annotation concept was not even invented yet. I remember that the first time I had contact with Spring was in 2006. As a young coder eager to learn to write Java code, writing applications using XML did not seem appealing. As soon as the idea of annotations emerged, Spring adopted it and rapidly provided its own annotations, (the stereotype annotations `@Component` and its specializations `@Service` and `@Repository`, etc.), in order to make configuring Spring applications more practical. This happened in 2007, when Spring 2.5 was released. In this version, XML was still needed. Starting with Spring 3.0 in 2009 and the introduction of Java Configuration, a configuration method based on annotations placed inside the Java code, XML became expendable. The code above doesn't need any XML at all. An application context can be created based on that configuration class using a corresponding class, `org.springframework.context.annotation.AnnotationConfigApplicationContext`. In order to test the configuration, a test class like the one below must be written.

```
public class ApplicationContextTest {
    @Test
    public void testDataSource4() {
        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(DataSourceConfig.class);

        DataSource dataSource = ctx.getBean("dataSource", DataSource.class);
        assertNotNull(dataSource);
    }
}
```

The following sections will be analogous to the ones for XML configuration where applicable.

The Annotations

The core annotation in Spring is the `@Component` from the `org.springframework.stereotype` package. This annotation marks a class from which a bean will be created. Such classes are automatically picked up using annotation-based configuration and classpath scanning. In Figure 2-20, all the annotations used in this book are depicted and are grouped by their purpose.

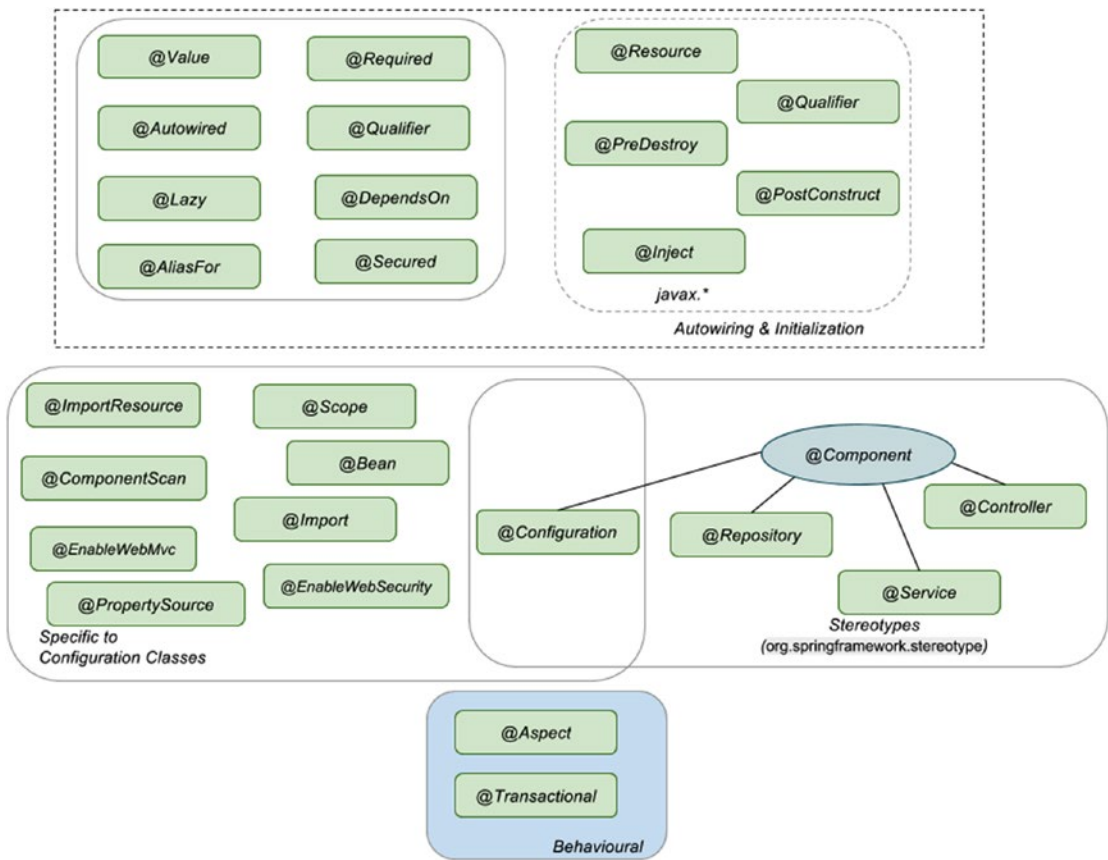


Figure 2-20. Annotations used in this book

- Stereotypes annotations are used to mark classes according to their purpose:
 - @Component: template for any Spring-managed component(bean).
 - @Repository: template for a component used to provide data access, specialization of the @Component annotation for the the Dao layer.
 - @Service: template for a component that provides service execution, specialization of the @Component annotation for the Service layer.
 - @Controller: template for a web component, specialization of the @Component annotation for the Web layer.
 - @Configuration: configuration class containing bean definitions (methods annotated with @Bean).
- Autowiring and initialization annotations are used to define which dependency is injected and what the bean looks like. For example:
 - @Autowired: core annotation for this group; is used on dependencies to instruct Spring IoC to take care of injecting them. Can be used on fields, constructors, and setters. Use with @Qualifier from Spring to specify name of the bean to inject.

- `@Inject`: equivalent annotation to `@Autowired` from `javax.inject` package. Use with `@Qualifier` from `javax.inject` to specify name of the bean to inject.
 - `@Resource`: equivalent annotation to `@Autowired` from `javax.annotation` package. Provides a name attribute to specify name of the bean to inject.
 - `@Required`: Spring annotation that marks a dependency as mandatory, used on setters.
 - `@Lazy`: dependency will be injected the first time it is used.
- Annotations that appear only in (and on) classes annotated with `@Configuration`; they define the configuration resources, the components, and their scope.
 - Behavioral annotations are annotations that define behavior of a bean. They might as well be named proxy annotations, because they involve proxies being created to intercept requests to the beans being configured with them.

JSR 250 annotations contained in JDK, package `javax.annotation` are supported. Also in Spring 3, support for JSR 330¹⁸ was added, and the Spring annotations are implementations of JSR 330 annotations. A complete list of the annotations in each package is depicted in Figure 2-21. Most of them are Java annotations that provide a minimum behavior of the Spring annotations.

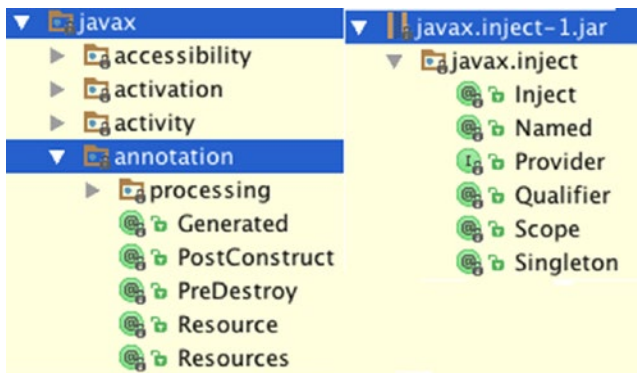


Figure 2-21. JSR 250 and JSR 330 annotations supported by Spring

Now that the stars of the following sections have been introduced, let the show begin!

Using Multiple Sources of Configuration

Classes annotated with `@Configuration` contain bean definitions. There can be one or more in a Spring application, and they can be combined with XML configuration files. These classes can be bootstrapped¹⁹ in many ways, depending on the chosen setup of the configuration. The `DataSourceConfig` class used in the previous example is a typical Java Configuration class, which contains two bean definitions and some properties that are injected from a property file using the `PropertySource` annotation.

¹⁸Extension of the Java dependency injection API <https://jcp.org/en/jsr/detail?id=330>.

¹⁹Bootstrapping in Spring means loading an application context.

```

import org.springframework.jdbc.datasource.DriverManagerDataSource;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration
@PropertySource("classpath:db/datasource.properties")
public class DataSourceConfig {
    @Value("${driverClassName}")
    private String driverClassName;
    ...

    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    public DataSource dataSource() throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}

```

The `@Bean` annotation is used to tell Spring that the result of the annotated method will be a bean that has to be managed by it. The `@Bean` annotation together with the method are treated as a bean definition, and the method name becomes the bean id.

The `PropertySource` annotation adds a bean of type `PropertySource` to Spring's environment that will be used to read property values from a property file set as argument. The configuration also requires a bean of type `PropertySourcesPlaceholderConfigurer` to replace the placeholders set as arguments for the `@Value` annotated properties. This class can be bootstrapped with an `AnnotationConfigApplicationContext` instance as shown earlier. But in combination with XML, it can also be bootstrapped with `ClassPathXmlApplicationContext`. In this case, we need an XML configuration file that has component scanning enabled, so the configuration class can be picked up.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <!-- does not pickup @Configuration so the configuration
         class must be declared as a bean-->
    <context:annotation-config/>
    <bean class="com.ps.config.DataSourceConfig" />

    <!-- or the more practical way -->
    <!-- Picks up everything-->
    <context:component-scan base-package="com.ps.config"/>
</beans>

```

There are two examples of configuration using XML in the following code snippet. The `<context:annotation-config/>` activates various annotations to be detected in bean classes: Spring's `@Required` and `@Autowired`, JSR 250's `@PostConstruct`, `@PreDestroy`, and `@Resource` and a few others. This configuration element is rarely used these days, because `<context:component-scan />` extends it and is more practical to use because it supports a lot of attributes for filtering and reducing scope of scanning.

In a Spring test environment, the `spring-test` library provides the `@ContextConfiguration` to be used in order to bootstrap a test environment using one or multiple configuration resources. The code below depicts this scenario, and it won't be explained now, because it will be covered in **Chapter 3**.

```
...
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {DataSourceConfig.class})
public class CfgToXmlTest {

    @Autowired
    DataSource dataSource;

    @Test
    public void testBoot() {
        assertNotNull(dataSource);
    }
}
```

When configuration classes and XML configuration files are used together to configure an application, they can be configured and used as in the XML snippet and bootstrap example above, but the reverse is possible also, to import a configuration from XML files and bootstrap everything with a `AnnotationConfigApplicationContext` instance, using the `@ImportResource` annotation on the configuration class. The following snippet of code depicts this situation.

```
<!-- user-repo-config.xml -->
<beans">

    <bean id="userRepo" class="com.ps.repos.impl.JdbcUserRepo"
        p:dataSource-ref="dataSource"/>

</beans>

//UserRepoConfig.java
import org.springframework.context.annotation.ImportResource;
...
@Configuration
@PropertySource("classpath:db/datasource.properties")
@ImportResource("classpath:spring/user-repo-config.xml")
public class UserRepoDSConfig {
```

```

    @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
    public DataSource dataSource() throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}

//Test class
...
import org.junit.Test;

public class XmlToCfgTest {

    private ApplicationContext ctx;

    @Before
    public void setup() {
        ctx = new AnnotationConfigApplicationContext(UserRepoDSConfig.class);
        assertNotNull(ctx);
    }

    @Test
    public void testStart3() {
        DataSource dataSource = ctx.getBean("dataSource", DataSource.class);
        assertNotNull(dataSource);

        UserRepo userRepo = ctx.getBean("userRepo", UserRepo.class);
        assertNotNull(userRepo);
    }
}

```

What actually happens in the previous example? The bean definitions in the XML file are imported into the `UserRepoDSConfig` that is bootstrapped for testing. And the test passes. The classes in this section can be found in `02-ps-container-02-practice`. The configuration file is located at `02-ps-container-02-solution/src/main/resources/spring/user-repo-config.xml`, the `UserRepoDSConfig` is located at `02-ps-container-02-solution/src/main/java/com/ps/config/UserRepoDSConfig` and the test class at `02-ps-container-02-solution/src/test/java/com/ps/XmlToCfgTest.java`, right next to the previous test class `CfgToXmlTest.java`.

Another situation that you will have to deal with when developing Spring applications is having multiple configuration classes. This is recommended for the same reason it is recommended to have multiple XML configuration files, to group together beans with the same responsibility and to make the application testable. Bootstrapping an application configured via multiple configuration classes can be done using an `AnnotationConfigApplicationContext` instance, which will be initialized with the two classes as parameters.

```
ApplicationContext ctx =
    new AnnotationConfigApplicationContext(DataSourceConfig.class,
        PetRepoConfig.class);
```

In case you have configuration classes with bean definitions with the same purpose in the application, the more practical approach would be to use the `@Import` annotation to import the bean definition in one class into the other.

```
import org.springframework.context.annotation.Import;

@Configuration
@Import({DataSourceConfig.class, UserRepoDSConfig.class})
public class AllRepoConfig {
    ...
}
```

In the configuration examples so far we have used bean definitions by methods annotated with `@Bean`, which are always found under a class annotated with `@Configuration`. For large applications, you can imagine that this is quite impractical, which is why there is another way. The simplest way to define a bean in Spring is to annotate the bean class with `@Component` (or any of the stereotype annotations that apply) and enable component scanning. Of course, this is applicable only to classes that are part of the project. For classes that are defined in third party libs like the `DataSource` in the examples presented so far, `@Bean` is the only solution. The same applies to Spring infrastructure beans too, like the `PropertySourcesPlaceholderConfigurer`.

In using Java Configuration, component scanning is enabled by annotating the configuration class with `@ComponentScan`. It works the same way as `<context:component-scan />` for XML. The code snippet below makes use of component scanning to find the bean definition for `JdbcRequestRepo`.

```
package com.ps.repos.impl;
...
import org.springframework.stereotype.Repository;

//JdbcRequestRepo.java
@Repository
public class JdbcRequestRepo extends
    JdbcAbstractRepo<Request> implements RequestRepo{
    ...
}
```

```
//RequestRepoConfig.java
...
import org.springframework.context.annotation.ComponentScan;
@Configuration
@Import(DataSourceConfig.class)
@ComponentScan(basePackages = "com.ps")
public class RequestRepoConfig {
}
```

This setup can be tested by bootstrapping the configuration and testing that `jdbcRequestRepo` can be found. The name of the `jdbcRequestRepo` bean is named like this by the Spring IoC, because no explicit bean name was configured. More details about bean naming using Java Configurations can be found in the bean naming section.

```
public class BootstrapTest {

    @Before
    public void setup() {
        ctx = new AnnotationConfigApplicationContext(DataSourceConfig.class,
            RequestRepoConfig.class);
        assertNotNull(ctx);
    }

    @Test
    public void testStart5() {
        DataSource dataSource = ctx.getBean("dataSource", DataSource.class);
        assertNotNull(dataSource);

        RequestRepo requestRepo = ctx.getBean("jdbcRequestRepo", RequestRepo.class);
        assertNotNull(requestRepo);
    }
}
```

Bean Naming

Bean naming is simpler when using JavaConfigurations for a number of reasons.

CC When the name is not defined for a bean declared with `Bean`, the Spring IoC names the bean with the annotated method name.

The name can be set by populating the `name` attribute. The same attribute can receive as argument an array of names. The first one becomes the name; the rest become aliases.

```
// bean name = dataSource

@Bean

public DataSource dataSource() throws SQLException {
```

```

...
}
//bean name = one
@Bean(name="one")
public DataSource dataSource() throws SQLException {
...
}
//bean name = one, alias = two
@Bean(name={"one", "two"})
public DataSource dataSource() throws SQLException {
...
}

```

CC When the name is not defined for a bean declared with `Component`, the Spring IoC creates the name of the bean from the name of the bean type, by lowercasing the first letter.

That is why in the previous example, the bean name was `jdbcTemplate`, because the class name was `JdbcTemplate`. If you want to rename it, all you have to do is give the name to the `@Repository` annotation (or any of the stereotype annotations) as argument:

```

// bean name = jdbcTemplate
@Repository
public class JdbcTemplate extends
    JdbcAbstractRepo<Request> implements RequestRepo{
    ...
}
// bean name = requestRepo
@Repository("requestRepo")

```



```

public class JdbcRequestRepo extends
    JdbcAbstractRepo<Request> implements RequestRepo{
    ...
}
// or
@Repository(value="requestRepo")
public class JdbcRequestRepo extends
    JdbcAbstractRepo<Request> implements RequestRepo{
    ...
}
// or
// bean name = requestRepo
@Component("requestRepo")
public class JdbcRequestRepo extends
    JdbcAbstractRepo<Request> implements RequestRepo{
    ...
}

```

Aliases cannot be defined using stereotype annotations, but a request has already been made to provide such a feature.²⁰

Related to naming is the `@Description` annotation, which was added in Spring 4.x. This annotation is used to add a description to a bean, which is quite useful when beans are exposed for monitoring purposes, as you will see in the JMX section. This can be used together with `@Bean` and `@Component` (and its specializations).

```

@Repository
@Description("This is not the bean you are looking for")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo {
    ...
}

```

²⁰If you are interested in this feature, you can follow the evolution of the issue here: <https://jira.spring.io/browse/SPR-6736>.

In Spring 4.2, the `@AliasFor` annotation was added. This annotation is set on annotation attributes to declare them as aliases for other annotation attributes. In the example below, a `@DsCfg` annotation was declared with a single attribute called `loginTimeout`.

```
package com.ps.repos.impl;

public @interface DsCfg {
    int loginTimeout() default 3600;
}
```

Repository bean definitions can be annotated with this annotation to automatically set the `loginTimeout` value for the `dataSource` to the one specified by the `loginTimeout` attribute value of the annotation.²¹ Injecting the value from the annotation into the appropriate field of the bean will obviously be done using a post processor bean (Remember those?), and the code needed for this is irrelevant for this example and will not be depicted here.

```
package com.ps.sample;
...
import com.ps.sample.DsCfg;

@DsCfg(loginTimeout = 4800)
@Repository("requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo {
    ...
}
```

The attribute name is quite long, right? If the application is big, renaming the attribute would be a pain, so this is where `@AliasFor` comes to the rescue.

```
package com.ps.sample;
import org.springframework.core.annotation.AliasFor;
public @interface DsCfg {

    int loginTimeout() default 3600;

    @AliasFor(attribute = "loginTimeout")
    int lTout() default 3600;
}
```

Using `@AliasFor`, the alias `lTout` was defined for the `loginTimeout` attribute, so the repository definition above can be written now like this:

```
package com.ps.sample;
...
import com.ps.sample.DsCfg;
```

²¹In practice, you will probably never need to do this; we are doing this to give a concrete example of the use of `@AliasFor`.

```

@DsCfg(lTout = 4800)
@Repository("requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo {
...
}

```

And even more can be done. Aliases for meta-annotation attributes can be declared. Let's say that we do not like the value attribute name of the `@Repository` annotation and we want in our application to make it more obvious that the `@Repository` annotation can be used to set the id of a bean, by adding an `id` attribute that will be an alias for the value attribute. The only conditions for this to work are that the annotation declaring the alias has to be annotated with the meta-annotation, and the annotation attribute must reference the meta-annotation.

```

package com.ps.sample;
import org.springframework.core.annotation.AliasFor;

@Repository
public @interface MyRepoCfg {

    @AliasFor(annotation = Repository.class, attribute = "value")
    String id() default "";
}

```

Thus, the repository beans can be declared as depicted in the following code snippet:

```

...
import com.ps.MyRepoCfg;
...

//JdbcRequestRepo. java bean definition
@MyRepoCfg(id = "requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo{
...
}

```

Field, Constructor, and Setter Injection

At the beginning of the XML configuration section it was mentioned that field injection is not supported in XML. This is because the `<bean />` element definition does not support this.²² In using annotations to configure beans, field injection is possible, since the central annotation used to define dependencies `@Autowired` can be used on fields, constructors, setters, and even methods.

The term `autowire` is the short version for automatic dependency injection. This is possible only in Spring applications using component scanning and stereotype annotations to create beans. The `@Autowired` annotation indicates that Spring should take care of injecting that dependency. This raises an interesting question: how does Spring know what to inject?

²²It only supports the `<constructor-arg />` element for constructor injection and `<property />` element for setter injection.

CC Out of the box, Spring will try to **autowire by type**, because rarely in an application is there need for more than one bean of a type. Spring will inspect the type of dependency necessary and will inject the bean with that exact type. Let's consider a configuration class called `RequestRepoConfig` that defines one bean of type `JdbcRequestRepo`. Let's specify an explicit name for the repository bean of type `JdbcRequestRepo` and then define an autowired dependency in a typical Spring test class. In using `@Autowire` on a field, we are making use of field injection.

```
//JdbcRequestRepo.java bean definition
@Repository("requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo{
    ...
}

//configuration class RequestRepoConfig.java
...
import org.springframework.context.annotation.ComponentScan;

@Configuration
@Import(DataSourceConfig.class)
@ComponentScan(basePackages = "com.ps")
public class RequestRepoConfig {
}

//AutowireTest.java Spring typical test class
import org.springframework.beans.factory.annotation.Autowired;
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {RequestRepoConfig.class})
public class AutowiringTest {

    @Autowired
    RequestRepo reqRepo;

    @Test
    public void testAutowiredField() {
        assertNotNull(reqRepo);
    }
}
```

The test above will pass as long as there is exactly one bean of type `RequestRepo` defined in the context. Spring does not care how the bean is named, unless there are more beans of the same type in the context.

CC By default, if Spring cannot decide which bean to autowire based on type (because there are more beans of the same type in the application), it defaults to autowiring by name. The name considered as the criterion for searching the proper dependency is the name of the field being autowired.

In the previous case, if Spring cannot decide what to autowire based on type, it will look for a bean named `reqRepo`. Let's define in the `RequestRepoConfig` another bean of type `RequestRepo` and let's name it `anotherRepo` and then see what happens.

```
@Configuration
@Import(DataSourceConfig.class)
@ComponentScan(basePackages = "com.ps")
public class RequestRepoConfig {

    @Bean
    public RequestRepo anotherRepo(){
        return new JdbcRequestRepo();
    }
}
```

The class `AutowireTest` is part of `02-ps-container-02-practice`, and the log for the Spring framework has been set to debug. When running the `testAutowiredField` test method, here is what can be seen in the log:²³

```
...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'requestRepo'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean 'anotherRepo'
ERROR o.s.t.c.TestContextManager - Caught exception while allowing TestExecutionListener
to prepare test instance com.ps.AutowiringTest@6ec8211c UnsatisfiedDependencyException
Error creating bean with name 'com.ps.AutowiringTest':
    Unsatisfied dependency expressed through field 'reqRepo'
No qualifying bean of type com.ps.repos.RequestRepo is defined:
    expected single matching bean but found 2: requestRepo,anotherRepo
```

In this case, we have two beans of type `RequestRepo`, and neither of them is named `reqRepo`, so Spring cannot decide. The first solution would be to rename the `anotherRepo` method. The second would be to tell Spring via `@Autowired + @Qualifier` (the one provided by Spring) that the bean named `anotherRepo` should be used for autowiring, because that is what the `Qualifier` annotation is used for, to tell Spring which bean qualifies for autowiring from the developers's point of view.

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

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {RequestRepoConfig.class})
public class AutowiringTest {

    @Qualifier("anotherRepo")
    @Autowired
    RequestRepo reqRepo;
}
```

²³The log entries were cleaned up a little, and only a snippet of the log is presented here for obvious reasons.

The `@Autowired` annotation can be used on constructors to tell Spring to use autowiring in order to provide arguments for that constructor. The way Spring will identify the autowiring candidate is by using the same logic presented before: it will try to find a unique bean of the parameter type. If it finds more than one, the one named as the parameter will be injected. The `JdbcRequestRepo` bean type needs a `dataSource` in order to function properly. That dependency will be provided using **constructor injection** by annotating the constructor with `@Autowired`. The test class used so far can still be used to test the `reqRepo` field, because if the `dataSource` cannot be injected, the bean can't be created, and the test will fail. And if you run the test and it passes, the log will be a pretty good source for understanding what happens *under the hood*.

```
@Repository("requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo{

    @Autowired
    public JdbcRequestRepo(DataSource dataSource) {
        super(dataSource);
    }
}
//AutowiringTest.java Spring typical test class
...
@Qualifier("requestRepo")
@Autowired
RequestRepo reqRepo;
...
// the log when bootstrapping the application
DEBUG o.s.b.f.s...Factory - Creating instance of bean 'requestRepo'
DEBUG o.s.b.f.s...Factory - Returning cached instance of singleton bean 'one'
DEBUG o.s.b.f.s...Factory - Returning cached instance of singleton bean 'dataSource'
DEBUG o.s.b.f.s...Factory -
    Autowiring by type from bean name 'requestRepo' via constructor
    to bean named 'dataSource'
DEBUG o.s.b.f.a.AutowiredAnnotationBeanPostProcessor -
    Autowiring by type from bean name 'com.ps.AutowiringTest'
    to bean named 'requestRepo'
}
```

To test that autowiring of the `dataSource` succeeded, a breakpoint can be placed in the `testAutowiredReq` method. Run the test in debug mode, and inspect the contents of the `reqRepo` field. In Figure 2-22, such an operation is depicted, and the target bean's contents are evidencediated.

```

17 @RunWith(SpringJUnit4ClassRunner.class)
18 @ContextConfiguration(classes = {RequestRepoConfig.class})
19 public class AutowiringTest {
20     ... // @Qualifier("anotherRepo")
21     ... @Qualifier("requestRepo")
22     ... @Autowired
23     ... RequestRepo reqRepo; reqRepo: JdbcRequestRepo@2649
24     ... @Test
25     ... public void testAutowireReq() {
26     ...     assertNotNull(reqRepo); reqRepo: JdbcRequestRepo@2649
27     ... }
28 }
29 }
30 }

```

Variables

```

this = {AutowiringTest@2723}
reqRepo = {JdbcRequestRepo@2649}
  dataSource = {DriverManagerDataSource@2657}
    url = "jdbc:h2:~/sample"
    username = "sample"
    password = "sample"
    connectionProperties = null
    logger = {SLF4JLocationAwareLog@2735}

```

Figure 2-22. Running a test in debug mode to test constructor injection

In case Spring finds more than candidate for autowiring when constructor injection is used, the solution is still to use `@Qualifier`, only in this case, this annotation is used on the parameter, because a constructor can have more parameters, and Spring must be instructed exactly where each bean will be injected. The definition of the constructor will change like this:

```

@Autowired
public JdbcRequestRepo(@Qualifier("oracleDataSource") DataSource dataSource) {
    super(dataSource);
}

```

// Configuration class that defines two datasource beans

```

@Configuration
public class DataSourceConfig {

    @Bean(name = "h2DataSource" )
    public DataSource dataSource() throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        //populate
        return ds;
    }
}

```

```

@Bean(name = "oracleDataSource" )
public DataSource dataSource() throws SQLException {
    OracleDataSource ds = new OracleDataSource();
    //populate
    return ds;
}
}

```

In using `@Autowired` on constructors, it makes not sense to have more than one constructor annotated with it, and Spring will complain about it because it will not know what constructor to use to instantiate the bean.

Everything that was presented about the capabilities of `@Autowired` and `@Qualifier` and how to use them to control bean autowiring applies for setter injection too. So except for a code snippet, there nothing more that can be said about **setter injection** via autowiring.

```

public class JdbcUserRepo extends JdbcAbstractRepo<User>
    implements UserRepo {
    protected DataSource dataSource;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    //or by name
    @Autowired
    public void setDataSource(@Qualifier("oracleDataSource") DataSource dataSource) {
        this.dataSource = dataSource;
    }
}

```

The `@Autowired` annotation by default requires the dependency to be mandatory, but this behavior can be changed, by setting the `required` attribute to `true`:

```

@Autowired(required=false)
public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

```

The `@Autowired` annotation works on methods, too. If Spring can identify the proper bean, it will autowire it. This is useful for development of special Spring configuration classes that have methods that are called by Spring directly and have parameters that have to be configured. The most common example is the setting method global security using classes annotated with `@EnableGlobalMethodSecurity`:

```

@Configuration
@EnableGlobalMethodSecurity
public class MethodSecurityConfig {

```



```
// method called by Spring
@Autowired
public void registerGlobal(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .inMemoryAuthentication()
        .withUser("john").password("test").roles("USER").and()
        .withUser("admin").password("admin").roles("USER", "ADMIN");
}

@Bean
public MethodSecurityService methodSecurityService() {
    return new MethodSecurityServiceImpl()
}
}
```

A more detailed version of this code snippet will be covered and explained in detail in the **Spring Security** section, in the chapter **Spring Web**.

A strong feature for `@Autowired` added in Spring 4.x is the possibility to use generic types as qualifiers. This is useful when you have classes that are organized in a hierarchy and they all inherit a certain class that is generic, like the repositories in the project attached to the book, all of which extend `JdbcAbstractRepo<T>`. Let's see how this feature can be used.

```
// repository classes extending JdbcAbstractRepo
public class JdbcReviewRepo extends JdbcAbstractRepo<Review>
    implements ReviewRepo { ...}
public class JdbcResponseRepo extends JdbcAbstractRepo<Response>
    implements ResponseRepo{ ...}

//configuration class defining beans of the previous types
@Configuration
@Import({DataSourceConfig.class, UserRepoDSConfig.class})
public class AllRepoConfig {

    @Bean
    public ReviewRepo reviewRepo(){
        return new JdbcReviewRepo();
    }

    @Bean
    public ResponseRepo responseRepo(){
        return new JdbcResponseRepo();
    }
}

// Test class
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {AllRepoConfig.class})
public class GenericQualifierTest {
```

```

@Autowired
JdbcAbstractRepo<Review> reviewRepo;

@Autowired
JdbcAbstractRepo<Response> responseRepo;
....
}

```

This helps a lot, because in related classes, the `Qualifier` annotation is no longer needed to name different beans of related types in order to make sure that Spring does not get confused.

It was mentioned before that the JSR 250 and JSR 330 annotations are supported and can be used alongside Spring annotations, but they have quite a few common functionalities. In Table 2-3, you can see the correspondences and the differences between them.

Table 2-3. Prefixes and corresponding paths

Spring	JSR	Comment
@Component	@Named	@Named can be used instead of all stereotype annotations except @Configuration
@Qualifier	@Qualifier	JSR Qualifier is a marker annotation used to identify qualifier annotations, like @Named, for example
@Autowired	@Inject	@Inject may apply to static as well as instance members
@Autowired + @Qualifier	@Resource(name="beanName")	@Resource is useful because replaces two annotations.

Injecting Dependencies That Are Not Beans

Because of its syntax and its definition, `@Autowired` cannot be used to autowire primitive values, or Strings, which is quite logical, considering that there is an annotation named `@Value` that specializes in this exactly. This annotation can be used to insert scalar values or can be used together with placeholders and SpEL in order to provide flexibility in configuring a bean. The `DataSourceConfig` is used to define a `DataSource` bean, and the property values needed to initialize the `DataSource` are injected using `@Value`.

```

import org.springframework.beans.factory.annotation.Value;
...
@Configuration
@PropertySource("classpath:db/datasource.properties")
public class DataSourceConfig {

    @Value("${driverClassName}")
    private String driverClassName;
    @Value("${url}")
    private String url;

    ...
}

```

```

@Bean
public DataSource dataSource() throws SQLException {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(driverClassName);
    ds.setUrl(url);
    ...
    return ds;
}
}

```

Also, the `@Value` annotation can also be used with SpEL in order to inject values from other beans, or objects treated as beans, in the following case from a `Properties` object:

```

@Configuration
public class DataSourceConfig1 {

    @Bean
    public Properties dbProps(){
        Properties p = new Properties();
        p.setProperty("driverClassName", "org.h2.Driver");
        p.setProperty("url", "jdbc:h2:~/sample");
        p.setProperty("username", "sample");
        p.setProperty("password", "sample");
        return p;
    }

    @Bean
    public DataSource dataSource(
        @Value("#{dbProps.driverClassName}")String driverClassName,
        @Value("#{dbProps.url}")String url,
        @Value("#{dbProps.username}")String username,
        @Value("#{dbProps.password}")String password) throws SQLException {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName(driverClassName);
        ds.setUrl(url);
        ds.setUsername(username);
        ds.setPassword(password);
        return ds;
    }
}

```

The `dbProps` is an object that was instantiated and initialized, and it is treated as a bean in the application because of the `@Bean` annotation. There is no equivalent for `@Value` in any JSR.

Bean Lifecycle and Scopes

Annotated beans and methods are discovered by component scanning. The bean lifecycle is the same no matter how the application is configured, and how the beans are defined, and although the behavior looks the same from outside, under the hood something different happens. And this is normal, since Java Configuration and XML configurations are two different things, but there are also common elements. Let's cover them step by step.

Bean Definitions Loading

- Recall from the XML bean lifecycle section that in the first step, the bean definitions are loaded. When XML is used, a bean of type `org.springframework.beans.factory.xml.XmlBeanDefinitionReader` is used to load the bean definitions from the XML file(s).
- For a configuration using Java Configuration annotations, the classpath is scanned by a bean of type `org.springframework.context.annotation.ClassPathBeanDefinitionScanner`, and the bean definitions are registered by a bean of type `org.springframework.context.annotation.ConfigurationClassBeanDefinitionReader`.²⁴
- For **mixed** configurations using XML and stereotype annotations, both classes mentioned in the previous cases are used.

Bean Creation

In all cases, XML and annotation beans are created by a factory of type `org.springframework.beans.factory.support.DefaultListableBeanFactory`.

Injecting Dependencies

- XML: the factory bean is used to inject dependencies.
- Java Configuration and all other annotations: a `org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor` bean is used to autowire dependencies. This is a post processor bean implementation that autowires annotated fields, setter methods, and arbitrary config methods. It is registered by the `@Configuration` annotation, but it can also be registered in mixed configurations by component scanning. This bean takes care of the autowiring configured with `@Autowired`, `@Value`, and `@Inject`.
- For **mixed** configurations, the two approaches above are used together.

Bean Destruction

In all cases, XML and annotations, the dirty job of disposing of a bean is the responsibility of `org.springframework.beans.factory.support.DisposableBeanAdapter` bean.

And this is all that can be said about the internals of processing annotations. All of them are picked up and treated accordingly by the bean post processor. If you are really curious regarding what other beans contribute to running an application, just set the log for the Spring Framework to `DEBUG` and enjoy the display.

²⁴`ConfigurationClassBeanDefinitionReader` is a Spring internal class and does not appear in the Spring API, but the source code can be found on GitHub at <https://github.com/spring-projects/spring-framework/blob/master/spring-context/src/main/java/org/springframework/context/annotation/ConfigurationClassBeanDefinitionReader.java>.

In the analogous XML section it was mentioned that there were equivalents for `init`-method and `destroy`-method from the `<bean/>` element that will be covered in the annotation section,²⁵ because they were represented by two attributes of the `@Bean` annotation. Please analyze the following example:

```
// JdbcRequestRepo.java
@Repository("requestRepo")=
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
    implements RequestRepo {
    private Logger logger = LoggerFactory.getLogger(JdbcRequestRepo.class);

    private void init() {
        logger.info(" ... initializing requestRepo ...");
    }
    private void destroy() {
        logger.info(" ... destroying requestRepo ...");
    }
    ...
}

//RequestRepoConfig.java configuration class
@Configuration
@Import(DataSourceConfig.class)
@ComponentScan(basePackages = "com.ps")
public class RequestRepoConfig {

    @Bean (initMethod = "init", destroyMethod = "destroy")
    public RequestRepo anotherRepo(){
        return new JdbcRequestRepo();
    }
}
}
```

The `initMethod` attribute of the Java Configuration annotation `@Bean` specifies the method name to be called in order to initialize `anotherRepo`, and the `destroyMethod` attribute specifies the method to call when the context is closed. The bean definition from the `RequestRepoConfig` class is basically equivalent to this:

```
<beans ...>
    <bean id="anotherRepo" class="com.ps.repos.impl.JdbcRequestRepo"
        init-method="init", destroy-method="destroy" />
</beans>
```

A bean that is created from a definition marked with `@Bean` or `@Component` or any of its specializations is by default a singleton. But the scope can be changed by annotating the class or method with `@Scope` and setting the value attribute to a different scope.

```
@Component
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class SimpleBean {
    ...
}
```

²⁵Java Configuration and stereotype annotations were introduced together, because the Java Configuration is just the second Spring step in totally removing configuration via XML.

! `@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)` is equivalent to `@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)`, and `@Scope("prototype")` as constant `SCOPE_PROTOTYPE` is of type string with the “prototype” value. Using Spring constants eliminates the risk of misspelling the scope value.

Another interesting annotation is `@Lazy`, which starting with Spring 4.x can be used on injection points (wherever `@Autowired` is used) too. This annotation can be used to postpone the creation of a bean until it is first accessed, by adding this annotation to the bean definition. This is useful when the dependency is a huge object and you do not want to keep the memory occupied with this object until it is really needed.

```
@Component
@Lazy
public class SimpleBean { ... }

// or on a @Bean
@Configuration
public class RequestRepoConfig {

    @Lazy
    @Bean
    public RequestRepo anotherRepo(){
        return new JdbcRequestRepo();
    }
}

// on injection point
@Repository
public class JdbcPetRepo extends JdbcAbstractRepo<Pet>
implements PetRepo {
    ...
    @Lazy
    @Autowired(required=false)
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Aside from all the annotations discussed so far, developers can develop their own annotations. Many of the annotations provided by Spring can be used as meta-annotations in your own code. You can view a meta-annotation as a super class. A meta-annotation can be used to annotate another annotation. All stereotype specializations are annotated with `@Component`, for example. Meta-annotations can be composed to obtain other annotations. For example, let’s create a `@CustomTx` annotation that will be used to mark service beans that will use a different transaction:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
import org.springframework.transaction.annotation.Transactional;

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Transactional("customTransactionManager", timeout="90")
public @interface CustomTx {
    boolean readOnly() default false;
}
```

! After all this talk about annotations and how great they are, you probably want to use them. The `02-ps-container-02-practice` contains all examples from the Java Config and Annotations sections. All the configuration classes are in the `com.ps.config` package. All the bean classes are under `com.ps.repo`. In the `resources` directory you can find some XML configuration classes used in the mixed examples. There is only one TODO task for this project. The TODO number 13 requires you to replace wherever you can the Spring annotations with JSR annotations.

Spring configuration using annotations is really powerful and flexible. In this book, Java Configuration and Annotations are covered together, because this is the way they are commonly used in practice. There definitely are old-style developers who still prefer XML and more grounded developers who like to *mix and match*. Configuration using annotations is practical, because it can help you reduce the number of resource files in the project, but the downside is that it is scattered all over the code. But being linked to the code, refactoring is a process that becomes possible without the torture of searching bean definitions in XML files, although smart editors help a lot with that these days. The annotations configuration is more appropriate for beans that are frequently changing: custom business beans such as services, repositories, and controllers. Imagine adding a new parameter for a constructor and then going hunting beans in the XML files so the definition can be updated. Java Configuration annotations should be used to configure infrastructure beans (data sources, persistence units, etc).

There are advantages to using XML too. The main advantage is that the configuration is centralized in a few parts of the application, but Java Configuration does the same. Then, XML is widely supported and known. Ultimately, it is only the context that decides the most appropriate solution. If you are a Spring expert starting to work on your own startup application, you will probably go with Java Configuration and annotations. And probably Spring Boot, which will make your work much easier by providing super-meta-annotations to configure much of the infrastructure needed for the project. If you are working on a big project with legacy code, you will probably be doomed with some XML configuration here and there. Whatever the case, respect good practices and read the manual, and you should be fine.

Summary

After reading this chapter you should possess enough knowledge to configure a Spring application, using XML, Java Configurations, and annotations or all of them, and to harness the power of beans to develop small Spring Applications.

- Spring is built for dependency injection and provides a container for Inversion of Control, which takes care of injecting the dependencies according to the configuration.
- The two flavors of configuration can also be mixed: XML-based when beans declarations are decoupled from code and Java Configuration when bean declarations are in the code.

- Spring promotes the use of interfaces, so beans of types implementing the same interface can be easily interchanged.
- The Spring way of configuration promotes testability. Since beans can be interchanged, it is easy to use stubs and mocks to isolate components.
- The bean lifecycle can be controlled; behavior can be added at specific points during the bean life.
- Spring offers many ways to simplify configuration: bean inheritance, nesting and namespaces in XML, and a lot of convention over configuration default behavior in Java Configuration.
- Bean definition sources can be coupled by importing them one into another, or by composing them in order to create an application context.

2.1 Quick quiz

Question 1: What is a bean? (choose all that apply)

- A. a Plain Old Java Object
- B. an instance of a class
- C. an object that is instantiated, assembled, and managed by a Spring IoC Container

Question 2: What is the default scope of a bean?

- A. default
- B. singleton
- C. protected
- D. prototype

Question 3: What are the types of dependency injection supported by Spring IoC Container? (choose all that apply)

- A. setter injection
- B. constructor injection
- C. interface-based injection
- D. field-based injection

Question 4: What class is used to bootstrap an XML configured application context?

- A. ClassPathXmlApplicationContext
- B. AnnotationConfigApplicationContext
- C. ApplicationContext

Question 5: Which of the following are stereotype annotations? (choose all that apply)

- A. @Component
- B. @Bean
- C. @PostConstruct
- D. @Scope
- E. @Configuration

Question 6: What is the difference between declaring a bean with @Bean and with @Component?

- A. @Bean annotates a method that creates a bean in a configuration class; @Component annotates classes to mark them as bean definitions for the Spring Container
- B. @Bean is a specialization of @Component and annotates classes to mark them as bean definitions for the Spring Container
- C. @Bean is the JSR 250 equivalent of @Component

Question 7: Given the following bean definition, what will be printed in the log when the application runs?

```
public class QuizBean {
    // assume correct logger instantiation here
    Logger log =...;

    private void initMethod() {
        logger.info("--> I'm calling it bean soon");
    }
}
// bean configuration
<bean id="quizBean" class="QuizBean"
    init-method="initMethod"/>
```

- A. the application won't run, it will crash in the initialization phase because the initMethod is private
- B. the application will run, but won't create the quizBean because its initialization method is private
- C. -> I'm calling it bean soon

Question 8: Given the **quizBean** bean definition, choose from the list the bean definitions that are equivalent.

```
<bean id="quizBean" class="QuizBean"
    init-method="initMethod">
    <property name="petitBean" ref="petitBean"/>
</bean>
```

A. @Component

```
public class QuizBean {
    public void setPetitBean(PetitBean petitBean) {
        this.petitBean = petitBean;
    }

    @PostConstruct
    private void initMethod() {
        logger.info("--> I'm calling it bean soon");
    }
}
```

B. @Configuration

```
public class QuizConfiguration {

    //assume petitBean() is defined here

    @Bean(initMethod="initMethod")
    private QuizBean getQuizBean() {
        return new QuizBean(petitBean());
    }
}
```

C. `<bean id="quizBean" class="QuizBean" init-method="initMethod" p:petitBean="petitBean" />`

Question 9: Which of the following beans is a bean factory post processor?

- A. ClassPathXmlApplicationContext
- B. PropertySourcesPlaceholderConfigurer
- C. CommonAnnotationBeanPostProcessor

Question 10: What is the correct way to import bean definitions from an XML file into a configuration class?

- A. `@Import("classpath:spring/app-config.xml")`
- B. `@Resource("classpath:spring/app-config.xml")`
- C. `@ImportResource("classpath:spring/app-config.xml")`

Question 11: Given the following configuration class and bean, the bean being autowired into another bean as dependency, what happens when the application runs?

```
package com.cfg;
//configuration
@Configuration
public class AppConfig {}
```

```
package com.quiz;
// bean
@Component
public class QuizBean {}

package com.another.quiz;
// autowiring the bean
@Component
public class AnotherQuizBean {
    @Autowired
    QuizBean quizBean;
}
```

- A. an exception will be thrown because there is no scope defined for the bean of type QuizBean
- B. an exception will be thrown because @ComponentScan is missing from the configuration class so the bean definitions are not discovered
- C. a bean of type QuizBean is created and autowired into a bean of type AnotherQuizBean

Question 12: The Spring IoC container by default tries to identify beans to autowire by type; if multiple beans are found, it chooses for autowiring the one that has the same name with the dependency being autowired

- A. True
- B. False

CHAPTER 3



Testing Spring Applications

Before an application is delivered to the client, it has to be tested and validated for use by a team of professionals called *testers*. As you can imagine, testing an application after development is complete is a little too late, because perhaps specifications were not understood correctly, or were not complete. Also, the behavior of an application on an isolated development system differs considerably from the behavior on a production system. That is why there are multiple testing steps that have to be taken, some of them even before development. And there is also the human factor. Since no one is perfect, mistakes are made, and testing helps find those mistakes and fix them before the application reaches the end user, thus ensuring the quality of the software. The purpose of software testing is to verify that an application satisfies the functional (application provides the expected functions) and nonfunctional (application provides the expected functions as fast as expected and does not require more memory than is available on the system) requirements and to detect errors, and all activities of planning, preparation, evaluation, and validation are part of it.

There are specific courses and certifications for testers that are designed to train them in functional and software testing processes that they can use to test an application, and the ISTQB¹ is the organization that provides the infrastructure for training and examination.

A Few Types of Testing

There are multiple types of testing classified by the development step in which they are executed, or by their implementation, but it is not the object of this book to cover them all. Only those that imply writing actual code using testing libraries and frameworks will be covered.

Test-Driven Development

Quality starts at the beginning of a project. Requirements and specifications are decided and validated, and based on them a process called **Test-Driven Development** can be executed. This process implies creation of tests before development of code. The tests will initially fail, but will start to pass one by one as the code is developed. The tests decide how the application will behave, and thus this type of testing is called test-driven development. This is a type of testing that ensures that the specifications were understood and implemented correctly. The tests for this process are designed by business analysts and implemented by developers. This approach puts the design under question: if tests are difficult to write, the design should be reconsidered. It is more suitable to JavaScript applications (because there is no compilation of the code), but it can be used in Java applications too when the development is done using interfaces.

¹International Software Testing Qualification Board: <http://www.istqb.org/>.

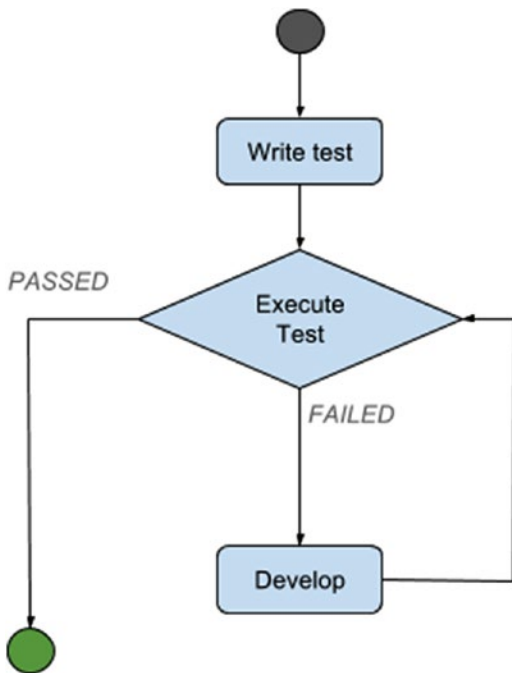


Figure 3-1. Test-driven development logical schema

In Figure 3-1, the test-driven development process is described for exactly one test case.

This testing technique is really good for finding problems early in the development process. And considering that the effort to solve a problem grows exponentially in proportion to the time it takes to find it, no project should ever be developed without it. Also, the tests should be run automatically using a continuous integration tool like Jenkins, Hudson, or Bamboo. Test-driven development can produce applications of high quality in less time than is possible with older methods, but it has its limitations. Sometimes tests might be incorrectly conceived or applied. This may result in units that do not perform as expected in the real world. Even if all the units work perfectly in isolation and in all anticipated scenarios, end users may encounter situations not imagined by the developers and testers. And since testing units was mentioned, this section will end here in order to cover the next one.

Unit and Integration Testing

Unit testing implies testing the smallest testable parts of an application individually and independently, isolated from any other units that might affect their behavior in an unpredictable way. The dependencies are kept to a minimum, and most of them will be replaced with pseudo-units reproducing the expected behavior. This means basically that the unit of functionality is taken out of context. The unit tests are written by developers and are run using automated tools, although they can be run manually too in case of need. A unit test exercises a single scenario with a provided input and compares the expected results for that input with the actual results. If they match, the test passes; if they don't, the test fails. This method of testing is fast and is one that is most used in many projects. The tests are written by developers, and the recommended practice is to cover every method in a class with positive and negative tests. *Positive tests* are the ones that test valid inputs for the unit, while *negative tests* test invalid inputs for the unit. These are tests that cover a failure of the unit, and they are considered to have failed if the unit does not actually fail when tested.

The core framework helping developers to easily write and execute unit tests in Java since 2000 is JUnit. The current version is 4.13, but version 5 is currently being developed, and an Alpha version was released in January 2016.

There are not many JUnit extensions, because there is little that this framework is missing, but there is a framework called Hamcrest that is quite interesting because it provides a set of matchers that can be combined to create flexible expressions of intent. It originated as a framework based on JUnit, and it was used for writing tests in Java, but managed to break the language barrier, and currently it is provided for most currently used languages such as Python and Swift. More about it can be found on the official site: <http://hamcrest.org/>.

Running a suite of unit tests together in a context with all their real dependencies provided is called **integration testing**. As the name of this technique implies, the infrastructure needed to connect objects being tested is also a part of the context in which tests are executed. In Figure 3-2, a simple diagram for comparing unit and integration testing concepts is depicted.

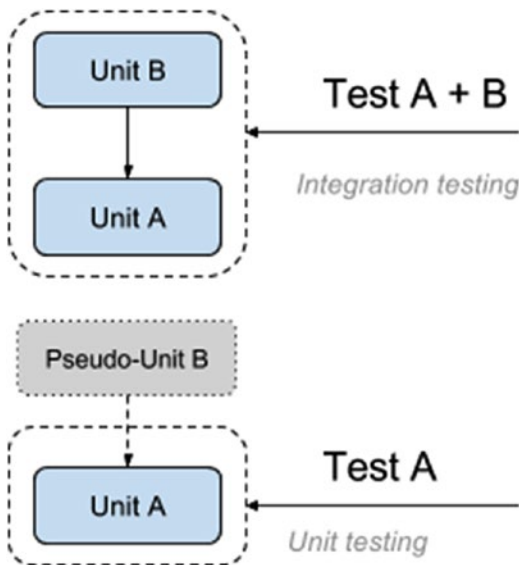


Figure 3-2. Unit and integration testing concepts

In order for functional units to be tested in isolation, dependencies must be replaced with pseudo-dependencies, fake objects with simple implementation that mimics the behavior of the real dependency as far as the dependent is concerned. The pseudo-dependencies can be *stubs* or *mocks*. Both perform the same function, to replace a real dependency, but they way they are created is what sets them apart.

Testing with Stubs

Stubs are created by the developer; they do not require extra dependencies. A stub is a concrete class implementing the same interface as the original dependency of the unit being tested. They should be designed to exhibit a small part or the whole behavior of the actual dependency.

For example, let's try to test one of the service classes that were introduced in the previous chapter, the `SimplePetService`.

```
package com.ps.services.impl;
...
public class SimplePetService extends SimpleAbstractService<Pet>
    implements PetService {

    // dependency that has to be stubbed
    private PetRepo repo;

    @Override
    public Pet createPet(User user, String name, int age,
        PetType petType, String rfid) {
        Pet pet = new Pet();
        pet.setOwner(user);
        pet.setName(name);
        pet.setAge(age);
        pet.setRfid(rfid);
        user.addPet(pet);
        repo.save(pet);
        return pet;
    }

    @Override
    public Set<Pet> findAllByOwner(User user) {
        return repo.findAllByOwner(user);
    }

    /**
     * @param user owner of the pet
     * @param name name of the pet
     * @return
     */

    @Override
    public Pet findByOwner(User user, String name) {
        return repo.findByOwner(user, name);
    }

    public void setRepo(PetRepo petRepo) {
        this.repo = petRepo;
    }

    public PetRepo getRepo() {
        return repo;
    }
}
```

```
// the interface defining the specific Pet behavior
public interface PetService {

    Pet createPet(User user, String name, int age, PetType petType,
        String rfid);

    Set<Pet> findAllByOwner(User user);

    Pet findByOwner(User user, String name);
}
```

This class inherits some behavior from the abstract class `SimpleAbstractService<Pet>`.

```
package com.ps.services.impl;
...
public abstract class SimpleAbstractService<T extends AbstractEntity>
    implements AbstractService<T>{

    public void save(T entity) {
        getRepo().save(entity);
    }

    public T findById(Long entityId){
        return getRepo().findById(entityId);
    }

    @Override
    public void delete(T entity) {
        getRepo().delete(entity);
    }

    @Override
    public void deleteById(Long entityId) {
        getRepo().deleteById(entityId);
    }

    abstract AbstractRepo<T> getRepo();
}
```

Although the number of tests varies depending on developer experience and what the code actually does, also related in a funny short story by Alberto Savoia, posted on the Google official blog,² my recommendation is to start unit testing by trying to write at least two tests for each method: one positive and one negative, for methods that can be tested in this way. There are seven methods in `SimplePetService`, besides the getter and setter, so the test class should have approximately fourteen tests. When testing the service class, the concern is that the class should interact correctly with the repository class. The behavior of the repository class is assumed known and fixed. The stub class will implement the typical repository behavior but without a database connection needed, because interaction with a database introduces an

²Here it is, in case you are curious: <http://googletesting.blogspot.ro/2010/07/code-coverage-goal-80-and-no-less.html>.

undesired lag in test execution. The implementation presented here will use a `java.util.Map` to simulate a database. As in the application, there are more repository classes extending `SimpleAbstractService<Pet>`; the stubs will follow the same inheritance design, so the abstract class will be stubbed as well.

```
package com.ps.repo.stub;
...
public abstract class StubAbstractRepo<T extends AbstractEntity>
    implements AbstractRepo<T> {

    protected Map<Long, T> records = new HashMap<>();

    @Override
    public void save(T entity) {
        if (entity.getId() == null) {
            Long id = (long) records.size() + 1;
            entity.setId(id);
        }
        records.put(entity.getId(), entity);
    }

    @Override
    public void delete(T entity) {
        if(records.containsKey(entity.getId())) {
            records.remove(entity.getId());
        } else {
            throw new NotFoundException("Entity with id "
                + entity.getId() + " could not be deleted because it does not exists");
        }
    }

    @Override
    public void deleteById(Long entityId) {
        if(records.containsKey(entityId)) {
            records.remove(entityId);
        } else {
            throw new NotFoundException("Entity with id "
                + entityId + " could not be deleted because it does not exists");
        }
    }

    @Override
    public T findById(Long entityId) {
        return records.get(entityId);
    }
}
```

The `StubPetRepo` class extends the previous stub class, adding its specific behavior, but skipping implementation methods that cannot be executed on the `Map` used to simulate the database. And since the `Map` contains `<id, Pet>` pairs, neither of the specific `PetRepo` methods can be stubbed, so a new `map` is needed.

```

package com.ps.repo.stub;
import org.apache.commons.lang3.NotImplementedException;
...
public class StubPetRepo extends StubAbstractRepo<Pet> implements PetRepo {

    // grouping pets by owner
    protected Map<User, Set<Pet>> records2 = new HashMap<>();

    //overriding the save method from StubAbstractRepo
    // to include stub behavior for PetRepo
    @Override
    public void save(Pet pet) {
        super.save(pet);
        addWithOwner(pet);
    }

    private void addWithOwner(Pet pet){
        if (pet.getOwner()!= null) {
            User owner = pet.getOwner();
            if (records2.containsKey(owner)) {
                records2.get(owner).add(pet);
            } else {
                Set<Pet> newPetSet = new HashSet<>();
                newPetSet.add(pet);
                records2.put(owner, newPetSet);
            }
        }
    }

    @Override
    public Pet findByOwner(User owner, String name) {
        Set<Pet> petSet = records2.get(owner);
        for (Pet pet: petSet) {
            if (pet.getName().equalsIgnoreCase(name)) {
                return pet;
            }
        }
        return null;
    }

    @Override
    public Set<Pet> findAllByOwner(User owner) {
        Set<Pet> petSet = records2.get(owner);
        // we never return null when returning collections
        // to avoid NullPointerException
        return petSet != null? petSet : new HashSet<>();
    }

    @Override
    public Set<Pet> findAllByType(PetType type) {
        throw new NotImplementedException("Not needed for this stub.");
    }
}

```

Now that we have the stubs, they have to be used. In order to write multiple unit tests and execute them together as a suite, a dependency is needed to make the implementation run more easily. This dependency is JUnit,³ a Java framework to write repeatable unit tests. It provides annotations to prepare and run unit test suites.

The recommended practice is to create a class named the same as the class to be tested but postfixed with Test, so the test class in this case will be named SimplePetServiceTest. Only a few test examples will be depicted here. For more, look in the code attached to the book for this chapter, project 03-ps-test-practice.

```
package com.ps.repo.services;

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
...

public class SimplePetServiceTest {
    public static final Long PET_ID = 1L;
    public static final User owner = buildUser("test@gmail.com", "test",
        UserType.OWNER);

    private StubPetRepo stubPetRepo = new StubPetRepo();

    // object to be tested
    private SimplePetService simplePetService;

    @Before
    public void setUp() {
        // create a few entries to play with
        stubPetRepo.save(build(owner, PetType.CAT, "John", 3, "0122345645"));
        stubPetRepo.save(build(owner, PetType.DOG, "Max", 5, "0222335645"));

        // create object to be tested
        simplePetService = new SimplePetService();
        simplePetService.setRepo(stubPetRepo);
    }

    //positive test, we know that a Pet with ID=1 exists
    @Test
    public void findByIdPositive() {
        Pet pet = simplePetService.findById(PET_ID);
        assertNotNull(pet);
    }
}
```

³JUnit official site: <http://junit.org/junit4/>.

```

//positive test, we know that pets for this owner exist and how many
@Test
public void findByOwnerPositive() {
    Set<Pet> result = simplePetService.findAllByOwner(owner);
    assertEquals(result.size(), 2);
}

//negative test, we know that pets for this owner do not exist
@Test
public void findByOwnerNegative() {
    User newOwner = buildUser("gigi@gmail.com", "test", UserType.OWNER);
    Set<Pet> result = simplePetService.findAllByOwner(newOwner);
    assertEquals(result.size(), 0);
}
...
// simple builder of User instances
private static User buildUser(String email, String password,
    UserType userType){
    ..
    return user;
}

// simple builder of Pet instances
private static Pet build(User owner, PetType petType, String name,
    Integer age, String rfid) {
    ...
    pet.setOwner(owner);
    return pet;
}
}

```

For the delete methods, the negative test looks a little different, since the delete methods do not return a result that you can make an assertion about. Instead, when they cannot delete a record, they throw an exception of type `com.ps.repos.NotFoundException`. Of course, this is a little extreme, but necessary in order to show you how these methods can be tested. Since we know the cases in which the exception is thrown, in order to make the test pass, we tell JUnit when this behavior is expected using the `expected` attribute of the `@Test` annotation.

```

//positive test, we know that a Pet with ID=1 exists
@Test
public void deleteByIdPositive() {
    simplePetService.deleteById(PET_ID);

    // we do a find to test the deletion succeeded
    Pet pet = simplePetService.findById(PET_ID);
    assertNull(pet);
}

//negative test, we know that a Pet with ID=99 does not exist
@Test(expected = NotFoundException.class)
public void deleteByIdNegative() {
    simplePetService.deleteById(99L);
}

```

In the code snippet above, the following JUnit components were used:

- The `@Before` annotation is used on methods that should be executed before executing a test method. These methods are used to prepare the context for that test to execute in. All objects used in the test methods should be created and initialized in this method. Methods annotated with `@Before` are executed before every method annotated with `@Test`.
- The `@Test` annotation is the annotation that tells JUnit that the code in this method should be run as a test case. Methods annotated with `@Test` should always be public and return void. It can also be used to treat expected exceptions.
- The `assert*` methods are defined in the `org.junit.Assert` class and can be used to simplify the code of a test method. Without them, the user would have to write the code that specifies when the test should pass or fail.

After a quick analysis of the code, one thing should be obvious: testing code with stubs is time-consuming, and writing stubs seems to take as much time as development itself. Indeed, testing using stubs is applicable only for really simple applications and scenarios. The worst thing about testing with stubs, though, is that if the interface changes, the stub implementation has to change too. So, not only do you have to adapt the tests, but the stubs too. The second-worst thing is that all methods have to be implemented when stubs are used, even those that are not used by a test scenario. Example:

```
@Override
public Set<Pet> findAllByType(PetType type) {
    throw new NotImplementedException("Not needed for this stub.");
}
```

This method was not involved in any test scenario, and to avoid providing an implementation, the decision was made to throw a `NotImplementedException`. The third-worst thing about stubs is that if we have a hierarchy of stubs, refactoring the one at the base of the hierarchy means refactoring all the stubs based on them, or else tests will fail. And this is the case in our example as well, since all stubs are based on `StubAbstractRepo`.

■ **Conclusion** Stubs make testing seem like a tedious job, so let's see what *mocks* can do to improve the situation.

Testing with Mocks

A mock object is also a pseudo object, replacing the dependency we are not interested in testing and helping to isolate the object in which we are interested. Mock code does not have to be written, because there are a few libraries and frameworks that can be used to generate mock objects. The mock object will implement the dependent interface on the fly. Before a mock object is generated, the developer can configure its behavior: what methods will be called and what will they return. The mock object can be used after that, and then expectations can be checked in order to decide the test result.

There are more libraries and frameworks for mock generation written in Java applications, and which one to use is up to you.

- **EasyMock** provides an easy way to replace collaborators of the unit under test. More about it on their official site: <http://easymock.org/>. This library is very popular and was used extensively in Spring development until version 3.1 made the switch to Mockito.
- **jMock** is a small library, and the team that created it brags about making it quick and easy to define mock objects. The API is very nice and is suitable for complex stateful logic. More about it on their official site: <http://www.jmock.org/>.

- **Mockito** is a mocking framework that provides a really clean and simple API for writing tests. The interesting thing about it is that it provides the possibility of partial mocking; real methods are invoked but still can be verified and stubbed. More about it on their official site: <http://mockito.org/>.
- **PowerMock** is a framework that extends other mock libraries such as EasyMock with more powerful capabilities. It was created in order to provide a way of testing code considered untestable. It uses a custom classloader and bytecode manipulation to enable mocking of static methods, constructors, final classes and methods, private methods, removal of static initializers, and more. You can read more about it here: <https://github.com/jayway/powermock>.

Each of these mocking tools will be covered in this chapter. All of them also provide annotations when used in a Spring context, when doing integration testing. But until Spring testing is covered, we will stick to simple unit testing.

EasyMock

The class to test with mocks generated by EasyMock is `SimpleUserService`. This class inherits all methods from `SimpleAbstractService<T>` and provides its own for user-specific behavior.

```
import static com.ps.util.TestObjectsBuilder.buildUser;
...
public class SimpleUserService extends SimpleAbstractService<User>
    implements UserService {

    private UserRepo repo;

    @Override
    public User createUser(String email, String password, UserType userType) {
        User user = buildUser(email);
        user.setPassword(password);
        user.setUserType(userType);
        repo.save(user);
        return user;
    }

    @Override
    public Set<User> findByName(String name, boolean exact) {
        return repo.findAllByUserName(name, exact);
    }

    // setters & getters
    public void setRepo(UserRepo repo) {
        this.repo = repo;
    }

    @Override
    public UserRepo getRepo() {
        return repo;
    }
}
```

```
// interface for User specific behavior
public interface UserService {
    User createUser(String email, String password, UserType userType);

    /**
     * Method used to search an User by his name
     * @param name name of the user searching for
     * @param exact if the search should be exact (name= :name),
     *             or not ( name like '%name%')
     * @return
     */
    Set<User> findByName(String name, boolean exact);
}

```

The `TestObjectsBuilder` is a java class containing utility methods for creating user and pet objects. Its code is not relevant in this context, but you can inspect its code in the project attached to this chapter.

To perform a test using mocks generated with `EasyMock`, the following steps have to be completed:

1. Declare the mock
2. Create the mock
3. Inject the mock
4. Record what the mock is supposed to do
5. Tell the mock the actual testing that is being done
6. Test
7. Make sure that the methods were called on the mock
8. Validate the execution

The test class will be named `SimpleUserServiceTest`, and in the following test snippet, the `findByName` method is tested, and the above-listed steps are underlined.

```
package com.ps.repo.services;

import org.junit.Before;
import org.junit.Test;
...
import static org.easymock.EasyMock.*;
import static org.junit.Assert.*;

public class SimpleUserServiceTest {

    (1)private UserRepo userMockRepo;
    private SimpleUserService simpleUserService;

    @Before
    public void setUp() {
        (2)userMockRepo = createMock(UserRepo.class);
    }
}

```

```

//create object to be tested
simpleUserService = new SimpleUserService();

(3)simpleUserService.setRepo(userMockRepo);
}

@Test
public void findByNamePositive() {
    //record what we want the mock to do
    User simpleUser = buildUser("gigi@gmail.com", "the_password", UserType.OWNER);
    Set<User> userSet = new HashSet<>();
    userSet.add(simpleUser);

    (4)expect(userMockRepo.findAllByUserName("gigi", false)).andReturn(userSet);
    (5)replay(userMockRepo);

    (6)Set<User> result = simpleUserService.findByName("gigi", false);
    (7)verify(userMockRepo);
    (8) assertEquals(result.size(), 1);
}
}

```

The EasyMock library provides static methods to process the mock, and when multiple mocks are needed, `replay` and `verify` methods are replaced with `replyAll` and `verifyAll`, and the mocks are picked up and processed without direct reference to any of them. The main advantage of using mocks is that there is no need to maintain any extra classes, because when using mocks, the behavior needed from the dependency is defined on the spot, inside the test method body, and the generating library takes care of mimicking the behavior. Young and inexperienced developers might have difficulty in understanding how mocking works, but if you are their mentor, just ask them to create stubs first and then switch them to mocks. They will understand more easily and will be enchanted by the possibility of not needing to write too much extra code in order to test an object.

jMock

The class to test with mocks generated by jMock is `SimpleRequestService`. This class inherits all methods from `SimpleAbstractService<T>` and provides its own for requesting specific behavior.

```

public class SimpleRequestService extends SimpleAbstractService<Request>
    implements RequestService {

    private RequestRepo repo;

    public Request createRequest(User user, Pet pet,
        Pair<DateTime, DateTime> interval, String details) {
        Request request = new Request();
        ...
        repo.save(request);
        return request;
    }
}

```



```

@Override
public Set<Request> findAllByUser(User user) {
    return repo.findAllForUser(user);
}

// setters & getters
public void setRepo(RequestRepo requestRepo) {
    this.repo = requestRepo;
}

@Override
public RequestRepo getRepo() {
    return repo;
}
}

// interface for Request specific behavior
public interface RequestService {

    Request createRequest(User user, Pet pet,
        Pair<DateTime, DateTime> interval, String details);

    Set<Request> findAllByUser(User user);
}

```

The Pair class is a java class containing two fields of generic types. It is a utility class, and its code is not relevant in this context, but you can inspect its code in the project attached to this chapter.

To perform a test using mocks generated with jMock, the following steps have to be completed:

1. Declare the mock
2. Declare and define the context of the object under test, an instance of the `org.jmock.Mockery` class
3. Create the mock
4. Inject the mock
5. Define the expectations we have from the mock
6. Test
7. Check that the mock was actually used
8. Validate the execution

The test class will be named `SimpleRequestServiceTest`, and in the following test snippet, the `findAllByUser` method is tested and the above listed steps are underlined.

```

package com.ps.repo.services;

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.joda.time.DateTime;
import org.junit.Before;
import org.junit.Test;
...
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

public class SimpleRequestServiceTest {

    (1)private RequestRepo requestMockRepo;

    (2)private Mockery mockery = new Mockery();

    private SimpleRequestService simpleRequestService;

    @Before
    public void setUp() {
        (3)requestMockRepo = mockery.mock(RequestRepo.class);

        simpleRequestService = new SimpleRequestService();
        (4)simpleRequestService.setRepo(requestMockRepo);
    }

    @Test
    public void findByUserPositive() {
        User user = buildUser("gigi@gmail.com", "the_password", UserType.OWNER);
        Request req = new Request();
        req.setUser(user);
        req.setStartAt(DateTime.parse("2016-09-06").toDate());
        req.setEndAt(DateTime.parse("2016-09-18").toDate());
        req.setRequestStatus(RequestStatus.NEW);
        Set<Request> expectedResult= new HashSet<>();
        expectedResult.add(req);
        (5)
        mockery.checking(new Expectations() {{
            allowing(requestMockRepo).findAllForUser(user);
            will(returnValue(expectedResult));
        }});
        (6)Set<Request> result = simpleRequestService.findAllByUser(user);
        (7)mockery.assertIsSatisfied();
        (8)assertEquals(result.size(), 1);
    }
}

```

The only observation needed here is that when multiple mocks are used, or multiple operations are executed by the same mock, defining the expectations can become a bit cumbersome. Still, it is easier than defining stubs.

Mockito

Mockito has the advantage of mocking behavior by writing code that is readable and very intuitive. The collection of methods provided was designed so well that even somebody without extensive programming knowledge can understand what is happening in that code, as long as that person also understands English. The class that will be tested with Mockito is `SimpleReviewService`. This class inherits all methods from `SimpleAbstractService<T>` and provides its own for request specific behavior.

```
public class SimpleReviewService extends SimpleAbstractService<Review>
    implements ReviewService {

    private ReviewRepo repo;

    @Override
    public Review createReview(ReviewGrade grade, String details) {
        Review review = new Review();
        review.setGrade(grade);
        review.setDetails(details);
        repo.save(review);
        return review;
    }

    @Override
    public Set<Review> findAllByUser(User user) {
        return repo.findAllForUser(user);
    }

    // setters & getters
    public void setRepo(ReviewRepo reviewRepo) {
        this.repo = reviewRepo;
    }

    @Override
    public ReviewRepo getRepo() {
        return repo;
    }
}

// interface for Review specific behavior
public interface ReviewService {
    Review createReview(ReviewGrade grade, String details);

    Set<Review> findAllByUser(User user);
}
```

To perform a test using mocks generated with Mockito, the following steps have to be completed:

1. Declare and create the mock
2. Inject the mock
3. Define the behavior of the mock
4. Test
5. Validate the execution

The test class will be named `SimpleReviewServiceTest`, and in the following test snippet, the `findAllByUser` method is tested and the above listed steps are underlined.

```
package com.ps.repo.services;

import org.junit.Before;
import org.junit.Test;

import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
...

public class SimpleReviewServiceTest {

    (1)private ReviewRepo reviewMockRepo = mock(ReviewRepo.class);

    private SimpleReviewService simpleReviewService;

    @Before
    public void setUp(){
        simpleReviewService = new SimpleReviewService();
        (2)simpleReviewService.setRepo(reviewMockRepo);
    }

    @Test
    public void findByUserPositive() {
        User user = buildUser("gigi@gmail.com", "the_password", UserType.OWNER);
        Request req = new Request();
        req.setUser(user);
        Review review = new Review();
        review.setRequest(req);

        Set<Review> reviewSet = new HashSet<>();
        reviewSet.add(review);

        (3) when(reviewMockRepo.findAllForUser(user)).thenReturn(reviewSet);
        (4) Set<Review> result = simpleReviewService.findAllByUser(user);
        (5) assertEquals(result.size(), 1);
    }
}
```

The `org.mockito.Mockito` class provides static methods for creation of the mock object and defining its behavior. Defining the behavior of the mock is so intuitive that looking at the line marked with (3), you can directly figure out how the mock works: when the `findAllForUser` method is called on it with argument `user`, it will return the `reviewSet` object defined previously. When multiple mocks are used or multiple methods of the same mock are called, then more when statements must be written.

When a mock method is being called multiple times, Mockito also has the possibility to check how many times the method was called with a certain argument using a combination of `verify` and `times` methods.

```
when(reviewMockRepo.findAllForUser(user)).thenReturn(reviewSet);
Set<Review> result = simpleReviewService.findAllByUser(user);
verify(reviewMockRepo, times(1)).findAllForUser(user);
assertEquals(result.size(), 1);
```

Quite practical, right? Probably this is the reason why the Spring team switched from EasyMock to Mockito. And there is more, because when using annotations, everything becomes even more practical.

```
package com.ps.repo.services;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Mockito;
import org.mockito.MockitoAnnotations;

import static com.ps.util.TestObjectsBuilder.buildUser;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

/**
 * Created by iuliana.cosmina on 4/17/16.
 */
public class MockPetServiceTest {

    public static final Long PET_ID = 1L;
    public static final User owner = buildUser("test@gmail.com",
        "the_password", UserType.OWNER);

    @InjectMocks
    SimplePetService simplePetService;

    //Creates mock instance of the field it annotates
    @Mock
    PetRepo petRepo;

    @Before
    public void initMocks() {
        MockitoAnnotations.initMocks(this);
    }
}
```

```

//positive test, we know that pets for this owner exist and how many
@Test
public void findByOwnerPositive() {
    Set<Pet> sample = new HashSet<>();
    sample.add(new Pet());
    Mockito.when(petRepo.findAllByOwner(owner)).thenReturn(sample);
    Set<Pet> result = simplePetService.findAllByOwner(owner);
    assertEquals(result.size(), 1);
}
}

```

The `InjectMock` has a behavior similar to the Spring IoC, because its role is to instantiate testing object instances and to try to inject fields annotated with `@Mock` or `@Spy` into private fields of the testing object.

The `MockitoAnnotations.initMocks(this);` call that initializes the mocks is not needed when the test class is annotated with `@RunWith(MockitoJUnitRunner.class)` and the `Mockito JUnit` runner class is used to execute the tests that will take care of the mock objects too.

Also, `Mockito` provides matchers that can be used to replace any variables needed for mocking environment preparation. These matchers are static methods in the `org.mockito.Mockito` class and can be called to replace any argument with a pseudo-value of the required type. For common types, the method names are prefixed with `any`, (`anyString()`, `anyLong()`, and others), while for every other object type, `any(Class<T>)` can be used. So the line

```
Mockito.when(petRepo.findAllByOwner(owner)).thenReturn(sample);
```

can be written using a matcher, and no `owner` variable is needed.

```
Mockito.when(petRepo.findAllByOwner(any(User.class))).thenReturn(sample);
```

PowerMock

`PowerMock` was born because sometimes code is not testable, perhaps because of bad design or because of some necessity. Below you can find a list of untestable elements:

- static methods
- classes with static initializers
- final classes and final methods; sometimes there is need for an insurance that the code will not be misused or to make sure that an object is constructed correctly
- private methods and fields

`PowerMock` is not that useful in a Spring application, since you will rarely find static elements there, but there is always legacy code and third-party libraries that might require mocking, so it is only suitable to know that it is possible to do so and the tool to use for this. If you want to know more, take a look on their official site: <https://github.com/jayway/powermock>

When it comes to testing applications, the technique and tools to use are defined by the complexity of the application, the experience of the developer, and ultimately legal limitations, because there are companies that are restricted to using software under a certain license. During a development career, you will probably get to use all the techniques and libraries mentioned in this book. Favor mocks for nontrivial dependencies and nontrivial interfaces. Favor stubs when the interfaces are simple with repetitive behavior, but also because stubs can log and record their usage.

That being said, you can switch over to the next section, which will show you how to use all these things to test a Spring application.

3.1 Testing with Spring

Spring provides a module called `spring-test` that contains Spring JUnit test support classes that can be used to make testing Spring applications a manageable operation. The core class of this module is `org.springframework.test.context.junit4.SpringJUnit4ClassRunner`, which is used to cache an `ApplicationContext` across test methods. All the tests are run in the same context, using the same dependencies; thus this is integration testing.

In order to define a test class for running in a Spring context, the following have to be done:

1. annotate the test class with `@RunWith(SpringJUnit4ClassRunner.class)`
2. annotate the class with `@ContextConfiguration` in order to tell the runner class where the bean definitions come from

```
// bean definitions are provided by class AllRepoConfig
@ContextConfiguration(classes = {AllRepoConfig.class})
public class GenericQualifierTest {...}
// bean definitions are loaded from file all-config.xml
@ContextConfiguration(locations = {"classpath:spring/all-config.xml"})
public class GenericQualifierTest {...}
```

■ **CC** If `@ContextConfiguration` is used without any attributes defined, the default behavior of Spring is to search for a file named `{testClassName}-context.xml` in the same location as the test class and load bean definitions from there if found.

3. use `@Autowired` to inject beans to be tested.

This method of testing was already introduced to test Java Configuration-based applications in the previous chapter, but in this chapter you will find out how you can manipulate the configuration so that tests can be run in a test context.

The following code snippet tests the class `SimplePetService` in a Spring context defined by two Spring configuration XML files. The test uses a stub implementation for `PetRepo`, declared as a bean and injected in `SimplePetService` using configuration. In Figure 3-3, the classes and files involved in defining the test context and running the test are depicted.

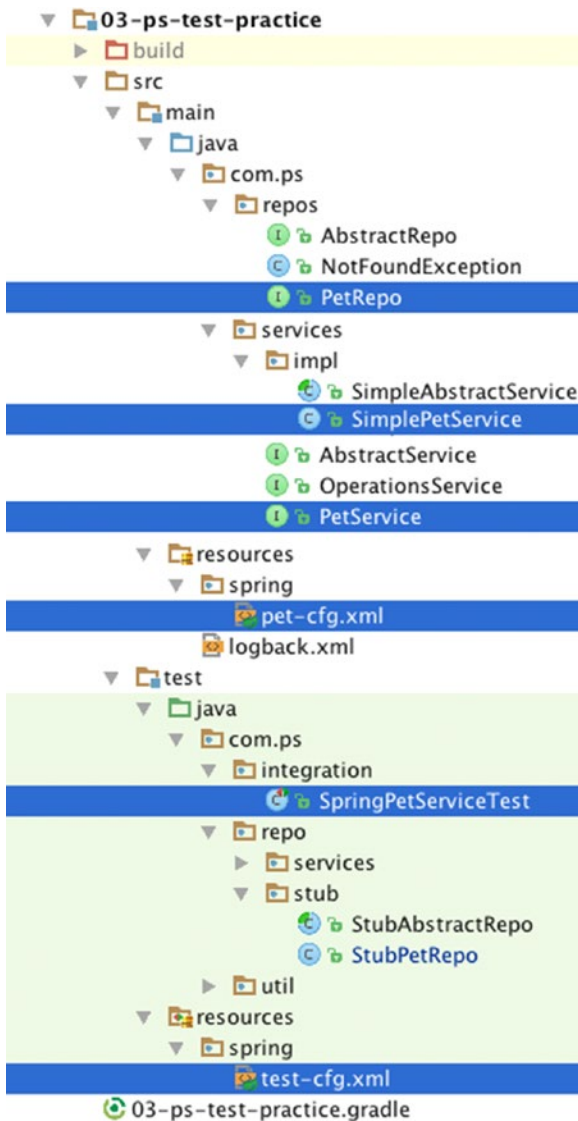


Figure 3-3. Spring test context for testing *SimplePetService*

The `pet-cfg.xml` file contains the definition for a single bean of type `SimplePetService`, the bean we are interested in testing.

```
//pet-cfg.xml
<beans ...>

  <bean name="simplePetService" class="com.ps.services.impl.SimplePetService">
    <property name="repo" ref="petRepo" />
  </bean>

</beans>
```


The `petRepo` is not defined in this file. It is decoupled from the `simplePetService` bean via configuration. It can be provided in a different file depending on the execution context. In this case, we are using a test context, and the dependency for executing this test is provided by the file `test-cfg.xml`.

```
//test-cfg.xml
<beans ...>

    <bean name="petRepo" class="com.ps.repo.stub.StubPetRepo"
        init-method="init"/>

```

For the stub to match the Spring environment, it was modified a little, and an `init` method was added in order to initialize the contents managed by the repo. The `TestObjectsBuilder` utility class also contains a `buildPet` method, used to create a pet object. The method body is also not relevant to the context, but can be inspected in the project attached to this chapter. To make things easier, methods building test objects were moved to a different class called `TestObjectsBuilder`.

```
package com.ps.repo.stub;
...
import static com.ps.util.TestObjectsBuilder.buildPet;
import static com.ps.util.TestObjectsBuilder.buildUser;

public class StubPetRepo extends StubAbstractRepo<Pet>
    implements PetRepo {

    protected Map<User, Set<Pet>> records2 = new HashMap<>();

    public void init() {
        // create a few entries to play with
        final User owner = buildUser("test@gmail.com", "the_password", UserType.OWNER);
        this.save(buildPet(owner, PetType.CAT, "John", 3, "0122345645"));
        this.save(buildPet(owner, PetType.DOG, "Max", 5, "0222335645"));
    }

    @Override
    public void save(Pet pet) {
        super.save(pet);
        addWithOwner(pet);
    }
    ...
}
```

Even if we are using a stub, the following test is an integration test. All the test methods will run in the same Spring test context. The object to be tested and its dependencies are created only once, when the application context is created, and they are used by all methods. The object to be tested is a bean, and it is injected in the class testing it using `@Autowired`.

```

package com.ps.integration;

import org.junit.Test;
import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import static com.ps.util.TestObjectsBuilder.buildUser;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations= {
    "classpath:spring/test-cfg.xml",
    "classpath:spring/pet-cfg.xml"} )
public class SpringPetServiceTest{

    public static final Long PET_ID = 1L;
    public static final User owner =
        buildUser("test@gmail.com", "the_password", UserType.OWNER);

    @Autowired
    PetService simplePetService;

    //positive test, we know that a Pet with ID=1 exists
    @Test
    public void findByIdPositive() {
        Pet pet = simplePetService.findById(PET_ID);
        assertNotNull(pet);
    }

    //positive test, we know that pets for this owner exist and how many
    @Test
    public void findByOwnerPositive() {
        Set<Pet> result = simplePetService.findAllByOwner(owner);
        assertEquals(result.size(), 2);
    }
}

```

The Spring test context is created using the two configuration files, which will make sure the stub dependency is the only one available to be injected into the `simplePetService` bean. As you can see, the `@Before` annotated method is no longer necessary. Also, there is no longer any need to manipulate the `petRepo` object, since it is created, initialized, and injected by Spring. So all that is left for the developer to do is to inject the bean being tested and jump right to writing tests.

Something similar can be done when configuration is provided using Java Configuration, because the `@Bean` annotation can be used to declare attributes. In the following code snippet the XML files were replaced by configuration classes.

```
// replaces test-cfg.xml
@Configuration
public class TestAppConfig2 {

    @Bean(initMethod = "init")
    public PetRepo petRepo(){
        return new StubPetRepo();
    }
}

// replaces pet-cfg.xml
public class PetConfigClass2 {

    @Bean
    public PetService simplePetService(){
        return new SimplePetService();
    }
}
}
```

The test class stays the same, only the attribute locations is replaced with classes in the `@ContextConfiguration` annotation, because the configuration is now provided using Java Configuration classes.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestAppConfig2.class,
    PetConfigClass2.class})
// previous declaration replaces @ContextConfiguration(locations= {
//     "classpath:spring/test-cfg.xml",
//     "classpath:spring/pet-cfg.xml" } )
public class SpringPetServiceTest2
    ...
}
```

Stereotype annotations can be added too. The `StubPetRepo` class must be defined as a bean, and the `init` method must be annotated with `@PostConstruct`. The `SimplePetService` class must also be defined as a bean, and the repo dependency must be autowired.

```

//StubPetRepo.java
@Component
public class StubPetRepo extends StubAbstractRepo<Pet> implements PetRepo {

    @PostConstruct
    public void init(){
        // create a few entries to play with
        final User owner = buildUser("test@gmail.com", "the_password", UserType.OWNER);
        this.save(buildPet(owner, PetType.CAT, "John", 3, "0122345645"));
        this.save(buildPet(owner, PetType.DOG, "Max", 5, "0222335645"));
    }
    ...
}

//SimplePetService.java
@Component
public class SimplePetService extends SimpleAbstractService<Pet>
    implements PetService {
    ...

    @Autowired
    public void setRepo(PetRepo petRepo) {
        this.repo = petRepo;
    }
}

```

The configuration classes must have scanning for components enabled and set to use exactly the packages we are interested in:

```

//PetConfigClass.java
@Configuration
@ComponentScan(basePackages = "com.ps.services.impl")
public class PetConfigClass {
}

//TestAppConfig.java
@Configuration
@ComponentScan(basePackages = "com.ps.repo.stub")
public class TestAppConfig {
}

```

The only thing that changes in the test class is the configuration for the `@ContextConfiguration` annotation; the autowiring of the tested components and the tests remain exactly the same.

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestAppConfig.class, PetConfigClass.class})
public class SpringPetServiceTest2 {
    ...
}

```

In Spring 3.1, the `org.springframework.test.context.support.AnnotationConfigContextLoader` class was introduced. This class loads bean definitions from annotated classes. So a configuration class can be created directly in the test class to create the beans needed to be tested in isolation. This is useful because configuration classes usually contain more than one bean definition, and loading them all just to test one is, well, inefficient. The class must be internal to the test class and static, and the `AnnotationConfigContextLoader` class must be used as a value for the loader attribute of the `@ContextConfiguration` annotation.

Please look at the following code snippet:

```
package com.ps.integration;
import
    org.springframework.test.context.support.AnnotationConfigContextLoader;
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(loader=AnnotationConfigContextLoader.class)
public class SpringPetServiceTest3 {

    public static final Long PET_ID = 1L;
    public static final User owner = buildUser("test@gmail.com", "test",
        UserType.OWNER);

    @Configuration
    static class TestCtxConfig {

        @Bean
        StubPetRepo petRepo(){
            return new StubPetRepo();
        }

        @Bean
        PetService simplePetService(){
            SimplePetService petService = new SimplePetService();
            petService.setRepo(petRepo());
            return petService;
        }
    }

    @Autowired
    PetService simplePetService;

    @Test
    public void findByOwnerPositive() {
        Set<Pet> result = simplePetService.findAllByOwner(owner);
        assertEquals(result.size(), 2);
    }
}
```

Also, Spring Test can be combined with mocking to write integration tests that skip heavy components, like the ones providing access to databases. To be able to cover this specific scenario, concrete implementations for the repository classes must be used. Since their implementation is irrelevant to this chapter and will be covered later in Chapter 5, Data Access, the full code will not be presented here. In order to manage Pet objects, a Repository class is needed. The most basic way to provide access to a database in Spring applications is to use a bean of type `org.springframework.jdbc.core.JdbcTemplate`, so repository

classes will be created with a dependency of that type. The methods querying the database from this bean require an object that can map database records to entities, which is why each repository class should define an internal class implementing the Spring-specific mapping interface `org.springframework.jdbc.core.RowMapper<T>`.

```
//abstract repository base class
package com.ps.repos.impl;
import org.springframework.jdbc.core.JdbcTemplate;
...
public class JdbcAbstractRepo<T> extends AbstractEntity<T>
    implements AbstractRepo<T> {

    protected JdbcTemplate jdbcTemplate;

    public JdbcAbstractRepo(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    ...
}

@Repository("petRepo")
public class JdbcPetRepo extends JdbcAbstractRepo<Pet> implements PetRepo {
    private String findByIdQuery = "select id, name from pet";
    private RowMapper<Pet> rowMapper = new PetRowMapper();

    @Autowired
    public JdbcPetRepo(JdbcTemplate jdbcTemplate) {
        super(jdbcTemplate);
    }
    ...
    @Override
    public Pet findById(Long entityId) {
        return jdbcTemplate.queryForObject(findByIdQuery, rowMapper, entityId);
    }
    ...
    // the DB record to entity mapper class
    private class PetRowMapper implements RowMapper<Pet> {
        public Pet mapRow(ResultSet rs, int rowNum) throws SQLException {
            Long id = rs.getLong("ID");
            String name = rs.getString("NAME");
            Pet pet = new Pet();
            pet.setId(id);
            pet.setName(name);
            return pet;
        }
    }
}
```

A configuration class that provides a mock that replaces the `JdbcTemplate` bean is needed. The mock will be created by a static method in the `org.mockito.Mockito` class called `mock(Class<T>)`. This configuration class will be used together with the configuration classes for repositories and services to create a test context.

```
// the mock configuration class
import static org.mockito.Mockito.mock;
...
@Configuration
public class MockTemplateConfig {

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return mock(JdbcTemplate.class);
    }
}

//repositories configuration class
@Configuration
@ComponentScan(basePackages = "com.ps.repos.impl")
public class JdbcRepoConfig {
}

//services configuration class
@Configuration
@ComponentScan(basePackages = "com.ps.services.impl")
public class ServiceConfig {
}
```

Now the test class can be created; it has to respect the following two rules:

- must be executed with `MockitoJUnitRunner` runner class
- two Spring-specific components must be added to the test class so the context can be loaded

```
@ClassRule
public static final SpringClassRule SPRING_CLASS_RULE = new SpringClassRule();

@Rule
public final SpringMethodRule springMethodRule = new SpringMethodRule();
```

`org.springframework.test.context.junit4.rules.SpringClassRule` is an implementation of `JUnit org.junit.rules.TestRule` that supports class-level features of the Spring TestContext Framework. The `@ClassRule` annotation is used on fields that reference rules or methods that return them. Fields must be public, static and a subtype of `org.junit.rules.TestRule`. Methods must be public, static and return an implementation of `TestRule`. The `org.springframework.test.context.junit4.rules.SpringMethodRule` is an implementation of `JUnit org.junit.rules.MethodRule` that supports instance-level and method-level features of the Spring TestContext Framework. The `@Rule` annotation is used on fields that reference rules or methods that return them. Fields must be public and not static and an implementation of `TestRule` or `org.junit.rules.MethodRule`; methods must be public and not static and return an implementation of `TestRule` or `MethodRule`.

The final test class is depicted in the following code snippet:

```
package com.ps.config;

import com.ps.ents.Pet;
import com.ps.mock.MockTemplateConfig;
import com.ps.services.impl.SimplePetService;
import org.junit.ClassRule;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mockito;
import org.mockito.runners.MockitoJUnitRunner;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.rules.SpringClassRule;
import org.springframework.test.context.junit4.rules.SpringMethodRule;

import static org.junit.Assert.assertNotNull;
import static org.mockito.Mockito.any;
import static org.mockito.Mockito.anyString;
import static org.mockito.Mockito.anyLong;

@RunWith(MockitoJUnitRunner.class)
@ContextConfiguration(classes = {JdbcRepoConfig.class,
    ServiceConfig.class, MockTemplateConfig.class})
public class TestAppConfig3 {

    public static final Long PET_ID = 1L;

    // mocking the database
    @Autowired
    JdbcTemplate jdbcTemplate;

    // tested object
    @Autowired
    SimplePetService simplePetService;

    @ClassRule
    public static final SpringClassRule SPRING_CLASS_RULE = new SpringClassRule();

    @Rule
    public final SpringMethodRule springMethodRule = new SpringMethodRule();

    //positive test, we know that a Pet with ID=1 exists
    @Test
    public void findByIdPositive() {
        Mockito.when(jdbcTemplate.queryForObject(anyString(),
            any(RowMapper.class), anyLong())).thenReturn(new Pet());
    }
}
```



```

        Pet pet = simplePetService.findById(PET_ID);
        assertNotNull(pet);
    }
}

```

If you haven't figured out yet what happened, here is the explanation: A test context was created containing all the beans in the configuration classes except the `JdbcTemplate` bean. This bean was replaced with a mock, which was defined as a bean in a test configuration class and that was injected automatically by Spring where needed (in the repository bean).

Using Profiles

Starting with Spring 3.1, the `@Profile` annotation became available. When classes are annotated with this annotation, they become eligible for registration when one or more profiles are active. Spring profiles have the same semantics as Maven profiles, but they are more practical to use, because they are much easier to configure and are not the builder tool of an application. It was mentioned before that different environments require different configurations, and much care should be used during development so that components are decoupled enough so they can be swapped depending on the context in which processes are executed. Spring profiles help considerably in this case. For example, during development, tests are run on development machines, and a database is not really needed, or if one is needed, an in-memory simple and fast implementation should be used. This can be set up by creating a test datasource configuration file that will be used only when the development profile is active. The two datasource classes, for production and test environment, are depicted in the code snippet below:

```

//production dataSource
package com.ps.config;
import org.springframework.context.annotation.Profile;
...
@Configuration
@PropertySource("classpath:db/datasource.properties")
@Profile("prod")
public class ProdDataConfig {

    @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
public DataSource underlineddataSource() throws SQLException {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(driverClassName);
    ds.setUrl(url);
    ds.setUsername(username);
    ds.setPassword(password);
    return ds;
}
}

// development dataSource
@Configuration
@Profile("dev")
public class TestDataConfig {

    @Bean
    public DataSource datasource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript("classpath:db/schema.sql")
            .addScript("classpath:db/test-data.sql")
            .build();
    }
}

```

In the sample above we have two configuration classes, each declaring a bean named `dataSource`, each bean specific to a different environment. The profiles are named simply *prod*, for the production environment, and *dev*, for the development environment. In the test class we can activate the development profile by annotating the test class with `@ActiveProfiles` annotation and giving the profile name as argument. Thus, in the test context only the beans defined in classes annotated with `@Profile("dev")` will be created and injected. The test class is depicted in the following code snippet.

```

import org.springframework.test.context.ActiveProfiles;
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {PetConfigClass.class})
@ActiveProfiles("dev")
public class PetServiceTest {

    public static final Long PET_ID = 1L;

    @Autowired
    PetService simplePetService;

    //positive test, we know that a Pet with ID=1 exists
    @Test
    public void findByIdPositive() {
        Pet pet = simplePetService.findById(PET_ID);
        assertNotNull(pet);
    }
}

```

By annotating the test class `@ActiveProfiles("dev")`, the development profile is activated, so when the context is created, bean definitions are picked up from the configuration class (or files) specified by the `@ContextConfiguration` annotation and all the configuration classes annotated with `@Profile("dev")`. Thus, when running the test, the in-memory database will be used, making the execution fast and practical.

■ **Conclusion** Using the Spring-provided test classes to test a Spring application is definitely easier than not doing it, since no external container is needed to define the context in which the tests run. If the configuration is decoupled enough, pieces of it can be shared between the test and production environment. For basic unit testing, Spring is not needed, but in order to implement proper integration testing, the ability to set up a test context in record time is surely useful.

Summary

After reading this chapter you should possess enough knowledge to test a Spring application using unit and integration testing.

- Testing is an important part of the development process.
- What is unit testing, and what library is useful for writing unit tests in Java?
- What is a stub?
- What is a mock?
- What is integration testing?
- How does one set up a Spring Test Context?

Quick Quiz

Question 1: Given the following integration test:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class SimpleTest {
    //test methods here
}
```

What can be said about the behavior of Spring when the test is executed? (choose all that apply)

- A. Spring will build an empty context and the tests will fail
- B. Spring will not be able to create a test context for the tests to run in
- C. Spring will look for a file named `SimpleTest-context.xml` in the same location as the `SimpleTest` class to use it to initialize the context

Question 2: Given the following unit test, what is missing from the class definition that prevents the test from being executed correctly?

```
public class SimplePetServiceTest {

    @InjectMocks
    SimplePetService simplePetService;

    PetRepo petRepo;

    @Before
    public void initMocks() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void findById() {
        Mockito.when(petRepo.findById(1L)).thenReturn(new Pet());
        Pet pet = simplePetService.findById(1L);
        assertNotNull(pet);
    }
}
```

- A. The class should be annotated with `@RunWith(MockitoRunner.class)`
- B. The `setUp` method is missing the following statement:
`simplePetService.setRepo(petRepo);`
- C. Nothing, the test will be executed correctly and it will pass.
- D. The `petRepo` field is missing annotation `@Mock`

Question 3: The class `SpringJUnit4ClassRunner` is used to set up a test context.

- A. true
- B. false

Question 4: What is the `@ContextConfiguration` used for?

- A. to load and configure a `TestApplicationContext` instance
- B. to load and configure an `ApplicationContext` for integration testing
- C. to inject beans used in unit testing

Question 5: What library is mandatory for writing unit tests for a Spring application?

- A. JUnit
- B. spring-test
- C. any mock generating library such as `jMock`, `Mockito`, or `EasyMock`

Practical Exercise

The project to use to test your understanding of testing is 03-ps-test-practice. This project contains part of the implementation depicted in the code snippets. The parts missing are marked with a TODO task and are visible in IntelliJ IDEA in the TODO view. There are six tasks for you to solve in order to test your acquired knowledge of testing, and they are focused on Mockito usage and Spring testing.

Tasks TODO 14 and 15 require you to complete two unit tests that test a `SimplePetService` object using a stub.

Task TODO 16 requires you to place the missing annotations in the `MockPetServiceTest`.

Task TODO 17 requires you to define the behavior of the mock object using Mockito methods.

Tasks TODO 18 and 19 require you to complete the test class definitions in order for the test cases to be executed correctly. In order to run a test case, just click anywhere on the class content or on the class name in the project view and select the Run `{TestClassName}` option. If you want to run a single test, just right click, and from the menu select Run `{TestMethodName}`. The options are depicted in Figure 3-4 for a test class and a test method in the project. You can even run the tests using debug in case you are interested in stopping the execution at specific times. After implementing all the solutions, you can run the Gradle test task to run all your tests. You should see a green toolbar, next to it a message telling you that all twenty tests passed, similar to what you see depicted in Figure 3-5.

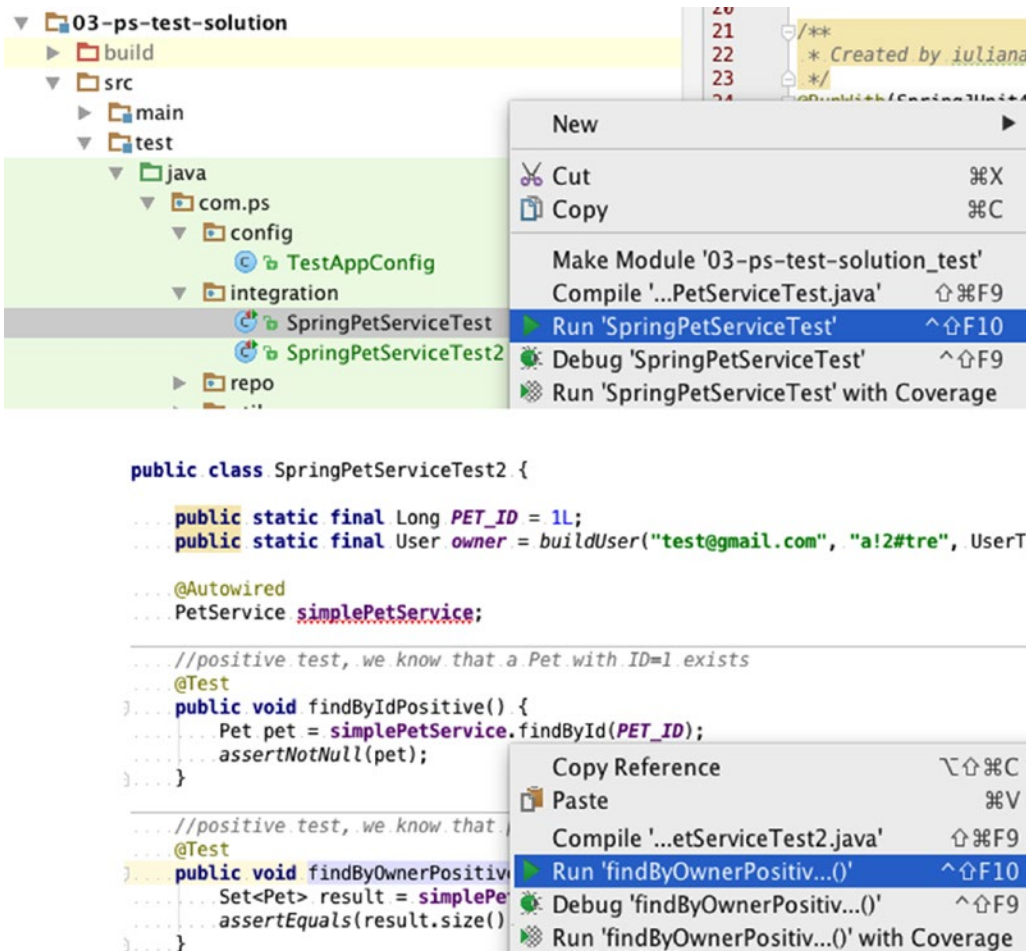


Figure 3-4. How to run tests in IntelliJ IDEA

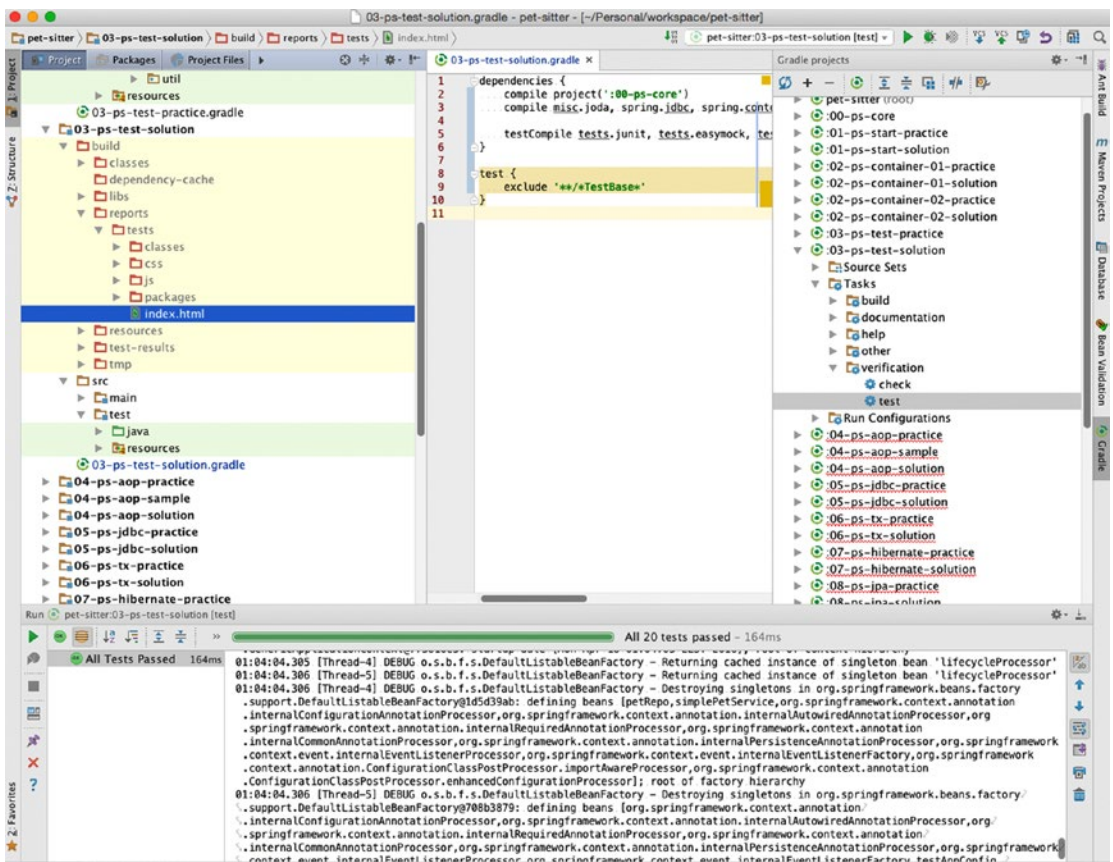


Figure 3-5. Running all tests for a module in IntelliJ IDEA, using the Gradle test task

In the Figure 3-5, the Gradle task appears on the right, in the Gradle view selected with gray. For any module it can be found under `ModuleName/Tasks/verification`.

On the left, the `index.html` file generated when Gradle tests are run is selected. You can open that file in the browser when writing the test in order to check percentages of the tests passing and detailed logs telling you why tests fail. If all is well at the end, you should see a page looking like the one depicted in Figure 3-6.

Test Summary



Generated by [Gradle 2.11](#) at Apr 18, 2016 1:04:04 AM

Figure 3-6. Gradle web report after all tests have passed

The project 03-ps-test-solution contains the proposed solution for this chapter. Please compare yours with the provided one to make sure all was implemented according to the specifications.

Also, because the project has grown in size, you can do a run with the `-Drecept` option and analyze its stability and execution time. Also, in order to build modules in parallel, you can add the `-Dorg.gradle.parallel=true` option.

```
gradle build -Drecept -Dorg.gradle.parallel=true
```

If there are `*-practice` projects with unimplemented tests, the build will unfortunately fail, and when analyzing your build in the browser, you will see something similar to what is depicted in Figure 3-7. But if the module you have just worked on is not among the ones listed, then all is ok.

BETA Gradle

Summary
Failure
Timing
Tasks
Switches
Infrastructure

luliana.cosmina

x pet-sitter build

Ran for 7 sec
Started just now
Gradle 2.11

3 task failures

The :01-ps-start-practice:test task failed.
There were failing tests. See the report at:
file:///Users/luliana.grajdeanu/Personal/workspace/pet-sitter/01-ps-start-practice/build/reports/tests/index.html

The :03-ps-test-practice:test task failed.
There were failing tests. See the report at: file:///Users/luliana.grajdeanu/Personal/workspace/pet-sitter/03-ps-test-practice/build/reports/tests/index.html

See failure details

7 sec

+03:00 Started on **Apr 18, 2016 at 1:44:55 AM** Finished on **Apr 18, 2016 at 1:45:02 AM**
EEST Started on **Apr 18, 2016 at 1:44:55 AM** Finished on **Apr 18, 2016 at 1:45:02 AM**

:02-ps-container-02-practice:test	5278 ms	32%
:02-ps-container-01-practice:test FAILED	3990 ms	24%
:03-ps-test-practice:test FAILED	2887 ms	18%
:01-ps-start-practice:test FAILED	2305 ms	14%
:00-ps-core:compileJava UP-TO-DATE	209 ms	1%
:02-ps-container-01-solution:compileJava UP-TO-DATE	84 ms	

See all tasks

Parallel ⓘ	On
Daemon ⓘ	On
Refresh dependencies ⓘ	Off
Re-run tasks ⓘ	Off
Continuous ⓘ	Off

Figure 3-7. Official Gradle report about the pet-sitter project build

CHAPTER 4



Aspect Oriented Programming with Spring

AOP is an acronym for **Aspect Oriented Programming**, a term that refers to a type of programming that aims to increase modularity by allowing the separation of cross-cutting concerns. A cross-cutting concern is a functionality that is tangled with business code, which usually cannot be separated from the business logic. Auditing, security, and transaction management are good examples of cross-cutting concerns. They are mingled with the business code, heavily coupled with the functionality that might be affected if they fail. These are good candidates for separation using *aspects*, because there is no design pattern that would allow writing the code in such a way that they would be separated from the business logic. This means that additional behavior is added to existing behavior when the application is compiled. So transaction management, security logic, and auditing can be developed separately and mingled with the functionality at compile time. This is done by defining an *advice* containing code that will be executed in a location named *join point* specified by a *pointcut*. This approach allows for code implementing behavior that is not related to business logic to be separated from functional code, the result being a more modular, less coupled, and less cluttered application.

The business or base code is not actually changed; you can imagine aspects as plugins. They modify the behavior, not the actual implementation.

AOP is a type of programming that aims to help with separation of cross-cutting concerns to increase modularity; it implies declaring an aspect class that will alter the behavior of base code, by applying advices to specific join points, specified by pointcuts.

AOP is a complement of OOP (**Object Oriented Programming**) and they can be used together to write powerful applications, because both provide different ways of structuring your code. OOP is focused on making everything an object, while AOP introduces the *aspect*, which is a special type of object that injects and wraps its behavior to complement the behavior of other objects. Other examples of cross-cutting concerns:

- Caching
- Internationalization
- Error detection and correction
- Memory management
- Performance monitoring
- Synchronization

Problems Solved by AOP

When databases are used to store data, a connection to the database is used to interact with the application. The connection to database needs to be opened before the communication and closed afterwards, to successfully complete the communication with the database. Every database implementation will allow a limited number of connections simultaneously; thus connections that are no longer used need to be closed, so that others can be opened. Using Spring, a JDBC a repository method that looks for a user based on its ID looks similar to the implementation in Figure 4-1:

```
public User findById(Long id) {
    String sql = "select u.ID as ID, u.USERNAME as USERNAME," +
                "u.EMAIL as EMAIL, u.PASSWORD as PASSWORD" +
                "from P_USER u where u.ID = ?";

    User user = null;
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        ps = conn.prepareStatement(sql);
        ps.setLong(1, id);
        rs = ps.executeQuery();
        Set<User> userSet = mapUsers(rs);
        if (!userSet.isEmpty()) {
            return userSet.iterator().next();
        }
    } catch (SQLException e) {
        throw new RuntimeException("User not found!", e);
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException ex) {
            }
        }
        if (ps != null) {
            try {
                ps.close();
            } catch (SQLException ex) {
            }
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException ex) {
            }
        }
    }
    return null;
}
```

Figure 4-1. Method using JDBC to search for a user based on its ID in a database, with cross-cutting concern-specific code highlighted

The code implementing the opening and closing of the connection is enclosed in red rectangles. This code was specifically chosen to underline a cross-cutting concern that tangles in the base code, leading to code cluttering. It is not a clean solution and not a stable one either, since in a big project, database communicating methods will be written by different programmers, and it is enough for one to make a mistake that leads to connections to the database not being closed correctly, and the application might end up not being able to communicate with the database because a new connection cannot be opened.

The example above is quite archaic, and the call diagram for the method above is depicted in Figure 4-2.

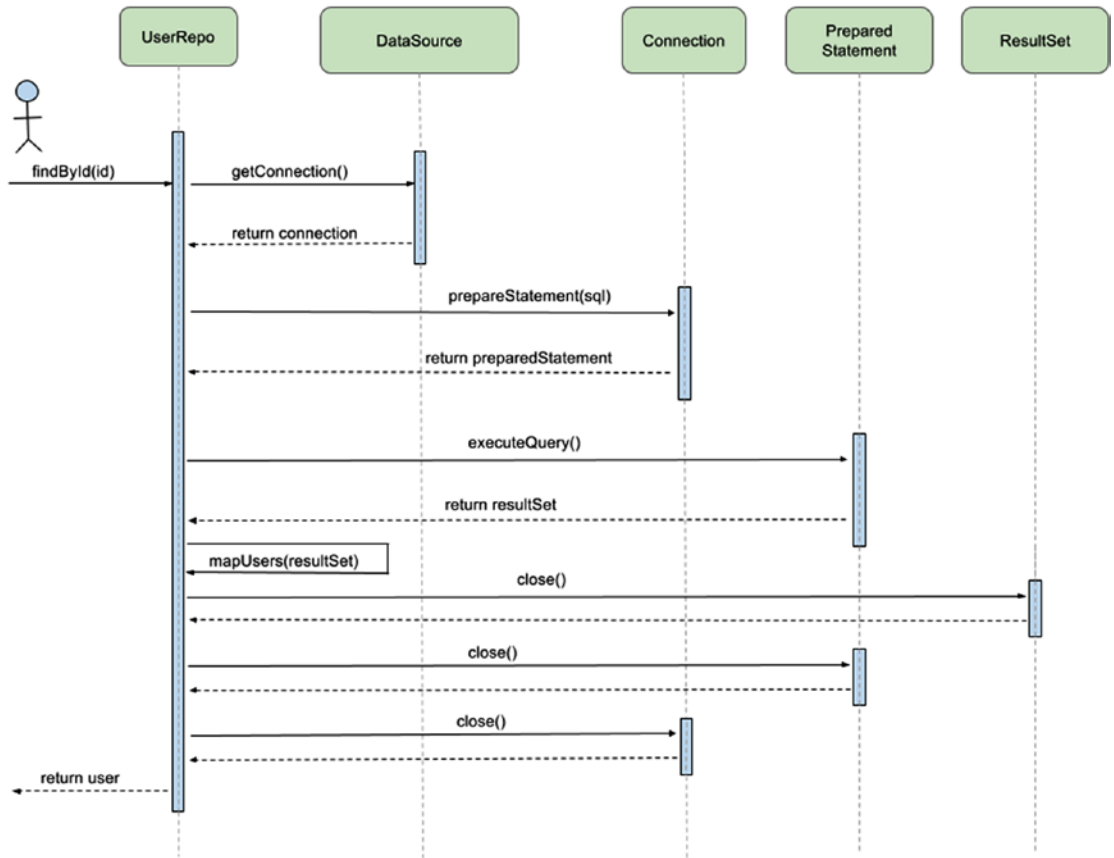


Figure 4-2. UML call diagram for the JDBC method extracting a user by its ID

Complicated, right? AOP can help decouple the connection management code from the business code. Using AOP, the previous diagram should be reduced to something like what is depicted in Figure 4-3.

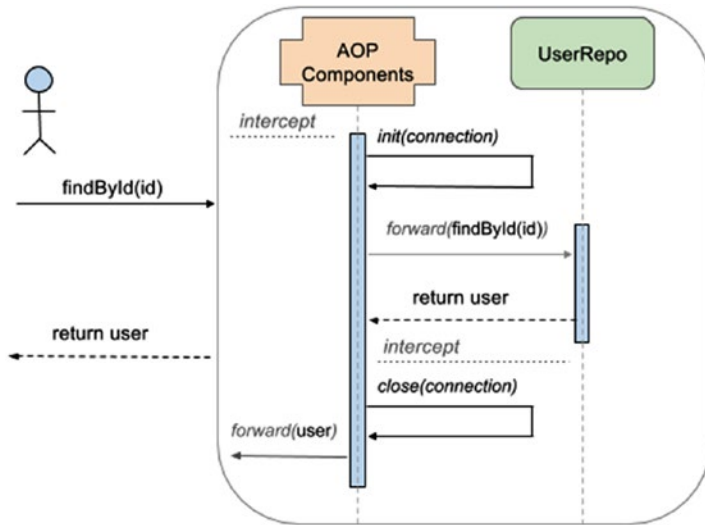


Figure 4-3. Conceptual schema for the `findById()` method with AOP

The schema in the previous image is a conceptual one, and it depicts the basic idea. AOP components intercept calls to certain objects and inject their own functionality, without being strongly connected to the initial target, and they do so in a transparent way.

There are not many applications left that use JDBC components directly. Currently there are frameworks that provide classes to manage the connections for you, so you do not have to write all that code. For example, in Spring, the class `org.springframework.jdbc.core.JdbcTemplate` does that for you, and it will be covered in **Chapter 5: Data Access**. Repository components can be created based on `JdbcTemplate` instances, and database connections are managed by connection pool components. Connection pooling is a technique of reusing connections: when a connection is closed, it can reside in memory to be reused the next time a connection is required, eliminating the cost of creating a new connection. But using connections directly is cumbersome, because application crashes can lead to inconsistent data. Aside from connections, applications use transactions to group database operations in units of work. When all operations have completed successfully, the transaction is committed, and the changes are persisted into the database. If one operation fails, the transaction is rolled back, leaving the database untouched.

In this case, transactions become a cross-cutting concern because a method that is supposed to be called into a transaction has to obtain a transaction, open it, perform its actions, and then commit the transaction.

But usually an application uses more than one cross-cutting concern, the most common grouping is: security + logging (or auditing) + transactions. So service classes that provide and regulate access to data end up looking like Figure 4-4.

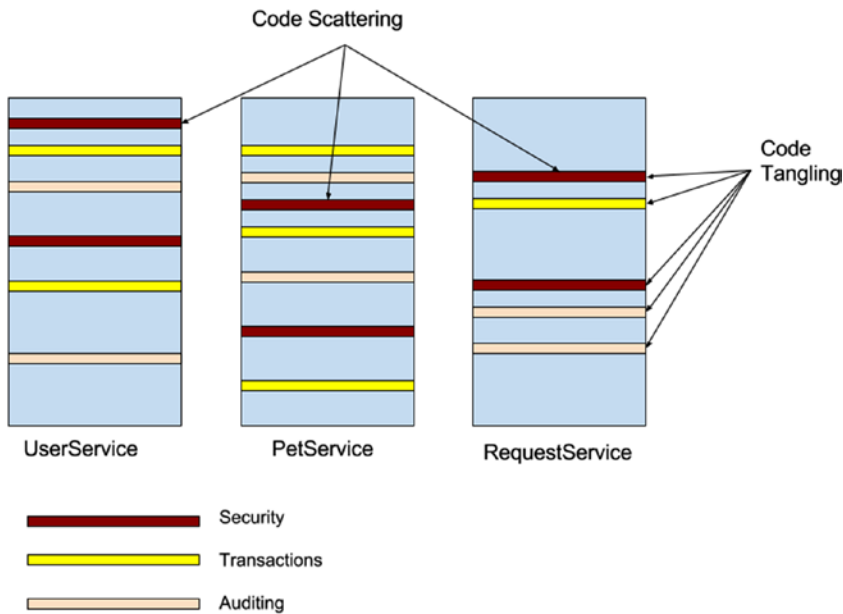


Figure 4-4. Code tangling and scattering in service classes without AOP

Code tangling and code scattering are two issues that come together when base code needs to be executed under certain conditions. Code tangling means that the same component is injected into many others, thus leading to significant dependencies coupling components together. Code scattering means duplication, because you can do the same thing in only one way. The resulting solution is difficult to extend and to maintain, harder to understand, and a pain to debug. Using AOP, a programmer can focus on the core implementation, then define aspect components to cover the cross-cutting concerns. AOP provides even more modularization than OOP does.

Spring AOP

The original library that provided components for creating aspects is named **AspectJ**. It was developed by the Xerox PARC company and released in 1995. It defined a standard for AOP because of its simplicity and usability. The language syntax used to define aspects was similar to Java and allowed developers to define special constructs called aspects. The aspects developed in AspectJ are processed at compile time, so they directly affect the generated bytecode.

The Spring AOP framework is a complement to the current version of AspectJ and contains many annotations that can be used to develop and configure aspects using Java code, but the Spring development team knows and recognizes its limitations. For example, it cannot advise fine-grained objects such as domain objects. Spring AOP functionality is based on AspectJ, which is why when Spring AOP libraries are used, `aspectjweaver` and `aspectjrt` must be added to the application classpath.

Also, Spring AOP cannot advise objects that are not managed by the Spring container. AspectJ can do that. Spring AOP uses dynamic proxies for aspect weaving, so the bytecode of the target objects is not affected in any way. Also, following the Spring convention, Spring AOP is non-invasive having been developed in such a way to keep AOP components decoupled from application components.

In earlier versions, XML was used to define aspects. The purpose of this chapter is to help you modularize your code so that tangling and scattering are eliminated (or at least reduced to a minimum) using everything Spring has to offer.

The Spring Framework is composed of a few libraries:

- **spring-aop** provides AOP alliance API compliant components that can be used to define method interceptors and pointcuts so that code with different responsibilities can be cleanly decoupled.
- **spring-aspects** provides integration with AspectJ.
- **spring-instrument** provides class instrumentation support and classloader implementations that can be used on application servers.
- **spring-instrument-tomcat** contains Spring's instrumentation agent for Tomcat.

Spring IoC can be used to write software applications, and for business applications that require the use of cross-cutting concerns, using Spring AOP is a must. Before digging into it, let's introduce the AOP concepts and how they are declared in Spring.

AOP Terminology

A few specific AOP terms were mentioned in the introduction of this chapter, but a more detailed explanation is required.

- **Aspect:** a class containing code specific to a cross-cutting concern. A class declaration is recognized in Spring as an aspect if it is annotated with the `@Aspect` annotation.
- **Weaving:** a synonym for this word is *interlacing*, but in software, the synonym is *linking*, and it refers to aspects being combined with other types of objects to create an advised object.
- **Join point:** a point during the execution of a program. In Spring AOP, a join point is always a method execution. Basically, the join point marks the execution point where aspect behavior and base behavior join.
- **Target object:** object to which the aspect applies.
- **Target method:** the advised method.
- **Advice:** action taken by an aspect at a join point. In Spring AOP there are multiple advice types:
 - **Before advice:** methods annotated with `@Before` that will execute before the join point. These methods do not prevent the execution of the target method unless they throw an exception.
 - **After returning advice:** methods annotated with `@AfterReturning` that will execute after a join point completes normally, meaning that the target method returns normally without throwing an exception.
 - **After throwing advice:** methods annotated with `@AfterThrowing` that will execute after a join point execution ends by throwing an exception.
 - **After (finally) advice:** methods annotated with `@After` that will execute after a join point execution, no matter how the execution ended.
 - **Around advice:** methods annotated with `@Around` intercept the target method and surround the join point. This is the most powerful type of advice, since it can perform custom behavior before and after the invocation. It has the responsibility of choosing to perform the invocation or return its own value.

- **Pointcut:** a predicate used to identify join points. Advice definitions are associated with a pointcut expression and the advice will execute on any join point matching the pointcut expression. Pointcut expressions are defined using AspectJ Pointcut Expression Language.¹ Pointcut expressions can be defined as arguments for Advice annotations or as arguments for the `@Pointcut` annotation.
- **Introduction:** declaring additional methods, fields, interfaces being implemented, annotations on behalf of another type. Spring AOP allows this using a suite of AspectJ `@Declare*` annotations that are part of the `aspectjrt` library.
- **AOP proxy:** the object created by AOP to implement the aspect contracts. In Spring, proxy objects can be JDK dynamic proxies or CGLIB proxies. By default, the proxy objects will be JDK dynamic proxies, and the object being proxied must implement an interface, that will also be implemented by the proxy object. But a library like CGLIB can be used to create proxies by subclassing too, so an interface is not needed.

Quick Start

To quickly introduce most Spring AOP terms in the previous section, let's consider a very simple example. Let's consider a `JdbcTemplateUserRepo` bean with a few simple methods used to query for or modify `User` instances.+

```
...
@Repository("userTemplateRepo")
public class JdbcTemplateUserRepo implements UserRepo {

    private RowMapper<User> rowMapper = new UserRowMapper();

    protected JdbcTemplate jdbcTemplate;

    @Autowired
    public JdbcTemplateUserRepo(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public Set<User> findAll() {
        String sql = "select id, username, email, password from p_user";
        return new HashSet<>(jdbcTemplate.query(sql, rowMapper));
    }

    public User findById(Long id) {
        String sql = "select id, email, username,password from p_user where id= ?";
        return jdbcTemplate.queryForObject(sql, rowMapper, id);
    }
}
```

¹Complete reference here: <https://eclipse.org/aspectj/doc/next/progguide/language.html>.

```

public Set<User> findAllByUsername(String username, boolean exactMatch) {
    String sql = "select id, username, email, password from p_user where ";
    if (exactMatch) {
        sql += "username= ?";
    } else {
        sql += "username like '% ' || ? || '%'";
    }
    return new HashSet<>(jdbcTemplate.query(sql, new Object{username}, rowMapper));
}

public void updatePassword(Long userId, String newPass) {
    String sql = "update p_user set password=? where ID = ?";
    jdbcTemplate.update(sql, newPass, userId);
}

// Maps a row returned from a query executed on the P_USER table to a User object.
//implementation not relevant for this chapter
private class UserRowMapper implements RowMapper<User> {
    ...
}
}

```

The `UserRowMapper` instance is not relevant for this chapter, but in case you are curious, this class implements the Spring `RowMapper` interface, and it is used by `JdbcTemplate` instances to transform database records into domain objects to be used by the application. Objects of this type are typically stateless and reusable. They will be covered in detail in **Chapter 5: Data Access**.

An aspect can be created to monitor the execution of the methods of this bean. The following code snippet depicts an aspect class definition that contains a single advice that prints a message every time the `findById` method is called.

```

package com.ps.aspects;

import org.apache.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

import org.aspectj.lang.annotation.Component;
import org.aspectj.lang.annotation.Aspect;
import underlineorg.aspectj.lang.annotation.Before;

@Aspect
@Component
public class UserRepoMonitor {
    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);

    @Before
    ("execution(public com.ps.repos.`JdbcTemplateUserRepo+.findById(..)`)")
    public void beforeFindById(JoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        logger.info(" ---> Method " + methodName + " is about to be called");
    }
}

```


The class containing the aspect definition must be declared as a bean. This can be done using any of the three methods covered in Chapter 2. In the example above, the declaration is done by annotating the class with `@Component`. The `@Before` annotation is used with a parameter that is called a pointcut expression. This is used to identify the method execution on which the behavior will be applied.

To test the following code, we need a configuration class and a test class. The configuration for the datasource will be decoupled in the class `com.ps.config.TestDataConfig`, that is not relevant for this chapter, but you can find it in the sources attached to this chapter in project 04-aop-practice. The application configuration will be provided by the class `AppConfig`. To enable aspect support, the configuration class must be annotated with `@EnableAspectJAutoProxy`.

```
package com.ps.config;
...
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@ComponentScan(basePackages = {"com.ps.repos.impl", "com.ps.aspects"})
@EnableAspectJAutoProxy
public class AppConfig {
}
```

To be able to add behavior to existing objects, Spring uses a method called proxying, which implies creating an object that wraps around the target object. By default, Spring creates JDK dynamic proxies, which are proxy objects implementing the same interface the target object does. The `@EnableAspectJAutoProxy` annotation can also be used to modify the type of proxies created if CGLIB is on the classpath. The CGLIB proxies are called sub-class proxies, because they extend the type of the target object. To change the type of proxies to CGLIB proxies, just set the `proxyTargetClass` parameter to true. But before getting deeper into this topic, let's do some testing.

```
... // imports here
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class TestJdbcTemplateUserRepo {

    @Autowired
    @Qualifier("userTemplateRepo")
    UserRepo userRepo;

    @Test
    public void testFindById() {
        User user = userRepo.findById(1L);
        assertEquals("John", user.getUsername());
    }
}
```

! Because the `UserRepoMonitor` only prints messages currently, to clearly see the advice in action all the logs will be set to OFF, except the one of the aspect class. This can be done by editing the `04-ps-aop-practice/src/test/resources/logback-test.xml` and setting all logs for other packages to OFF and the log level for the `com.ps.aspects` on INFO, as the following code snippet shows:

```
<?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="org.springframework" level="off"/>
  <logger name="org.springframework.jdbc" level="off"/>
  <logger name="org.h2" level="off"/>
  <logger name="com.ps.aspects" level="info"/>

  <root level="info">
    <appender-ref ref="console" />
  </root>
</configuration>
```

When the `testFindById` method is executed, in the log a single line will be printed.

```
INFO c.p.a.UserRepoMonitor - ---> Method findById is about to be called.
```

It is obvious that the before advice was executed, but how does it actually work? Spring IoC creates the `userTemplateRepo` bean. Then the aspect definition with an advice that has to be executed before the `findById` method tells Spring that this bean has to be wrapped up in a proxy object that will add additional behavior, and this object will be injected instead of the original everywhere needed. And because we are using JDK dynamic proxies, the proxy will implement the `UserRepo` interface. Figure 4-5 depicts this situation.

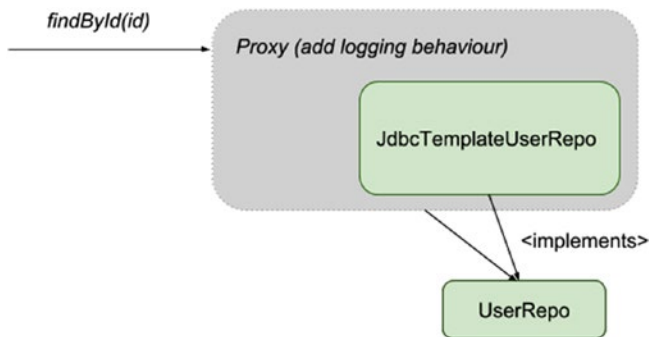


Figure 4-5. The userTemplateRepo proxy bean

The bean that is injected into the test bean can be inspected by placing a breakpoint inside the `testFindById` method and starting the test in debug mode. In the debugger view, the `userRepo` can be expanded and its contents inspected. In Figure 4-6 you can see side by side the structure of a `userRepo` bean when aspects are used and when they aren't.²

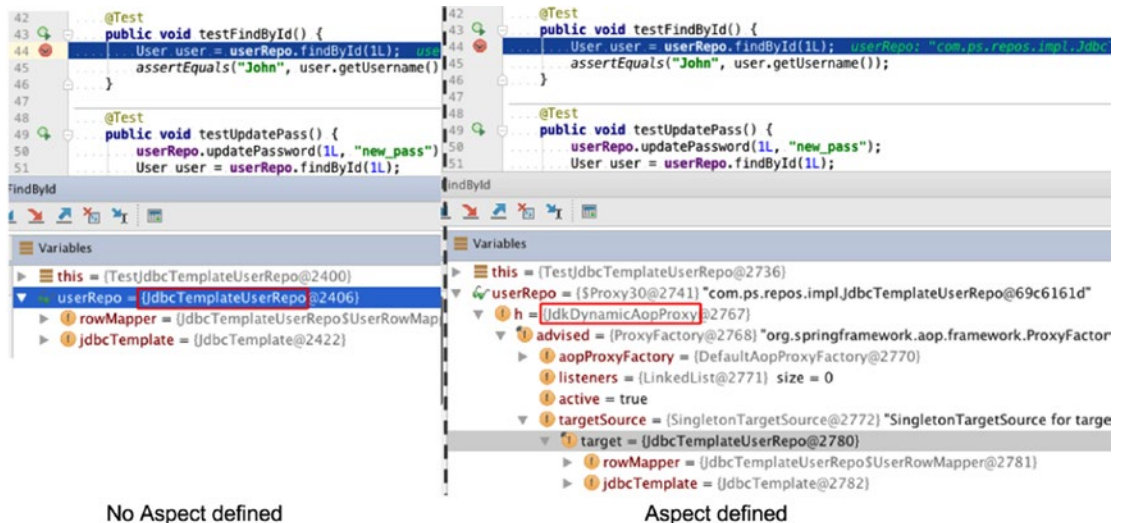


Figure 4-6. Injected bean `userRepo` structure when aspects are used and when they are not as displayed in the IntelliJ IDEA Debugger view. On the right you notice the type of the injected bean: `JdkDynamicAopProxy`

In this case, it is obvious that the type of proxy created is the default JDK dynamic proxy, which is useful when the target class implements one or more interfaces, because Spring will create a proxy that implements all of them. The JDK proxies are created at runtime using the JDK Reflection API, and reflection is known to affect performance.

²You can disable aspects support by commenting the `@EnableAspectJAutoProxy` annotation, and Spring will just ignore the `Aspect` annotation.

If the CGLIB library is to be added to the application classpath, Spring must be told that we want subclass-based proxies by modifying the aspect enabling annotation to `@EnableAspectJAutoProxy(proxyTargetClass = true)`. This approach is suitable when the target class does not implement any interface, so Spring will create a new class on the fly that is a subclass of the target class. CGLIB is suitable for that because it is a bytecode generation library. The proxies generated with CGLIB are called subclass-based, and one of their advantages is that they can override methods optionally. They use specialized interceptors to call methods, and this can actually improve performance.

In Figure 4-7 you can see the structure of the `userRepo` bean when CGLIB is used to create proxies.

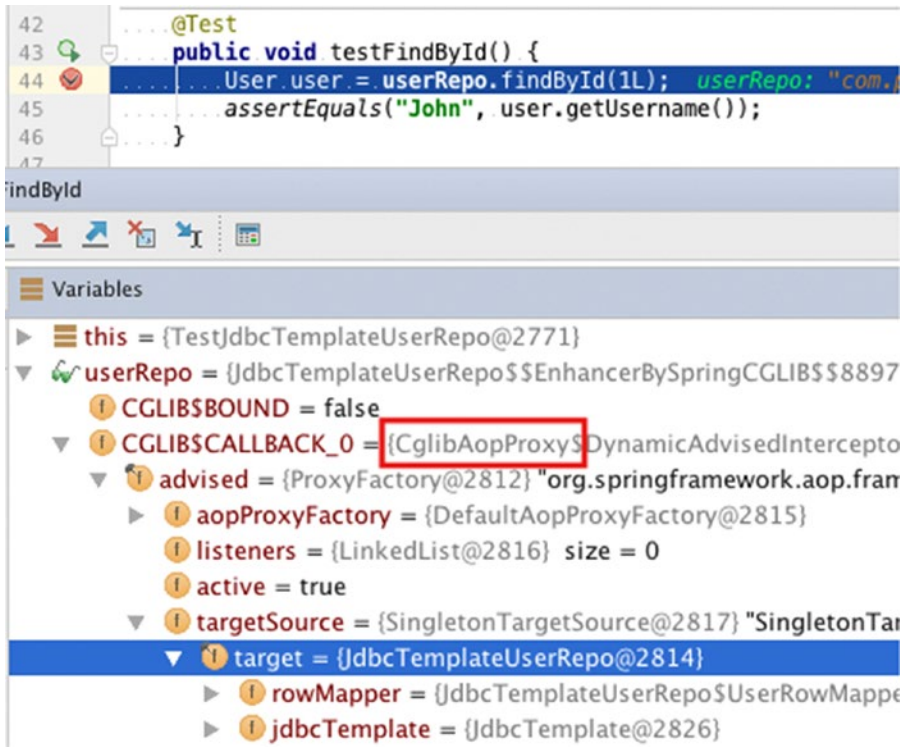


Figure 4-7. Injected bean `userRepo` structure when proxies are created using CGLIB in the IntelliJ IDEA Debugger view

So basically, in order to use aspects in a Spring application you need the following:

- `spring-aop` as a dependency
- declare an `@Aspect` class and declare it as a bean as well (using `@Component` or `@Bean` or XML typical bean declaration element)
- declare an advice method annotated with a typical advice annotation (`@Before`, `@After`, etc.) and associate it to a pointcut expression
- enable aspects support by annotating a configuration class with `@EnableAspectJAutoProxy`
- (optional) add CGLIB as a dependency and enable aspects support using subclassed proxies by annotating a configuration class with `@EnableAspectJAutoProxy(proxyTargetClass = true)`

Aspect Support Configuration using XML

Although aspect support configuration has been based on annotations since Spring 3.1, before that version, the configuration was done using XML. Since your experience as a developer you might end up working on projects using Spring < 3.1, we find it useful to cover how to configure aspect support using XML.

Obviously, there's a namespace for that, which provides an element definition equivalent to `@EnableAspectJAutoProxy`. The specific element is `<aop:aspectj-autoproxy../>`, which has a child element `<aop:include ../>` for each aspect defined in the application.

If you want to avoid configuring aspects using annotations altogether, the reason for that being that you are using a Java version < 5 and annotations are not supported, there is also an `<aop:config />` element that can be used to declare methods of a bean as advice and associate pointcuts.

```
<?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"
  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">
  <!-- Configuration for the aspects that apply to the application -->
  <bean id="userRepoMonitor" class="com.ps.aspects.UserRepoMonitor" />

  <!-- Configure Aspect support -->
  <aop:aspectj-autoproxy>
    <aop:include name="userRepoMonitor"/>
  </aop:aspectj-autoproxy>

  <!-- Configure advice -->
  <aop:config>
    <aop:aspect ref="userRepoMonitor">
      <aop:before
        pointcut="execution(public com.ps.repos.*JdbcTemplateUserRepo+.findById(..))"
        method="beforeFindById" />
    </aop:aspect>
  </aop:config>
</beans>
```

To configure the use of subclass-based proxies, the `<aop:aspectj-autoproxy../>` must have the `proxy-target-class` attribute set to `true`.

Defining Pointcuts

In the Quick Start section, the `before` advice was covered and the pointcut expression associated with it, which provided means to identify where to apply the advice. The pointcut expression looked like the following snippet:

```
execution(public * com.ps.repos.*.JdbcTemplateUserRepo+.findById(..))
```

The template that a pointcut expression follows can be defined as follows:

```
execution( [Modifiers] [ReturnType] [FullClassName].[MethodName]
           ([Arguments]) throws [ExceptionType])
```

The expression can contain wildcards like + and * and can be made of multiple expressions concatenated by boolean operators such as &&, ||, etc. The * wildcard replaces any group of characters; the + wildcard is used to specify that the method to advise can also be found in subclasses identified by [FullClassName] criteria.

There is also a list of designators that can be used to define the reach of the pointcut; for example, the within(...) designator can be used to limit the pointcut to a package

```
public * com.ps.repos.*.JdbcTemplateUserRepo+.findById(..) && +underlinewithin(com.ps.*)
```

Also, a pointcut expression can identify only methods defined in a class annotated with a specific annotation:

```
execution(@org.springframework.transaction.annotation.Transactional
         public * com.ps.repos.*.Repo+.findById(..))
```

A pointcut expression can even identify methods that return values with a specific annotation:

```
execution(public (@org.springframework.stereotype.Service *) *(..))
```

And by using the @annotation() designator, only methods annotated with a specific annotation can be taken into consideration:

```
execution(public (public * com.ps.service.*.Service+.*(..)
             && @annotation(org.springframework.security.access.annotation.Secured))
```

This is useful when security is involved, because the @annotation can be bound to a method parameter so that the annotation object is available in the advice body.

```
@Around("execution(* *(..) && @annotation(securedObj)")
public void calculateExecutionTime(ProceedingJoinPoint pjp,
    Secured securedObj) throws Throwable {
    // extra security checks using securedObj.allowedRoles()
    ...
}
```

! Observations:

- The [ReturnType] is mandatory. If the return type is not a criterion, just use *. If it is missing the application crashes at boot time, throwing a java.lang.IllegalArgumentException with a message explaining that the pointcut is not well-formed.
- The [Modifiers] is not mandatory and if not specified defaults to public.

- The [MethodName] is not mandatory, meaning no exception will be thrown at boot time. But if unspecified, the join point where to execute the advice won't be identified. It's safe to say that if you want to define a technically useful pointcut expression, you need to specify it.
- The [Arguments] is mandatory. If it is missing the application crashes at boot time, throwing a `java.lang.IllegalArgumentException` with a message explaining that the pointcut is not well formed. If the arguments are not a criterion just use `(. .)`, which will match a method with 0 or many arguments. If you want the match to be done on a method with no arguments, use `()`. If you want the match to be done on a method with a single argument, use `(*)`.

Using pointcut expressions can become tedious when the criterion for determining the method to be advised is complicated. But declaring pointcuts can be made pleasant by breaking down the expressions into several subexpressions that can be combined and reused in other composed expressions. Spring also offers the possibility to externalize expressions in dedicated classes—yes, even pointcuts declarations can be decoupled. In the following code snippets we'll explain how this can be done.

Let's declare a service class that can be used to manage `User` instances. This service class will use the `UserRepo` to interact with the database.

```
...
@Service
public class UserServiceImpl implements UserService {

    private UserRepo userRepo;

    @Autowired
    public void setUserRepo(UserRepo userRepo) {
        this.userRepo = userRepo;
    }

    @Override
    public Set<User> findAll() {
        return userRepo.findAll();
    }

    @Override
    public void updateUsername(Long id, String username) {
        userRepo.updateUsername(id, username);
    }

    @Override
    public void updatePassword(Long id, String password) {
        userRepo.updatePassword(id, password);
    }

    @Override
    public User findById(Long id) {
        return userRepo.findById(id);
    }
}
```

We want to write an aspect that will write a message every time an `update*` method is called only from a repository or a service class managing `User` instances. So the pointcut has to look for a method whose name begins with `update` that has arguments and is found in any of the classes implementing `UserRepo` or `UserService`.

There are many ways the pointcut expression can be written, and in this section a few of them will be presented. The aspect class and advice look like this:

```
@Aspect
@Component
public class UserRepoMonitor {
    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);

    @Before("execution( * com.ps.repos.*UserRepo+.update*(..))
           || execution( * com.ps.services.*Service+.update*(..))")
    public void beforeUpdate(JoinPoint joinPoint) throws Throwable {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String methodName = joinPoint.getSignature().getName();
        logger.info(" ---> Method " + className + "." + methodName
                   + " is about to be called");
    }
}
```

Of course in a small application, when you know you only have those two classes, this pointcut expression will do too: `"execution(* update*(..))"`, but in a big application, this relaxed expression might cause the advice to be applied on methods you did not intend to be advised. The expression used in the first code snippet is composed of two expressions.

```
execution( * com.ps.repos.*UserRepo+.update*(..))
|| execution( * com.ps.services.*Service+.update*(..))
```

The two expressions can be split into two pointcut declarations that can be associated to methods. The name of these methods will then be used in a composite expression to identify a pointcut. And that is why these pointcuts are called `Named Pointcuts`.

```
@Aspect
@Component
public class UserRepoMonitor {
    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);

    @Before("repoUpdate() || serviceUpdate()")
    public void beforeUpdate(JoinPoint joinPoint) throws Throwable {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String methodName = joinPoint.getSignature().getName();
        logger.info(" ---> Method " + className + "." + methodName
                   + " is about to be called");
    }
}
```



```

@Pointcut("execution( * com.ps.repos.*UserRepo+.update*(..))")
public void repoUpdate() {}

@Pointcut("execution (* com.ps.services.*Service+.update*(..))")
public void serviceUpdate() {}
}

```

It was mentioned before that a dedicated class can be created to group together pointcuts. In this case, the composite expression must be modified to contain the package and classname where the methods are located.

```

package com.ps.aspects;
import org.aspectj.lang.annotation.Pointcut;

public class PointcutContainer {

    @Pointcut("execution( * com.ps.repos.*UserRepo+.update*(..))")
    public void repoUpdate() {
    }

    @Pointcut("execution (* com.ps.services.*Service+.update*(..))")
    public void serviceUpdate() {
    }
}

```

Methods `repoUpdate` and `serviceUpdate` are moved to the class `com.ps.aspects.PointcutContainer`, so the expression for the `beforeUpdate` advice changes to:

```

@Before("com.ps.aspects.PointcutContainer.repoUpdate() ||
        com.ps.aspects.PointcutContainer.serviceUpdate()")
public void beforeUpdate(JoinPoint joinPoint) throws Throwable {
    ...
}

```

So, now that the pointcuts are externalized, what more can be done? The methods annotated with `@Pointcut` and associated with pointcut expressions can be used to process data. So far, we have been using the `JoinPoint` argument to extract context data. But context data can be injected by Spring, if told so by using special designators.

The update methods in our example have a string argument that must be processed, and an exception must be thrown if the value contains special characters like `$`, `#`, `&`, `%`. This should be done only in the service methods, because there is no point in calling a repo method with a bad argument. This means that we have to separate the advice in two parts: one for repositories, one for services.

```

@Aspect
@Component
public class UserRepoMonitor {
    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);

    @Before("com.ps.aspects.PointcutContainer.serviceUpdate()")
    public void beforeServiceUpdate(JoinPoint joinPoint) throws Throwable {
        Object[] args = joinPoint.getArgs();
        String text = (String)args[1];
    }
}

```

```

    if (StringUtils.indexOfAny(text, new String[]{"$", "#", "&", "%"}) != -1) {
        throw new IllegalArgumentException("Text contains weird characters!");
    }
}

@Before("com.ps.aspects.PointcutContainer.repoUpdate()")
public void beforeRepoUpdate(JoinPoint joinPoint) throws Throwable {
    String className = joinPoint.getSignature().getDeclaringTypeName();
    String methodName = joinPoint.getSignature().getName();
    logger.info(" ---> Method " + className + "." + methodName
        + " is about to be called");
}
}

```

Notice how all information about the context of the advices is extracted from the `joinPoint` object. The `JoinPoint` interface provides a couple of methods to access information about the target method, all accessible via the proxy object. A reference to the target object can be obtained by calling `joinPoint.getTarget()`. In Figure 4-8, a test run was stopped at a breakpoint after calling the `joinPoint.getTarget()` method, and the target object is the focus on the Debugger view in IntelliJ IDEA. You can clearly notice its type.

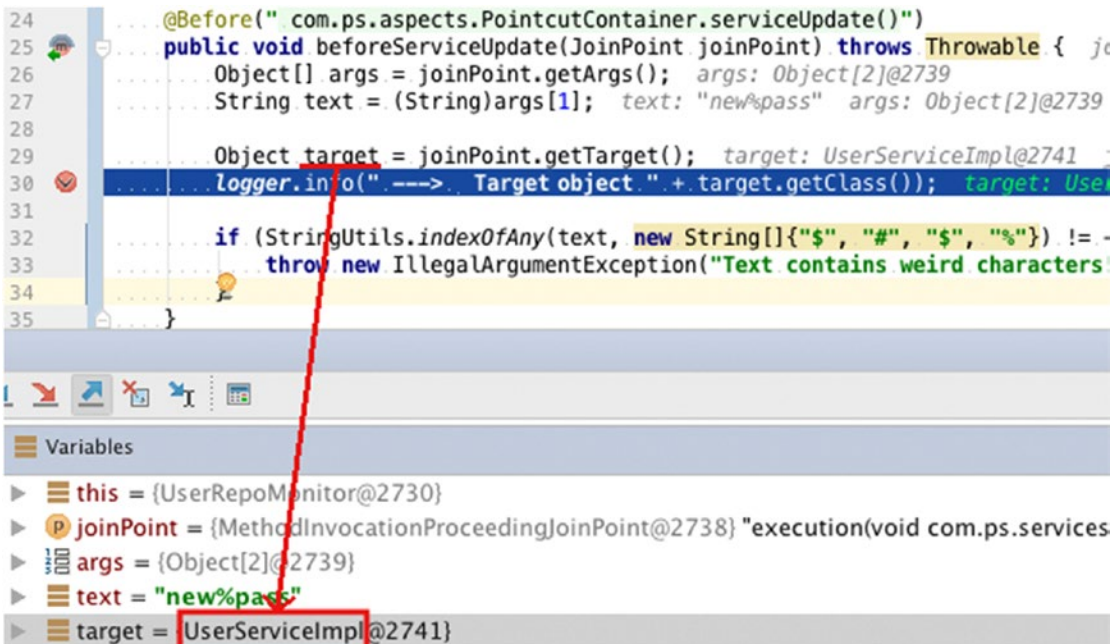


Figure 4-8. The advice target object

Instead of calling `JoinPoint` methods, we'll modify the implementation and use Spring to automatically populate the information we need. The `pointcut` expression becomes:

```
public class PointcutContainer {
...
    @Pointcut("execution (* com.ps.services.*Service+.update*(..))
              && args(id,pass) && target (service)")
    public void serviceUpdate(UserService service, Long id, String pass) {
    }
}
```

! The `args()` designator is used to identify methods with a parameter configuration defined by it. It can be used as `args(com.ps.ents.User)` when it identifies methods with one parameter of type `User`. But this designator can be used to make method arguments available in the advice body.

In the code snippet above, it is used in its binding form to ensure that the value of the corresponding argument will be passed as the parameter value when the advice is invoked. In this case, the values of the `(id,pass)` method arguments will be injected into `(id,pass)` advice arguments. And yes, as you probably suspected, the name of the parameter in the designator must match the advice parameter name.

The advice can now be modified to declare exactly the fields we need Spring to populate for us and use them directly. No need to use a `JoinPoint` object to extract them and take care of the conversions in the advice body, because Spring has already done that for us.

```
@Aspect
@Component
public class UserRepoMonitor {
    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);

    @Before("com.ps.aspects.PointcutContainer.serviceUpdate(service, id, pass)")
    public void beforeServiceUpdate
        (UserService service, Long id, String pass) throws Throwable {
        logger.info(" ---> Proxied object " + service.getClass());

        if (StringUtils.indexOfAny(pass, new String{"$", "#", "%", "%"}) != -1) {
            throw new IllegalArgumentException("Text for " + id
                + " contains weird characters!");
        }
    }
}
```

And now that we know how to define pointcuts, it is time to come back and dig deeper into advice types and definitions.

Implementing Advice

In the **Quick Start** section, some technical details about how aspects and advice are implemented in Spring were covered, but the surface was only scratched. The pointcut specifics had to be covered first to make advice declaration easier to understand. This section will cover in detail every type of advice that you will most likely need during development of Spring Applications.

Before

You were already introduced to the before advice. In the previous section there was a before advice defined that tested an argument of type string.

```
import org.aspectj.lang.annotation.Before;
...

@Before("com.ps.aspects.PointcutContainer.serviceUpdate(id, pass)")
public void beforeServiceUpdate (Long id, String pass) throws Throwable {

    if (StringUtils.indexOfAny(pass, new String{"$", "#", "$", "%"}) != -1) {
        throw new IllegalArgumentException("Text for " + id
            + " contains weird characters!");
    }
}
```

The UML sequence sequence that describes what happens when `userService.update*(..)` is called is depicted in Figure 4-9.

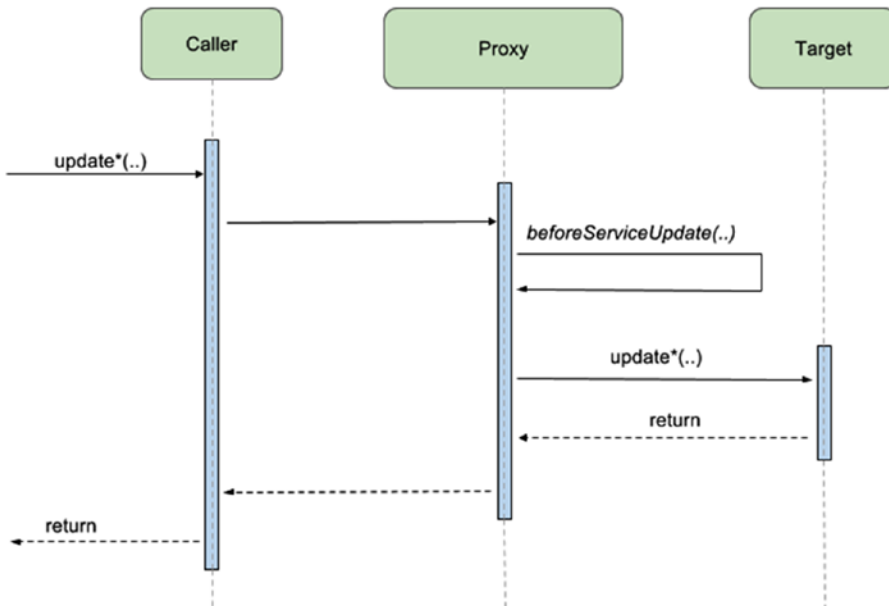


Figure 4-9. Before advice UML call diagram

The proxy object receives the call destined for the target bean and calls first the advice method. If the advice method returns successfully, it then forwards the initial call to the target bean and forwards the result back to the caller. If the advice method throws an exception, the exception gets propagated to the caller, and the target method is no longer executed. A test method to cover the case in which the advice method throws an exception should expect an exception of type `IllegalArgumentException`, and the code is depicted after this paragraph.

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})

@ActiveProfiles("dev")
public class TestUserService {

    @Autowired
    UserService userService;
    @Test(expected = IllegalArgumentException.class)
    public void testUpdatePass() {
        userService.updatePassword(1L, "new%pass");
        User user = userService.findById(1L);
        assertEquals("new%pass", user.getPassword());
    }
}
```

Because the before advice does not have any effect on the arguments, it is used most of the times to perform checks and stop the execution of the target method by throwing an exception if those checks are not passed. Obviously, the most likely candidate for this is the security concern. In an application that requires users to provide credentials before using it, security rights and roles must be checked every time a method handling sensitive information is called. Including the security check in every method body would be a pain. So an advice that tests the provided credentials is definitely a more practical solution. The Spring Security Framework, which will be covered in **Chapter 6: Spring Web**, makes handling security even easier by providing out of the box security aspects.

After Returning

This type of advice is executed only if the target method executed successfully and does not end by throwing an exception. An aspect of this type is defined in the code snippet below for the same update methods in the target bean. This advice will just print a message confirming that the update process executed correctly. The `@AfterReturning` has an attribute that can be used to access the result of the target method execution named `returning`. The value of this attribute must match the name of the parameter name in the advice method signature, because Spring will inject there the result of the execution of the target method.

```
import org.aspectj.lang.annotation.AfterReturning;
...

@AfterReturning(value="execution (* com.ps.services.*Service+.update*(..))",
    returning = "result")
public void afterServiceUpdate(JoinPoint joinPoint, int result) throws Throwable {
    String className = joinPoint.getSignature().getDeclaringTypeName();
    String methodName = joinPoint.getSignature().getName();
    if(result == 0) {
        logger.info(" ---> Update method " + className + "." + methodName
            + " performed as expected.");
    }
}
```

The UML sequence diagram that describes what happens when `userService.update*(..)` is called is depicted in Figure 4-10. Because the after returning advice has access to the value returned by the target method, this advice is suitable for use when caching is implemented.

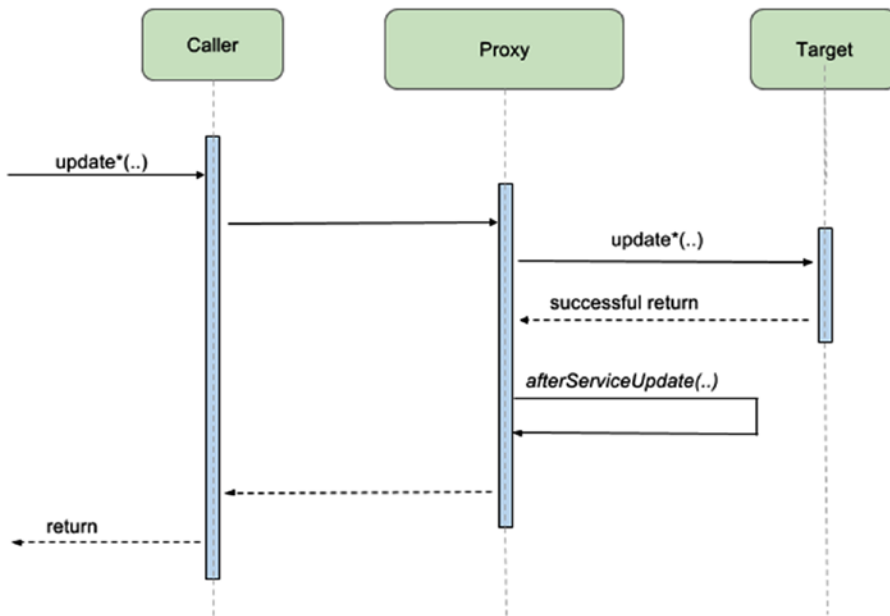


Figure 4-10. After Returning advice UML sequence diagram

After Throwing

The after throwing advice is similar to the after returning. The only difference is that its criterion of execution is exactly the opposite. That is, this advice is executed only when when the target method ends by throwing an exception. The `@AfterThrowing` annotation has an attribute named `throwing` that can be used to access the exception thrown by the target method. The value of this attribute must match the name of the exception parameter of the advice method.

```

//custom exception type for this example
import com.ps.exception.UnexpectedException;
...
import org.aspectj.lang.annotation.AfterThrowing;
...
@AfterThrowing(value="execution
    (* com.ps.services.*Service+.updateUsername(..)", throwing = "e")
public void afterBadUpdate(JoinPoint joinPoint, Exception e) throws Throwable {
    String className = joinPoint.getSignature().getDeclaringTypeName();
    String methodName = joinPoint.getSignature().getName();
    if(e instanceof DuplicateKeyException) {

```

```

        logger.info(" ---> Update method " + className + "."
            + methodName + " failed. Existing username found.");
    } else {
        throw new UnexpectedException(" Ooops!", e);
    }
}

```

This type of advice does not stop the propagation of the exception, but it can change the type of exception being thrown.

For this example, it is an advice that is executed only before the `userService.updateUsername()` is used. The repository method called by the service method throws a `DuplicateKeyException` if the username is already found in the database. This type of exception is expected to be thrown, so the advice just prints a notification, and the original exception propagates. But if a different type of exception is thrown, then the advice wraps up the original exception into an `UnexpectedException` to tell the original caller that something wrong has happened that needs to be handled.

To test that this advice is correctly executed, the test method must be declared to expect the `DuplicateKeyException` to be thrown and provide a username that already exists in the the database.

```

@Test(expected = DuplicateKeyException.class)
public void testAfterUpdateUsernameBad() {
    int result = userService.updateUsername(1L, "Johnny");
    assertEquals(1, result);
}

```

This test must pass, and in the console a single line must be printed. (This is, of course, true only if you do not have other advice defined for the same method.)

```

INFO c.p.a.UserRepoMonitor - ---> Update method
    com.ps.services.UserServiceImpl.updateUsername failed. Existing username found.

```

The UML sequence diagram that describes what happens when `userService.updateUsername(..)` is called is depicted in Figure 4-11. This type of advice can be used to restore the system state after an unexpected failure.

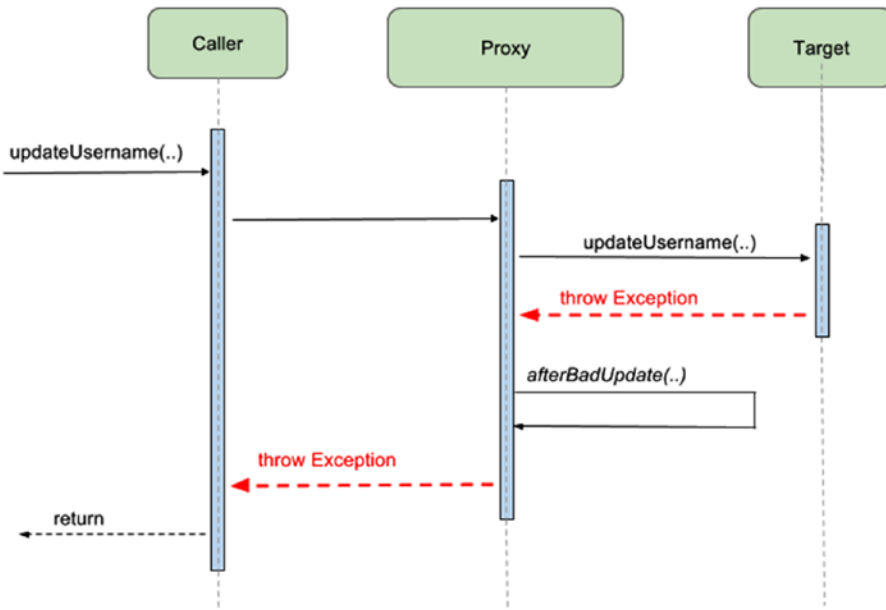


Figure 4-11. After Throwing advice UML sequence diagram

After

The after advice is executed after the target method regardless of how its execution ended, whether successfully or with an exception, and because of this, it is most suitable to use for auditing or logging.

```

import org.aspectj.lang.annotation.After;
...
@Aspect
@Component
public class UserRepoMonitor {

    private static final Logger logger = Logger.getLogger(UserRepoMonitor.class);
    private static long findByIdCount = 0;

    @After(
        "execution(public * com.ps.repos.*.JdbcTemplateUserRepo+.updateUsername(..)")
    )
    public void afterFindById(JoinPoint joinPoint) throws Throwable {
        ++findByIdCount;
        String methodName = joinPoint.getSignature().getName();
        logger.info(" ---> Method " + methodName + " was called "
            + findByIdCount + " times.");
    }
}

```

The UML sequence diagram that describes what happens when `userService.updateUsername(..)` is called is depicted in Figure 4-12.

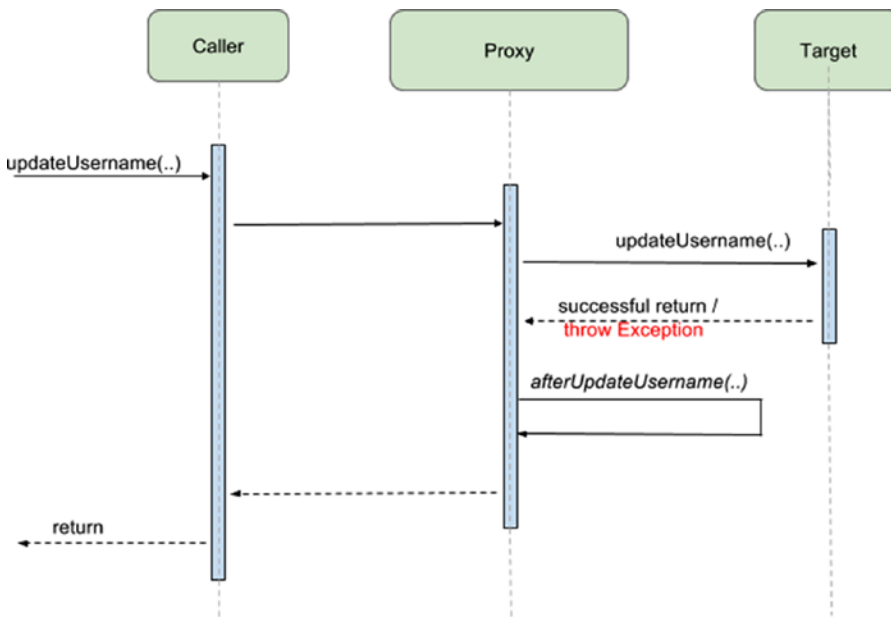


Figure 4-12. After advice UML sequence diagram

! Until now, every advice in the default implementation (without using designators for binding parameters like args or target) has as a parameter a reference of type `org.aspectj.lang.JoinPoint`. The object that Spring injects at runtime provides access to both the state available at a join point and static information about it: type of the target, name of the method target, arguments, reference to the target itself. This information can be used for tracing and logging; it does not give direct control over the execution of the target method.

The type allowing this is `org.aspectj.lang.ProceedingJoinPoint`, an extension of `org.aspectj.lang.JoinPoint` that can be used as type for the join point parameter only in around advice.

Around

The around advice is the most powerful type of advice, because it encapsulates the target method and has control over its execution, meaning that the advice decides whether the target method is called, and if so, when. For this section an advice that logs in the duration of each find operation was created.

```

import org.aspectj.lang.annotation.Around;
...
@Around("execution(public * com.ps.repos.*.*Repo+.find*(..))")
public Object monitorFind(ProceedingJoinPoint joinPoint) throws Throwable {
    String methodName = joinPoint.getSignature().getName();
    logger.info(" --> Intercepting call of: " + methodName);
    long t1 = System.currentTimeMillis();
    try {

```

```

        //put a pause here so we can register an execution time
        Thread.sleep(1000L);
        return joinPoint.proceed();
    } finally {
        long t2 = System.currentTimeMillis();
        logger.info(" ---> Execution of " + methodName + " took: "
            + (t2 - t1) / 1000 + " ms.");
    }
}

```

The type `ProceedingJoinPoint` inherits from `JoinPoint` and adds the `proceed()` method that is used to call the target method. And because in this case the advice method calls the target method directly, exceptions can be caught and treated in the advice method, instead of propagating them. The UML sequence diagram that describes what happens when any `userService.find*(..)` is called is depicted in Figure 4-13.

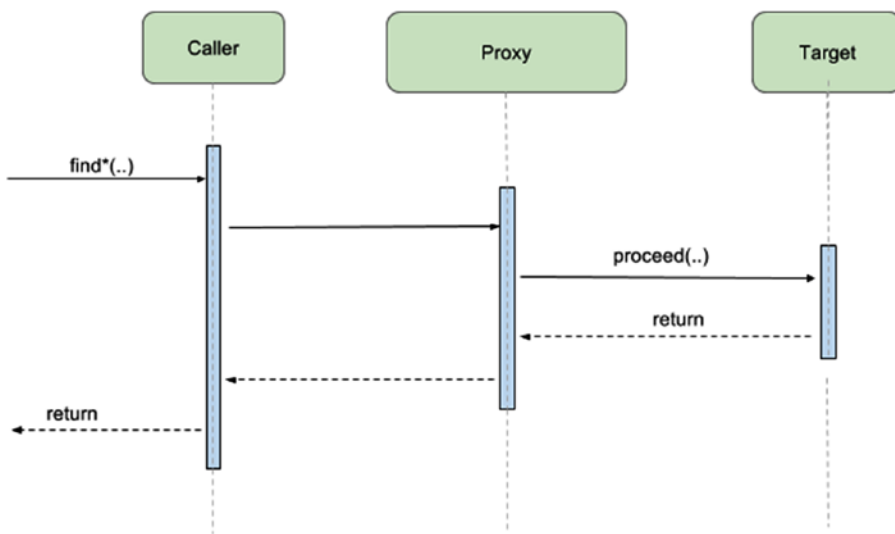


Figure 4-13. Around advice UML sequence diagram

Conclusions

Spring AOP together with AspectJ provides a very practical way for separating cross-cutting concerns in targeted components. Other Spring modules and frameworks would have been much more difficult to develop and use without AOP. Spring transactions, a module that will be covered in the next chapter, makes use of Spring AOP to transparently manage transactions. Spring Security, which will be covered in Chapter 6 makes use of AOP to control access to sensitive data.

However Spring AOP does have its limitations:

- Only public Join Points can be advised (you probably suspected that).
- Aspects can be applied only to Spring Beans.

- Even if Spring AOP is not set to use CGLIB proxies, if a Join Point is in a class that does not implement an interface, Spring AOP will try to create a CGLIB proxy.
- If a method in the proxy calls another method in the proxy, and both match the pointcut expression of an advice, the advice will be executed only for the first method. This is the proxy's nature: it executes the extra behavior only when the caller calls the target method.

Let's consider the following example. We'll edit the `JdbcTemplateUserRepo` to add a method named `updateDependencies` that is supposed to update some records that depend on the object being updated. The actual body is not important for this scenario.

```
@Repository("userTemplateRepo")
public class JdbcTemplateUserRepo implements UserRepo {
    ...
    @Override
    public int updateUsername(Long userId, String username) {
        String sql = "update p_user set username=? where ID = ?";
        updateDependencies(userId);
        return jdbcTemplate.update(sql, username, userId);
    }

    @Override
    public int updateDependencies(Long userId) {
        //mock method to test the proxy nature
        return 0;
    }
}
```

Let's consider the following pointcut expression and advice declaration:

```
// PointcutContainer.java
public class PointcutContainer {
    ...

    @Pointcut("execution( * com.ps.repos.*.*UserRepo+.update*(..))")
    public void proxyBubu() {
    }
}

// UserRepoMonitor.java
@Aspect
@Component
public class UserRepoMonitor {
    ...
    @Before("com.ps.aspects.PointcutContainer.proxyBubu()")
    public void bubuHappens(JoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        String className = joinPoint.getSignature().getDeclaringTypeName();
        logger.info(" ---> BUBU when calling: " + className + "." + methodName);
    }
}
```

To test the previous example, let's use a simple method:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class TestUserService {

    @Autowired
    UserService userService;
    ...

    @Test
    public void testProxyBubu() {
        int result = userService.updateUsername(3L, "Iuliana");
        assertEquals(1, result);
    }
}
```

If the previous test method is executed, in the console only one line can be seen:

```
INFO c.p.a.UserRepoMonitor - ---> BUBU when calling:
        com.ps.repos.impl.JdbcTemplateUserRepo.updateUsername
```

But how do we know that the `updateDependencies` matches the joinpoint expression? We create a method in the service bean that calls this method, and we write a test for it as well.

```
// UserServiceImpl.java
@Service
public class UserServiceImpl implements UserService {
    ...
    @Override
    public int updateDependencies(Long id) {
        return userRepo.updateDependencies(id);
    }
}
// TestUserService.java
...@Test
public void testProxyBubuDeps() {
    int result = userService.updateDependencies(3L);
    assertEquals(0, result);
}
```

If the test `testProxyBubuDeps` method is executed, in the console only one line can be seen:

```
INFO c.p.a.UserRepoMonitor - ---> BUBU when calling:
        com.ps.repos.impl.JdbcTemplateUserRepo.updateDependencies
```

Now we can be sure that both methods are matching the pointcut expression.

What really happens here is that the advice method does its thing and then the target method is called. The target method executes, and it calls the `updateDependencies` method on the target object. The proxy is not involved in this call in any way, so that is why it cannot apply any advice. The UML sequence diagram in Figure 4-14 should make things even clearer.

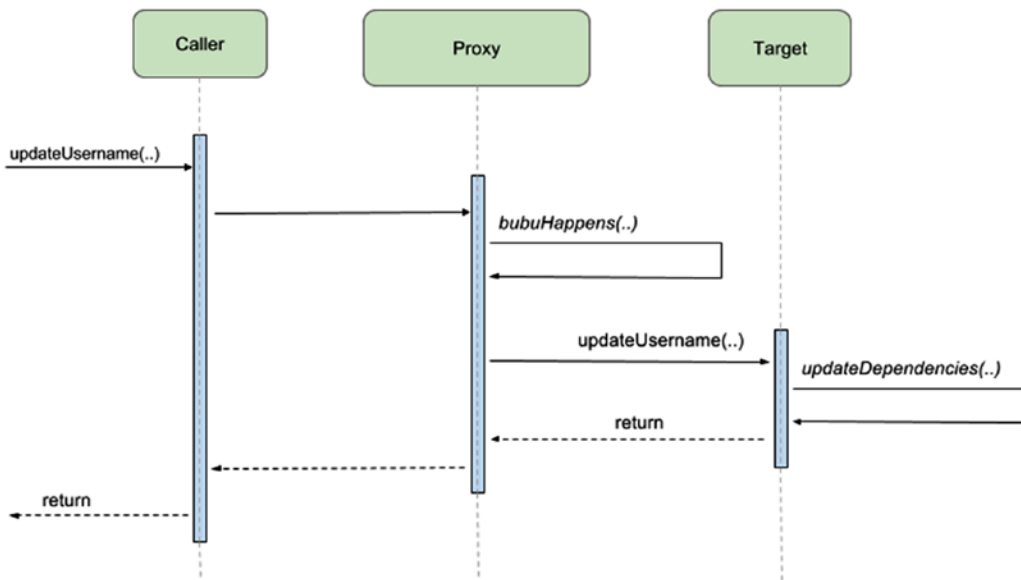


Figure 4-14. The proxy nature explained via UML sequence diagram

The code snippets used to explain this specific scenario are part of the 04-ps-aop-solution.

Summary

After reading this chapter, you should possess enough knowledge to write and configure aspects with Spring AOP. Keep in mind the following:

- Cross-cutting concerns cause code scattering and tangling.
- AOP is a programming type that can help reduce code scattering and tangling by modularizing cross-cutting concerns.
- The most common cross-cutting concerns.
- AOP Concepts: aspect, advice, join point, pointcut, AOP proxy, etc.
- How aspect support is configured in a Spring application.
- How many advice types there are.

Quick Quiz

Question 1: Which options in the list below represent a cross-cutting concern?(choose all that apply)

- connecting to the database
- caching
- security
- transactions

Question 2: Which options in the list are used to declare advice in Spring AOP? (choose all that apply)

- A. @Aspect
- B. @Before
- C. @Pointcut
- D. @AfterReturning
- E. @After

Question 3: What is true about the after advice?(choose all that apply)

- A. it is not executed if the target method execution ends by throwing an exception
- B. it can catch the target method exception and prevent propagation
- C. it can catch the target method exception and throw a different exception

Question 4: Which methods will match the following pointcut expression? (choose all that apply)
`execution(protected * com.ps.repos.*.*Repo.find*(..))`

- A. none: Spring AOP supports only advising public methods
- B. `com.ps.repos.impl.JdbcTemplateUserRepo.findById(Long id)`
- C. `com.ps.repos.impl.JdbcUserRepository.findAll()`

Question 5: Which of the following pointcut expressions match the following method definition?
 Method definition: a method named `update` that has the first parameter of type `Long` and more parameters may follow

- A. `execution(update(*))`
- B. `execution(* update(Long, *))`
- C. `execution(* update(..))`
- D. `execution(* update(Logn,..))`

Question 6: What is the XML equivalent of `@EnableAspectJAutoProxy` ?

- A.

```
<aop:aspectj-autoproxy>
  <aop:aspect base-package="..."/>
</aop:aspectj-autoproxy>
```
- B.

```
<aop:aspectj-autoproxy>
  <aop:include name="..."/>
</aop:aspectj-autoproxy>
```
- C.

```
<aop:config>
  <aop:aspect ref="...">
</aop:config>
```

Question 7: Which of the following is true about Spring AOP proxies?

- A. A proxy object must implement the interface that the target implements or be a subclass of the target's type.
- B. A proxy object has scope prototype.
- C. Spring AOP uses subclass-based proxies by default.

Question 8: What is a pointcut?

- A. a parameter that every advice method must specify in its signature that provides access to the execution context
- B. an expression to identify methods to which the advice applies to
- C. a predicate used to identify join points.

Practical Exercise

In the source code for this book there is a project called 04-ps-aop-practice. This project can be used to test your understanding of Spring AOP. This project contains part of the implementation depicted in the code snippets. The parts missing are marked with a TODO task and are visible in IntelliJ IDEA in the TODO view. There are six tasks for you to solve in order to test your acquired knowledge of Spring AOP.

Task TODO 20 requires you to enable aspect support by modifying the `AppConfig` class. If you want to use JDK proxies or CGLIB, it is up to you. The CGLIB dependency is on the classpath and can be used.

Task TODO 21 requires you to declare the `UserRepoMonitor` as an aspect.

Tasks 22–26 require that you configure methods declared inside the `UserRepoMonitor` class as different types of advice. The pointcuts expressions are your responsibility as well.

The tests that can be used to verify that your advices work, are declared in the class `TestUserService`. Each test method is commented with the type of advice it tests. Since some of the pointcut expressions might match more than one method, all advices that apply will be executed. To stop that from happening, when working on configuring an advice, comment the annotations on the others.

To run a test case, just click anywhere on the class content, or on the class name in the project view and select the Run `'{TestClassName}'` option. If you want to run a single test, just right click and from the menu select Run `'{TestMethodName}'`. These instructions were already explained to you in the practice section of **Chapter 3: Testing**. You might need to review it.

After you have resolved all the TODOs, you should be able to run the full test suite, and you should see something similar to what is depicted in Figure 4-15.

```

TestUserService (3)
/Library/Java/JavaVirtualMachines/jdk1.8.0_92_jdk/Contents/Home/bin/java ... All 6 tests passed - 2s 148ms
TestUserService (com.ps.config) 2s 148ms
  testBeforeUpdatePass 18ms
  testAll 1s 37ms
  testAfterUpdatePass 9ms
  testAfterUpdateUsernameGood 1ms
  testFindById 1s 7ms
  testAfterUpdateUsernameBad 76ms
00:07:31.143 [main] INFO c.p.a.UserRepoMonitor -> Target object class com.ps.services.UserServiceImpl
00:07:31.162 [main] INFO c.p.a.UserRepoMonitor -> Intercepting call of: findAll
00:07:32.187 [main] INFO c.p.a.UserRepoMonitor -> Execution of findAll took: 1 ms.
00:07:32.188 [main] INFO c.p.a.UserRepoMonitor -> Target object class com.ps.services.UserServiceImpl
00:07:32.189 [main] INFO c.p.a.UserRepoMonitor -> Method com.ps.repos.impl.JdbcTemplateUserRepo.updatePassword is about to be called
00:07:32.198 [main] INFO c.p.a.UserRepoMonitor -> Target object class com.ps.services.UserServiceImpl
00:07:32.198 [main] INFO c.p.a.UserRepoMonitor -> Method com.ps.repos.impl.JdbcTemplateUserRepo.updateUsername is about to be called
00:07:32.199 [main] INFO c.p.a.UserRepoMonitor -> Method updateUsername was called 1 times.
00:07:32.200 [main] INFO c.p.a.UserRepoMonitor -> Intercepting call of: findById
00:07:33.206 [main] INFO c.p.a.UserRepoMonitor -> Execution of findById took: 1 ms.
00:07:33.207 [main] INFO c.p.a.UserRepoMonitor -> Target object class com.ps.services.UserServiceImpl
00:07:33.208 [main] INFO c.p.a.UserRepoMonitor -> Method com.ps.repos.impl.JdbcTemplateUserRepo.updateUsername is about to be called
00:07:33.282 [main] INFO c.p.a.UserRepoMonitor -> Method updateUsername was called 2 times.
00:07:33.282 [main] INFO c.p.a.UserRepoMonitor -> Update method com.ps.services.UserServiceImpl.updateUsername failed. Existing username found.
Process finished with exit code 0
  
```

Figure 4-15. Test suite for the Aspects section with all tests passing

If you have difficulties solving the TODOs, you can take a peek at `04-ps-aop-solution`.

! In case you are still hungry for more information, there are two blog entries on the Spring official blog. One is about proxies

<https://spring.io/blog/2012/05/23/transactions-caching-and-aop-understanding-proxy-usage-in-spring>

and one is about named pointcuts

<https://spring.io/blog/2007/03/29/aop-context-binding-with-named-pointcuts>

CHAPTER 5



Data Access

Software applications usually handle sets of data that must be stored in an organized manner so they can be easily accessed, managed, and updated. The fact that data can be made to persist means that it can be available even when the application is down. So storage is decoupled from the rest of the application. The most common way of organizing data for storage is a database. Any storage setup that allows for data to be organized in such a way that can be queried and updated represents a database. The most widely known and used databases nowadays are relational databases (such as MySQL, PostgreSQL, and Oracle) and key-value databases, also known as NoSQL. Of course, XML files can be used as a database as well, but only for small applications, because using XML files for data storage implies that all data must be loaded into memory in order to be managed.

From an architectural point of view, software applications are multilayered or multitiered, and the database is the base layer, where data gets stored. The interface between the base layer (also called the infrastructure layer) and the rest of the application is the data access layer, also known as the repository layer. On top of this, there is the logic (or service) layer. This layer transforms user data and prepares it for storage or transforms database content into data proper for display in the user interface. On top of the service layer there is the presentation (or interface) layer, which is responsible for receiving commands and data and forwarding them for processing to the service layer or displaying results. And across all layers, as mentioned in the previous chapter, are the cross-cutting concerns, as depicted in Figure 5-1.

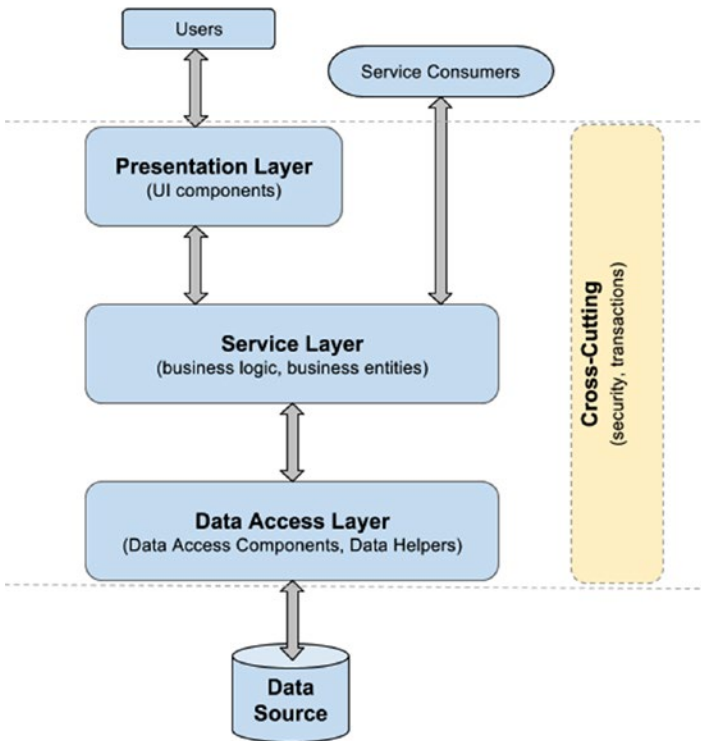


Figure 5-1. Typical software application architecture

The database is a central resource in a software application, and thus it can be a source of bottlenecks. That is why accessing a database to perform typical operations like searching for data, creating data, and deleting data need to be managed properly. Spring supports many database implementations and can be integrated with most frameworks that provide efficient data access management (Hibernate, EclipseLink, or MyBatis.¹) This chapter is focused on how Spring can be used to manage database access, and the code samples cover service, data layer, and transaction-cutting concerns.

¹My Batis was previously known as iBatis, before Apache dropped it. More information on the new official site: <http://blog.mybatis.org/>

Basic Data Access Using JDBC

To perform data operations, a connection to the database is needed. The most basic way to get a connection in Java is to use JDBC. But the basic way is also cumbersome, since the developer has to write the code to close the connection and any other stream-based objects used during the processing. Because a connection is not enough, transporting data between the application and the database requires additional objects to contain the data in a format that can be read. Any objects that build on top of a connection, such as prepared statement and result set objects, must be safely discarded as well. This leads to ugly cluttered code that is difficult to maintain, difficult to test, and difficult to write. To access a database using JDBC, the following things are needed:

- a database driver used to communicate with the database;
- a connection url that is the entry point for the communication;
- database credentials (usually user and password).

Using the previous information, a `javax.sql.DataSource` object is created. This is a Java interface that must be implemented by every `DataSource` class used to access a database, and each database driver library contains its own implementation.

Example: the class used to provide access to an oracle database is `oracle.jdbc.pool.OracleDataSource`. To interact with the database using JDBC, the following steps must be implemented:

- The `DataSource` is used to open a connection to the database. The connection object is of a type specific to the database that implements `java.sql.Connection`.
- To extract or save data to the database, a statement instance of type implementing `java.sql.PreparedStatement` is created based on the connection.
- After the statement has been executed, the results are stored in an object of type implementing `java.sql.ResultSet`.

Just look at the following example, which uses JDBC to extract all users from a database:

```
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
...
public Set<User> findAll() {
    Set<User> userSet = new HashSet<>();
    String sql = "select u.ID as ID, u.USERNAME as USERNAME," +
                "u.EMAIL as EMAIL," +
                "u.PASSWORD as PASSWORD from P_USER u ";

    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        ps = conn.prepareStatement(sql);
        rs = ps.executeQuery();
```

```

        userSet = mapUsers(rs); // (*)
    } catch (SQLException e) {
        throw new RuntimeException("Unexpected problem when"+
            " extracting users from database!", e);
    } finally {
        if (rs != null) {
            try {
                rs.close();
            } catch (SQLException ex) {
                ...
            }
        }
        if (ps != null) {
            try {
                ps.close();
            } catch (SQLException ex) {
                ...
            }
        }
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException ex) {
                ...
            }
        }
    }
    return userSet;
}

```

The code above opens a connection, extracts the data, processes the results, then closes the connection. Each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. And another complication here is that domain objects (entities) that are Java objects corresponding to database record conversion must also be implemented, because JDBC does not know how to do this on its own. In the example above, notice the line marked with (*). This is done by the `mapUsers`, which is depicted in the code snippet below:

```

private Set<User> mapUsers(ResultSet rs) throws SQLException {
    Set<User> userSet = new HashSet<>();
    User user = null;
    while (rs.next()) {
        user = new User();
        // set internal entity identifier (primary key)
        user.setId(rs.getLong("ID"));
        user.setUsername(rs.getString("USERNAME"));
        user.setEmail(rs.getString("EMAIL"));
        user.setPassword(rs.getString("PASSWORD"));
        userSet.add(user);
    }
    return userSet;
}

```

Basically, the `ResultSet` object is converted to one or more `User` objects depending on the number of results returned by the statement execution. And as food for thought, imagine how problematic the handling of `Timestamp` and `Date` values becomes with the method above.

What happens if a query is not executed correctly? A `java.sql.SQLException` is thrown, and the developer has to write the appropriate code to treat it, of course, because it is a checked exception. The traditional JDBC usage is redundant, prone to error, poor in exception handling, and overall cumbersome to use.

Spring Data Access

With Spring, there is no need to write basic traditional JDBC code. The Spring framework provides components that were created to remove the necessity of “manually” handling database connections. In writing enterprise applications (using Spring or not), classes called repositories are used to communicate with the database. In Figure 5-2, you can see the different types of repository classes that can be used and the persistence framework used as a bridge to the database.

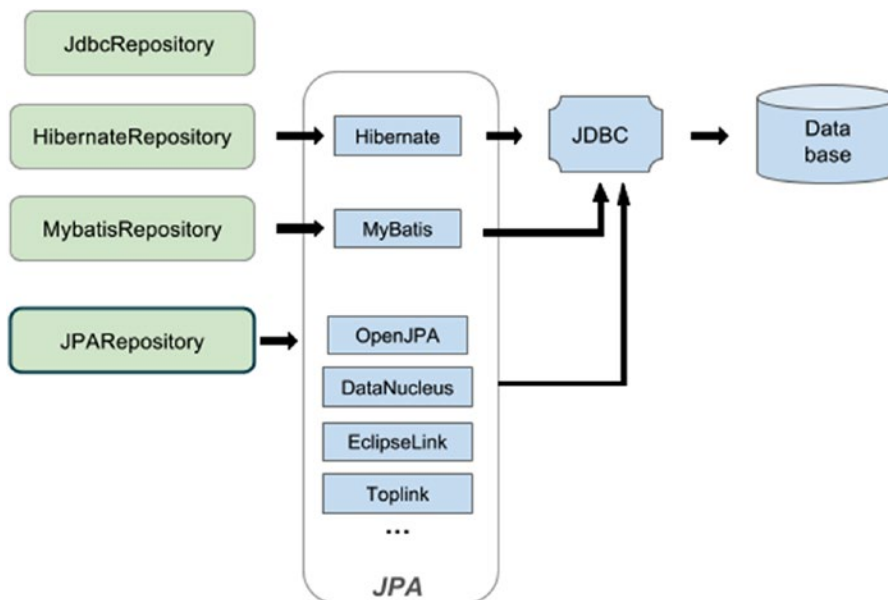


Figure 5-2. How repository classes interact with the database

As mentioned earlier, data storage is a resource. Access to resources has to be managed properly to avoid bottlenecks. In software, the term *bottleneck* refers to a situation in which multiple processes require access to a resource, and if the access to the resource is not managed properly, some processes end up never having access to the resource. Using pure JDBC access to the resource, the database is handled by the developer explicitly. Spring comes in, together with persistence frameworks, to unburden the developer of this responsibility. So, let’s begin with the most basic way of accessing data using Spring: using a JDBC repository class.

Introducing JdbcTemplate

The Spring Framework's JDBC abstraction framework consists of four main packages: `core`, `datasource`, `object`, and `support`.² There is also a small `config` package containing two infrastructure beans used in the configuration of a Spring JDBC environment. Spring simplifies the code that needs to be written in using JDBC to interact with a database by introducing the `org.springframework.jdbc.core.JdbcTemplate` class, which is part of the `core` package. This class hides large amounts of JDBC boilerplate code and unburdens the developer of connection management, since database connections are acquired and released automatically. The `datasource` package contains utility classes for `DataSource` management and a few simple `DataSource` implementations that can be used for testing. The `object` package contains classes that represent RDBMS queries, updates, and stored procedures as thread-safe, reusable objects. The `support` package provides `SQLException` translation functionality and some utility classes. Exception handling is covered, providing clear exceptions with root causes always being reported and also making sure that resources are always released properly, even in case of failures. The `JdbcTemplate` class is designed according to the *template method pattern*, which is a behavioral design pattern that defines the skeleton of an algorithm, the effective implementation being defined later or delegated to subclasses.³

The simplest example that can be used to demonstrate how the `JdbcTemplate` can be used will be covered in this section. For this, we need the following:

- a configuration file containing the database details, let's call it *db.properties*:

```
#db.properties contents
# For running application with H2 in memory database
driverClassName=org.h2.Driver
url=jdbc:h2:~/sample
username=sample
password=sample
```

- configuration files containing SQL queries for initializing the database structure: *schema.sql* containing SQL code for creating tables and *test-data.sql* containing insert statements that populate the tables with some initial data. The content of the files is not relevant for this chapter, but they are part of the `05-ps-jdbc-practice` project.
- a configuration class for the test database, let's call it `TestDataConfig`. This configuration class will contain all the beans necessary to create a `DataSource` bean. We will be using an in-memory database called H2. The following configuration seems complicated, but this offers the possibility to define the connection URL and the credentials to access the database. At this point, the *db.properties* configuration file can just be replaced with a file describing a different type of database, and the rest of the application will still work. The `DataSource` bean is of type `SimpleDriverDataSource`, which is a simple Spring-specific implementation of `javax.sql.DataSource`. Two specialized Spring beans are used to initialize the `DataSource` bean, `DataSourceInitializer`, and to populate the `DataSource` bean, `DatabasePopulator`.

²More information on the official Spring Reference page: <http://docs.spring.io/spring/docs/4.3.2.RELEASE/spring-framework-reference/htmlsingle/#jdbc-packages>.

³Other Spring classes to access a different type of resource follow the same pattern: *JmsTemplate*, *RestTemplate*, etc.

```

package com.ps.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.*;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;
import org.springframework.core.io.Resource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.DatabasePopulator;
import org.springframework.jdbc.datasource.init.DatabasePopulatorUtils;

import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;

@Configuration
@Profile("dev")
@PropertySource("classpath:db/db.properties")
public class TestDataConfig {

    @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);
            DatabasePopulatorUtils.execute(databasePopulator(), dataSource);
            return dataSource;
        } catch (Exception e) {
            return null;
        }
    }

    @Value("classpath:db/schema.sql")

```

```

private Resource schemaScript;

@Value("classpath:db/test-data.sql")
private Resource dataScript;

@Bean
public DataSourceInitializer dataSourceInitializer
    (final DataSource dataSource) {
    final DataSourceInitializer initializer = new DataSourceInitializer();
    initializer.setDataSource(dataSource);
    initializer.setDatabasePopulator(databasePopulator());
    return initializer;
}

private DatabasePopulator databasePopulator() {
    final ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScript(schemaScript);
    populator.addScript(dataScript);
    return populator;
}

@Bean
public JdbcTemplate userJdbcTemplate() {
    return new JdbcTemplate(dataSource());
}
}

```

The configuration class probably looks familiar to you, since it was already introduced in Chapter 2: *Bean Lifecycle and Configuration*. There is an extra bean declaration here, though: `dataSourceInitializer`. This bean uses an instance of `DatabasePopulator` to read scripts necessary to create and populate the `P_USER` and the `P_PET` tables. The `db/schema.sql` file contains SQL DDL statements that define the structure of the tables. The `db/test-data.sql` contains SQL DML insert statements that are executed by the `databasePopulator` component to insert data into the previously created table.

The simple version, using an embedded database, looks like the following code snippet and uses Spring JDBC support components like `EmbeddedDatabaseBuilder` to create a lightweight in-memory database.

```

package com.ps.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@Profile("dev")
public class TestDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()

```



```

        .setType(EmbeddedDatabaseType.H2)
        .addScript("classpath:db/schema.sql")
        .addScript("classpath:db/test-data.sql")
        .build();
    }

    @Bean
    public JdbcTemplate ~underlinejdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}

```

! An equivalent configuration can be set up using XML and the JDBC namespace. For an embedded in-memory database with no concern for credentials or connection url, the `<jdbc:embedded-database .../>` configuration can be used:

```

<?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: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/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">
    <!-- Creates an in-memory "sample" database
         populated with test data for fast testing -->
    <jdbc:embedded-database id="dataSource">
        < jdbc:script location="classpath:db/schema.sql"/>
        <jdbc:script location="classpath:db/test-data.sql"/>
    </jdbc:embedded-database>
</beans>

```

A more complex configuration that allows for credentials to be read from an external configuration file can be set up as well using the `jdbc:initialize-database .../>` element and a `SimpleDriverDataSource` bean:

```

<?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: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/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util.xsd">

```

```

<util:properties id="dbProp" location="classpath:db/db.properties"/>

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
  <property name="driverClass" value="#{dbProp.driverClassName}"/>
  <property name="url" value="#{dbProp.url}"/>
  <property name="username" value="#{dbProp.username}"/>
  <property name="password" value="#{dbProp.password}"/>
</bean>
<jdbc:initialize-database data-source="dataSource">
  <jdbc:script location="classpath:db/schema.sql"/>
  <jdbc:script location="classpath:db/test-data.sql"/>
</jdbc:initialize-database>
</beans>

```

- an application configuration class declaring the repository bean to be tested, let's call it AppConfig and define it like this:

```

@Configuration
@ComponentScan(basePackages = {"com.ps.repos.impl"})
public class AppConfig {
}

```

- a repository bean that uses a bean of type JdbcTemplate to access the database: JdbcTemplateUserRepo

```

package com.ps.repos.impl;
import com.ps.ents.User;
// interface to define User specific methods
// to be implemented by the repository
import com.ps.repos.UserRepo;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashSet;
import java.util.Set;

@Repository("userTemplateRepo")
public class JdbcTemplateUserRepo implements UserRepo {

    private RowMapper<User> rowMapper = new UserRowMapper();

    protected JdbcTemplate jdbcTemplate;

    @Autowired
    public JdbcTemplateUserRepo(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}

```

```

@Override
public User findById(Long id) {
    String sql = "select id, email, username, password from p_user where id= ?";
    return jdbcTemplate.queryForObject(sql, rowMapper, id);
}

... // other repository methods

/**
 * Maps a row returned from a query executed
 * on the P_USER table to a com.ps.ents.User object.
 */
private class UserRowMapper implements RowMapper<User> {

    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        Long id = rs.getLong("ID");
        String email = rs.getString("EMAIL");
        String username = rs.getString("USERNAME");
        String password = rs.getString("PASSWORD");
        User user = new User();
        user.setId(id);
        user.setUsername(username);
        user.setEmail(email);
        user.setPassword(password);
        return user;
    }
}
}

```

In a software application, data is managed as objects that are called *entities* or *domain objects*. The entity classes are written by the developer, and their fields match the columns in a database table. The `UserRowMapper` internal class depicted in the previous code snippet is used to do just that, and it will be covered in more detail a little bit later in this section.

- a test class: `TestJdbcTemplateUserRepo`

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class TestJdbcTemplateUserRepo {

    @Autowired
    @Qualifier("userTemplateRepo")
    UserRepo userRepo;

    @Before
    public void setUp() {
        assertNotNull(userRepo);
    }
}

```

```

@Test
public void testFindById() {
    User user = userRepo.findById(1L);
    assertEquals("John", user.getUsername());
}

@Test
public void testNoFindById() {
    User user = userRepo.findById(99L);
    assertEquals("John", user.getUsername());
}
}

```

This test class is a simple one that searches for a user object by its ID. The `testFindById` method passes, the `testNoFindById` test does not, and this method will be covered in detail in the section about Spring data access exceptions. Here, the method is incomplete, missing the expected attribute, because it is out of scope at the moment.

To create a `JdbcTemplate` bean, a `dataSource` is needed to access database records. Every repository bean will have this bean as a dependence, and it will use it to manipulate data when `JdbcTemplate` bean methods are called, as in the following code snippet:

```
String sql = "select id, email, username,password from p_user where id= ?";
jdbcTemplate.queryForObject(sql, rowMapper, id);
```

The next sequence of steps is taken by Spring in the background:

1. A database connection is acquired.
2. A transaction is acquired or created so the statement can be executed in its context if in a transactional environment. In the previous case, there is no transaction support, defined declarative using `@Transactional` or programmatic. Both will be introduced later in the chapter.
3. The statement is executed.
4. The `ResultSet` is processed and transformed into entity object(s) using `RowMapper` instances.
5. Exceptions are handled.
6. The database connection is released.

What we are interested in when running methods of this test class is what happens in the background, and this is because there is no code needed to be developed for connection management. It was already mentioned that Spring takes care of that, but it would be nice to actually have a proof, right? To be able to see this happening, the `logback-test.xml` file can be customized to show the Spring JDBC detailed log by adding the following line:

```
<logger name="org.springframework" level="debug"/>
```

Before we look at the log produced when the test class is run, the location of the files mentioned until now must be covered, and you can see it in [Figure 5-3](#).

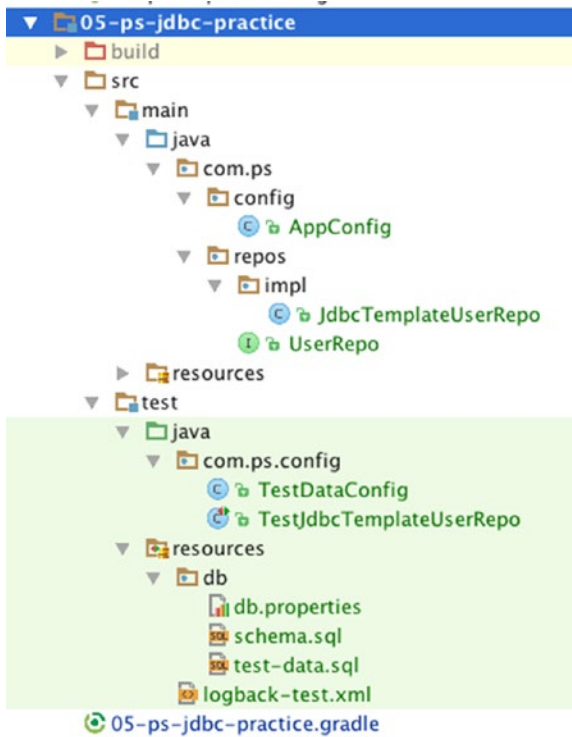


Figure 5-3. Location of the files in the **05-ps-jdbc-practice** project

The project has the typical Java project structure consecrated by Maven, and the *.sql, *.xml, *.properties are resource files, so they are stored in the project in the resources directory from the test or main module. By running the `TestJdbcTemplateUserRepo.testFindById()` method with the logging setup mentioned previously, if you scan the log, you will find the following lines:

```
#booting up environment
... creating beans ...
DEBUG o.s.j.d.DataSourceUtils - Fetching JDBC Connection from DataSource
DEBUG o.s.j.d.SimpleDriverDataSource - Creating new JDBC Driver Connection
    to [jdbc:h2:sample]
DEBUG o.s.j.d.i.ScriptUtils - Executing SQL script from class path
    resource [db/schema.sql]
... listing SQL scripts ...
INFO o.s.j.d.i.ScriptUtils - Executed SQL script from class path
    resource [db/schema.sql] in 23 ms.
INFO o.s.j.d.i.ScriptUtils - Executing SQL script from class path
    resource [db/test-data.sql]
... listing SQL scripts ...
INFO o.s.j.d.i.ScriptUtils - Executed SQL script from class path
    resource [db/test-data.sql] in 5 ms.
DEBUG o.s.j.d.DataSourceUtils - Returning JDBC Connection to DataSource
#executing the test
```

```

DEBUG o.s.j.c.JdbcTemplate - Executing prepared SQL query
DEBUG o.s.j.c.JdbcTemplate - Executing prepared SQL statement
    [select id, email, username,password from p_user where id= ?]
DEBUG o.s.j.d.DataSourceUtils - Fetching JDBC Connection from DataSource
DEBUG o.s.j.d.SimpleDriverDataSource - Creating new JDBC Driver Connection
    to [jdbc:h2:sample]
DEBUG o.s.j.d.DataSourceUtils - Returning JDBC Connection to DataSource
...

```

Basically, every time a `JdbcTemplate` method is called, a connection is automatically opened for query execution, and then the same connection is released and all the magic is done by the Spring utility class `org.springframework.jdbc.datasource.DataSourceUtils`. This class is abstract and it is used internally by `JdbcTemplate` and by transaction manager classes that will be covered in the following sections.

Using Spring's `JdbcTemplate` as depicted so far is suitable for small applications designed for academic use. Use it whenever you need to get rid of traditional JDBC. In an enterprise application, transactions will most likely be needed, so more Spring components must be added to the mix. Working with transactions will be covered shortly, but before that, let's see what happens when an unexpected event takes place during communication with a database using a `JdbcTemplate` bean.

Querying with JdbcTemplate

The `JdbcTemplate` provides the developer a multitude of methods to query for entities (also called domain objects), generic maps, and lists and simple types (long, int, String). `JdbcTemplate` methods use queries containing the '?' placeholder for parameters and variables that are bound to it. It makes use of the advantage of varargs and autoboxing to simplify `JdbcTemplate` use. But simple queries, without any bind variables, can be used too. The following example uses a query to count all the users in the `p_user` table and returns an integer.

```

public int countUsers() {
    String sql = "select count(*) from p_user";
    return jdbcTemplate.queryForObject(sql, Integer.class);
}

```

The method `queryForObject` used here does not need a `RowMapper` object, since the result is a number of type `Integer`. The type of the result is provided as argument to the method as well. This method has replaced, in Spring version 3.2.2, specialized methods like `queryForInt` and `queryForLong`. Examples of how these methods were used until 3.2.2 (when they became deprecated) can be seen in the following code snippet:

```

public int countUsers() {
    String sql = "select count(*) from p_user";
    return jdbcTemplate.queryForInt(sql);
}

public Long findIdByUsername(String username) {
    String sql = "select id from p_user where username = ?";
    return jdbcTemplate.queryForLong(sql, username);
}

```

Sometimes there might be a need to extract a record from a table in a different format from an entity. `JdbcTemplate` provides a method called `queryForMap` that extracts a row from the `ResultSet` as a `Map<String, Object>`.

```
public Map<String, Object> findByIdAsMap(Long id) {
    String sql = "select * from p_user where id= ?";
    return jdbcTemplate.queryForMap(sql, id);
}
```

The map returned by this method contains pairs of column [name, column value]. Example:

```
{
    ID=1,
    FIRST_NAME=null,
    USERNAME=John,
    LAST_NAME=null,
    PASSWORD=test,
    ADDRESS=null,
    EMAIL=john@pet.com
}
```

The `queryForList` does the same as the previous methods but for `ResultSet` instances that contain multiple rows. This method returns a `List` of `Map<String, Object>` objects, each map containing a row from the `ResultSet`.

```
public List<Map<String, Object>> findAllAsMaps() {
    String sql = "select * from p_user";
    return jdbcTemplate.queryForList(sql);
}
```

Sample output:

```
[
    {
        ID=1,
        FIRST_NAME=null,
        USERNAME=John,
        LAST_NAME=null,
        PASSWORD=test,
        ADDRESS=null,
        EMAIL=john@pet.com
    },
    {
        ID=2,
        FIRST_NAME=null,
        USERNAME=Mary,
        LAST_NAME=null,
        PASSWORD=test,
        ADDRESS=null,
        EMAIL=mary@pet.com
    }
]
```

These two methods can be used for testing and auditing.

The first example using `JdbcTemplate` returns an entity object (also called domain object). To transform a table record into a domain object, Spring provides the `org.springframework.jdbc.core.RowMapper<T>` interface. This should be implemented for every entity type in an application, because an object of the implementing type is required as a parameter for `JdbcTemplate` methods that return entities. `RowMapper` objects are usually stateless, so creating one per repository class and using it as a converter from table records to Java objects is a good practice.

Spring converts contents of a `ResultSet` into domain objects using a callback approach. The `JdbcTemplate` instance first executes the query and populates the `ResultSet`; then the `mapRow` method of the `RowMapper` instance used as argument is called. In case the `ResultSet` contains more than one table row, the method is called for each row.

```
@Repository("userTemplateRepo")
public class JdbcTemplateUserRepo implements UserRepo {

    private RowMapper<User> rowMapper = new UserRowMapper();

    protected JdbcTemplate jdbcTemplate;

    public Set<User> findAll() {
        String sql = "select id, username, email, password from p_user";
        return new HashSet<>(jdbcTemplate.query(sql, rowMapper));
    }
    ...

    class UserRowMapper implements RowMapper<User> {

        public User mapRow(ResultSet rs, int rowNum) throws SQLException {
            Long id = rs.getLong("ID");
            String email = rs.getString("EMAIL");
            String username = rs.getString("USERNAME");
            String password = rs.getString("PASSWORD");
            User user = new User();
            user.setId(id);
            user.setUsername(username);
            user.setEmail(email);
            user.setPassword(password);
            return user;
        }
    }
}
```

The interface is generic, and the type of the domain object that the repository manages should be used as parameter. Usually, it is declared as an internal class of a repository, but when more than one repository manages the same type of domain object, it can be declared as a top-level class. However, as you can notice, the code to transform a `ResultSet` into a domain object, or a collection of them, has to be written by the developer. This too will no longer be necessary when ORM (object relational mapping) is used. But more about that later.

So far, only `JdbcTemplate` methods that return some kind of result have been covered. An example for an execution that does not return a result is appropriate right about now. The most important method in the `JdbcTemplate` class that executes queries but returns no result is named `query`. This method is polymorphic,⁴ and the method signature differs depending on the purpose of the query result.

⁴*Polymorphism* is one of the object oriented programming principles. The term is of Greek etymology and means one name, many forms. Polymorphism manifests itself in software by having multiple methods all with the same name but slightly different functionalities.

Spring provides the `org.springframework.jdbc.core.RowCallbackHandler` interface, which can be implemented to stream the rows returned by the query to a file, to convert them to XML, or to filter them before adding them to a collection. An instance of this type is provided to the query method along with query parameters if necessary, and the `JdbcTemplate` will use it accordingly. In the code snippet below, the code of this interface is depicted.

```
public interface RowCallbackHandler {
    void processRow(ResultSet rs) throws SQLException;
}
```

The implementation of the method is used by the `JdbcTemplate` to process the `ResultSet` row as implemented by the developer. The exceptions are caught and treated by the `JdbcTemplate` instance. In the following code snippet, the `HTMLUserRowCallbackHandler` is used by the `JdbcTemplate` instance to extract all users named “John” from the `p_user` table, write the resulting rows in HTML, and print them to the console.

```
...
import org.springframework.jdbc.core.RowCallbackHandler;

public class JdbcTemplateUserRepo implements UserRepo {

    private class HTMLUserRowCallbackHandler
        implements RowCallbackHandler {
        private PrintStream out;

        public HTMLUserRowCallbackHandler(PrintStream out) {
            this.out = out;
        }

        @Override
        public void processRow(ResultSet rs)
            throws SQLException {
            StringBuilder htmlSb =
                new StringBuilder("<p>user ID: " + rs.getLong("ID") + "</p></br>\n")
                .append("<p>username: " + rs.getString("USERNAME") + "</p></br>\n")
                .append("<p>email: " + rs.getString("EMAIL") + "</p></br>");
            out.print(htmlSb.toString());
        }
    }

    public void htmlAllByName(String name) {
        String sql = "select id, username, email from p_user where username = ?";
        jdbcTemplate.query(sql, new HTMLUserRowCallbackHandler(System.out), name );
    }
    ...
}
```

When running this method, among all the Spring logs at the end we will see the following too in the console:

```
<p>user ID: 1</p></br>
<p>username: John</p></br>
<p>email: john@pet.com</p></br>
```

If the `ResultSet` contains more than one row, the `JdbcTemplate` instance will process each of them using the `HTMLUserRowCallbackHandler.processRow` method.

Spring also provides the option of processing a full `ResultSet` at once and transforming it into an object, via the `org.springframework.jdbc.core.ResultSetExtractor<T>`. This capability is very useful when the results are extracted from more than one table, but must be treated in the application as a single object. The developer is responsible of iterating the `ResultSet` and setting the properties of the object with other objects mapped to entries in the database. This interface is parametrizable by type.

```
public interface ResultSetExtractor<T> {
    T extractData(ResultSet rs) throws SQLException, DataAccessException;
}
```

The following code snippet depicts the usage of `UserWithPetsExtractor` to extract a user by id from the `p_user` table with the pets from the `p_pet` table and create an object of type `User` that contains the populated list of `Pet` objects linked to the user.

```
...
import org.springframework.jdbc.core.ResultSetExtractor;

@Repository("userTemplateRepo")
public class JdbcTemplateUserRepo implements UserRepo {
    ...
    @Override
    public User findByIdWithPets(Long id) {
        String sql = "select u.id id," +
            " u.username un," +
            " u.email email, " +
            "p.id pid, " +
            "p.name pname," +
            " p.age age," +
            " p.pet_type pt" +
            " from p_user u, p_pet p where u.id=p.owner and u.id=" + id;
        return jdbcTemplate.query(sql, new UserWithPetsExtractor());
    }

    private class UserWithPetsExtractor implements
        ResultSetExtractor<User> {
        @Override
        public User extractData(ResultSet rs) throws SQLException,
            DataAccessException {
            User user = null;
            while (rs.next()) {
                if (user == null) {
                    user = new User();
                    // set internal entity identifier (primary key)
                    user.setId(rs.getLong("ID"));
                    user.setUsername(rs.getString("UN"));
                    user.setEmail(rs.getString("EMAIL"));
                }
                Pet p = new Pet();
                p.setId(rs.getLong("PID"));
                p.setName(rs.getString("PNAME"));
                p.setAge(rs.getInt("AGE"));
                p.setPetType(PetType.valueOf(rs.getString("PT")));
                user.addPet(p);
            }
        }
    }
}
```

```

        return user;
    }
}

// test method for findByIdWithPets()
@Test
public void testExtractor(){
    User user = userRepo.findByIdWithPets(1L);
    assertEquals(2, user.getPets().size());
}

```

This is pretty much all you need to know about `JdbcTemplate`, so you can use it properly in your code. To summarize:

- `JdbcTemplate` works with queries that specify parameters using the '?' placeholder.
- Use `queryForObject` when it is expected that execution of the query will return a single result.
- Use `RowMapper<T>` when each row of the `ResultSet` maps to a domain object.
- Use `RowCallbackHandler` when no value should be returned.
- Use `ResultSetExtractor<T>` when multiple rows in the `ResultSet` map to a single object.

Querying with NamedParameterJdbcTemplate

Besides the `JdbcTemplate` class, Spring provides another template class for executing queries with named parameters:

`org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate`. This class provides methods analogous to `JdbcTemplate` that require as a parameter a map containing a pair of named parameters and their values.⁵ Once the named parameters have been replaced with the values, the call is delegated behind the scenes to a `JdbcTemplate` instance. The relationship between the two classes is depicted in Figure 5-4.

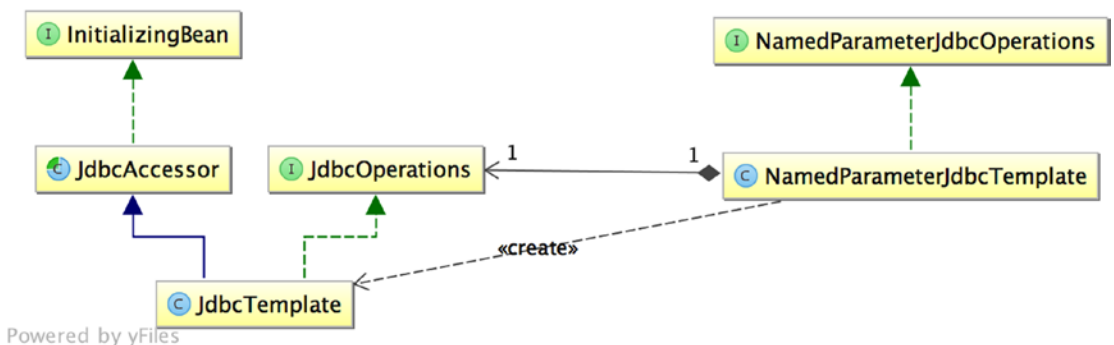


Figure 5-4. Relationship between `JdbcTemplate` and `NamedParameterJdbcTemplate`

⁵This approach of providing parameters is used by JPA and Hibernate as well.

If you look at the code of the `NamedParameterJdbcTemplate`,⁶ you will see the following:

```
...
public class NamedParameterJdbcTemplate
    implements NamedParameterJdbcOperations {
    ...
    /** The JdbcTemplate we are wrapping */
    private final JdbcOperations classicJdbcTemplate;
    ...
    /**
     * Expose the classic Spring JdbcTemplate to allow invocation of
     * less commonly used methods.
     */
    @Override
    public JdbcOperations getJdbcOperations() {
        return this.classicJdbcTemplate;
    }
    ...
}
```

The `JdbcTemplate` is therefore exposed to the developer via the `getJdbcOperations`, and traditional `JdbcTemplate` can be called as well.

In the code snippet below, you can compare the `findById` method written with the two types of JDBC template classes:

```
//using JdbcTemplate
public User findById(Long id) {
    String sql = "select id, email, username,password from p_user where id= ?";
    return jdbcTemplate.queryForObject(sql, rowMapper, id);
}

//using NamedParameterJdbcTemplate
public User findById(Long id) {
    String sql = "select id, email, username,password from p_user where id= :id";
    Map<String, Object> params = new HashMap<>();
    params.put("id", id);
    return jdbcNamedTemplate.queryForObject(sql, params, rowMapper);
}
```

In writing your application, the decision of using `JdbcTemplate` or `NamedParameterJdbcTemplate` is up to you. But consider this: named queries are easier to read and safer, since there is little chance to assign the value to the wrong parameter.

⁶Code accessible on GitHub: <https://github.com/spring-projects/spring-framework/blob/master/spring-jdbc/src/main/java/org/springframework/jdbc/core/namedparam/NamedParameterJdbcTemplate.java>, or in IntelliJ IDEA, click on the class name and press the Control (Command on MacOS) key, and the sources will be opened for you.

Aside from SELECT queries, JdbcTemplate can execute INSERT, UPDATE, and DELETE operations using the update method. This method is polymorphic as well⁷ and can be called with or without parameters. It returns an integer representing the number of lines that were affected. Below you can see code snippets for each of them:

- INSERT: the method createUser below inserts a new user

```
// JdbcTemplateUserRepo.java
public int createUser(Long userId, String username, String password,
                    String email) {
    return jdbcTemplate.update(
        "insert into p_user(ID, USERNAME, PASSWORD, EMAIL)" +
        " values(?,?,?,?)",
        userId, username, password, email
    );
}

...

public Set<User> findAllByUserName(String username,
    boolean exactMatch) {
    String sql = "select id, username, email, password from p_user where ";
    if (exactMatch) {
        sql += "username= ?";
    } else {
        sql += "username like '%" || ? || '%'";
    }
    return new HashSet<>(jdbcTemplate.query(sql,
        new Object{username}, rowMapper));
}

// test method in TestJdbcTemplateUserRepo.java
@Test
public void testCreate(){
    int result = userRepo.createUser(5L, "Diana", "mypass",
        "diana@opympus.com");
    assertEquals(1, result);
    Set<User> dianas = userRepo.findAllByUserName("Diana", true);
    assertTrue(dianas.size() == 1);
}
}
```

- UPDATE: the method below updates the password for a user, identifying it by its ID.

```
// JdbcTemplateUserRepo.java
public int updatePassword(Long userId, String newPass) {
    String sql = "update p_user set password=? where ID = ?";
    return jdbcTemplate.update(sql, newPass, userId);
}
}
```

⁷Its many signatures and recommended uses can be inspected in the official JavaDoc available online <http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html>.

```
// test method in TestJdbcTemplateUserRepo.java
@Test
public void testUpdate(){
    int result = userRepo.updatePassword(1L, "newpass");
    assertEquals(1, result);
}
```

- DELETE: the method below deletes a user identified by its ID.

```
// JdbcTemplateUserRepo.java
public int deleteById(Long userId) {
    return jdbcTemplate.update(
        "delete from p_user where id =? ",
        userId);
}

// test method in TestJdbcTemplateUserRepo.java
@Test
public void testDelete(){
    int result = userRepo.deleteById(4L);
    assertEquals(1, result);
}
```

DML stands for Data Manipulation Language, and the database operations presented so far are part of it, the commands SELECT, INSERT, UPDATE, and DELETE are database statements used to create, update, or delete data from existing tables.

DDL stands for Data Definition Language, and database operations that are part of it are used to manipulate database objects: tables, views, cursors, etc. DDL database statements can be executed with JdbcTemplate using the execute method. The following code snippet depicts how to create a table using a JdbcTemplate instance. After creation, a query to count the records in the table is executed. If the table was created, the query will successfully be executed and return 0.⁸

```
// JdbcTemplateUserRepo.java
public int createTable(String name) {
    jdbcTemplate.execute("create table " + name +
        " (id integer, name varchar2)" );
    String sql = "select count(*) from " + name;
    return jdbcTemplate.queryForObject(sql, Integer.class);
}

// test method in TestJdbcTemplateUserRepo.java
@Test
public void testCreateTable(){
    int result = userRepo.createTable("new_p_user");
    // table exists but is empty
    assertEquals(0, result);
}
```

⁸This kind of approach leaves an application sensitive to SQL Injection attacks, which is why in enterprise applications, the tables are never created by executing DDL scripts in the code with parameter values provided by the user.

But things do not always go as intended, and sometimes exceptions are thrown. In the next section, what can be done in such unexpected cases and the components involved in treating the situation accordingly are presented.

Spring Data Access Exceptions

When working with data sources, there are a few particular cases when things do not go well: sometimes, operations might be called on records that no longer exist; sometimes records cannot be created because of some database restrictions; sometimes the database takes too long to respond; and so on. In pure JDBC, the same type of checked exception is always thrown, `java.sql.SQLException`, and in its message there is an SQL code that the developer must identify and treat accordingly.

Spring provides a big family of data access exceptions. All of them are subclasses of Java `RuntimeException` and provide a practical way for the developer to handle database access exceptions without knowing the details of the data access API. The `JdbcTemplate` instance takes care of transforming cryptic `java.sql.SQLException`s into clear and concise `org.springframework.dao.DataAccessException` implementations⁹ behind the scenes using an infrastructure bean of a type that implements `org.springframework.jdbc.support.SQLExceptionTranslator`. Spring data access exceptions are unchecked and contain human-readable messages that point exactly to the cause of the problem. There are translator components that can be used to transform any type of data access exceptions into Spring-specific exceptions, which is quite useful when one is migrating from one database type to another. A simplified version of the Spring data access exception hierarchy can be seen in Figure 5-5.

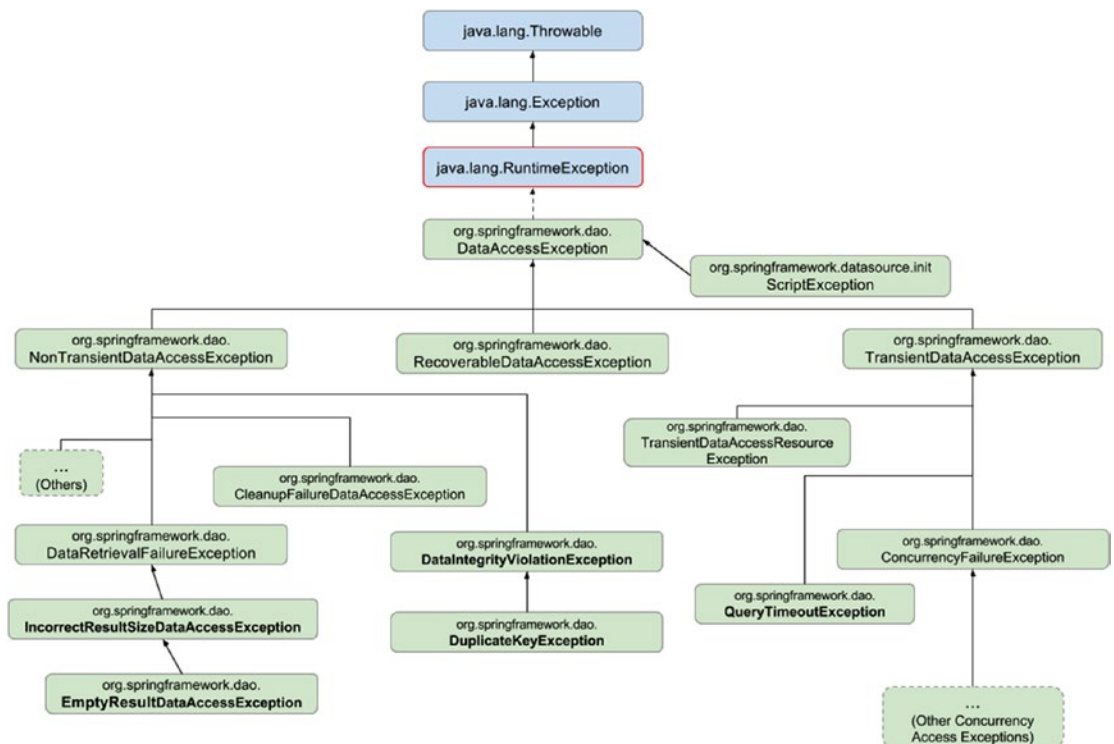


Figure 5-5. Spring data access exception hierarchy

⁹Class `org.springframework.dao.DataAccessException` is an abstract class, parent of all the family of Spring Data Access exceptions.

The `java.sql.SQLException` is a checked exception that forces developers to catch it and handle it or declare it as being thrown over the call method call hierarchy. This type of exception is general and is thrown for every database error. This introduces a form of tight coupling.

In Spring, the data access exceptions are unchecked (they extend `java.lang.RuntimeException`, which is why this class is exhibited in the previous image with a red border) and can be thrown up the method call hierarchy to the point where they are most appropriate to handle. The Spring data access exceptions hide the technology used to communicate with the database. They are thrown when `JdbcTemplate`, JPA, Hibernate, etc. are used. That is probably why they are part of the `spring-tx` module.

As you can notice from Figure 5-5, the Spring data access exception family has three main branches:

- exceptions that are considered *non-transient*, which means that retrying the operation will fail unless the originating cause is fixed. The parent class of this hierarchy is the `org.springframework.dao.NonTransientDataAccessException`. The most obvious example here is searching for an object that does not exist. Retrying this query will fail until a user that is the one being searched for exists. The code snippet below searches for a user that does not exist. Right below is a snippet from the console log depicting the exception type and message.

```
// test method in TestJdbcTemplateUserRepo.java
@Test
public void testNoFindById() {
    User user = userRepo.findById(99L);
    assertEquals("Darius", user.getUsername());
}
```

```
#log from the console
org.springframework.dao.EmptyResultDataAccessException:
    Incorrect result size: expected 1, actual 0
```

Obviously, this test will fail until a user with ID equal to 99 and username Darius is found in the `p_user` table.

- `org.springframework.dao.RecoverableDataAccessException` is thrown when a previously failed operation might succeed if some recovery steps are performed, usually closing the current connection and using a new one. This type of exception is very useful when the database is on a different computer from that of the application and connection to the database fails because of a temporary network hiccup. The exception can be caught and the query retried when the network is up, and the execution will succeed in a new connection.
- exceptions that are considered *transient*, which extend the abstract class `org.springframework.dao.TransientDataAccessException`, which means that retrying the operation might succeed without any intervention. These are concurrency or latency exceptions. For example, when the database becomes unavailable because of a bad network connection in the middle of the execution of a query, an exception of type `QueryTimeoutException` is thrown. The developer can treat this exception by retrying the query.

The fourth branch contains only an exception type used when initialization of a test database with a bean of type `DataSourceInitializer` fails because of a script error; thus the exception type is named `ScriptException`. And that is all that can be said about Spring data access exceptions. Sounds pretty simple, right?

! If you want to test your understanding of Spring `JdbcTemplate` and related components presented so far, you can open the `05-ps-jdbc-practice` project and try to complete the TODOs. There are three that you should definitely solve and one that is a bonus in case you really like this section and you enjoy the practice. The parts missing are marked with a TODO task and are visible in IntelliJ IDEA in the TODO view.

Task TODO 27, located in the `TestJdbcTemplateUserRepo` class, requires you to complete the body of a negative test method that searches for a user that does not exist. The test should pass. To see the user IDs present in the database, inspect the `05-ps-jdbc-practice/src/test/resources/db/test-data.sql`.

Task TODO 28, located in the `TestJdbcTemplateUserRepo` class, requires you to complete the body of a test method that counts the users in the database.

Task TODO 29, located in `JdbcNamedTemplateUserRepo`, requires you to complete the body of repository methods. You can use either the `JdbcNamedTemplate` bean or the underlying `JdbcTemplate` accessed with `JdbcNamedTemplate.getJdbcOperations()`.

The bonus task TODO 30, located in test class `TestNamedJdbcTemplateUserRepo`, if you decide to complete it, will help you test the methods implemented by completing TODO 29.

If you have trouble, you can take a peek at the proposed solution in the `05-ps-jdbc-solution` project.

Data Access Configuration In a Transactional Environment

Until now, data access has been covered with Spring in a nontransactional environment, which means that when a query was executed, a connection was obtained, the query was executed, then the connection was released. In enterprise applications, though, there is a need to group certain SQL operations together so that in case one of them fails, all the results of previous queries in the same group should be rolled back to avoid leaving the database in an inconsistent state. The context of execution for a group of SQL operations is called a *transaction* and has the following properties:

- **Atomicity** is the main attribute of a transaction and is the characteristic mentioned earlier, that if an operation in a transaction fails, the entire transaction fails, and the database is left unchanged. When all operations in a transaction succeed, all changes are saved into the database when the transaction is committed. Basically it is “all or nothing.”
- **Consistency** implies that every transaction should bring the database from one valid state to another.
- **Isolation** implies that when multiple transactions are executed in parallel, they won’t hinder one another or affect each other in any way. The state of the database after a group of transactions is executed in parallel should be the same as if the transactions in the group had been executed sequentially.
- **Durability** is the property of a transaction that should persist even in cases of power off, crashes, and other errors on the underlying system.¹⁰

¹⁰An exception to this rule would be if the server catches fire or gets really wet.

In a transactional environment, transactions have to be managed. In Spring, this is done by an infrastructure bean called the *transaction manager*. The transaction manager bean’s configuration is the only thing that has to be changed when the environment changes. There are four basic flavors:

1. **Local JDBC Spring environment:** a local JDBC configuration declaring a basic datasource to be used (even an embedded one will do) and a bean of type `org.springframework.jdbc.datasource.DataSourceTransactionManager`, a Spring-specific implementation. The connections to use can come from a connection pool, and the transaction manager bean will associate connections to transactions according to the configured behavior. Configuring transactional behavior is done declaratively by annotating methods with `@Transactional`. It is mandatory to use this annotation on repository classes, too, in order to ensure transactional behavior. Without an ORM, a component of type `JdbcTemplate` must be used to execute queries at the repository level. All this will be detailed later in this chapter. The abstract schema of this configuration is depicted in Figure 5-6.

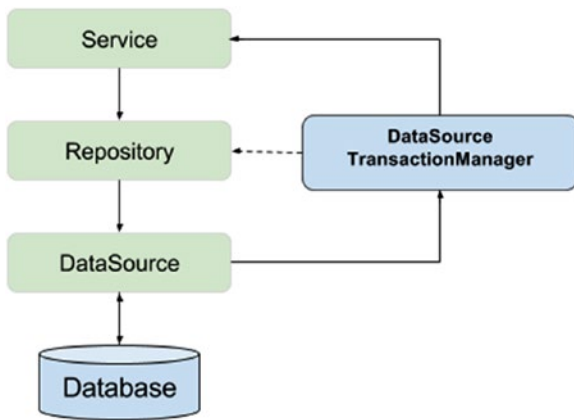


Figure 5-6. Spring Local JDBC Configuration abstract schema

2. **Local Hibernate Spring environment:** a local hibernate configuration declaring a basic datasource to be used (even an embedded one will do) and a bean of type `org.springframework.orm.hibernate5.HibernateTransactionManager`, a Spring-specific implementation that uses a hibernate session object created by an infrastructure bean of type that extends `org.springframework.orm.hibernate5.LocalSessionFactoryBean` to manage entities in a transactional context. The abstract schema of this configuration is depicted in Figure 5-7.

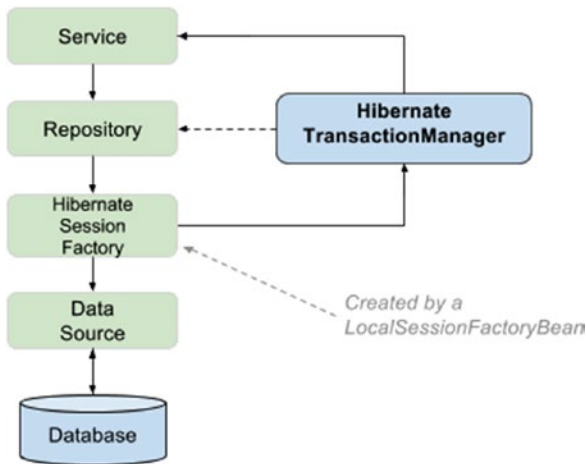


Figure 5-7. Spring Local Hibernate Configuration abstract schema

- Local JPA Spring environment:** a local configuration declaring a basic datasource to be used (even an embedded one will do) and a bean of type `org.springframework.orm.jpa.JpaTransactionManager`, a Spring-specific implementation that uses an entity manager object created by an infrastructure bean of type `org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean` to manage entities in a transactional context. The abstract schema of this configuration is depicted in Figure 5-8. To create the `LocalContainerEntityManagerFactoryBean` bean, a persistence manager and a JPA adapter bean are needed. These can be provided by Hibernate, Apache OpenJPA, or any other Spring-supported persistence framework.

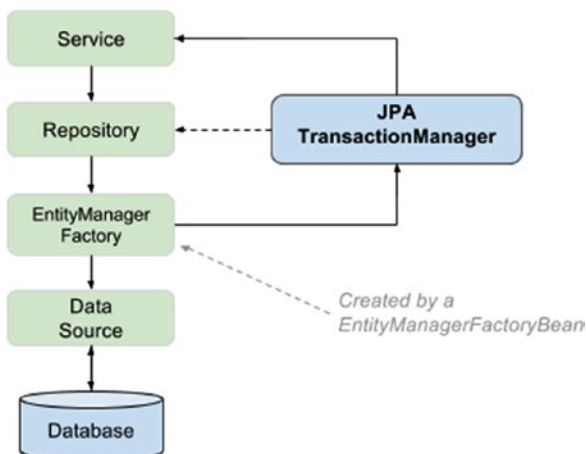


Figure 5-8. Spring Local JPA Configuration abstract schema

4. **Enterprise JTA Spring environment:** this setup requires an application server that will configure and provide a datasource bean using JNDI. Spring will load a bean of type extending `org.springframework.transaction.jta.JtaTransactionManager` specific to the application server used. This transaction manager is appropriate for handling distributed transactions, which are transactions that span multiple resources, and for controlling transactions on application server resources. The abstract schema of this configuration is depicted in Figure 5-9.

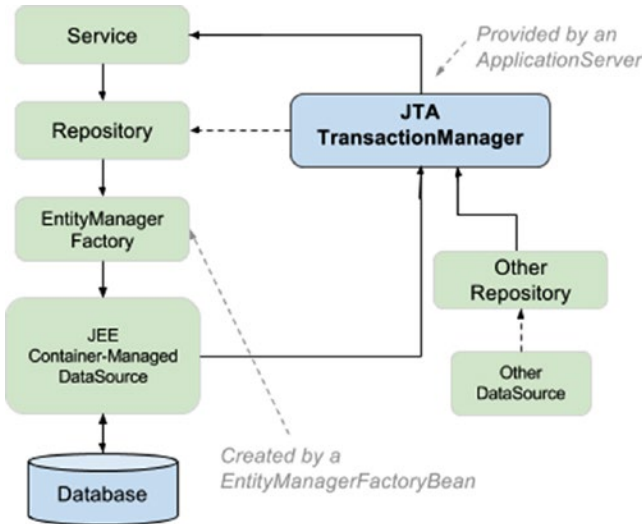


Figure 5-9. Enterprise JTA Spring environment

How Transaction Management Works in Spring

In the previous section, a repository class used a `JdbcTemplate` to execute methods within a connection. To introduce transactions, we need a service class that will call the repository methods in the context of a transaction. Figure 5-10 depicts the abstract UML sequence diagram that describes the examples covered in this section. The diagram in the figure mentions only the `findById(...)` method, but the call sequence for every service method that involves managing database-stored information is the same.

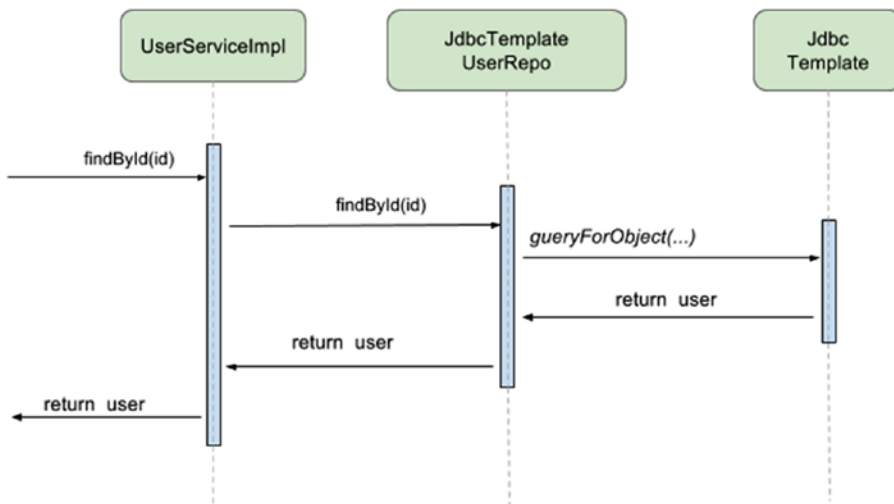


Figure 5-10. UML call diagram for a service class

Transaction management is implemented in Spring by making use under the hood of a framework presented in a previous chapter, **AOP**, and that is because transactions are a cross-cutting concern. Basically, for every method that must be executed in a transaction, retrieving or opening a transaction before execution and committing it afterward is necessary. For beans that have methods that must be executed in a transactional context, AOP proxies are created that wrap the methods in an Around advice that takes care of getting a transaction before calling the method and committing the transaction afterward. The AOP proxies use two infrastructure beans for this: an `org.springframework.transaction.interceptor.TransactionInterceptor` in conjunction with an implementation of `org.springframework.transaction.PlatformTransactionManager`. Spring provides a flexible and powerful abstraction layer for transaction management support. At the core of it is the `PlatformTransactionManager` interface. Any transaction manager provider framework that is supported can be used. JTA providers can as well, and this will have no impact on the rest of the classes of the application. The most common transaction management providers are:

- `DataSourceTransactionManager`, Spring basic transaction management provider class;
- `HibernateTransactionManager`, when Hibernate is used for persistence support;
- `JpaTransactionManager`, when an entity manager is used for persistence support;
- `JtaTransactionManager`, used to delegate transaction management to a Java EE server. Can be used with Atomikos too, removing the need for an application server.¹¹
- `WebLogicJtaTransactionManager`, transaction support provided by the WebLogic Application Server.
- etc.

Conceptually, what happens when a transactional method is called is depicted in Figure 5-11.

¹¹You can find more about Atomikos from their official site <https://www.atomikos.com/>.

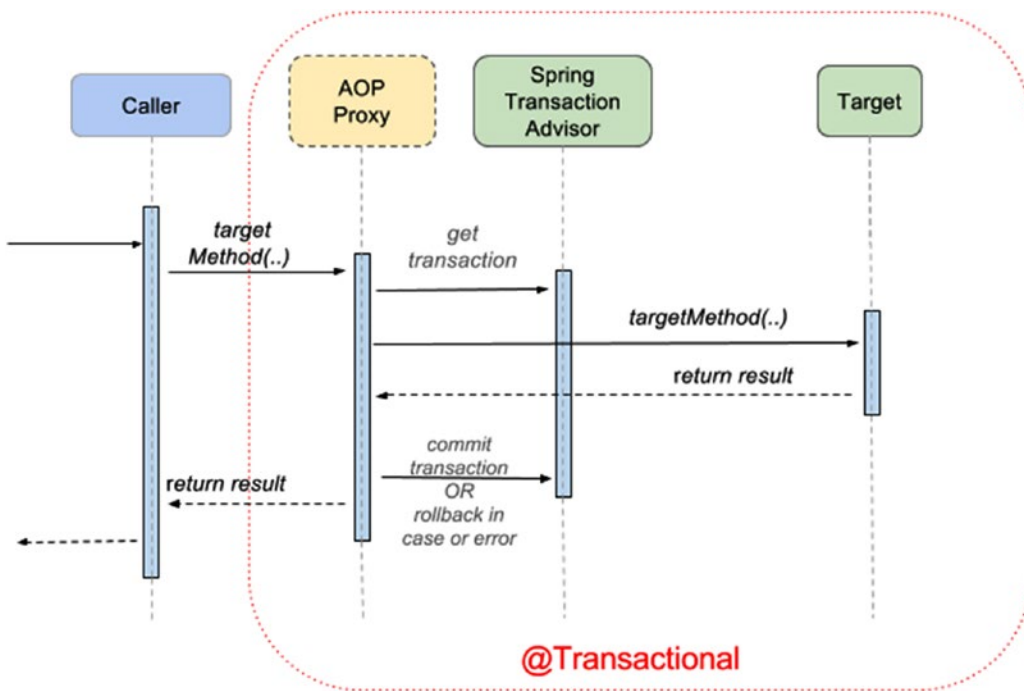


Figure 5-11. Conceptual UML sequence diagram for a transactional operation

Methods that need to be executed in a transactional context are annotated with `@Transactional` Spring annotation. The body of these methods is a functional unit that cannot be subdivided. The official Spring course calls these methods atomic units of work. As you probably remember from the AOP chapter, this annotation must be used only on public methods; otherwise, the transactional proxy won't be able to apply the transactional behavior. When the application context is created and support for this annotation is enabled via configuration (this will be covered in the following section), here is what happens under the hood: an internal infrastructure Spring-specific bean of type `org.springframework.aop.framework.autoproxy.InfrastructureAdvisorAutoProxyCreator` is registered and acts as a bean postprocessor that modifies the service and repository bean to add transaction-specific logic. Basically, this is the bean that creates the transactional AOP proxy.

Configure Transactions Support

The simplest way to add transaction management to the application covered in the previous section is to do the following:

- Configure transaction management support:** add a declaration of a bean of type `org.springframework.jdbc.datasource.DataSourceTransactionManager`. Using XML and activating it with the `<tx:annotation-driven .. />` (be careful when you are using transactions XML configuration because it is a common mistake to forget this element) configuration element provided by the `tx` namespace is easy.

```

<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-2.0.xsd">

        <bean id="transactionManager"
            class="org.springframework.jdbc.datasource.
            DataSourceTransactionManager">
            <property name="dataSource" ref="dataSource"/>
        </bean>

        <tx:annotation-driven transaction-manager="transactionManager"/>
    </beans>

```

The configuration above declares a bean named `transactionManager` and sets it as the general transaction manager for the application using `<tx:annotation-driven ../>`. Or a Configuration class:

```

public class TestDataConfig {
    ...
    @Bean
    public PlatformTransactionManager txManager(){
        return new DataSourceTransactionManager(dataSource());
    }
}

```

And we activate it with the `@EnableTransactionManagement`:

```

import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = {"com.ps.repos.impl", "com.ps.services.impl"})
public class AppConfig {
}

```

- **Declare transactional methods:** write a service class containing methods that will call the `UserRepo` bean methods in a transaction. A method that is to be executed in a transaction must be annotated with the `Spring @Transactional` annotation. Since the transactional behavior must be propagated to the repository, the repository class or methods are annotated with `@Transactional` too.

```

import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class UserServiceImpl implements UserService {

    private UserRepo userRepo;

    @Autowired
    public UserServiceImpl(UserRepo userRepo) {
        this.userRepo = userRepo;
    }
}

```

```

    @Transactional
    @Override
    public User findById(Long id) {
        return userRepo.findById(id);
    }
}

```

■ **CC** Both `@EnableTransactionManagement` and `<tx:annotation-driven ..>` enable all infrastructure beans necessary for supporting transactional execution. But there is a minor difference between them. The XML configuration element can be used like this: `<tx:annotation-driven />` without the `transaction-manager` attribute. In this case, Spring will look for a bean named `transactionManager` by default, and if it is not found, the application won't start. The `@EnableTransactionManagement` is more flexible; it looks for a bean of any type that implements the `org.springframework.transaction.PlatformTransactionManager`, so the name is not important. In case the default transaction manager bean must be established without a doubt, this can be done by making the configuration class annotated with `@EnableTransactionManagement` implement the `org.springframework.transaction.annotation.TransactionManagementConfigurer` interface and declare the default transaction manager by providing an implementation for the method `annotationDrivenTransactionManager`.

```

import org.springframework.transaction.annotation.TransactionManagementConfigurer;
...
@EnableTransactionManagement
@Configuration
public class AppConfig implements TransactionManagementConfigurer {

    ...

    @Bean
    public PlatformTransactionManager txManager(){
        return new DataSourceTransactionManager(dataSource());
    }

    @Override
    public PlatformTransactionManager annotationDrivenTransactionManager() {
        return txManager();
    }
}

```

This is useful, because in bigger applications requiring more than one `datasource`, multiple transaction manager beans need to be declared. If the default to use is not specified, Spring cannot choose for itself, and the application will fail to start with an ugly and explicit exception:

```

org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [org.springframework.transaction.PlatformTransactionManager]
is defined: expected single matching bean but found 2: txManager,simpleManager

```


The transaction manager to use when executing a method in a transactional context can also be specified by the `transactionManager` attribute of the `@Transactional` annotation. So considering that we have two transaction manager beans, `txManager`, which is configured as default, and `simpleManager`, code similar to that in the next snippet will work like a charm.

```
@Service
public class UserServiceImpl implements UserService {
    private UserRepo userRepo;

    @Autowired
    public UserServiceImpl(UserRepo userRepo) {
        this.userRepo = userRepo;
    }

    //default txManager is used
    @Transactional
    @Override
    public User findById(Long id) {
        return userRepo.findById(id);
    }

    @Transactional(transactionManager = "simpleManager",readOnly = true)
    @Override
    public void htmlAllByNameAll(String name){
        userRepo.htmlAllByName(name);
    }
}
```

The transactions in the context of which methods are executed in Spring applications are called *declarative transactions* when defined with `@Transactional`, and this type of declaring transactions is not connected to a JTA (Java Transaction API), which is very practical, because this means that transactions can be used in any environment: local with JDBC, JPA, Hibernate, or JDO (Java Data Objects), or together with an application server. Transactional behavior can be added to any method of any bean as long as the method is public, because declarative transaction behavior is implemented in Spring using AOP. Spring also provides declarative rollback rules and the possibility to customize transactional behavior through attributes of the `@Transactional` annotation. And at this point, a list of these attributes is appropriate:

1. The `transactionManager` attribute value defines the transaction manager used to manage the transaction in the context of which the annotated method is executed.
2. The `readOnly` attribute should be used for transactions that involve operations that do not modify the database (example: searching, counting records). The default value is `false`, and the value of this attribute is just a hint for the transaction manager, which can be ignored depending of the implementation. Although if you tell Spring that the transaction is supposed only to read data, some performance optimizations will be done for read-only data access. Although it seems useless to use a transaction for reading data, it is recommended to do so to take into account the isolation level configured for the transactions. The isolation attribute will be covered shortly.

3. The propagation attribute can be used to define behavior of the target methods: if they should be executed in an existing or new transaction, or no transaction at all. The values for this attribute are defined by the Spring `org.springframework.transaction.annotation.Propagation` enum, and they match the ones defined in JEE for EJB transactions. There are seven propagation types:
 - (a) **REQUIRED**: an existing transaction will be used or a new one will be created to execute the method annotated with `@Transactional(propagation = Propagation.REQUIRED)`.
 - (b) **REQUIRES_NEW**: a new transaction is created to execute the method annotated with `@Transactional(propagation = Propagation.REQUIRES_NEW)`. If a current transaction exists, it will be suspended.
 - (c) **NESTED**: an existing nested transaction will be used to execute the method annotated with `@Transactional(propagation = Propagation.NESTED)`. If no such transaction exists, it will be created.
 - (d) **MANDATORY**: an existing transaction must be used to execute the method annotated with `@Transactional(propagation = MANDATORY)`. If there is no transaction to be used, an exception will be thrown.
 - (e) **NEVER**: methods annotated with `@Transactional(propagation = Propagation.NEVER)` must not be executed within a transaction. If a transaction exists, an exception will be thrown.
 - (f) **NOT_SUPPORTED**: no transaction is used to execute the method annotated with `@Transactional(propagation = Propagation.NOT_SUPPORTED)`. If a transaction exists, it will be suspended.
 - (g) **SUPPORTS**: an existing transaction will be used to execute the method annotated with `@Transactional(propagation = Propagation.SUPPORTS)`. If no transaction exists, the method will be executed anyway, without a transactional context.

In the following code snippet, the `findById` method is executed in a transaction and the `getPetsAsHtml` is executed within a nested transaction. Actual creation of nested transactions will work only on specific transaction managers. And as you can see from the log at the end of the code snippet, `DataSourceTransactionManager` is not one of them when one is working with an in-memory database. This transaction manager supports nested transactions via the JDBC 3.0 “savepoint” mechanism, but it needs a database management system that supports savepoints, such as Oracle, for example.

Nested transactions allow for complex behavior; a transaction can start before the enclosing transaction is completed. A commit in a nested transaction has no effect, and all changes will be applied to the database when the enclosing transaction is committed. If a nested transaction is rolled back, though, the enclosing transaction is rolled back to prevent leaving the database in an inconsistent state: partial changes will not be kept. Nested transactions can be used to force atomic execution of multiple methods.

If instead of **NESTED**, **REQUIRED** is used, there will be no guarantee of atomic execution.

```

//UserServiceImpl.java
@Service
public class UserServiceImpl implements UserService {
    ...
    @Transactional(propagation = Propagation.REQUIRED, readOnly= true)
    @Override
    public User findById(Long id) {
        return userRepo.findById(id);
    }
}

//PetServiceImpl.java
@Service
public class PetServiceImpl implements PetService {

    ...

    @Override
    @Transactional(propagation = Propagation.NESTED, readOnly = true)
    public String getPetsAsHtml(User owner) {
        Set<Pet> pets = petRepo.findByOwner(owner);
        if(pets.isEmpty()) {
            return "<p>User " + owner.getUsername() + " has no pets.</p>\n";
        }
        // build html from pets
        StringBuilder htmlSb = ...;
        return htmlSb.toString();
    }
}

//PetServiceTest.java test class
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class PetServiceTest {
    @Test
    public void testFindById() {
        User user = userService.findById(1L);
        String out = petService.getPetsAsHtml(user);
        assertTrue(out.contains("Name"));
        System.out.println(out);
    }
}

```

In Figure 5-12, the behavior of nested transactions is depicted.

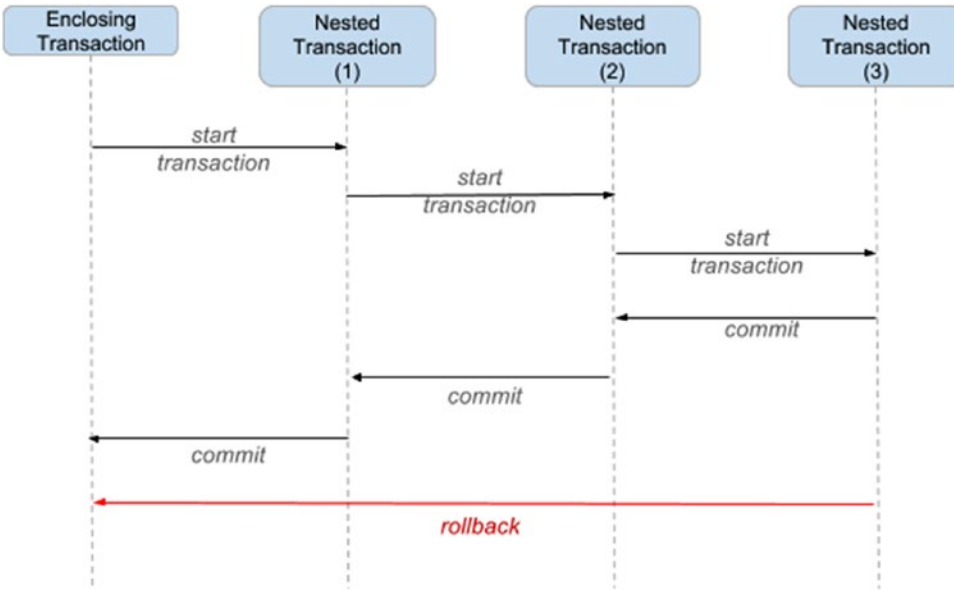


Figure 5-12. Nested transaction behavior

How do we know that at least Spring is trying to execute the second method in a nested transaction? Simple, we look in the console log:

```

...
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance
of singleton bean 'transactionManager'
DEBUG o.s.j.d.DataSourceTransactionManager - Creating new transaction:
with name [com.ps.services.impl.UserServiceImpl.findById]
PROPAGATION_REQUIRED,ISOLATION_DEFAULT,readOnly; "
DEBUG o.s.j.d.SimpleDriverDataSource -
Creating new JDBC Driver Connection to [jdbc:h2:sample]
DEBUG o.s.j.d.DataSourceTransactionManager - Acquired Connection
[conn2: url=jdbc:h2:sample user=SAMPLE] for JDBC transaction
DEBUG o.s.j.d.DataSourceUtils - Setting JDBC Connection
[conn2: url=jdbc:h2:sample user=SAMPLE] read-only
DEBUG o.s.j.d.DataSourceTransactionManager - Switching JDBC Connection
[conn2: url=jdbc:h2:sample user=SAMPLE] to manual commit
DEBUG o.s.j.c.JdbcTemplate - Executing prepared SQL statement
[select id, email, username, password, user_type from p_user where id= ?]
DEBUG o.s.j.d.DataSourceTransactionManager - Initiating transaction commit
DEBUG o.s.j.d.DataSourceTransactionManager -
Committing JDBC transaction on Connection
[conn2: url=jdbc:h2:sample user=SAMPLE]
DEBUG o.s.j.d.DataSourceTransactionManager - Releasing JDBC Connection
[conn2: url=jdbc:h2:sample user=SAMPLE] after transaction
DEBUG o.s.j.d.DataSourceUtils - Returning JDBC Connection to DataSource

# Not actually nested.
    
```

```

DEBUG o.s.j.d.DataSourceTransactionManager - Creating new transaction with name
      [com.ps.services.impl.PetServiceImpl.getPetsAsHtml]:
      PROPAGATION_NESTED,ISOLATION_DEFAULT,readOnly; "
DEBUG o.s.j.d.SimpleDriverDataSource - Creating new JDBC Driver Connection
      to [jdbc:h2:sample]
DEBUG o.s.j.d.DataSourceTransactionManager - Acquired Connection
      [conn3: url=jdbc:h2:sample user=SAMPLE] for JDBC transaction
DEBUG o.s.j.d.DataSourceUtils - Setting JDBC Connection
      [conn3: url=jdbc:h2:sample user=SAMPLE] read-only
DEBUG o.s.j.d.DataSourceTransactionManager - Switching JDBC Connection
      [conn3: url=jdbc:h2:sample user=SAMPLE] to manual commit
DEBUG o.s.j.c.JdbcTemplate - Executing SQL query
      [select id, name, age, pet_type from p_pet where owner=1]
DEBUG o.s.j.d.DataSourceTransactionManager - Initiating transaction commit
DEBUG o.s.j.d.DataSourceTransactionManager - Committing JDBC transaction
      on Connection [conn3: url=jdbc:h2:sample user=SAMPLE]
DEBUG o.s.j.d.DataSourceTransactionManager - Releasing JDBC Connection
      [conn3: url=jdbc:h2:sample user=SAMPLE] after transaction
DEBUG o.s.j.d.DataSourceUtils - Returning JDBC Connection to DataSource
User John has:
<p>Name: Mona, type: CAT, Age: 2</p></br>
<p>Name: Max, type: DOG, Age: 10</p></br>

```

4. The isolation attribute value defines how data modified in a transaction affects other simultaneous transactions. As a general idea, transactions should be isolated. A transaction should not be able to access changes from another uncommitted transaction. There are four levels of isolation, but every database management system supports them differently. In Spring, there are five isolation values that are defined in the `org.springframework.transaction.annotation`. Isolation enum:
 - (a) `DEFAULT`: the default isolation level of the DBMS.
 - (b) `READ_UNCOMMITTED`: data changed by a transaction can be read by a different transaction while the first one is not yet committed, also known as *dirty reads*.
 - (c) `READ_COMMITTED`: dirty reads are not possible when a transaction is used with this isolation level. This is the default strategy for most databases. But a different phenomenon could happen here: *repeatable read*: when the same query is executed multiple times, different results might be obtained. (Example: a user is extracted repeatedly within the same transaction. In parallel, a different transaction edits the user and commits. If the first transaction has this isolation level, it will return the user with the new properties after the second transaction is committed.)
 - (d) `REPEATABLE_READ`: this level of isolation does not allow dirty reads, and repeatedly querying a table row in the same transaction will always return the same result, even if a different transaction has changed the data while the reading is being done. The process of reading the same row multiple times in the context of a transaction and always getting the same result is called *repeatable read*.

- (e) **SERIALIZABLE**: this is the most restrictive isolation level, since transactions are executed in a serialized way. So no dirty reads, no repeatable reads, and no phantom reads are possible. A *phantom read* happens when in the course of a transaction, execution of identical queries can lead to different result sets being returned.
5. **timeout**. By default, the value of this attribute is defined by the transaction manager provider, but it can be changed by setting a different value in the annotation: `@Transactional(timeout=3600)`. The value represents the number of milliseconds after which a transaction is considered failed, and the default value is `-1` which means timeouts are not supported.
 6. **rollbackFor** attribute values should be one or more exception classes, subclasses of `Throwable`. When this type of exception is thrown during the execution of a transactional method, the transaction is rolled back. By default, a transaction is rolled back only when a `RuntimeException` is thrown. In using this attribute, the rollback can be triggered for checked exceptions as well.

In the code snippet below, `MailSendingException` is a checked exception that is thrown when a notification of successful user modification cannot be sent via email. The basic idea is this: the main operation was performed successfully, and the user data has been updated. It does not make sense to roll back the changes because a notification could not be sent.

```
@Transactional(rollbackFor = MailSendingException.class)
public int updatePassword(Long userId, String newPass)
    throws MailSendingException {
    User u = userRepo.findById(userId);
    String email = u.getEmail();
    sendEmail(email);
    return userRepo.updatePassword(userId, newPass);
}

private void sendEmail(String email)
    throws MailSendingException {
    ... // code not relevant here
}
```

7. **noRollbackFor** attribute values should be one or more exception classes, subclasses of `Throwable`. When this type of exception is thrown during the execution of a transactional method, the transaction is not rolled back. By default, a transaction is rolled back only when a `RuntimeException` is thrown. Using this attribute, rollback of a transaction can be avoided for a `RuntimeException` as well.

The `@Transactional` annotation can be used at the class level too. In this case, all the methods in the class become transactional, and all properties defined for the transaction are inherited from the `@Transactional` class level definition, but they can be overridden by a `@Transactional` defined at the method level.

```
@Service
@Transactional(readOnly = true,
    propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {
```

```

...
@Transactional(propagation = Propagation.REQUIRES_NEW)
@Override
public User findById(Long id) {
    return userRepo.findById(id);
}

@Transactional(propagation = Propagation.SUPPORTS)
@Override
public void htmlAllByNameAll(String name){
    userRepo.htmlAllByName(name);
}
}

```

`@Transactional` can also be used on abstract classes and interfaces. This leads to all implementation methods inheriting the transactional behavior defined by the annotation on the parent class/interface, but it is not the recommended practice.

Testing transactional methods

For test cases involving datasource operations, a few practical annotations were added in Spring 4.1 to the `org.springframework.test.context.jdbc` package:

- The `@Sql` annotation can be used to specify SQL scripts to be executed against a given database during integration tests. It can be used on classes and on methods. It can be used multiple times when the SQL files to be executed have different syntax, error handling rules, or different execution phases. The tests specific to this section are integration tests, because they test the communication between a service and a repository. So they are a perfect fit to introduce this annotation.
- The `@SqlGroup` annotation can be used on classes and methods to group together `@Sql` annotations. It can be used as a meta-annotation to create custom composed annotations. For example, when more than one datasource is involved, it can be used together with `@SqlConfig` to group scripts to be executed to prepare the test environment for a single test case involving both datasources:

```

@SqlGroup({
    @Sql(scripts = "script1.sql", config = @SqlConfig(dataSource =
        "dataSource1")),
    @Sql(scripts = "script2.sql", config = @SqlConfig(dataSource = "dataSource2"))
})

```

- The `@SqlConfig` is used to specify the configuration of the SQL script.

In the code snippet below, you can see how these annotations can be used to test your service classes. The first example uses multiple `@Sql` annotations; the second groups them with `@SqlGroup`. The example can be tested by running the `UserServiceTest.testCount()` method from the `06-ps-tx-practice` project/

```

import org.springframework.test.context.jdbc.Sql;
import org.springframework.test.context.jdbc.SqlConfig;
import org.springframework.test.context.jdbc.SqlGroup;
...

```

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class UserServiceTest {
    @Test
    @Sql(value = "classpath:db/extra-data.sql",
        config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--"))
    @Sql(
        scripts = "classpath:db/delete-test-data.sql",
        config = @SqlConfig(transactionMode = SqlConfig.TransactionMode.ISOLATED),
        executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD
    )
    public void testCount(){
        int count = userService.countUsers();
        assertEquals(8, count);
    }
}

```

Although the syntax is quite obvious, just in case extra clarifications are necessary, here they are:

- the first `@Sql` annotation specifies a script to be executed to save some data into the test database before executing the test method. The `@SqlConfig` is used to declare specific SQL syntax details, so Spring can execute the `extra-data.sql` script correctly.
- The second `Sql` annotation is used to execute the script that will clean the test database after the test execution. The attribute that specifies when the script is executed is `executionPhase`, and in this case, the value used to tell Spring to execute the script after the test method is `Sql.ExecutionPhase.AFTER_TEST_METHOD`.

The `@SqlConfig` annotation is quite powerful and provides attributes to declare isolation level (`transactionMode = SqlConfig.TransactionMode.ISOLATED`) and a `transactionManager` to be used.

```
(transactionManager="txMng")
```

The following example has the same behavior as the one above, but for teaching purposes, the two `Sql` annotations have been composed using `@SqlGroup`.

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class UserServiceTest {
    // @SqlGroup version
    @Test
    @SqlGroup({
        @Sql(value = "classpath:db/extra-data.sql",
            config = @SqlConfig(encoding = "utf-8", separator = ";", commentPrefix = "--")),
        @Sql(
            scripts = "classpath:db/delete-test-data.sql",
            config = @SqlConfig(transactionMode = SqlConfig.TransactionMode.ISOLATED),
            executionPhase = Sql.ExecutionPhase.AFTER_TEST_METHOD
        )
    })
    public void testCount() {

```



```

        int count = userService.countUsers();
        assertEquals(8, count);
    }
}

```

The `@Sql` annotation has more attributes, but there is one in particular that you will definitely find useful: the `statements` attribute. This attribute allows you to provide a statement to be executed before the test method.

```

@Test
@Sql(statements = {"drop table NEW_P_USER if exists;"})
public void testCreateTable(){
    int result = userRepo.createTable("new_p_user");
    // table exists but is empty
    assertEquals(0, result);
}

```

Repository methods can be tested in a transactional context, decoupled from the service methods. This is useful when the test database must be left unaffected by a test execution. The `@Transactional` annotation can be used on the test method and configured appropriately, and the `@Rollback` annotation can be used to fulfill this specific purpose: leave a test database unchanged.

```

import org.springframework.test.annotation.Rollback;
...
RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class TransactionalJdbcRepoTest {
    @Transactional
    @Test
    public void testFindById() {
        User user = userRepo.findById(1L);
        assertEquals("John", user.getUsername());
    }

    @Test
    @Transactional @Rollback(false)
    public void testCreate(){
        int result = userRepo.createUser
            (5L, "Diana", "mypass", "diana@opypmus.com", UserType.BOTH);
        assertEquals(1, result);
        Set<User> dianas = userRepo.findAllByUserName("Diana", true);
        assertTrue(dianas.size() == 1);
    }
}

```

The `@Rollback` annotation can be used to specify that no rollback should be performed after a test method execution, by making use of its default attribute: `@Rollback(false)`. This is equivalent to another annotation introduced in Spring 4.2: `@Commit`. As you can probably imagine, you can use one or the other for this purpose, but you should not use them together unless you really want to confuse the Spring Container. The results will thus be unpredictable. Both annotations can be used at the class level too, and class level annotation configuration can be overridden by method level annotation configurations.

Before Spring 4.0, the `@TransactionConfiguration` could be used on a test class used to define the transactional context for tests. It became deprecated in Spring 4.0, with the introduction of profiles and `@EnableTransactionManagement`. But in case you are interested in an example, here it is:

```
import org.springframework.test.context.transaction.TransactionConfiguration;
...
@TransactionConfiguration(defaultRollback = false,
    transactionManager = "txManager")
@Transactional
public class TransactionalJdbcRepoTest {
    @Test
    public void testFindById() {
        User user = userRepo.findById(1L);
        assertEquals("John", user.getUsername());
    }
    ...
}
```

The default value for the `defaultRollback` attribute is `true`, and it was set to `false` in the previous example just to introduce this attribute.

Another useful test annotation is `@BeforeTransaction`, which can be used to set up or check a test environment before executing the transactional test method. The `@Before` annotated method is executed in the context of the transaction. The method in the following code snippet checks that the test database was initialized properly outside of a transaction context.

```
import org.springframework.test.context.transaction.BeforeTransaction;
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestDataConfig.class, AppConfig.class})
@ActiveProfiles("dev")
public class UserServiceTest {
    ...
    @BeforeTransaction
    public void checkDbInit(){
        int count = userService.countUsers();
        assertEquals(4, count);
    }
}
```

Making Third-Party Components Transactional

For situations in which annotations cannot be used to configure transactional execution, this being the case when a version earlier than Java 1.5 is used or the service is a third-party implementation that cannot be changed, the Spring transaction management configuration can be achieved using XML to declare a combination of AOP and tx configuration elements.

```
<beans ..>
  <!-- dataSource bean -->
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="org.h2.Driver"/>
  </bean>
</beans>
```

```

        <property name="url" value="jdbc:h2:test"/>
        <property name="username" value="test"/>
        <property name="password" value="test"/>
    </bean>

    <!-- target bean -->
    <bean id="userService" class="...UserServiceImpl" />

    <!-- transaction manager bean -->
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- AOP pointcut to select target methods -->
    <aop:config>
        <aop:pointcut id="allMethods"
            expression="execution(* com.ps.service.*.UserService+.*(..))">
            <aop:advisor pointcut-ref="userService"
                advice-ref="transactionalAdvice" />
        </aop:pointcut>
    </aop:config>

    <!-- Transactional Around advice -->
    <tx:advice id="transactionalAdvice">
        <tx:attributes>
            <tx:method name="find*" read-only="true" timeout="10"/>
            <tx:method name="update*" read-only="false" timeout="30"/>
        </tx:attributes>
    </tx:advice>
</beans>

```

! If you want to test your understanding of Spring transaction management, you can open now the `06-ps-tx-practice` project and try to complete the TODOs. There are five tasks for you to solve, numbered from 31 to 35. The parts missing are marked with a TODO task and are visible in IntelliJ IDEA in the TODO view.

Task TODO 31, located in the `TestDataConfig` class, requires you to define a transaction manager bean to be used to manage transactions.

Task TODO 32, located in the `AppConfig` class, requires you to enable use of declarative transactions.

Task TODO 33, located in `UserServiceImpl`, requires you to make all the methods transactional.

Task TODO 34, located in class `UserServiceImpl`, requires you to complete the transaction definition of the `updatePassword(...)` method, to make the transaction writable, and to roll back for the checked exception thrown by the method.

Task TODO 35, located in test class `UserServiceTest`, requires you to complete the body of the method testing the `userService.updatePassword(...)` method.

If you have trouble, you can take a peek at the proposed solution in the `06-ps-tx-solution` project.

! Spring Declarative Model Clarification

There is heated argument on the Internet (most of it happening on [StackOverflow](https://stackoverflow.com)¹²) right now regarding whether `@Transactional` should be used on repository classes/methods when there is a service layer involved and service classes/methods are already annotated with `@Transactional`. In order to settle this debate, we have to consider how declarative transactions are implemented in Spring. It was already mentioned at the beginning of Section 9, **How Transaction Management Works in Spring**, that AOP is used to decorate beans with transactional behavior. This means that when we annotate classes or methods with `@Transactional`, a proxy bean will be created to provide the transactional behavior, and it is wrapped around the original bean.

In an application that does not use a service layer, to ensure transactional behavior when interacting with the database, the repository classes/ methods must be annotated with `@Transactional`. This will tell Spring to create transactional proxies for the repository classes. The abstract UML diagram for this scenario is depicted in Figure 5-11, and the target object is the repository bean. When a service layer and service classes are added, there are two possibilities: we annotate the new service classes with `@Transactional` and remove the annotation from the repository, or we annotate the service classes and keep `@Transactional` on the repository classes as well. Let's analyze each of these cases in detail.

Case 1: Only service classes are annotated with `@Transactional`. In this case, the target object is the service bean, which contains a reference to the repository bean. When a service method is called, the following happens:

1. The transactional proxy calls the transactional advisor to get a transaction.
2. The transactional proxy forwards the initial call to the target service bean.
3. The target service object calls the repository method and returns the result to the proxy.
4. The proxy calls the transactional advisor to commit the transaction.
5. The proxy returns the result to the caller.

The Spring service class has a reference to a repository class, which means that at runtime, the service bean will be created by aggregating the two objects. Actually, when the Spring application context is created, this is what happens:

1. A repository bean declaration is found in the configuration, so a repository bean is created.
2. A service bean declaration is found in the configuration, so a service bean is created. The service bean depends on the previously created repository bean, so the dependency is provided using autowiring. So now the repository is a member of the service bean.
3. Then the `InfrastructureAdvisorAutoProxyCreator` bean creates the proxy object that wraps around the service bean to provide transactional behavior. So the service bean becomes now a target object.

So, to summarize: the service method calls the repository method, and this call is done internally by the target object. The full execution is atomic in the context of the transaction obtained by the proxy from the Spring transactional advisor. This is important, especially when a service method calls more than one repository method, because this approach ensures that all repository methods will be executed within the same transaction. This situation is depicted by the diagram in Figure 5-13.

¹²The most popular programmers' social network <http://stackoverflow.com>

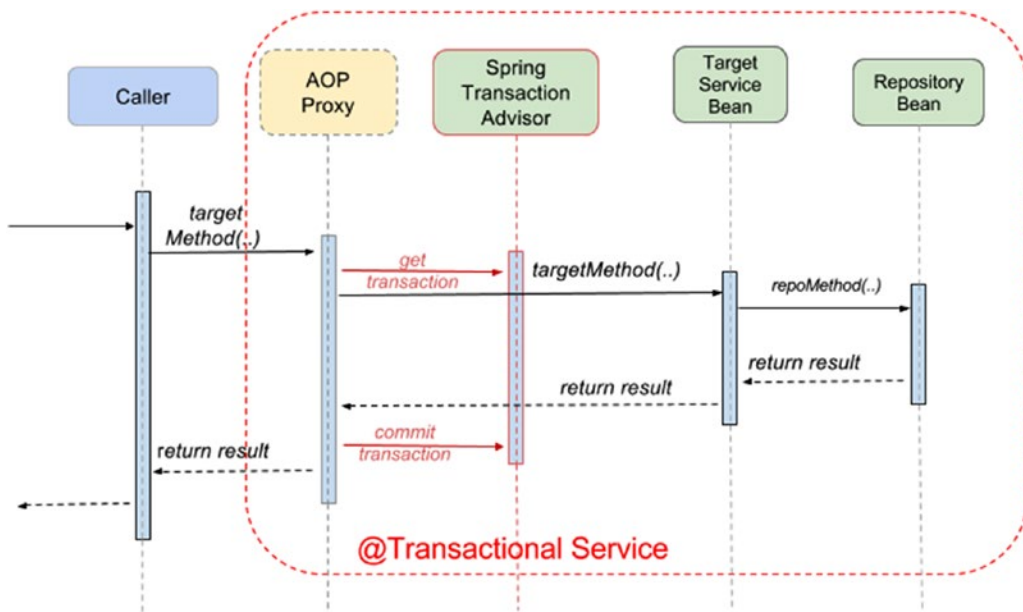


Figure 5-13. @Transactional annotated service class

To get a look under the hood, you can execute the `UserServiceTest.testFindById()` method in debug mode, from project 6-ps-tx-solution in IntelliJ IDEA and take a look at the `userService` bean. In the Variables console, you should see what is depicted in Figure 5-14.

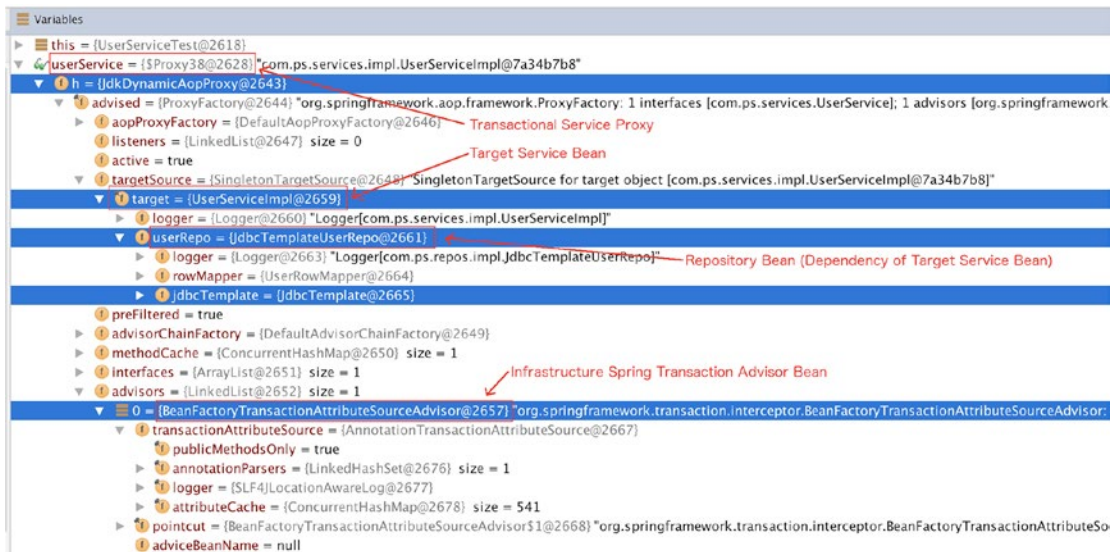


Figure 5-14. IntelliJ IDEA execution of test paused to analyze service transactional bean

Case 2 Service and repository classes are **both** annotated with `@Transactional`. In this case, there are two transactional proxies involved, one for service and one for the repository bean. The service bean contains a reference to the repository proxy in this case. When a service method is called, the following happens:

1. The transactional service proxy calls the transactional advisor to get a transaction.
2. The transactional proxy forwards the initial call to the target service bean.
3. The service bean calls the repository method.
4. The transactional repository proxy calls the transactional advisor to get a transaction. The transaction returned depends on the propagation configuration of the `@Transactional` annotation declared on the repository class.
5. The target repository object method is executed, and the result is returned to the transactional repository proxy.
6. The transactional repository proxy calls the transactional advisor to commit the transaction.
7. The transactional repository proxy returns the result to the caller, in this case the target service object.
8. The target service object returns the result to the transactional service proxy.
9. The transactional service proxy calls the transactional advisor to commit the transaction.
10. The transactional service proxy returns the result to the caller.

So, to summarize: the service target object calls the repository method on the repository proxy, which takes care of establishing a transactional context for the execution of the repository method. This situation is depicted by the diagram in [Figure 5-15](#).

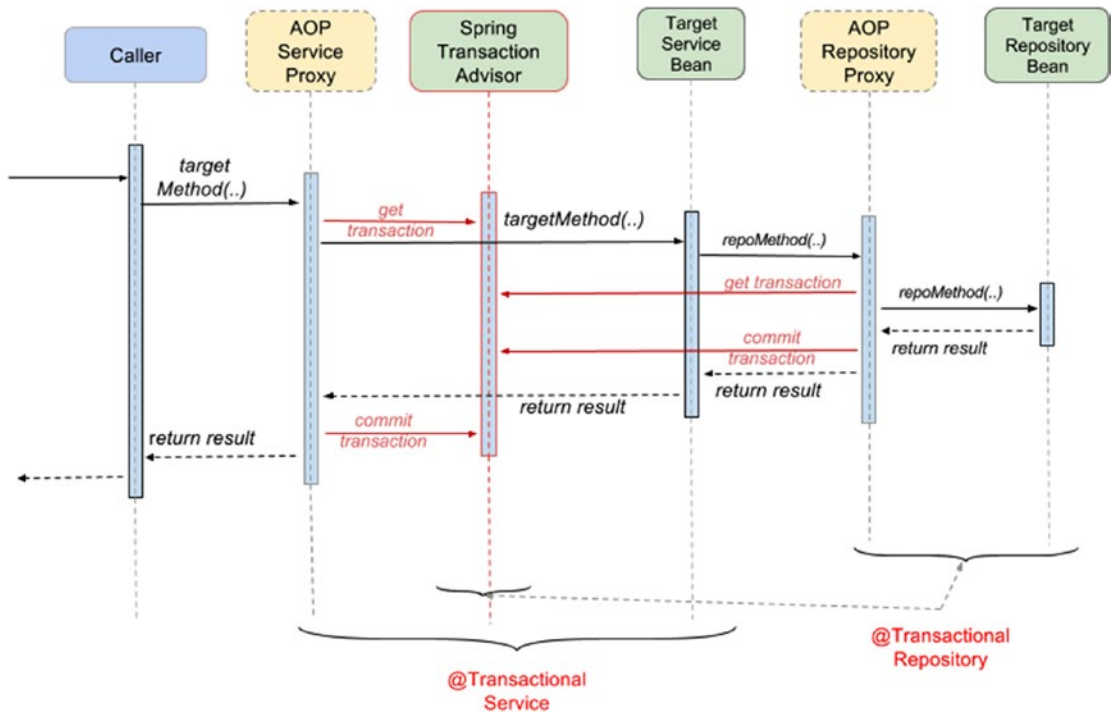


Figure 5-15. *@Transactional annotated service and repository class*

To get a look under the hood, first uncomment the `@Transactional` annotation from the class `JdbcTemplateUserRepo` and then execute the `UserServiceTest.testFindById()` method from project `6-ps-tx-solution` in debug mode in IntelliJ IDEA and take a look at the `userService` bean. In the Variables console you should see what is depicted in Figure 5-16.

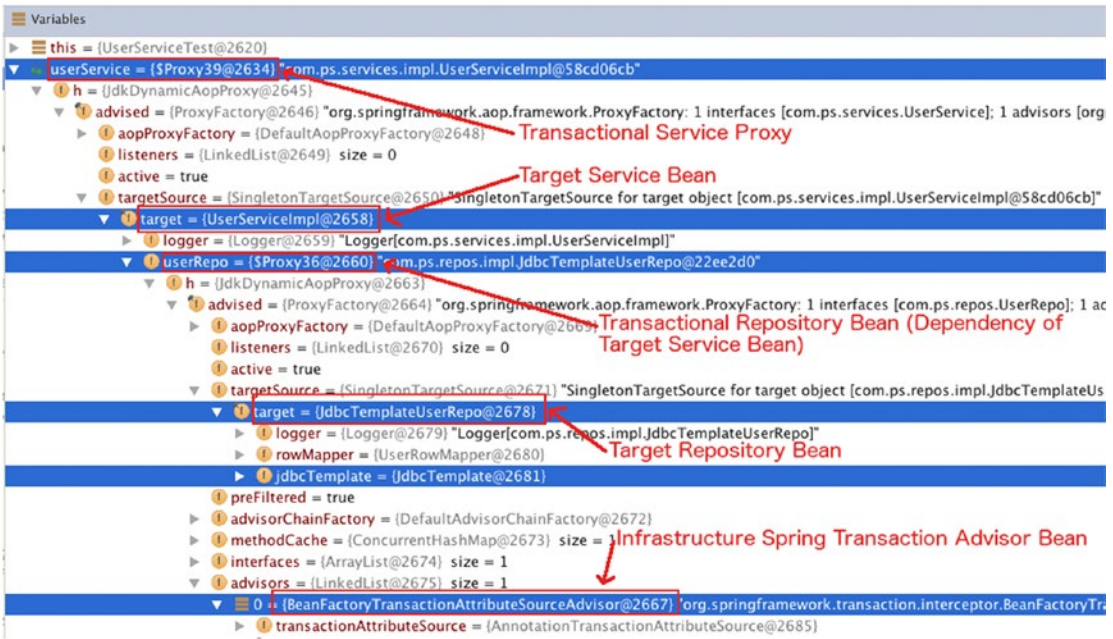


Figure 5-16. IntelliJ IDEA execution of test paused to analyze service and repository transactional bean

So there you have it, the end of the debate. Use `@Transactional` in the service layer or the DAO/repository layer, but not both. The service layer is the usual choice, because service methods call multiple repository methods that need to be executed in the same transaction. The only reason to make your repositories transactional is if you do not need a service layer at all, which is usually the case for small educational applications.

Spring Programatic Transaction Model

With the transaction declarative model, there is the benefit of flexible configuration and clean code, but transaction management is left fully to the transaction management provider. The programatic model, although a little more tedious to use, is practical when some control over transaction management is needed. To be able to have some control over what happens to the transactions, Spring provides the `TransactionTemplate` class. In the code snippet below, a programatic transactional service is depicted that makes use of an instance of `TransactionTemplate` to manage transactions.

```
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallback;
import org.springframework.transaction.support.TransactionTemplate;
...
@Service("programaticUserService")
public class ProgramaticUserService implements UserService {
```

```
    private UserRepo userRepo;
```



```

private TransactionTemplate txTemplate;

@Autowired
public ProgramaticUserService(UserRepo userRepo,
    PlatformTransactionManager txManager) {
    this.userRepo = userRepo;
    this.txTemplate = new TransactionTemplate(txManager);
}

@Override
public int updatePassword(Long userId, String newPass)
    throws MailSendingException {
    return txTemplate.execute(new TransactionCallback<Integer>() {
        @Override
        public Integer doInTransaction(TransactionStatus status) {
            try {
                User user = userRepo.findById(userId);
                String email = user.getEmail();
                sendEmail(email);
                return userRepo.updatePassword(userId, newPass);
            } catch (MailSendingException e) {
                status.setRollbackOnly();
            }
            return 0;
        }
    });
}

private void sendEmail(String email) throws MailSendingException {
    ... //not relevant for this section
}
}

```

The `status.setRollbackOnly()` method is called to instruct the transaction manager that the only possible outcome of the transaction may be a rollback, and not the throwing of an exception, which would in turn trigger a rollback.

**Distributed Transactions

A distributed transaction is a transaction that involves two or more transactional resources. The most obvious example here is an application that involves JMS and JDBC. Conceptually, what happens when a distributed transaction involving JMS and JDBC resources is executed is depicted in Figure 5-17.

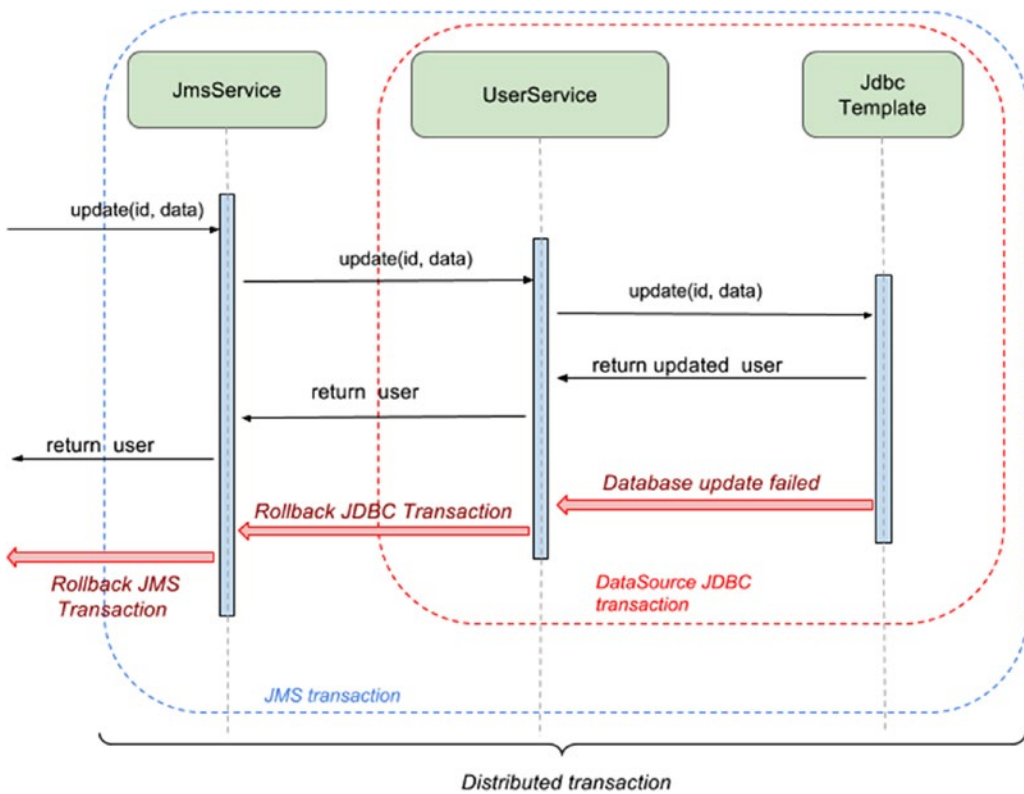


Figure 5-17. Conceptual UML sequence diagram for a distributed transaction involving JMS and JDBC resources

Both JmsService and UserService are transactional. The sequence or execution steps are as follows:

1. Start messaging transaction.
2. Receive message requesting update of a user record.
3. Start database transaction to edit the user record.
4. Commit the database transaction if database was updated with success. Otherwise, roll back the transaction.
5. Commit the messaging transaction if the database transaction was committed with success. If the database transaction was rolled back, roll back the messaging transaction.

Using distributed transactions requires a JTA and specific XA drivers.¹³ There are many open-source and commercial JTA providers: JBossTS, Java Open Transaction Manager (JOTM), and Atomikos.¹⁴ Since distributed transactions are not a topic for the official certification exam, nor part of the Spring Core components, this section will end here, because something way more interesting follows.

¹³The most popular article about Spring distributed transactions has since 2009 been David Sayer’s <http://www.javaworld.com/article/2077963/open-source-tools/distributed-transactions-in-spring-with-and-without-xa.html>.

¹⁴A complete list of steps showing how to configure and use the Atomikos JTA provider is available on the Spring official blog: <https://spring.io/blog/2011/08/15/configuring-spring-and-jta-without-full-java-ee/>.

Introducing Hibernate and ORM

The term JPA was introduced earlier, but the time has come for it to be covered in detail. JPA (Java Persistence API) is an interface for persistence providers to implement. There are many JPA providers available for Java applications: Hibernate, EclipseLink, Apache OpenJPA, etc.

The most popular of them is Hibernate, one of the open source Red Hat projects. Hibernate ORM is an **object relational mapping** framework that provides support for mapping an object oriented domain model to a relational database. This framework is open-source, and Hibernate has grown over the years into a full-fledged technology that has been split into numerous tools for domain model validation, indexing and searching a JPA for NoSQL databases (Hibernate OGM),¹⁵ and so on. If you are curious about the Red Hat Hibernate project family, you can find out more on their official site <http://hibernate.org/>. In this section, the basic details for configuring JPA with Hibernate will be covered.

Session and Hibernate Configuration

To configure JPA with Hibernate in a Spring application, the following components must be introduced:

- The `org.hibernate.SessionFactory` interface is the core component of Hibernate. An object of this type is thread-safe, shareable, and immutable. Usually, an application has a single `SessionFactory` instance, and threads servicing client requests obtain `Session` instances from this factory. Once a `SessionFactory` instance is created, its internal state is set. This internal state includes all of the metadata about Object/Relational Mapping.
- The `org.hibernate.Session` interface is the hibernate component representing a single functional unit, and it is the main runtime interface between a Java application and Hibernate. The session is a stateful object that manages persistent objects within the functional unit. It acts as a transactional-scoped cache, operations executed in a session are basically cached, and the changes are persisted to the datasource (the second-level cache) when the transaction is committed. A `Session` object can be obtained by calling `sessionFactory.getCurrentSession()`.
- `org.springframework.orm.hibernate*.HibernateTransactionManager`: this class is an implementation of `PlatformTransactionManager` for a single `SessionFactory`. Spring 4 currently contains three hibernate packages, one for each version supported (3, 4, and 5). There is a `HibernateTransactionManager` in each of them. The version for Hibernate 3 is currently deprecated and will probably be removed in Spring 5.

```
import org.springframework.transaction.PlatformTransactionManager;
...
@Bean
public PlatformTransactionManager transactionManager() {
    return new HibernateTransactionManager(sessionFactory());
}
```

¹⁵Hibernate OGM provides Java Persistence (JPA) support for NoSQL solutions. It reuses Hibernate ORM's engine but persists entities into a NoSQL datastore instead of a relational database. Read more about it on the official site: <http://hibernate.org/ogm/>.

- `org.springframework.orm.hibernate5.LocalSessionFactoryBuilder`: this is a utility class that can be used to create a `SessionFactory` bean. The session factory bean requires as parameters the `datasource` used by the application, the package where the entity classes can be found, and the hibernate properties.

```
import org.hibernate.SessionFactory;
import org.springframework.orm.hibernate5.LocalSessionFactoryBuilder;
...

@Bean
public SessionFactory sessionFactory() {
    return new LocalSessionFactoryBuilder(dataSource())
        .scanPackages("com.ps.ents")
        .addProperties(hibernateProperties())
        .buildSessionFactory();
}

@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.use_sql_comments", true);
    hibernateProp.put("hibernate.show_sql", true);
    return hibernateProp;
}
```

This class was introduced in Spring 3.1 to replace the class `org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean` that was used with Hibernate 3, and it was designed to work with Hibernate 4. As you can probably infer at this point, a version for Hibernate 5 exists, which was introduced in Spring 4.2. In case you are curious, an XML sample configuration for Hibernate 3 using the deprecated class is depicted in the following code snippet.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.
        AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="annotatedClasses">
        <list>
            <value>com.ps.ents.User</value>
            <value>com.ps.ents.Pet</value>
            <value>com.ps.ents.Request</value>
            <value>com.ps.ents.Response</value>
            <value>com.ps.ents.Review</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <value>
```

```

                hibernate.format_sql=true
                hibernate.show_sql=true
            </value>
        </property>
    </bean>

```

The "com.ps.ents" is the package in which the entity classes reside. They represent the metadata that Hibernate needs so it can map them to database objects. XML configuration of entity classes and Hibernate were used in previous versions, though not that much now. In the previous code sample, the class annotation `AnnotationSessionFactoryBean` does not have a `scanPackages` property, so the entities have to be listed in the XML configuration file as values in a list that is used as argument to set the `annotatedClasses` property. The new approach is more practical, since it removes the necessity of modifying the XML Spring configuration file when a new entity class is added to the project.

The `hibernateProperties` is a `java.util.Properties` that contains specific Hibernate properties. The most useful are listed below:

- `hibernate.dialect`: the value is a dialect class matching the database used in the application (ex: `org.hibernate.dialect.H2Dialect`).
- `hibernate.hbm2ddl.auto`: the value represents what Hibernate should do when the application starts: update the database to apply changes done in the metadata, re-create the database altogether or do nothing. Possible values: `none` (default value), `create-only`, `drop`, `create`, `create-drop`, `validate`, `update`. This is very practical with embedded test databases, because the developer can focus on the code instead of setting up the test database. If `create-drop` is used, Hibernate will scan all the entities and generate tables and relationships among them according to the specific annotations placed on their fields.
- `hibernate.format_sql`: if true, and the next property is true also, the generated SQL statements are printed to the console in a pretty and readable way.
- `hibernate.show_sql`: if true, all the generated SQL statements are printed to the console.
- `hibernate.use_sql_comments`: if true, Hibernate will put a comment inside the SQL statement to tell the developer what that statement is trying to do.
- Because Hibernate is an evolved tool, it is about time to introduce connection pooling. For this section, `HikariCP` was chosen to create the `dataSource` bean. `HikariCP` is open-source, small, and practical, since only one library needs to be added to the project, and it is said to be the fastest connection pool in the Java universe.¹⁶

```

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
...
@Bean(destroyMethod = "close")
public DataSource dataSource() {

```

¹⁶You can read more about the project on their GitHub page <https://github.com/brettwooldridge/HikariCP>.

```

try {
    HikariConfig hikariConfig = new HikariConfig();
    //datasource connection data
    hikariConfig.setDriverClassName(driverClassName);
    hikariConfig.setJdbcUrl(url);
    hikariConfig.setUsername(username);
    hikariConfig.setPassword(password);

    //connection pool specific configuration
    hikariConfig.setMaximumPoolSize(5);
    hikariConfig.setConnectionTestQuery("SELECT 1");
    hikariConfig.setPoolName("springHikariCP");
    hikariConfig.addDataSourceProperty("dataSource.cachePrepStmts", "true");
    hikariConfig.addDataSourceProperty("dataSource.prepStmtCacheSize", "250");
    hikariConfig.addDataSourceProperty("dataSource.prepStmtCacheSqlLimit", "2048");
    hikariConfig.addDataSourceProperty("dataSource.useServerPrepStmts", "true");

    HikariDataSource dataSource = new HikariDataSource(hikariConfig);
        return dataSource;
    } catch (Exception e) {
        return null;
    }
}

```

The SessionFactory bean is then injected into repositories and is used to create objects of types implementing `org.hibernate.query.Query<R>` that are executed and the result is returned, but more about that after entity classes are covered, just a little to keep things clear.

Hibernate supports all JPA 2.x annotations in the `javax.persistence.*` package and extends this package to provide behavior not supported by JPA and to perform specific enhancements. The metadata for Hibernate is made of annotations placed on classes, fields, and methods that define how those objects should be treated, what restrictions they have, and so on. The most important annotation is the `@Entity` annotation, which is part of the `javax.persistence.*` and marks classes as templates for domain objects, also called entities.

■ **CC** Classes annotated with `@Entity` are mapped to database tables matching the class name, unless specified otherwise using the `@Table` annotation.

```

import javax.persistence.Table;
import javax.persistence.Entity;
...
@Entity
@Table(name="P_USER")
public class User extends AbstractEntity {
...
}

```

Entity class members are annotated to specify purpose, column name (if different from the field name), validation rules, relationships with other entities, and many more that you will discover later in the chapter.

By default, all class members are treated as persistent unless annotated with `@Transient`. The `@Column` annotation is not necessary unless the database column name is required to be different from the field name or restrictions need to be applied for a field.

Example: unique, nullable, insertable, updatable, length, precision, scale.

The `@Id` annotation marks the field as the unique identifier for this entity type and matches the primary-key of the database table. `@Entity` and `@Id` are mandatory for a domain class.

The access type for entity class members annotated with persistence-specific annotations is FIELD. Their values are populated by Hibernate using reflection. Using setters and getters can be forced by using the `@Access` annotation, but this is not recommended, since it might interfere with other components. (Example: listeners and AOP advice)

Relationships between tables are defined by fields annotated with `@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@OneToOne`, and they match the database-equivalent relationship definitions. In the following code snippet, a `User` entity is depicted with most of its fields and annotations. The table that Hibernate will create in the database will be named `P_USER`.

The `@OneToMany` annotation is used to define a foreign key in the `P_PET` table linking pet records to records in `P_USER` and can have attributes defined that specify the behavior of the child entities. For example, in the code snippet below, using `@OneToMany(mappedBy = "owner", cascade = {CascadeType.REMOVE})` tells Hibernate that on deleting a user entity, all children entities of type `Pet` must be deleted as well.

```
import org.hibernate.validator.constraints.NotEmpty;
import javax.validation.constraints.NotNull;
import javax.persistence.Column;
import javax.persistence.OneToMany;
...
@Entity
@Table(name="P_USER")
@SequenceGenerator(name = "seqGen", allocationSize = 1)
public class User extends AbstractEntity {

    @NotEmpty
    @Column(nullable = false, unique = true)
    private String username;

    @Column(name="first_name")
    public String firstName;

    @NotNull
    @Enumerated(EnumType.STRING)
    @Column(name = "user_type")
    private UserType userType;

    @NotEmpty
    @Column(unique = true)
    private String email;

    @JsonIgnore
    @OneToMany(mappedBy = "owner",
        cascade = {CascadeType.REMOVE})
    private Set<Pet> pets = new HashSet<>();
...
}
```

In the `Pet` entity class, the `@ManyToOne` part of the relationship is declared, and it defines an exact name of the foreign key column using `@JoinColumn`. The foreign key field will be populated with a reference to the `User` entity that is a parent of this domain object. In JPA, the entity declaring the `@OneToMany` relationship is usually called a *parent* entity, and the one declaring the `@ManyToOne` is called a *child* entity.

```
import org.hibernate.validator.constraints.NotEmpty;
import javax.persistence.JoinColumn;
```

```
import javax.persistence.ManyToOne;
...
@Entity
@Table(name="P_PET")
public class Pet extends AbstractEntity {
...

    @ManyToOne
    @JoinColumn(name = "OWNER_ID", nullable = false)
    private User owner;
}

```

Also, since we are making use of inheritance to avoid writing duplicated code, an abstract class that defines common fields for all entity classes was introduced called `AbstractEntity`, to tell Hibernate that this class is not an entity class, but a template for other entity classes, the `@MappedSuperclass`. An architectural decision was made to group all infrastructure-related fields in this class: the primary key field marked with `@Id` and `@GeneratedValue`, which is used to allow auto-generation of the value, the Date fields that are modified when the object is edited, and the `@Version` annotated field that is used to ensure integrity when one is performing the merge operation and for optimistic concurrency control. An entity class can have only one field annotated with `@Version`.

```
import com.fasterxml.jackson.annotation.JsonIgnore;
import org.springframework.format.annotation.DateTimeFormat;
import javax.validation.constraints.NotNull;
...
@MappedSuperclass
public abstract class AbstractEntity implements Serializable {

    @JsonIgnore
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(updatable = false)
    protected Long id;

    @JsonIgnore
    @Column(name = "CREATED_AT", nullable = false)
    @NotNull
    @DateTimeFormat(pattern = "yyyy-MM-dd")
    protected Date createdAt;
    ...
    @JsonIgnore
    @Version
    public int version;
}

```

Session and Hibernate Querying

Entities are manipulated by `Session` instances that provide methods for search, persist, update, delete. This instance is also in charge of managing transactions and is a good replacement for JPA's `EntityManager`. The current session is obtained from the `SessionFactory` bean by calling

```
sessionFactory.getCurrentSession();
```


The queries for these operations are written in Hibernate Query Language, which allows for a more practical way of writing the SQL queries. HQL queries operate on domain objects and are transformed under the hood into matching SQL queries.

Example: retrieving an object when we know its ID becomes really easy to do with Hibernate.

```
@Override
public User findById(Long id) {
    return session.get(User.class, id);
}
```

And if we want to see the SQL native query generated by Hibernate, all we have to do is look in the log with the two hibernate SQL-specific parameters `show_sql` and `format_sql` set to true for the `SessionFactory` bean.

```
select
    user0_.id as id1_5_,
    user0_.CREATED_AT as CREATED_2_5_,
    user0_.MODIFIED_AT as MODIFIED3_5_,
    user0_.version as version4_5_,
    user0_.active as active5_5_,
    user0_.address as address6_5_,
    user0_.email as email7_5_,
    user0_.first_name as first_na8_5_,
    user0_.last_name as last_nam9_5_,
    user0_.password as passwor10_5_,
    user0_.rating as rating11_5_,
    user0_.user_type as user_ty12_5_,
    user0_.username as usernam13_5_
from
    P_USER user0_
where
    user0_.id=?
```

HQL supports placeholders and named parameters in queries written by the developer and can return single results or collections and does not need a mapping object. Remember the `RowMapper<T>?` No need for it with Hibernate. This is what ORM is good at. Based on the metadata represented by the annotations in the entity classes, Hibernate can easily transform database records into Java objects and vice versa under the hood. In the code snippet below, you can see a few different HQL queries.

```
//an empty list is returned when no record matches the criterion
List<Users> list = session.createQuery("from User u where username= ?")
//null is returned when no record matches the criterion
User user = (User) session.createQuery("from User u where u.id= :id").
    setParameter("id", userId).uniqueResult();

// update the user
User user = (User) session().createQuery("from User u where u.id= :id").
    setParameter("id", userId).uniqueResult();
user.setUsername(username);
session.update(user);

//save a new user in the database
public void save(User user) {
    session.persist(user);
}
```

```
// delete a user
public void deleteById(User user) {
    session.delete(user);
}
```

To synchronize domain objects with the database, the `Session` instance provides quite a few methods. The following are the most used:

- `update(entity)` is used to persist changes done to an existing database object.
- `persist(entity)` is used to save a new domain object into the database. If this object has other domain objects associated with it and the association is mapped with `cascade="persist"`, the `persist` operation will include them as well. This method does not return a value.
- `save(entity)` is used to save a new domain object into the database. Before saving the object, an identifier is generated. This operation applies to associated instances if the association is mapped with `cascade="save-update"`. This method returns the generated identifier.
- `saveOrUpdate(entity)` is used to save a domain object to the database. If the object exists, `update` is performed; otherwise, `save` is performed and the operation applies to associated instances if the association is mapped with `cascade="save-update"`.

When Hibernate is used in the application, a repository class will use the `SessionFactory` bean to manipulate data objects. The Hibernate-specific implementation of the `UserRepo` is depicted in the following code snippet. Notice the `SessionFactory` bean being injected and obtaining the current session.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;
...

@Repository
@Transactional
public class HibernateUserRepo implements UserRepo {
    private SessionFactory sessionFactory;
    @Autowired
    public HibernateUserRepo(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    /**
     * @return the transactional session
     */
    protected Session session() {
        return sessionFactory.getCurrentSession();
    }

    @Override
    public List<User> findAll() {
        return session().createQuery("from User u").list();
    }
    ...
}
```

The UML sequence diagram when Hibernate is used with Spring is depicted in Figure 5-18.

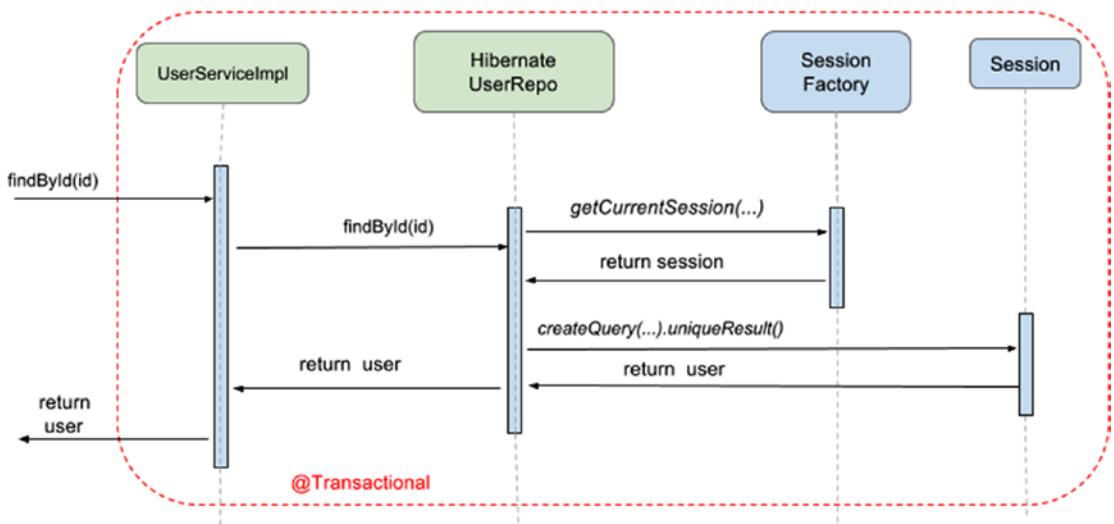


Figure 5-18. Conceptual UML sequence diagram for operations involving a Hibernate repository class

Aside from all these methods, Hibernate supports execution of native queries with `Session` instances as well.

```

import org.hibernate.query.NativeQuery;
...
public List<String> findAll() {
    NativeQuery<String> nq = session()
        .createNativeQuery("select first_name from P_USER");
    return nq.getResultList();
}
  
```

Exception Mapping

When things go wrong, Hibernate throws its own exceptions, which are equivalent in meaning to the Spring data access exceptions that were already covered in a previous section. Hibernate throws runtime data access exceptions that extend `HibernateException`, but these exceptions can be translated to Spring exceptions by declaring an exception translator bean. To enable this behavior, a postprocessor bean has to be declared that will look for all exception translator beans implementing `org.springframework.dao.support.PersistenceExceptionTranslator` and will advise all repository beans (classes annotated with `@Repository`) so that the translators can intercept the hibernate exceptions and apply the appropriate translation. In the code snippet below, you can see the postprocessor bean and the Hibernate-specific translator bean being declared in the configuration class.

```

import org.springframework.dao.annotation.
    PersistenceExceptionTranslationPostProcessor;
import org.springframework.orm.hibernate5.
    HibernateExceptionTranslator;
...
@Bean
public PersistenceExceptionTranslationPostProcessor petpp() {
  
```

```

        return new PersistenceExceptionTranslationPostProcessor();
    }
}
@Bean
public HibernateExceptionTranslator hibernateExceptionTranslator() {
    return new HibernateExceptionTranslator();
}

```

When exception translation using the `PersistenceExceptionTranslationPostProcessor` processor is not possible, perhaps because the repository classes are part of a third-party library, XML can be used to define an AOP advice that does the translation using the class `PersistenceExceptionTranslationInterceptor`.

```

<bean id="pExInterceptor"
    class="org.springframework.dao.support.PersistenceExceptionTranslationInterceptor" />
...
<aop:advisor pointcut="execution(* * ..Repo+.*(..))"
    advice-ref="pExInterceptor" />

```

! Hibernate is not a subject for the official certificate exam, but if you want to test your understanding of it, there is a project called `07-ps-hibernate-practice` that has four TODO tasks defined, numbered from 36 to 39.

Tasks 36–38 ask you to write some missing HQL queries in the `HibernateUserRepo` that can be tested with classes `TestHibernateUserRepo` and `TestUserService`.

Task 39 asks you to complete the `SessionFactory` bean declaration that is missing in the `TestDataConfig` class.

The proposed solutions can be found in project `07-ps-hibernate-solution`.

When executing any of the tests, search in the console log for the words `session` and `transaction` and notice how Hibernate and Spring work together:

```

...
DEBUG o.s.b.f.a.AutowiredAnnotationBeanPostProcessor - Autowiring by type
    from bean name 'hibernateUserRepo' to bean named 'sessionFactory'
DEBUG o.s.b.f.s.DefaultListableBeanFactory - Creating instance of bean
    'transactionManager'
INFO o.s.o.h.HibernateTransactionManager - Using DataSource
    HikariDataSource (springHikariCP)
    of Hibernate SessionFactory for HibernateTransactionManager
DEBUG o.s.t.a.AnnotationTransactionAttributeSource - Adding transactional method
    'HibernateUserRepo.findById' with attribute: PROPAGATION_MANDATORY,ISOLATION_DEFAULT; "
DEBUG o.s.o.h.HibernateTransactionManager - Participating in existing transaction
Hibernate:
    select
        user0_.id as id1_5_0_,
            ...
        user0_.username as usernam13_5_0_
    from
        P_USER user0_
    where
        user0_.id=?

```

```

DEBUG o.s.o.h.HibernateTransactionManager - Initiating transaction commit
DEBUG o.s.o.h.HibernateTransactionManager -
    Committing Hibernate transaction on Session ...
22:34:50.204 main DEBUG o.s.o.h.HibernateTransactionManager -
    Closing Hibernate Session ...
...

```

Hibernate API can be used to implement data access and participate in Spring-managed transactions. It is also completely agnostic regarding the database used. It has no dependency on Spring or the entity classes and provides hooks so Spring can manage transactions in a transparent manner. Implementation and configuration location for Hibernate `SessionFactory` instances, transaction manager, and database can be swapped with other implementations without any changes required in the code. If you use the `LocalSessionFactoryBuilder` Spring class to create the `SessionFactory` bean, it will be wrapped up in a proxy that will make sure that every session opened by this bean will participate in the current transaction. When Hibernate is used, declaring transactions as `readOnly` when they do not perform write operations could lead to considerable performance optimizations, because Hibernate will skip the flushing of the session (there is, after all, nothing to flush).

And now that Hibernate has been introduced, some more information about ORM is appropriate before we dive deep into JPA.

Object Relational Mapping

Object Relational Mapping, or ORM, is a method of mapping database objects to application objects and vice versa. This makes for easy handling of data objects. Aside from that, it also provides the possibility of querying the database using an object oriented approach. Hibernate's full name is Hibernate ORM, because it implements this technique to allow querying and persisting of data objects. A data object is also called a domain object or entity. Whatever its name, this object corresponds to a database object, usually a row from a table. There are more complex objects that can be defined and can encapsulate data from multiple tables, but since this topic is irrelevant for this book, the details will be kept to a minimum. Domain objects are easy to use in the application, because they encapsulate all data related to a table row stored in accessible fields. When a row in a table becomes a domain object, the following correspondences are made:

- The primary key value is stored in an ID field. This field is used to identify the object in the application. In the database, the identity is a simple topic. In Java, because objects are involved, things are a little different. Two domain objects can be logically equivalent, but only one of them has the ID field set with a primary key value, so the `equals` and `hashCode` methods must be adjusted to take this aspect into consideration.
- One-to-many relationships from the database are mapped to the HAS-A relationship. The domain object has a collection of domain objects as a field. Usually this relationship is bidirectional, and each object in the collection has a field referencing the parent object. This field is mapped in the database to the foreign key column.
- Entries from the same table can be mapped to different types of domain objects in the same hierarchy using a column value as discriminator.

Using an ORM framework introduces the following benefits:

- It provides mapping of database records to application objects.
- So no extra code needs to be written for this.

- It usually provides a rich object query language that is more intuitive and easier to use than native SQL.
- It provides easy navigation through objects using their relationships.
- It provides persistence through reachability. Look at the following code snippet:

```
public class User extends AbstractEntity {
    ...
    @OneToMany(mappedBy = "owner",
        cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Set<Pet> pets = new HashSet<>();
}
```

The cascade attribute defines what happens with child records from the P_PET table, which are mapped to Pet objects, when the parent domain object of type User is modified. If the User object is created together with the Pet objects at the same time, only the User object must be persisted to the database, since the persist operation will propagate to the child domain objects because of the CascadeType.PERSIST value. If the User object is deleted, child domain objects are deleted as well because of the CascadeType.REMOVE.

- It provides concurrency support: multiple processes can update the same data in parallel.
- It provides cache management per level:
 - per transaction (first-level cache): when an object is first loaded from the database, the object is stored in this cache, and subsequent requests for it will use the cache instead of going to the database.
 - per datasource (second-level cache at SessionFactory level): reduces trips to the database for read-heavy data and is shared by all sessions created by the SessionFactory bean. When a domain object is not found in the first-level cache, the next place to look is this cache. If found, the model object is also stored in the first-level cache before it is returned.
- transaction management and isolation.
- key management, since identifiers are automatically propagated and managed. Example: when a User object and its Pet instances are created in the system, the user ID (primary key) is generated and used automatically to populate the foreign key fields in the pet domain objects as well.
- ORM-specific code can be reused. (In the sample projects for this book, inheritance is used for domain objects, repositories, and services).
- ORM code has already been tested and is maintained by the creators of the framework; thus using an ORM reduces the effort of testing.

Aside from Hibernate, Spring supports integration with all major ORM/persistence providers such as EclipseLink, MyBatis, and Open JPA. Although really practical, ORM introduces a little lag for large amounts of data, so in this case, JDBC and native SQL should be used when speed of data processing is a performance criterion.

Java Persistence API

The previous example, depicting a Spring repository class using `SessionFactory` directly, ties Spring with Hibernate. Java Persistence API, also called **JPA**, introduces a common interface for object relational mapping and persistence that allows the ORM framework used to be switched easily. JPA is designed for operating on domain objects defined as POJOs. It replaces previous persistence mechanisms: EJB and JDO (Java Data Objects). It was first introduced in 2006 and has overcome initial limitations to successfully provide a set of specific JPA annotations that are supported by all ORM frameworks and persistence frameworks for Java. The annotations introduced in the previous section to configure domain objects are part of the `javax.persistence` package, which contains all JPA components.

The core JPA components are as follows:

- **Persistence Context:** a context containing a set of domain objects/entities in which for every persistent entity there is a unique entity instance.
- **Entity Manager:** as the name clearly states, such an object will manage entities, will take care of creation, update, querying, deletion. Entity Manager classes must extend `javax.persistence.EntityManager`, and instances are associated with a persistence context by annotating them with `@PersistenceContext`. Usually, these instances' life cycles are bound to the transaction in which the method is being executed, so they are managed by the container (in our case, the Spring container).
- **Entity Manager Factory:** again, the naming is very relevant for the purpose of such an object. Entity Manager Factory beans have the responsibility of creating application-managed Entity Manager instances. These factory classes must implement `javax.persistence.EntityManagerFactory`. They are thread-safe, shareable, and they represent a single datasource and persistence context.
- **Persistence Unit:** a group of entity classes defined by the developer to map database records to objects that are managed by an Entity Manager, basically all classes annotated with `@Entity`, `@MappedSuperclass`, and `@Embedded` in an application. All entity classes must define a primary key, must have a non-arg constructor or not allowed to be final. Keys can be a single field or a combination of fields. This set of entity classes represents data contained in a single datasource. Multiple persistence units can be defined within the same application. Configuration of persistence units can be done using XML. In the official documentation it is specified that a `persistence.xml` file must be defined under the `META-INF` directory, but if Spring and Java Configuration are used, there is no need for that file. The following code snippet depicts a `persistence.xml` sample file.

```
<persistence>
  <persistence-unit name="petSitterPU">
    <description>This unit manages pets, users,
      requestsm responses and reviews.
    </description>
  </persistence-unit>
</persistence>
```

- **JPA Provider:** the framework providing the backend for the JPA, the one that it is actually doing the heavy lifting. The most frequently used frameworks that implement the JPA interface are these:
 - **Hibernate** provides an implementation for `EntityManager`, the `org.hibernate.jpa.HibernateEntityManager`, but starting with Hibernate 5.2, `org.hibernate.engine.spi.SessionImplementor` is used, because it now extends `EntityManager` directly. Of course, these are internal details, since in the configuration, only the `org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter` class is needed. Hibernate is used inside the JBoss application server.
 - **EclipseLink:** is used inside the Glassfish application server
 - **Apache OpenJPA:** is used inside Weblogic, WebSphere, and TomEE.
 - **Data Nucleus:** used by Google App Engine.

Any of them can be used outside the application servers as well.

Configure Spring and JPA with Hibernate support

For the code associated with this section, which you can find under project `09-ps-data-jpa-practice`, Hibernate will be used as a JPA provider. The most recent version of Hibernate is production ready, compliant with JSR-338 for JPA 2.1 Specification, and is still compatible with JPA 2.0. Modifying the application in project `07-ps-hibernate-practice` to support JPA implies the following changes:

1. The `SessionFactory` bean declaration is no longer needed, and it will be replaced by a bean declaring an `Entity Manager Factory` bean. The Spring-specific `LocalContainerEntityManagerFactoryBean` class will be used for this.

```
import javax.persistence.EntityManagerFactory;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
...
@Configuration
@EnableTransactionManagement
public class TestDataConfig {

    @Bean
    public EntityManagerFactory entityManagerFactory(){
        LocalContainerEntityManagerFactoryBean factoryBean =
            new LocalContainerEntityManagerFactoryBean();
        factoryBean.setPackagesToScan("com.ps.ents");
        factoryBean.setDataSource(dataSource());
        factoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        factoryBean.setJpaProperties(hibernateProperties());
        factoryBean.afterPropertiesSet();
        return factoryBean.getNativeEntityManagerFactory();
    }

    ...
}
```


In the method above, the `LocalContainerEntityManagerFactoryBean` is created by explicit instantiation, and is not created by Spring. So the `afterPropertiesSet()` that initializes the factory object must be called explicitly. The `LocalContainerEntityManagerFactoryBean` object is used to create a raw `EntityManagerFactory` bean as returned by the `PersistenceProvider` implementation, in this case `org.hibernate.jpa.HibernatePersistenceProvider`. The `EntityManagerFactory` instance created by the factory bean is retrieved by calling `factoryBean.getNativeEntityManagerFactory()`.

The `HibernateJpaVendorAdapter` exposes Hibernate's persistence provider and `EntityManager` extension interface. The locations where the persistence metadata can be found are set by the `setPackagesToScan(...)` method, and the `datasource` bean is required as well to properly create an `EntityManagerFactory`.

2. The `EntityManager` will be replaced by a bean of type `JpaTransactionManager` that will use the `EntityManagerFactory` implementation to associate Entity Manager operations with transactions.

```
import org.springframework.orm.jpa.JpaTransactionManager;
...
@Configuration
@EnableTransactionManagement

public class TestDataConfig { @Bean
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new JpaTransactionManager(entityManagerFactory());
    }

    ...
}
```

3. The repository classes will be modified to use an instance of type `EntityManager` mapped to the application persistence context. The annotation `@PersistenceContext` expresses a dependency on a container-managed `EntityManager` and its associated persistence context.¹⁷ This field does not need to be autowired, since the `@PersistenceContext` annotation is picked up by an infrastructure Spring bean postprocessor bean of type `org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor` (class) that makes sure to create and inject an `EntityManager` instance. To create this instance, the backend ORM is used, in this case Hibernate 5.x, so the `entityManager` bean will be of type `org.hibernate.engine.spi.SessionImplementor`.

```
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
```

¹⁷The `EntityManager` instance annotated with `@PersistenceContext` cannot be accessed from a constructor, since it cannot be created and associated with the persistence context in the constructor. The reason for this is the definition of the `@PersistenceContext`. This annotation has the following meta-annotation defined: `@Target(value=TYPE,METHOD,FIELD)`. Full JavaDoc API here: <http://docs.oracle.com/javase/7/api/javax/persistence/PersistenceContext.html>.

```

...
@Repository
public class JpaUserRepo implements UserRepo {

    private EntityManager entityManager;

    @PersistenceContext

    void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    ...
}

```

In Figure 5-19, the execution of a test was paused in debug mode to depict the type of entity manager being injected into the repository class.

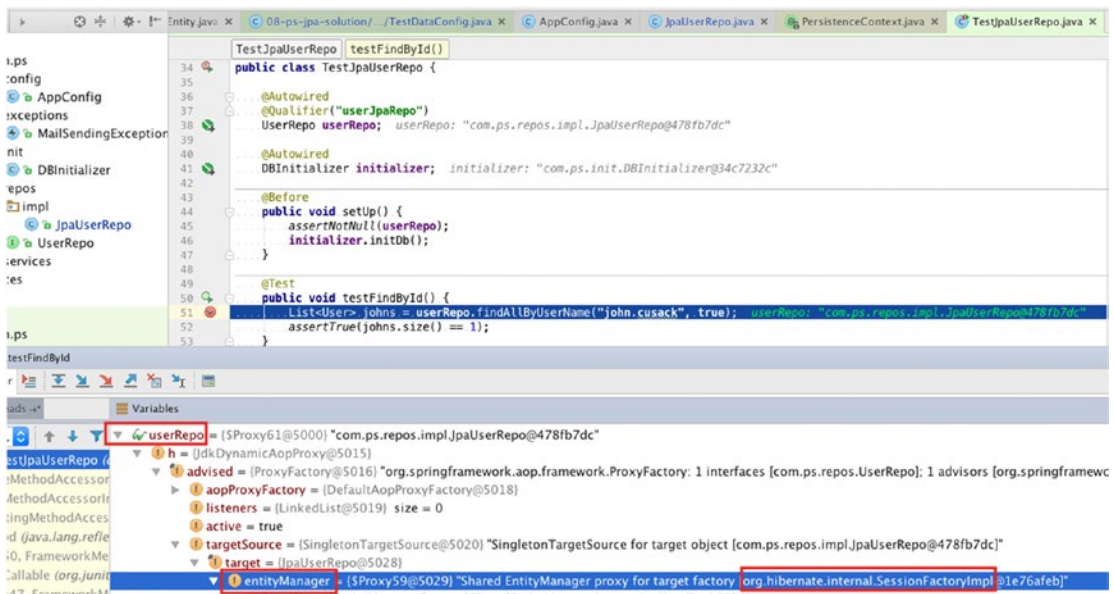


Figure 5-19. `TestJpaUserRepo.testFindById()` execution paused in debug mode to show the type of the `entityManager` bean that was injected into the `JpaUserRepo` class.

The methods of the repository need to be modified to make use of the `entityManager` bean. The most interesting part of the `EntityManager` API is depicted in Table 5-1.

Table 5-1. *EntityManager JPA methods*

Method	Action	Comment
<code><T> T find(Class<T> cl, Object pk);</code>	Find an entity by its primary key	Equivalent to: <i>select from table t where t.pk= PK_VAL</i>
Query <code>createQuery(String ql);</code>	Create a JPQL query	Returns a collection if <i>getResultList()</i> is called on the JPQL Query object. Returns a single object if <i>getSingleResult()</i> is called on the JPQL Query object.
Query <code>createNamedQuery(String name)</code>	Create a JPQL query from a named query in the metadata	Returns a collection if <i>getResultList()</i> is called on the JPQL Query object. Returns a single object if <i>getSingleResult()</i> is called on the JPQL Query object.
<code>void persist(Object obj)</code>	Adds the entity to the persistence context.	Equivalent to: <i>insert into table...</i>
<code><T> T merge(T entity);</code>	Merge the state of the given entity into the current persistence context.	Equivalent to: <i>update table t... where t.pk=PK_VAL</i>
<code>void flush()</code>	Persist persistence context contents to the database immediately.	Use carefully.
<code>void refresh(Object entity)</code>	Reload a state for an entity from the database.	Changes in the persistence context are discarded, so use carefully.
<code>void remove(Object entity)</code>	Removes the entity from the persistence context.	Equivalent to: <i>delete from table t where t.pk=PK_VAL</i>

JPQL is an acronym for Java Persistence Query Language.

Comparing with the pure Hibernate implementation with Spring, only the declaration of the transaction manager is changed, the `sessionFactory` bean, being replaced with `entityManagerFactory`. The `datasource` declaration does not change, and the hibernate properties are still needed, since Hibernate is still the backing ORM framework. Also, transactions are still needed, so `@EnableTransactionManagement` remains as well.

The conceptual UML sequence diagram when JPA is used to search a user by its ID looks like the one in Figure 5-20.

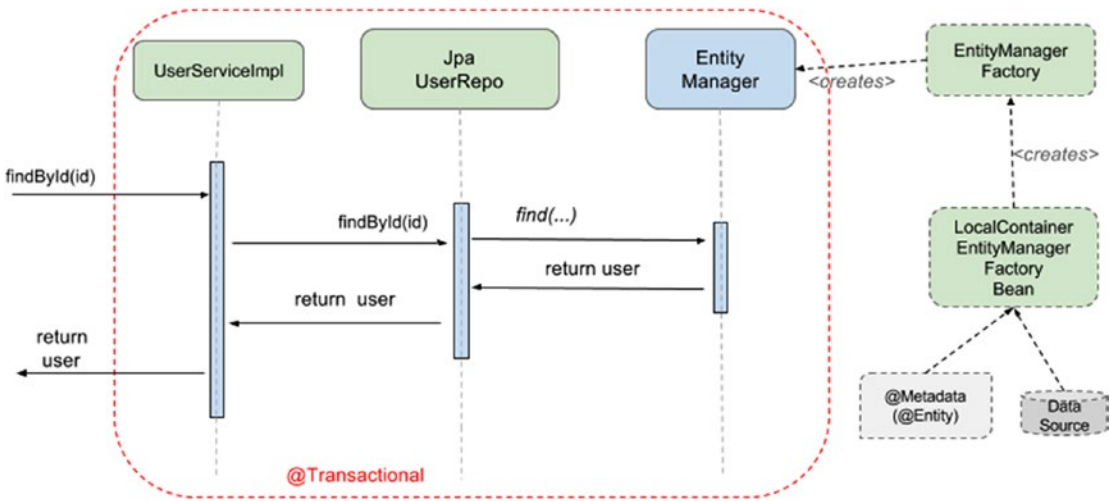


Figure 5-20. Conceptual UML sequence diagram when Spring is used with JPA

JPA Querying

The simplest operation with JPA is querying an object by its ID. There is no need to write a query, since EntityManager provides a method for this. It returns null when a user cannot be found. It returns an object of the entity type provided as argument. There is no need for casting, since under the hood, generics are used, just as with pure Hibernate.

```

public User findById(Long id) {
    return entityManager.find(User.class, id);
}
    
```

JPQL stands for JPA Query language. It is used to write domain object queries in a similar manner to HQL.

```

public List<User> findAllByUsername(String username, boolean exactMatch) {
    if (exactMatch) {
        return entityManager.createQuery("from User u where username= ?")
            .setParameter(0, username).getResultList();
    } else {
        return entityManager.createQuery("from User u where username like ?")
            .setParameter(0, "%" + username + "%").getResultList();
    }
}
    
```

And named parameters are supported too:

```

public List<User> findAllByUsername(String username, boolean exactMatch) {
    if (exactMatch) {
        return entityManager.createQuery("from User u where username= :un")
            .setParameter("un", username).getResultList();
    } else {
        return entityManager.createQuery("from User u where username like :un")
    
```

```

        .setParameter("un", "%" + username + "%").getResultList();
    }
}

```

Named queries are part of the metadata, and are defined with the annotation `@NamedQuery`, which must be placed on the entity class that the query manages. The example above could be simplified by declaring two named queries. The annotation `@NamedQueries` can be used to group multiple queries together.

```

//User.java
import javax.persistence.NamedQuery;
import javax.persistence.NamedQueries;
...
@Entity
@Table(name="P_USER")
@SequenceGenerator(name = "seqGen", allocationSize = 1)
@NamedQueries({
    @NamedQuery(name=User.FIND_BY_USERNAME_EXACT,
        query = "from User u where username= ?"),
    @NamedQuery(name=User.FIND_BY_USERNAME_LIKE,
        query = "from User u where username like ?")
})
public class User extends AbstractEntity {
    public static final String FIND_BY_USERNAME_EXACT = "findByUsernameExact";
    public static final String FIND_BY_USERNAME_LIKE = "findByUsernameLike";

    ...// entity fields and methods
}

//JpaUserRepo.java
import static com.ps.ents.User.FIND_BY_USERNAME_EXACT;
import static com.ps.ents.User.FIND_BY_USERNAME_LIKE;
...

@Repository("userJpaRepo")
public class JpaUserRepo implements UserRepo {

    public List<User> findAllByUserName(String username, boolean exactMatch) {
        if (exactMatch) {
            return entityManager.createNamedQuery(FIND_BY_USERNAME_EXACT)
                .setParameter(0, username).getResultList();
        } else {
            return entityManager.createNamedQuery(FIND_BY_USERNAME_LIKE)
                .setParameter(0, "%" + username + "%").getResultList();
        }
    }
}
...
}

```

Named queries support named parameters as well, and this is the recommended way of writing all queries, because it makes them more readable and prevents errors caused by mistaken parameter indexes.

```

//User.java
@Entity

```

```

@Table(name="P_USER")
@SequenceGenerator(name = "seqGen", allocationSize = 1)
@NamedQueries({
    @NamedQuery(name=User.FIND_BY_USERNAME_EXACT,
        query = "from User u where username= :un"),
    @NamedQuery(name=User.FIND_BY_USERNAME_LIKE,
        query = "from User u where username like :un")
})
public class User extends AbstractEntity {
    public static final String FIND_BY_USERNAME_EXACT = "findByUsernameExact";
    public static final String FIND_BY_USERNAME_LIKE = "findByUsernameLike";
    ...// entity fields and methods
}

//JpaUserRepo.java
import static com.ps.ents.User.FIND_BY_USERNAME_EXACT;
import static com.ps.ents.User.FIND_BY_USERNAME_LIKE;
...

@Repository("userJpaRepo")
public class JpaUserRepo implements UserRepo {
    public List<User> findAllByUserName(String username, boolean exactMatch) {
        if (exactMatch) {
            return entityManager.createNamedQuery(FIND_BY_USERNAME_EXACT)
                .setParameter("un", username).getResultList();
        } else {
            return entityManager.createNamedQuery(FIND_BY_USERNAME_LIKE)
                .setParameter("un", "%" + username + "%").getResultList();
        }
    }
    ...
}

```

When the query is expected to return multiple results, the method `.getResultList()` should be used. When a single result is expected, the method `getSingleResult()` should be used, and a casting must be done. JPA provides another method for querying entities using Criteria Queries. They are a part of the JPA API that can be used for creating ad hoc queries. It was introduced in JPA 2. Although it might look complicated, it is quite useful for complicated queries. For a simple query like the one implemented in the following code snippet, which only searches for users having a common last name, it looks quite impractical.

```

import javax.persistence.criteria.*;
...
@Override
public List<User> findAllByLastName(String username) {
    //create the query
    CriteriaBuilder builder= entityManager.getCriteriaBuilder();
    CriteriaQuery<User> query = builder.createQuery(User.class);

```

```

Root<User> userRoot = query.from(User.class);
ParameterExpression<String> value = builder.parameter(String.class);
query.select(userRoot).where(builder.equal(userRoot.get("lastName"), value));

// execute the query
TypedQuery<User> tquery = entityManager.createQuery(query);
tquery.setParameter(value,username);
return tquery.getResultList();
}

```

Aside from these methods, JPA also supports execution of native queries, using the method `Query createNativeQuery(String sqlString)`; although when not using managed objects, for more control and efficiency, `JdbcTemplate` is more appropriate.

```

import javax.persistence.Query;
...
public List<String> findAllFirstNames() {
    Query query = entityManager.createNativeQuery(
        "select first_name from P_USER"
    );
    return query.getResultList();
}

```

Persistence operations that fail throw JPA-specific exceptions. But the Spring exception translator bean takes care of translating these types of exceptions into Spring Data Access as well, thus actually hiding the persistence provider.

Advanced JPA, JTA, JNDI

When JTA is used in a project, the `persistence.xml` file **must be changed to include the datasource and the persistence unit declaration**:

```

<persistence>
  <persistence-unit name="petSitterPU">
    <description>This unit manages pets, users,
    requests, responses and reviews.
    </description>
    <jta-data-source>dataSource</jta-data-source>
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="hibernate.dialect"
        value="org.h2.Driver"/>
      <!-- hibernate properties here -->
    </properties>
  </persistence-unit>
</persistence>

```

When JTA is used and the `EntityManagerFactory` is provided by an application server such as JBoss or WebSphere, it can be retrieved using a JNDI lookup.

```

<jee:jndi-lookup id="entityManagerFactory" jndi-name="persistence/petSitterEMF" />

```

! If you want to test your understanding of working with Spring and JPA, being backed up by Hibernate, you can take a look at `08-ps-jpa-practice`. It contains only three TODOs, numbered from 40 to 42.

Task 40 is located in the `TestDataConfig` class and asks you to provide all the properties necessary for the `LocalContainerEntityManagerFactoryBean` class so that it can be used to configure a `EntityManagerFactory` bean correctly.

Task 41 is also located in the `TestDataConfig` class and asks to provide a proper transaction manager bean declaration.

Task 42 is located in the `JpaUserRepo` class and asks you to annotate the setter for the `EntityManager` instance correctly so the tests in `TestJpaUserRepo` can be executed and they pass.

As a bonus task, you can enrich the `UserServiceImpl` class with new methods that will be tested by the `UserServiceTest` class.

Spring Data JPA

Spring Data is a Spring project designed to help defining repository classes in a more practical way. Repository classes have a lot of common functionality, so the Spring team tried to provide the possibility to offer this functionality out of the box. The solution was to introduce abstract repositories that can also be customized by the developer to reduce the boilerplate code required for data access. To use Spring Data components in a JPA project, a dependency on the package `spring-data-jpa` must be introduced.

The central interface of Spring Data is `Repository<T, ID extends Serializable>`. The full hierarchy can be seen in [Figure 5-21](#).

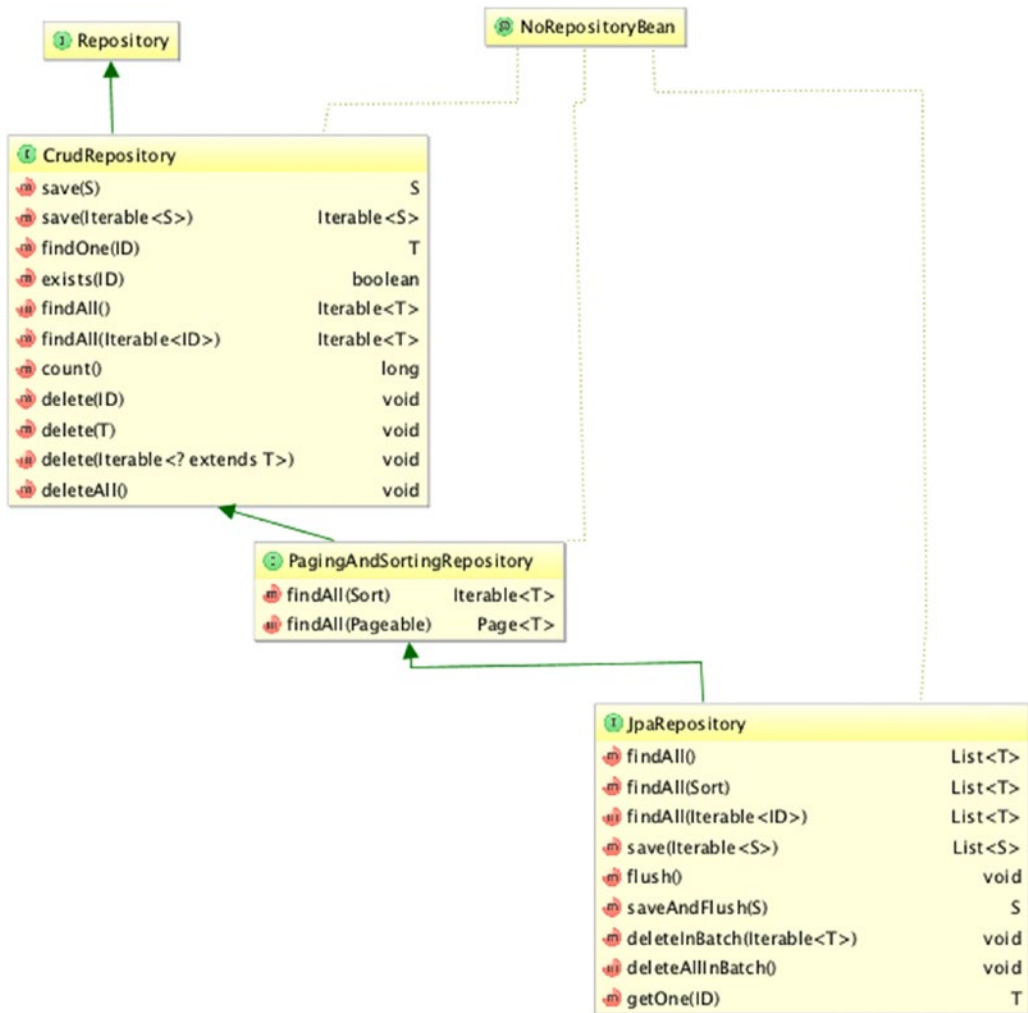


Figure 5-21. Spring Data JPA Repository hierarchy

The `NoRepositoryBean` annotation is used to exclude repository interfaces from being picked up, and repository instances will not be created for them.

Typically, a repository interface defined by a developer will extend one of the interfaces in the `Repository<T, ID extends Serializable>` hierarchy. This will expose a complete set of methods to manipulate entities.

If all that is needed is to extend the `Repository<T, ID extends Serializable>` interface, but you do not like the idea of extending a Spring component, you can avoid that by annotating your repository class with `@RepositoryDefinition`. This will have the same effect as extending the `Repository` interface, since this interface is empty, so it is used as a marker interface that can be replaced by an annotation.

Since the project for this section needs JPA, the repositories will extend the interface `JpaRepository<T, ID extends Serializable>`, which contains a set of default method skeletons out of the box, sparing the developer quite a lot of work. When a custom repository interface extends `JpaRepository`, it will automatically be enriched with functionality to save entities, search them by ID, retrieve all of them from the database, delete entities, flush, etc. (Just look at Figure 5-21, where all methods are listed.)

Usually, repository classes must perform custom and more complex queries that are not covered by the default methods provided by a Spring Data repository. In this case, the developer must define its own methods for Spring to implement when the repository instance is created. To tell Spring what those methods should do, the `@Query` annotation is used to annotate them. Inside that annotation should be a query definition that will be executed at runtime and the results returned. In the code snippet below, you can see what the `UserRepo` component looks like when Spring Data JPA is used.

```
package com.ps.repos;

import com.ps.ents.User;
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 UserRepo extends
    JpaRepository<User, Long> {

    @Query("select u from User u where u.username like %?1%")
    List<User> findAllByUsername(String username);

    // using named parameters
    @Query("select u from User u where u.username= :un")
    User findOneByUsername(@Param("un") String username);

    @Query("select u.username from User u where u.id= :id")
    String findUsernameById(Long id);

    @Query("select count(u) from User u")
    long countUsers();
}
```

Every `Repository` interface must be linked to the type of domain object it handles and the type of the primary key, which is why in the example above, the `UserRepo` interface extends `JpaRepository<User, Long>`. So `UserRepo` will manage `User` domain objects with a `Long` primary key value.

Interfaces like this are called *instant repositories*, because they can be created instantly by extending one of the Spring-specialized interfaces. So under the hood, as you are probably suspecting by now, Spring

creates a proxy object that is a fully functioning repository bean. Any additional functionality that is not provided by default can be easily implemented by defining a method skeleton and providing the desired functionality using annotations.

To tell Spring that it must create repository instances, a new configuration component must be introduced. The `@EnableJpaRepositories` tells Spring that it must create repository instances. As a parameter, the base package where the custom repository interfaces have been declared must be provided. Aside from that, the configuration also requires a persistence unit manager. In the code snippet below, you can see all the beans involved in the Spring JPA configuration.

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager;
import org.springframework.orm.jpa.persistenceunit.PersistenceUnitManager;
...

```

```
@Configuration
@EnableJpaRepositories(basePackages = {"com.ps.repos"})
@Import(DataSourceConfig.class)
public class PersistenceConfig {

    @Autowired
    DataSource dataSource;

    @Autowired
    Properties hibernateProperties;

    @Bean
    public EntityManagerFactory entityManagerFactory(){
        LocalContainerEntityManagerFactoryBean
            factoryBean = new LocalContainerEntityManagerFactoryBean();
        factoryBean.setPersistenceUnitManager(persistenceUnitManager());
        factoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
        factoryBean.setJpaProperties(hibernateProperties);
        factoryBean.afterPropertiesSet();
        return factoryBean.getNativeEntityManagerFactory();
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new JpaTransactionManager(entityManagerFactory());
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor exceptionTranslation(){
        return new PersistenceExceptionTranslationPostProcessor();
    }

    @Bean
    public PersistenceUnitManager persistenceUnitManager(){
        DefaultPersistenceUnitManager
            persistenceUnitManager = new DefaultPersistenceUnitManager();
        persistenceUnitManager.setPackagesToScan("com.ps.ents");
        persistenceUnitManager.setDefaultDataSource(dataSource);
        return persistenceUnitManager;
    }
}

```

The configuration is not different from the JPA backed up by Hibernate, and Hibernate is used as a persistence tool in this case, too. The only difference is the introduction of the `PersistenceUnitManager` bean. This is used by Spring to provide the internally generated repository with access to datasource and transactional behavior. Because Spring creates the repositories itself for this exact purpose, the `@Repository` annotation is no longer needed.

Because this way of creating repositories is fast and practical, it is currently the preferred way to implement JPA in the Spring application. The current version of this library as this book is being written is 1.10.2.RELEASE. This library is part of the Spring Data family¹⁸, a project designed to make the creation of repository components as practical as possible, regardless of the datasource used. In the following section, you will be introduced to Spring Data MongoDB, the library that provides utilities to integrate Spring with the NoSQL database called MongoDB.

! Before starting the next section, you can play with Spring Data JPA in the `09-ps-data-jpa-practice` project. There is a bonus TODO task numbered with 44, located in the `UserRepo` Java file, that challenges you to turn that interface into a Spring Data JPA repository so the tests in `TestUserRepo` class will pass.

**Spring and MongoDB

This is a bonus section, and information covered here does not appear in the official exam. But since NoSQL databases are getting used more and more, it was considered appropriate at least to scratch the surface. The NoSQL databases are a product of more and more content being generated and shared via the Internet. Relational databases are robust, designed to store objects and connections between them in a very structured manner. But content is not always structured, and the effort to normalize it to be stored in a relational database can become cumbersome and very costly in resources.

A more appropriate solution would be to have a database that does not require perfectly structured data, that is cloud friendly and scalable. And thus NoSQL databases were born to support storage and fast access of such amounts of poorly organized, complex, and unpredictable content, also referred to as BigData. There are currently over 225 NoSQL databases,¹⁹ and depending on their internal organization, they can be categorized as follows:

- **Key-Values Stores:** a hash table is used with a unique key and a pointer to a particular item of data (e.g., Amazon SimpleDB, Redis). It is the simplest to implement, but it is inefficient when only part of a value must be queried or updated.
- **Column Family Stores:** data is organized in columns, keys are still used, but they point to a column family. They were created to store and process very large amounts of data distributed over many machines (e.g., HBase, Cassandra).
- **Document Databases:** similar to Key-Values Stores, but the model is versioned documents of collections of key-value pairs. The semistructured documents are stored in formats like JSON. This type of database supports efficient querying (e.g., MongoDB, CouchDB).
- **Graph Databases:** to store the data, a flexible graph model is used that can scale across multiple machines (Infinite Graph, Neo4j).

¹⁸Project official page: <http://projects.spring.io/spring-data/>

¹⁹You can read about them here: <http://nosql-database.org/>.

For the code in this section, MongoDB Community edition²⁰ was used, because it is lightweight and easy to install on any operating system. The following steps were taken to set up the project 09-ps-mongo-sample:

- The first step is to install it on your system. Instructions to install on any operating system can be found on their official site here: <https://docs.mongodb.com/manual/administration/install-community/>.
- The next step is to start MongoDB. On every operating system there is an executable that can do this. The database service will be started on the 27017 port.

```
iuliana.cosmina@home - $ mongod --dbpath temp/mongo-db/
CONTROL initandlisten MongoDB starting : pid=89126 port=27017
      dbpath=/Users/iuliana.cosmina/temp/mongo-db/ 64-bit host=home
CONTROL initandlisten db version v3.2.8
```

- Open the mongo shell and test what database is used by executing the db command. The returned result should be test.

```
iuliana.cosmina@home - $ mongo
MongoDB shell version: 3.2.8
connecting to: test
Server has startup warnings:
2016-07-25T22:00:24.783+0300 I CONTROL  initandlisten
2016-07-25T22:00:24.783+0300 I CONTROL  initandlisten
      ** WARNING: soft rlimits too low.
         Number of files is 256, should be at least 1000
> db
test
>
```

For the purpose of the code sample in this section, there is no need to create a new database, so test will do.

- Create a domain object class that will be mapped to a MongoDB object. The class must have an identified field that will be annotated with the Spring Data special annotation @Id from the package org.springframework.data.annotation. Instances of this type will become entries in a collection named the same as the class but lowercased: user.

```
package com.ps.ents;

import org.springframework.data.annotation.Id;
import java.math.BigInteger;

public class User {

    @Id
    private BigInteger id;
    private String email;
    private String username;
    ... // other fields
```

²⁰Official site here: <https://www.mongodb.com/community>.

```

    public BigInteger getId() {
        return id;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
... // other setters and getter
@Override
public String toString() {
    return "User{" +
        "id=" + id +
        ", email='" + email + '\'' +
        ", username='" + username + '\'' +
        ", password='" + password + '\'' +
        ", rating=" + rating +
        ", active=" + active +
        ", firstName='" + firstName + '\'' +
        ", lastName='" + lastName + '\'' +
        '}';
}
}

```

- Create a new UserRepo interface that will extend the Spring Data MongoDB-specialized interface `MongoRepository<T, ID extends Serializable>`.

```

package com.ps.repos;

import com.ps.ents.User;
import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;
import java.util.List;

public interface UserRepo extends MongoRepository<User, Long> {

    @Query("{'username': { '$regex' : ?0 }}")
    List<User> findAllByUsername(String username);
}

```

Every `RepoMongoRepository` interface must be linked to the type of object it handles and the type of unique identifier, which is why in the example above, the `UserRepo` interface extends `MongoRepository<User, Long>`. So `UserRepo` will manage `User` objects with a `Long` unique identifier.

- Create a configuration class and annotate it with `@EnableMongoRepositories` to enable creation of MongoDB repository instances. This annotation is similar in functionality to `@EnableJpaRepositories`, but needs for the package(s) where the Mongo repository classes are declared to be provided as a value for its `basePackages` attribute.

```
package com.ps.config;

import com.mongodb.MongoClient;
...
import org.springframework.data.mongodb.MongoDbFactory;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoDbFactory;
import org.springframework.data.mongodb.repository.config.
EnableMongoRepositories;

@Configuration
@EnableMongoRepositories(basePackages = "com.ps.repos")
@ComponentScan(basePackages = { "com.ps.init"})
public class AppConfig {

    public static final String DB_NAME = "test";
    public static final String MONGO_HOST = "127.0.0.1";
    public static final int MONGO_PORT = 27017;

    @Bean
    public MongoDbFactory mongoDb() throws Exception {
        return new SimpleMongoDbFactory(new MongoClient(
            MONGO_HOST, MONGO_PORT), DB_NAME);
    }

    @Bean
    public MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongoDb());
    }
}
```

The `MongoTemplate` is used by the repository instances to manipulate data in the user collection. Create a test class to test the User data manipulation.

- Create a test class to test the User data manipulation.

```
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { AppConfig.class})
public class TestUserRepo {
    @Autowired
```

```

    UserRepo userRepo;

    @Test
    public void testFindById() {
        List<User> johns = userRepo.findAllByUserName("john");
        assertTrue(johns.size() == 2);
        logger.info(johns.toString());
    }

    @Test
    public void testFindAll() {
        List<User> users = userRepo.findAll();
        assertTrue(users.size() == 5);
    }

    @Test
    public void testNoFindById() {
        User user = userRepo.findOne(99L);
        assertNull(user);
    }

    @Test
    public void testCreate() {
        User diana = DBInitializer.buildUser("diana.ross@pet.com");
        diana.setPassword("test");
        diana = userRepo.save(diana);
        assertNotNull(diana.getId());
    }
    ...
}

```

The first time an object is saved in the collection, the collection is created as well. After the first run of one of the test methods, the mongo shell can be used to inspect the contents of the user collection using the `db.user.find()` function. The contents of the collection will be formatted in JSON form. For a pretty print, use `db.user.find().pretty()`.

```

> db.user.find().pretty()

{ "_id" : ObjectId("57966a4a17c62e0eae6b4b20"),
  "_class" : "com.ps.ents.User",
  "email" : "johnny.big@pet.com",
  "username" : "johnny.big",
  "rating" : 0,
  "active" : true,
  "firstName" : "johnny",
  "lastName" : "big" }

{ "_id" : ObjectId("57966a4a17c62e0eae6b4b24"),
  "_class" : "com.ps.ents.User",
  "email" : "john.cusack@pet.com",
  "username" : "john.cusack",

```



```
"rating" : 0,
"active" : true,
"firstName" : "john",
"lastName" : "cusack" }
```

As you have probably noticed, transactions have been mentioned nowhere in the MongoDB section. This is because transactional behavior is handled by a transaction manager. As this book is being written, there is not yet an implementation of Spring's `PlatformTransactionManager` for MongoDB, since this type of database is not transactional in the ACID sense.

The Spring team has set up a public GitHub repository <https://github.com/spring-projects/spring-data-book> containing an example of a Spring simple application with most of the supported NoSQL databases. If you are curious, feel free to clone it and try the code examples.

There will be no practice section at the end of the chapter, since it was scattered in the chapter after the essential sections.

Summary

Below, a list of core concepts and important details related to spring Data Access has been compiled for you:

- Spring supports data access with a layered architecture; higher layers have no knowledge about data management.
- Spring provides Data Access smart unchecked exceptions that get propagated to higher layers if untreated.
- Spring provides consistent transaction management: declarative and programmatic.
- Spring supports most popular persistence and ORM providers, which provide great support for caching, object management, automatic change detection.
- Spring can be used with Hibernate directly, without JPA API.
- Spring can be used with JPA API, but must be backed up by a persistence and ORM providers.
- Spring Data JPA helps the developer to avoid boilerplate code when creating repository components.
- Spring Data family project also provides support for NoSQL databases.
- Transaction management can be done by Spring-specialized infrastructure beans or by a transaction manager provided by an Application Server.
- Databases can be defined within the Spring configuration or can also be provided provided by an Application Server.

Quiz

Question 1: What data access technology is supported by the Spring framework?(choose all that apply)

- JDBC
- Hibernate
- JPA
- NoSQL

Question 2: Does Spring support transactional execution when using `JdbcTemplate` ?

- A. yes, if the methods encapsulating `JdbcTemplate` are annotated with `@Transactional`
- B. no

Question 3: Analyze the following code snippet:

```
public Set<User> findAll() {
    String sql = "select id, username, email, password from p_user";
    return new HashSet<>(jdbcTemplate.query(sql, rowMapper));
}
```

What can be said about the `rowMapper` object ? (choose all that apply)

- A. must implement `RowMapper<T>` interface
- B. is stateful
- C. provides a method to transform a `ResultSet` content into `User` objects

Question 4: Does `JdbcTemplate` support execution of SQL native queries?

- A. yes
- B. no
- C. if configured so

Question 5: What is not provided by `JdbcTemplate` ?

- A. JDBC exception wrapping into `DataAccessException`
- B. data source access
- C. JDBC query execution with named parameters
- D. method to open/close a connection

Question 6: What needs to be done to implement transactional behavior in a Spring application? (choose all that apply)

- A. enable declarative transaction support by annotating a configuration class with `@EnableTransactionManagement`
- B. declare a transaction manager bean
- C. annotate service methods with `@Transactional`
- D. activate transactional profile
- E. when using XML, declare `<tx:annotation-driven/>`

Question 7: Which of the following are valid transaction propagation values?(choose all that apply)

- A. MANDATORY
- B. REQUIRED
- C. PREFERRED
- D. NOT_ALLOWED

Question 8: When should a transaction be declared as `readOnly` ?

- A. when it does not include any writing statements execution
- B. when a large set of data is read
- C. when no changes should be allowed to the databases

Question 9: Analyze the following code snippet:

```
@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    @Override
    public User findById(Long id) {
        return userRepo.findById(id);
    }
}
```

What can be said about the `findById()` method?

- A. the method is executed in a new transaction
- B. when a method is executed, an exception will be thrown because the transaction is not declared as `readOnly`
- C. the method is executed in an existing transaction

Question 10: Spring provides programatic transaction management. What are the Spring type of beans that are used to implement a repository class using programatic transaction management? (choose all that apply)

- A. `TransactionTemplate`
- B. `TransactionDefinition`
- C. `TransactionService`
- D. `TransactionCallback`
- E. `PlatformTransactionManager`
- F. `TransactionStatus`

Question 11: What is the default rollback policy in transaction management?

- A. Rollback for any Exception
- B. Rollback for `RuntimeException`
- C. Rollback for checked exceptions
- D. Always commit

Question 12: What is used “under the hood” to implement the transactional behavior in Spring?

- A. JDBC
- B. JPA
- C. AOP

Question 13: What happens if a method annotated with `@Transactional` calls another method annotated with `@Transactional` ?

- A. a transaction is created for each method
- B. a single transaction is created and the methods are executed as a single unit of work
- C. depends on the configuration for each `@Transactional`

Question 14: Which of the following are Hibernate-specific Spring infrastructure beans? (choose all that apply)

- A. `SessionFactory`
- B. `Session`
- C. `HibernateTransactionManager`
- D. `HikariDataSource`
- E. `LocalSessionFactoryBuilder`

Question 15: What can be said about the `@Entity` annotation? (choose all that apply)

- A. is part of the JPA official API
- B. can be used without the JPA API
- C. marks the class as an entity; instances of this class will be mapped to table records
- D. it can be placed only at class level

Question 16: What is needed to work with JPA in Spring? (choose all that apply)

- A. declare an `EntityManagerFactory` bean
- B. declare a `PersistenceProvider` bean
- C. declare a `JpaTransactionManager` bean
- D. all of the above

Question 17: What can be done to declare an instant repository with Spring Data JPA? (choose all that apply)

- A. extend interface `Repository`
- B. no need to extend an interface; just annotate the interface with `@RepositoryDefinition`
- C. for multiple methods out of the box extend `JpaRepository`
- D. implement interface `Repository`

Question 18: What configuration annotation enables creation of repository instances with SpringDataJPA?

- A. `@EnableJpaRepositories`
- B. `@EnableInstantRepositories`
- C. `@EnableRepositoryGeneration`
- D. `@EnableTransactionManagement`

Question 19: Analyze the following code snippet:

```
public interface UserRepo extends JpaRepository<User, Long> {  
    @Query("select u from User u where u.username like %?1%")  
    List<User> findAllByUsername(String username);  
}
```

What can be said about the `findAllByUsername()` method?

- A. it is not a valid Spring Data JPA repository method
- B. the argument for `@Query` is a JPQL query
- C. the class is not a repository, because it is not annotated with `@Repository`
- D. this code will not compile

CHAPTER 6

Spring Web

In previous chapters, multilayered style projects were depicted. On top of the service layer was always the presentation layer, or the Web layer. This layer is the top layer of an application, and its main function is to translate user actions into commands that lower-level layers can understand and transform results from them into user-understandable data. Web applications can be accessed using clients such as browsers or specific applications that can interpret correctly the interface provided by the application such as mobile applications. So far in this book, only Spring components specific to lower-level layers have been introduced. This chapter will scratch the surface on components specific to the presentation layer that make the user interface and implement security.

Spring provides support for development of the web layer as well through frameworks such as Spring Web MVC and Spring WebFlow. A typical Java Web application architecture is depicted in Figure 6-1.

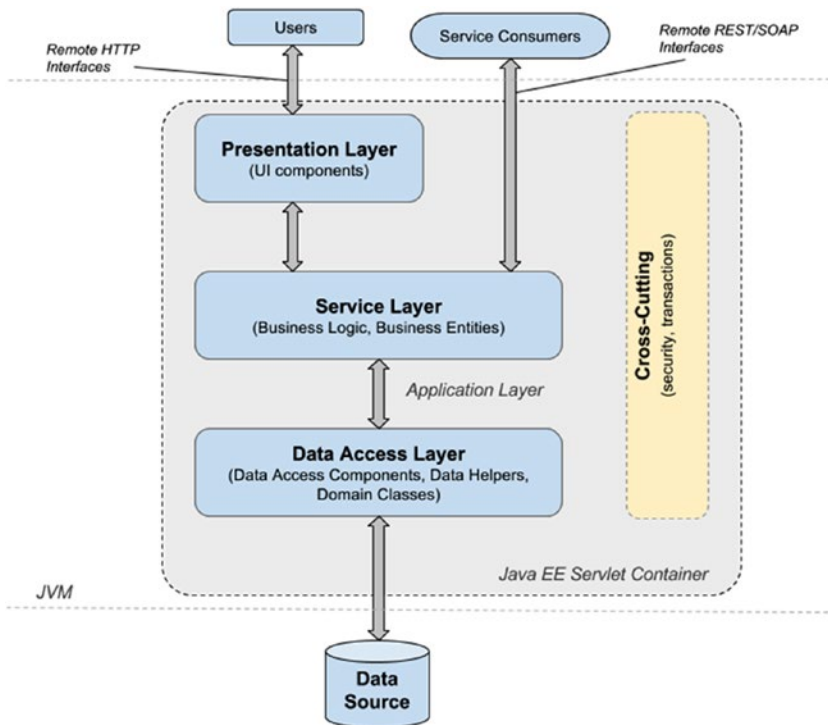


Figure 6-1. Typical Java Web application architecture

There can be more layers, but usually web applications have at least these three:

- **DAO**, where data mapping components (domain objects or entity classes) are defined.
- **Service** (also known as **Business**), where all classes needed to transform user data to be passed to the DAO layer are located. The business entities are POJOs that help with this conversion. All components specific to this layer implement how data can be created, displayed, stored, and changed.
- **Presentation**, where components that implement and display the user interface and manage user interaction reside.

Spring Web MVC is a popular request-driven framework based on the Model-View-Controller software architectural pattern, which was designed to decouple components that by working together make a fully functional user interface. The typical Model-View-Controller behavior is displayed in Figure 6-2.

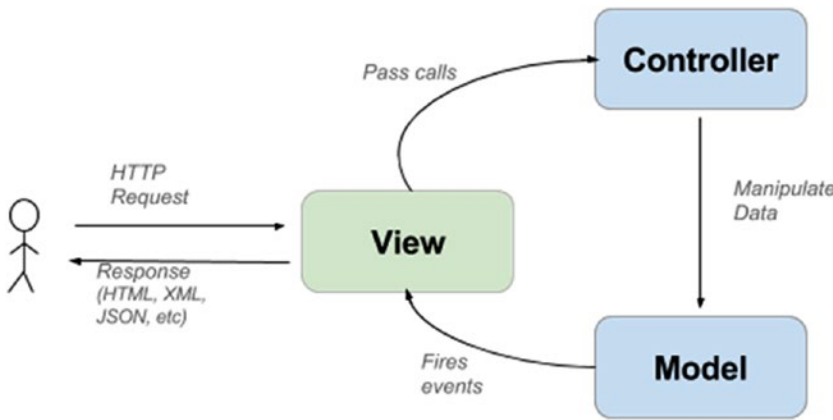


Figure 6-2. Typical MVC behavior

The main idea of MVC is that there are three decoupled components. Each of them can be easily swapped with a different implementation, and together they provide the desired functionality. The *view* represents the interface with which the user interacts. It displays data and passes requests to the *controller*. The controller calls the business components and sends the data to the *model*, which notifies the view that an actualization is needed. The *model* content is displayed by the *view*.

The central piece of Spring Web MVC is the `DispatcherServlet` class, which is the entry point for every Spring Web application. It dispatches requests to handlers, with configurable handler mappings, view resolution, locale, timezone, and support for uploading files. The `DispatcherServlet` converts HTTP requests into commands for controller components and manages rendered data as well. It basically acts as a Front Controller for the whole application. The basic idea of the Front Controller software design pattern, which implies that there is a centralized point for handling requests, is depicted in Figure 6-3.

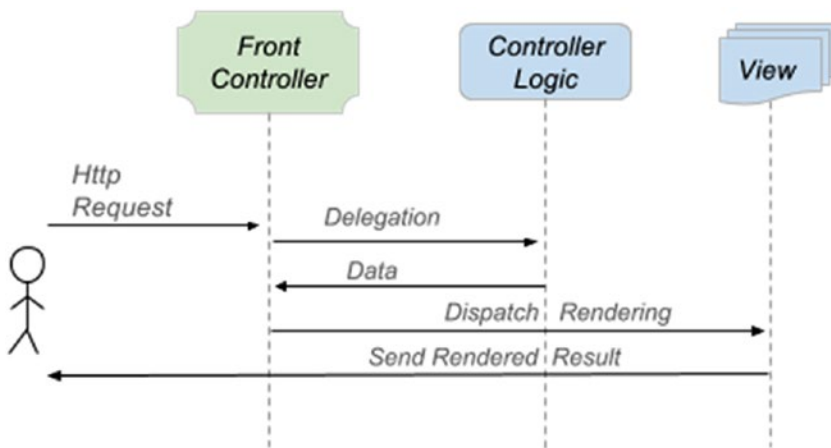


Figure 6-3. Front Controller software design pattern idea

The Spring Web MVC provides already configured beans for the implementation of this behavior, and these beans are contained in two main modules:

- spring-web.jar
- spring-webmvc.jar

A standard servlet listener is used to bootstrap and shut down the Spring application context. The application context will be created and injected into the `DispatcherServlet` before any request is made, and when the application is stopped, the Spring context will be closed gracefully as well. The Spring servlet listener class is `org.springframework.web.context.ContextLoaderListener`

Spring Web MVC is Spring's web framework, and thus Spring's typical configuration style is used, and controller components are beans. Starting with Spring 2.5, annotation configurations were introduced, so the controller components can be declared with a specialized stereotype annotation: `@Controller`. A Spring application can be deployed on an application server, can run on an embedded server like Jetty, or can be run with Spring Boot. All three flavors will be covered in this chapter.

Spring can be integrated with other frameworks:

- **Struts 2:** an open source framework for building Servlet/JSP from Apache-based web applications based on the Model-View-Controller (MVC) design paradigm. A previous version of this framework, Struts 1, was discontinued in 2016, but Struts 2 is alive and kicking, and on July 7, 2016, version 2.5.2 GA was released, as announced on their official site. More information can be found on the official site <https://struts.apache.org/>.
- **Wicket:** another open source Java web framework from Apache that was born in 2004 and is still alive and kicking; version 1.5.16 was released on August 5, 2016. This framework was designed with simplicity and separation of concerns as main purposes. More information can be found on the official site <http://wicket.apache.org/>.

- **Tapestry 5:** a component-oriented framework for creating highly scalable web applications in Java, also supported by Apache. More information can be found on the official site <http://tapestry.apache.org/>.
- **Spring WebFlow:** a Spring framework designed for implementation of stateful flows and destined for applications that require navigation through sequential steps in order to execute a business task. More information can be found on the official site <http://projects.spring.io/spring-webflow/>.

Using Spring Web MVC writing and running web applications is as easy as writing a simple standalone application as done in the previous chapters. Spring provides infrastructure beans for everything that is needed to boot up a web application and even a framework called Spring Boot that makes it easy to create standalone production-grade Spring-based applications by providing infrastructure beans already set up with a default general configuration. All this will be covered in this chapter, but keep in mind that the surface is only scratched on this topic, since Spring Web MVC is part of the set of topics for the Pivotal Certified Spring Web Application Developer exam and for this, Apress has published a different book.¹

Spring Web App Configuration

When a user accesses a URL, an HTTP request is generated to a web application hosted on an application server. Based on that request, inside the application context an action must be performed. The result of the action performed is returned, and it needs to be represented using a view. In a Spring Web application, all HTTP requests first reach the `DispatcherServlet`. Based on the configuration, a handler is called, which is a method of a class called *controller*. The result is used to populate a *model* that is returned to the `DispatcherServlet`, which based on another set of configurations decides which *view* is to be used to display the result.

This process is depicted in Figure 6-4.

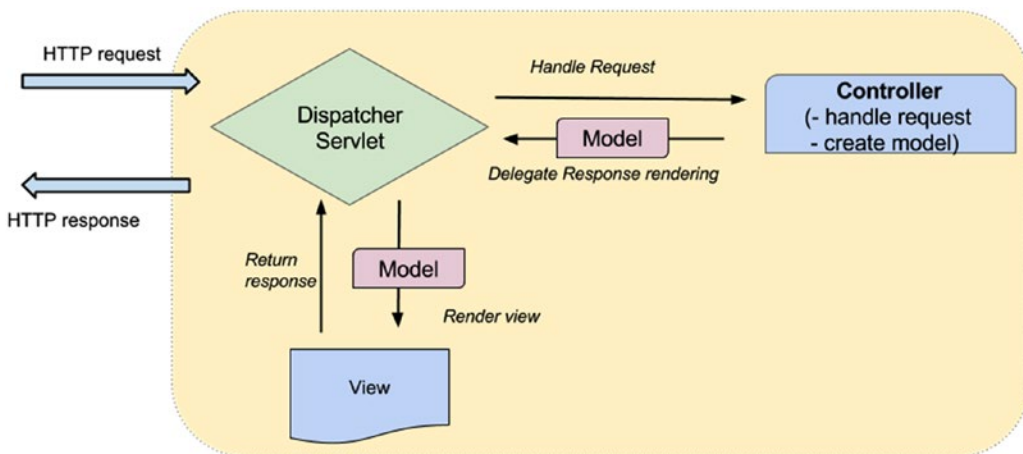


Figure 6-4. Request processing steps in a Spring Web application

¹Pivotal Certified Spring Web Application Developer Exam; <http://www.apress.com/9781484208090>.

The `DispatcherServlet` is the entry point of the application, the heart of Spring Web MVC that coordinates all request handling operations. It is equivalent to `ActionServlet` from Struts and `FacesServlet`² from JEE. It delegates responsibilities to Web infrastructure beans and invokes user web components. It is customizable and extensible. The `DispatcherServlet` must be defined in `web.xml` when the application is configured using the old-style XML configuration. When using a configuration without `web.xml`, a configuration class that extends `AbstractDispatcherServletInitializer` or `AbstractAnnotationConfigDispatcherServletInitializer` must be declared. These two are Spring specialized classes from the `org.springframework.web.servlet.support` package that implement `org.springframework.web.WebApplicationInitializer`. Objects of types implementing this interface are detected automatically by `SpringServletContainerInitializer`, which is bootstrapped automatically by every Servlet 3.0+ environment. More about specialized classes to be used to bootstrap Spring and configuring the `DispatcherServlet` will be covered in this chapter, along with information regarding the context in which each is used will be provided.

The `DispatcherServlet` uses Spring for its configuration, so programming using interfaces is a must to allow swapping different implementations easily.

The `DispatcherServlet` creates a separate “servlet” application context containing all specific web beans (controller, views, view resolvers). This context is also called the *web context* or the *DispatcherServletContext*.

The application context is also called *RootApplicationContext*. It contains all non-web beans and is instantiated using a bean of type `org.springframework.web.context.ContextLoaderListener`. The relationship between the two contexts is a parent-child relationship, with the application context being the parent. Thus, beans in the web context can access the beans in the parent context, but not conversely.³ This separation is useful when there is more than one servlet defined for an application, for example, one that handles web requests and one that handles web services calls, because they can both inherit beans from the root application context.

Of course, the beans in the root context can be included in the web context, but most nontrivial Spring applications require separate contexts for the web and the non-web part of the application, since usually the web part of an application is only a small part of the overall application, production applications having many concerns that lie outside the web context.

²`FacesServlet` is a servlet that manages the request processing lifecycle for web applications that utilize JavaServer Faces to construct the user interface. `org.springframework.web.context`

³This resembles class inheritance: the superclass inherits the parent class and can access its members.

The `DispatcherServlet` uses some infrastructure beans called handling mappings to identify a controller method to call and handler adapters to call it. An abstract diagram describing the call sequence is depicted in Figure 6-5.

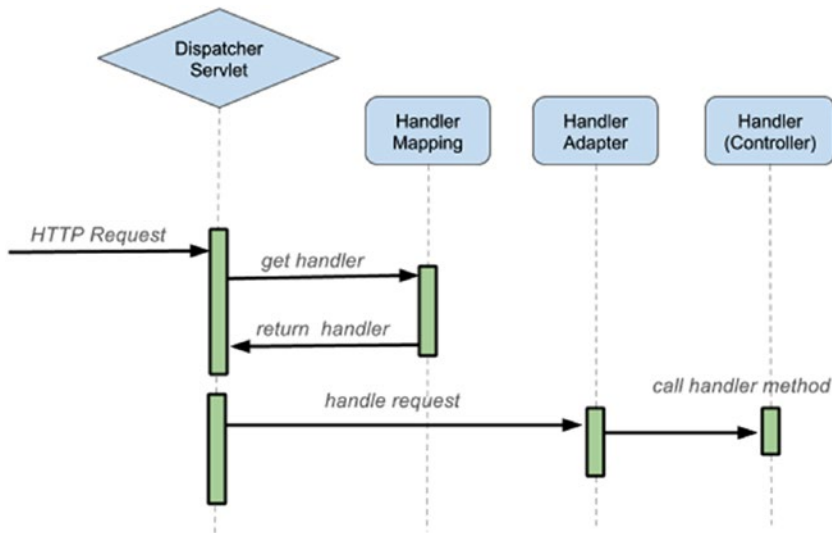


Figure 6-5. Abstract UML diagram for handling a HTTP request

In the Spring MVC versions previous to 2.5, all configuration was done using XML, and the controller classes had to extend the `org.springframework.web.servlet.mvc.AbstractController` Spring class. In Spring 2.5, the new `@MVC` model and support for annotations were introduced. Configuration still relied on XML for MVC infrastructure-specific beans, but controller classes no longer need to extend the Spring specialized class, since only annotating them with the stereotype annotation `@Controller` was enough for Spring to know what they are used for. Starting with Spring 3.0, Java Configuration was introduced, which led to totally removing XML configuration. Almost. Because the `web.xml` typical configuration file for a Java web application was still needed. Starting with Servlet 3.0, `web.xml` is no longer needed as well.

This book focuses more on `@MVC` and Java Configuration for web applications, since XML is probably going to be dropped in the future. The surface will be scratched a little on the XML configuration topic, just to give you an idea.

Quickstart

These are the usual steps taken to develop a Spring Web MVC application:

- develop backend application beans (service and DAO) and configuration
- develop MVC functional beans, also known as *controllers* or *handlers*, that contain methods also known as *handler methods*, which will be used to handle HTTP requests
- develop views used to display the results returned by handlers (common *.jsp files)
- declare and configure MVC infrastructure beans
- configure the web application

Controllers

Controllers are classes that define methods used to handle HTTP requests. They are annotated with the stereotype Spring annotation `@Controller`. Each handler method is annotated with `@RequestMapping`, which provides information regarding when this method should be called. In the following code snippet a controller class is defined, containing one handler method, which will be called when the user sends a request with the following URL:

```
http://localhost:8080/mvc-basic/users/list
```

The URL contains the following elements:

- `http` is HTML protocol definition
- `localhost` is the name/domain/IP of the machine where the application server is installed
- `8080` is the port where the application server can be accessed
- `mvc-basic` is the web application context
- `/users/list` is the request mapping value

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
...

@Controller
public class UserController {
    @Autowired
    private UserService userService;

    /**
     * Handles requests to list all users.
     */
    @RequestMapping(value = "/users/list", method = RequestMethod.GET)
    public String list(Model model) {
        logger.info("Populating model with list...");
        model.addAttribute("users", userService.findAll());
        //string used to identify the view
        return "users/list";
    }
}
```

Spring provides a way to specify the type of HTTP request on which the handler method should be called, via the method attribute of the `@RequestMapping` annotation.⁴ The mapping rules are usually URL based and they can also contain wildcards and regular expressions. The `@RequestMapping` can also be used at

⁴HTTP GET request is used when the user is requesting data from the application. HTTP POST and PUT are requests that send data to the application.

class level, to simplify mappings at method level when the mappings have common elements. For example, in case you have many `@RequestMapping` methods with the common pattern `/users`, the `UserController` class can be annotated with `@RequestMapping("/users")`], situation depicted in the code snippet below.

```
...
@Controller
@RequestMapping("/users")
public class UserController {
    //matches http://localhost:8080/mvc-basic/users/list
    @RequestMapping(value = "/list", method = RequestMethod.GET)
    public String list(Model model) { ...}

    //matches http://localhost:8080/mvc-basic/users/105
    @RequestMapping(value =("/{id:\\d*})", method = RequestMethod.GET)
    public String show(Long id, Model model) {
        ...
    }
}
```

The handler methods have flexible signatures with parameters that provide information about the request context. Spring will transparently provide arguments when the handler methods are called. Depending on the URL type, different annotations can be used to tell Spring what values to use.

- URL (Uniform Resource Locator)
- `http://localhost:8080/mvc-basic/showUser?userId=105`

Is handled by a method that has a parameter annotated with `@RequestParam` because the request is parametrized. If the url parameter has a name different from the method parameter, the annotation is used to inform Spring of that.

■ **CC** The convention in this case is that if the name of the parameter is not specified in the `@RequestParam` annotation, then the request parameter is named the same as the method parameter.

```
import org.springframework.web.bind.annotation.RequestParam;
...
@Controller
@RequestMapping("/users")
public class UserController {

    @RequestMapping(value = "/showUser", method = RequestMethod.GET)
    public String show(@RequestParam("userId") Long id, Model model) {
        ...
    }
}
```

- URI (Uniform Resource Identifier) is a cleaner URL request parameters. Handling URIs is supported starting with Spring 3.0.
- `http://localhost:8080/mvc-basic/users/105`

The previous URI is handled by a method that has a parameter annotated with `@PathVariable` because the request URI contains a piece that is variable. That piece is called a *path variable* and is defined by the `@RequestMapping` annotation. If the path parameter has a name different from the method parameter, the annotation is used to inform Spring of that.

■ **CC** The convention in this case is that if the name of the path variable name is not specified in the `PathVariable` annotation, then the path variable is named the same as the method parameter.

```
import org.springframework.web.bind.annotation.PathVariable;
...
@Controller
@RequestMapping("/users")
public class UserController {

    @RequestMapping(value =("/{userId}", method = RequestMethod.GET)
    public String show(@PathVariable("userId") Long id, Model model) {
        ...
    }
}
```

A handler method typically returns a string value representing a logical view name, and the view is populated with values in the `Model` object.

■ **CC** By default, the logical view name is interpreted as a path to a JSP page. Controller methods can also return null or void, and in this case, the default view is selected based on the request URL. Controller methods can also return concrete views, but this is usually avoided because it couples the controller with the view implementation.

```
import org.springframework.web.servlet.ModelAndView;
...
@Controller
@RequestMapping("/users")
public class UserController {

    @RequestMapping(value =("/{userId}", method = RequestMethod.GET)
    public ModelAndView show(@PathVariable("userId") Long id) {
        User user = ...; // no relevant
        ModelAndView modelAndView = new ModelAndView("user");
        return modelAndView;
    }
}
```

The model contains the data that will be used to populate a view. Spring provides view resolvers in order to do this to avoid ties to a specific view technology. Out of the box Spring supports JSP, Velocity templates, and XSLT views. The interfaces needed to be used to make this possible are `ViewResolver` and `View`. The first provides a mapping between view names and actual views. The second takes care of preparing the request and forwards it to a view technology. The `DispatcherServlet` delegates to a `ViewResolver` to map logical view names to view implementations. Spring comes out of the box with a few *view resolver* implementations. All handler methods must resolve to a logical view name that corresponds to a file, either explicitly by returning a `String`, `View`, or `ModelAndView` instance or implicitly based on internal conventions. The core view resolver provided by Spring is the `InternalResourceViewResolver`; it is the default view resolver. Inside `spring-webmvc.jar` there is a file called `DispatcherServlet.properties`, and in it all default infrastructure beans are declared. If you look for the default view resolver, you will find the previously mentioned implementation:

```
org.springframework.web.servlet.ViewResolver=
    org.springframework.web.servlet.view.InternalResourceViewResolver
```

The default implementation can be overridden by registering a `ViewResolver` bean with the `DispatcherServlet`. The default view resolver implementation is also customizable, and can be configured to look for view template files in a different location. The previous code snippets returned string values as view names. In Figure 6-6, the configuration and the mechanism of resolving views is depicted.

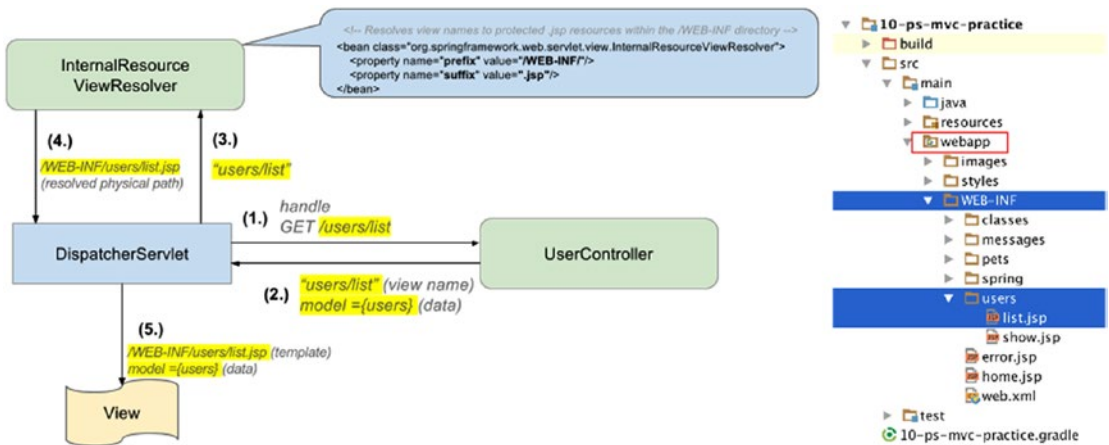


Figure 6-6. Custom Internal Resource View Resolver Example

The previous image depicts how `DispatcherServlet`, `UserController` and an `InternalResourceViewResolver` bean configured by the developer work together to produce a view. Each action has a number, and the steps are explained below:

1. The `DispatcherServlet` identifies the handler method for the `users/list` GET request and requests the `UserController` method to handle the request.
2. The handler method is executed and a logical view name `users/list` and a model object containing the collection named `users` are returned to the `DispatcherServlet`. Note that `model={users}` is a notation that means that the `users` object is contained by the `model` object.
3. The `DispatcherServlet` asks the `InternalResourceViewResolver` to which view template the `users/list` logical view name corresponds.

4. The `InternalResourceViewResolver`, which is configured as follows:

```
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/" />
    <property name="suffix" value=".jsp" />
</bean>
```

takes the logical view name, applies the prefix (`/WEB-INF/`) and the suffix (`.jsp`) and sends the result (`/WEB-INF/users/list.jsp`), which is the physical path of the view template to the `DispatcherServlet`.

5. The `DispatcherServlet` will further use the view template provided by the `InternalResourceViewResolver` and the model object received from the `UserController` to create a view.

To summarize everything so far, and to be able to dive completely into the Spring MVC configuration, to configure a Spring web application you must do the following:

- define the `DispatcherServlet` as the main servlet of the application that will handle all requests in the `web.xml` file and link it to the Spring configuration or configure this servlet programmatically by using a class implementing `WebApplicationInitializer` (possible only in a Servlet 3.0+ environment)
- Define the application configuration (usually in a Spring configuration file named `mvc-config.xml` or a Java Configuration class), which should contain the following:
 - Define the MVC context used (handler adapter, handler mapping, and other infrastructure beans)
 - Define a view resolver (or more)

Sounds pretty straightforward, right?

XML

To define the `DispatcherServlet` as the main servlet of the application, the `web.xml` file located under the `webapp\WEB-INF` directory must be modified to include this Spring specialized servlet and the parameters for it.

The following configuration snippet is suitable for this scenario:

```
<!-- web.xml -->
<web-app ...>
  <!-- The front controller, the entry point for all requests -->
  <servlet>
    <servlet-name>pet-dispatcher</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        /WEB-INF/spring/mvc-config.xml
      </param-value>
```



```

        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Map all requests to the DispatcherServlet for handling -->
    <servlet-mapping>
        <servlet-name>pet-dispatcher</servlet-name>
        <url-pattern></url-pattern>
    </servlet-mapping>
    ...
</web-app>

```

The `DispatcherServlet` was named `pet-dispatcher`, and the Spring Web MVC configuration file is referred into the `web.xml` file as value for the `contextConfigLocation` parameter. In the above case, the `mvc-config.xml` contains the Spring configuration for the front end (controllers and MVC infrastructure beans), the file is loaded by the `DispatcherServlet`, and a web execution context for the application is created.

The `servlet-mapping` elements map all incoming requests to the `DispatcherServlet`, and the `url-pattern` element can even contain a context to further filter the requests, in case other servlets are used (example: security handler servlet).

The declaration of a view resolver bean should be located in the `mvc-config.xml` file, since it is a Spring infrastructure bean. To match the internal structure of the web application presented earlier, the bean is customized like this:

```

<beans ...>
    <bean class="
        org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/" />
        <property name="suffix" value=".jsp" />
    </bean>
    ...
</beans>

```

The `InternalResourceViewResolver` configured like this receives the logical view name from the `DispatcherServlet`, applies the prefix (`/WEB-INF/`) and the suffix (`.jsp`), and sends the result, which will be a something similar to (`/WEB-INF/[template_path/template_name].jsp`), which is the physical path of the view template, to the `DispatcherServlet`, which will use the view template and the model object received from the controller to create a view that will be displayed to the end user.

The `InternalResourceViewResolver` has more properties, and an important one is the `requestContextAttribute`, which is the context holder for request-specific states, such as current web application context, current locale, current theme, and potential binding errors. It provides easy access to localized messages and `Errors` instances.

```

<beans ...>
    <bean class="
        org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/" />
        <property name="suffix" value=".jsp" />
        <property name="requestContextAttribute" value="requestContext" />
    </bean>
    ...
</beans>

```

Localization and personalization are advanced subjects that are not needed for the CORE certification exam, so they won't be covered in detail here. The `requestContextAttribute` was mentioned, because most developers new to Spring often forget about it and get really confused when changing locale, and themes doesn't work even if they declare and configure all the necessary beans. Also, `requestContext` is used in the `*.jsp` files to decide locale and themes. The following JSP snippet depicts how the attribute is used to implement some of the URLs that will set the locale of the application when clicked. Additional taglibs are used: JSTL taglib for conditions, and the Spring taglib for internationalization. The complete code sample can be found in the code attached to the book.

```
<!-- JSTL taglib import -->
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!-- Spring taglib import -->
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
<c:choose>
  <!-- when locale is English -->
  <c:when test="{requestContext.locale.language eq 'en'}">
    <c:url var="localeUrl" value="/">
      <c:param name="locale" value="de"/>
    </c:url>
    <a href="{localeUrl}"><spring:message code="locale.de"/></a>
    <!-- text: German -->
    <!-- URL: http://localhost:8080/mvc-basic/?locale=de-->
  </c:when>
  <c:otherwise>
    <!-- when locale is German -->
    <c:url var="localeUrl" value="/">
      <c:param name="locale" value="en"/>
    </c:url>
    <a href="{localeUrl}"><spring:message code="locale.en"/></a>
    <!-- text: English -->
    <!-- URL: http://localhost:8080/mvc-basic/?locale=en-->
  </c:otherwise>
</c:choose>
```

The next step: defining a controller. The class was already introduced previously, so the following code snippet only depicts the core element of a controller class. The method covered here handles a request to display a user. The `id` value is part of the URI and is automatically populated by Spring. The `userService` is a bean used to extract the data from the database, and the `users/show` is the logical name of the view, which will be resolved to the `/WEB-INF/users/show.jsp` view.

```
//custom exception, implementation not relevant for context
import com.ps.problem.NotFoundException;
...
@Controller
@RequestMapping("/users")
public class UserController {
```

```

private UserService userService;

public void setUserService(UserService userService) {
    this.userService = userService;
}

@RequestMapping(value =("/{id:\\d*}", method = RequestMethod.GET)
public String show(@PathVariable Long id, Model model)
    throws NotFoundException {
    User user = userService.findById(id);
    if (user == null) {
        throw new NotFoundException(User.class, id);
    }
    model.addAttribute("user", user);
    return "users/show";
}
}

```

The controller bean also has to be declared in the `mvc-config.xml` file.

```

<beans ...>
  <!-- by component scanning -->
  <context:component-scan base-package="com.ps.web"/>

  <!-- OR by XML declaration -->
  <bean id="UserController" class="com.ps.web.UserController>
    <property name="userService" ref="userService" />
  </bean>

  <bean id="userService" class=" com.ps.service.UserServiceImpl">
    ... <!-- bean definition -->
  </bean>
</beans>

```

The model is populated with the user object. Now that we have a view resolver and a handler, the `show.jsp` view needs to be defined. The view will reference the model object by its name `user` and display the values of its properties. The following JSP snippet depicts how the user object is used in the code of a JSP page.

```

<!-- Spring taglib import -->
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
<html>
<body>
<!-- other html elements not relevant here -->
<div class="userDetails">
  <table>
    <tr>
      <th><spring:message code="label.User.firstname"/></th>
      <td>${user.firstName}</td>
    </tr>

```

```

<tr>
  <th><spring:message code="label.User.lastname"/></th>
  <td>${user.lastName}</td>
</tr>
<tr>
  <th><spring:message code="label.User.username"/></th>
  <td>${user.username}</td>
</tr>
<tr>
  <th><spring:message code="label.User.type"/></th>
  <td>${user.userType}</td>
</tr>
<tr>
  <th><spring:message code="label.User.since"/></th>
  <td><fmt:formatDate value="${user.createdAt}"/></td>
</tr>
<tr>
  <th><spring:message code="label.User.rating"/></th>
  <td>${user.rating}</td>
</tr>
</table>
</div>
</body>
</html>

```

The JSP pages are usually developed using taglibs that extend the collection of JSP elements that can be used in a page and provide a simpler syntax. The Spring taglib declared in the JSP code snippet is used to extract the text for the table headers according to the locale of the application. For example, when creating the view to be displayed to the end user, the `<spring:message code="label.User.firstname"/>` will be transformed in *Firstname* if the locale is English and into *Vorname* if the locale is German.

@MVC

In Spring 3.0, the @MVC introduced a new configuration model. The main component of an MVC XML configuration is the `<mvc:annotation-driven/>` element, which registers all necessary default infrastructure beans for a web application to work: handler mapping, validation, conversion beans, and many others.

Another component that is important is the `<mvc:default-servlet-handler/>`. Usually in Spring Web applications, the default servlet mapping `"/` is mapped to the `DispatcherServlet`. This means that static resources will have to be served by it too, which might introduce a certain lag in providing a response, since the `DispatcherServlet` has to find the resources to which the request url is mapped. Think about it like this: a request comes to the `DispatcherServlet`. The default behavior of `DispatcherServlet` is to start looking for a controller to handle this request. And when it does not find one, it assumes that this is a request for a static resource and then asks the static resources handler whether it can handle the request. By declaring `<mvc:default-servlet-handler/>`, a handler for static resources is defined with the lowest precedence, and this handler will be asked first whether it can handle the request.

Technically, the `<mvc:default-servlet-handler/>` configures a `DefaultServletHttpRequestHandler` with a URL mapping of `"/*` and the lowest priority relative to all others URL mappings, and its sole responsibility is to serve static resources.

To use the `@MVC` elements, the `mvc` namespace must be specified:

```
<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:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.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">
<!-- Defines basic MVC defaults (handler mapping, date formatting, etc) -->
  <mvc:annotation-driven/>

  <!-- Configures a handler for serving static resources by forwarding to the
Servlet container's default Servlet.-->
  <mvc:default-servlet-handler/>
</beans>
```

There are many out of the box features that can be used with the `@MVC` model, but they are out of the scope of this book. If you are interested in knowing more, there is a good Apress book covering Spring Web: *The Pivotal Certified Spring Web Application Developer Exam Study Guide*.

Java Configuration for Spring MVC

In Spring 3.1, the `@EnableWebMvc` annotation was introduced. This annotation, when used on a configuration class, is the equivalent of `<mvc:annotation-driven/>`. Of course, there are a few extra details to take care of to make sure that the configuration works as expected, and they will all be covered in this section.

The first step is transforming the `mvc-config.xml` into a configuration class. Each bean definition, each scanning element must have a correspondence in the web configuration class. The configuration class has to be annotated with the `@Configuration` and `@EnableWebMvc` annotation and has either to implement `WebMvcConfigurer` or extend an implementation of this interface, for example `WebMvcConfigurerAdapter`, which gives the developer the option to override only the methods he or she is interested in. Annotating a configuration class with `@EnableWebMvc` has the result of importing the Spring MVC configuration implemented in the `WebMvcConfigurationSupport` class and is equivalent to `<mvc:annotation-driven/>`. This class registers many infrastructure Spring components necessary for a web application, and only a part of them will be covered in this section, since the this topic is not fully needed for the Spring Core certification exam.

To tell the `DispatcherServlet` that the configuration will be provided by a configuration class instead of a file, the following changes have to be made in `web.xml`:

- An initialization parameter named `contextClass` having as value the full class name of the Spring class used to create an annotation-based context is defined.
- The initialization parameter named `contextConfigLocation` will have as value the full class name of the configuration class written by the developer.

```

<web-app ...>
<!-- The front controller, the entry point for all requests -->
  <servlet>
    <servlet-name>pet-dispatcher</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextClass</param-name>
      <param-value>
        org.springframework.web.context.
          support.AnnotationConfigWebApplicationContext
      </param-value>
    </init-param>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
        com.ps.config.WebConfig
      </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>

```

The configuration class for what was configured with XML in the previous section looks like this:

```

package com.ps.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.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import org.springframework.web.servlet.view.InternalResourceViewResolver;

...
@Configuration
// <=><mvc:annotation-driven/>
@EnableWebMvc
// <=> <context:component-scan base-package="com.ps.web"/>
@ComponentScan(basePackages = {"com.ps.web"})
public class WebConfig extends WebMvcConfigurerAdapter {

    // <=> <mvc:default-servlet-handler/>
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurator) {
        configurator.enable();
    }
}

```

```

@Bean
InternalResourceViewResolver getViewResolver(){
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/");
    resolver.setSuffix(".jsp" );
    resolver.setRequestContextAttribute("requestContext");
    return resolver;
}
... // other beans and method implementations that are not in scope
}

```

The `WebMvcConfigurerAdapter` class that was extended by the configuration class here contains more methods for which the developer must provide implementation, and they are out of scope, but the `10-ps-mvc-practice` project that is the corresponding one for this section contains a full working and documented configuration that you can take a look at.

Getting Rid of web.xml

Starting with Servlet 3.0+, the `web.xml` file is no longer necessary to configure a web application. It can be replaced with a class implementing the `WebApplicationInitializer` interface (or a class extending any of the Spring classes that extend this interface). This class will be detected automatically by `SpringServletContainerInitializer` (an internal Spring-supported class that is not meant to be used directly or extended), which itself is bootstrapped automatically by every Servlet 3.0+ container.

The `SpringServletContainerInitializer`⁵ extends `javax.servlet.ServletContainerInitializer` and provides a Spring-specific implementation for the `onStartup` method. This class is loaded and instantiated, and the `onStartup` is invoked by every Servlet 3.0+ compliant container during container startup assuming that the `spring-web` module JAR is present on the classpath. Thus, the `web.xml` can be deleted, and, considering we are still using XML configuration, can be replaced with this class:

```

package com.ps.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class WebInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        ServletRegistration.Dynamic registration =
            servletContext.addServlet("pet-dispatcher", new DispatcherServlet());
        registration.setLoadOnStartup(1);
    }
}

```

⁵The code of this class can be found here: <https://github.com/spring-projects/spring-framework/blob/master/spring-web/src/main/java/org/springframework/web/SpringServletContainerInitializer.java>.

```

        registration.addMapping("/");
        registration.setInitParameter("contextConfigLocation",
            "/WEB-INF/spring/mvc-config.xml");
    }
}

```

There are more ways than one to write the implementation above. For example, the application context can be constructed first and then injected into the `DispatcherServlet`, but that is again out of the scope of this book.

```

package com.ps.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class WebInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/mvc-config.xml");
        ServletRegistration.Dynamic registration =
            servletContext.addServlet("dispatcher", new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}

```

If a configuration class annotated with `@EnableWebMvc` is used, then the implementation of the `onStartup(...)` method changes to this:

```

package com.ps.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class WebInitializer implements WebApplicationInitializer {

```



```

@Override
public void onStartup(ServletContext servletContext) throws ServletException {
    ServletRegistration.Dynamic registration =
        servletContext.addServlet("dispatcher", new DispatcherServlet());
    registration.setLoadOnStartup(1);
    registration.addMapping("/");
    registration.setInitParameter("contextConfigLocation",
        "com.ps.config.WebConfig");
    registration.setInitParameter("contextClass",
        "o.s.w.c.s.AnnotationConfigWebApplicationContext");
    }
}

```

There are many ways to write a configuration using Java Configuration classes as well, but only the most common way will be depicted here, namely extending the `AbstractAnnotationConfigDispatcherServletInitializer` class:

```

package com.ps.config;

import org.springframework.web.servlet.support.
    AbstractAnnotationConfigDispatcherServletInitializer;
...
public class WebInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?> getRootConfigClasses() {
        return new Class<?>{
            ServiceConfig.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};
    }
}

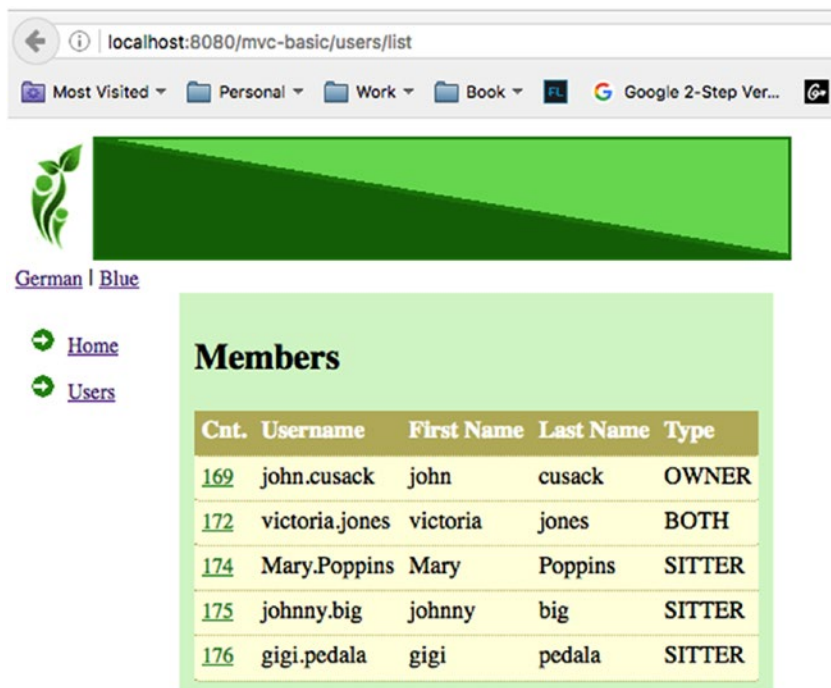
```

This class defines a set of methods that can be overridden by the developer to provide a complete configuration for a web application without needing any XML. It is most suitable and easy to use for multimodule applications in which the root context should be separated from web context. In the code depicted previously, the `getRootConfigClasses()` is called internally, and the configuration classes are used to create the root application context. The `getServletMappings()` method is used to set the context of the `DispatcherServlet`, and the `getServletFilters()` is used to set an additional filter for the application.

Using Spring to create web applications is easy and practical, but it is also a wide subject that cannot be covered in this book, since only the basic idea is a topic for the certification exam.

Running a Spring Web Application

The `10-ps-mvc-practice` is one of the projects that can be used to test your knowledge and basic understanding of Spring Web MVC. The project also contains internationalization and theme configuration beans, which are properly documented, so the `WebConfig` configuration class for the Spring Web application is still pretty straight-forward. The application is simple. It only lists the list of users in the system and allows you to see the details for each of them by clicking on a link. In Figure 6-7, the `users/list.jsp` page is depicted.



German | Blue

[Home](#)

[Users](#)

Members

Cnt.	Username	First Name	Last Name	Type
169	john.cusack	john	cusack	OWNER
172	victoria.jones	victoria	jones	BOTH
174	Mary.Poppins	Mary	Poppins	SITTER
175	johnny.big	johnny	big	SITTER
176	gigi.pedala	gigi	pedala	SITTER

Copyright © 2016 by Iuliana Cosmina and Apress. All rights reserved.

Figure 6-7. Practice Spring Web application

■ ! There are three TODO tasks for this section numbered from 45 to 47. If you want to test your understanding of Spring Web MVC configuration, you can try solving them.

Task TODO 45, located in the `WebConfig` class, requires you to complete the configuration of this class to make it usable in creating a web application context.

Task TODO 46, located in the `WebConfig` class, requires you to provide the configuration for the `InternalResourceViewResolver` bean.

Task TODO 47, located in `UserControllerTest`, requires you to complete the body of a test method that checks whether the `UserController show()` method works correctly: whether the view logical name returned is the correct one and if the model is populated with the correct data. Inspect the implementation of the `UserController` class to find the expected values that should be used in the assertion statements. In the same class, there is a fully implemented method that tests the `list()` method of the `UserController` class. Analyze it and use it for inspiration.

If you have trouble, you can take a peek at the proposed solution in the `10-ps-mvc-solution` project.

Running with Jetty

The project can be run on a Jetty embedded server, using the Gretty plugin, which is added in the Gradle configuration file for this module. The file is named as the module `10-ps-mvc-practice.gradle`. This was a development choice designed to help you identify easily which configuration file matches which module. The default name for a Gradle configuration file is `build.gradle`. In a big project such as `pet-sitter`, it would have been very difficult to identify the proper build file for a module while using `CTRL+SHIFT+N` in IntelliJ IDEA if all of them were named `build.gradle`. The following code snippet depicts the contents of the configuration file for the `10-ps-mvc-practice` module.

```
apply plugin: 'war'
apply from:
    'https://raw.githubusercontent.com/akhikh1/gretty/master/pluginScripts/gretty.plugin'

dependencies {
    compile project(':09-ps-data-jpa')
    compile misc.joda, spring.jdbc, spring.contextSupport, misc.h2, misc.commons,
        spring.webmvc, misc.hikari,...
    //etc
    testCompile tests.junit, tests.easymock, tests.jmock, tests.mockito,
        spring.test, tests.hamcrestCore, tests.hamcrestLib
}
gretty {
    port = 8080
    // application context
    contextPath = '/mvc-basic'
}
```

With this plugin set up, the Spring application can be run in two ways:

- from the terminal by running

```
gradle appRun
```
- If you look in the console, the Gretty log tells you where you can access the web application.

```
...
INFO Jetty 9.2.15.v20160210 started and listening on port 8080
INFO mvc-basic runs at:
INFO http://localhost:8080/mvc-basic
Run 'gradle appStop' to stop the server.
```
- from IntelliJ IDEA, go to the Gradle projects view, expand the pet-sitter project, look for 10-ps-mvc-practice, expand the node, and from under it go to Tasks -> gretty. Under it you will see the appStart task. Double click it, and a console will appear telling you when the application is started and for how long. Of course, initially the application won't be visible in the browser until you solve the TODO tasks. In Figure 6-8, you can see the Gradle task to run and the console available in IntelliJ IDEA.

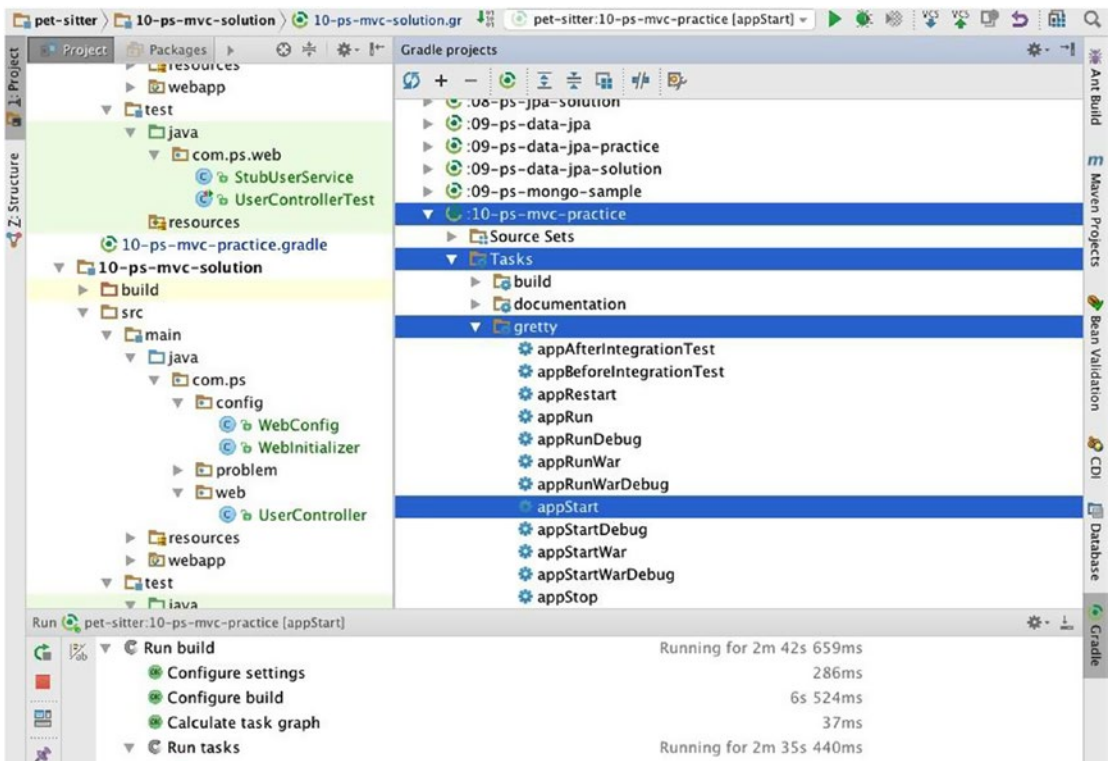


Figure 6-8. Spring Web application run with Gradle appStart task in IntelliJ IDEA

Running with Tomcat

A Spring Web application can be run on an application server as well. The most used application server is Apache Tomcat. It is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language, and Java WebSocket technologies. It is currently at version 9 and has been around since 1999. Installing Tomcat is easy on any operating system; just follow the instructions on the official site:

<http://tomcat.apache.org/>

When building a web application, the result is a *.war archive named the same as the project. In the previous `10-ps-mvc-practice.gradle` file, an extra configuration must be added to make sure that the war name is `mvc-basic`, so that when it is deployed on Tomcat, the context of the application will be the same.

```
war {
    archiveName = 'mvc-basic.war'
}
```

After running gradle build on the `10-ps-mvc-practice` module, under `10-ps-mvc-solution/build/libs` the `mvc-basic.war` should be present. Deploying this web application to Tomcat is easy: just copy the `mvc-basic.war` under `tomcat/webapps/` and start the server. The application will be deployed and will be available at `http://localhost:8080/mvc-basic/`.

The previous method of deployment is the manual way. When you use an intelligent Java IDE as IntelliJ IDEA, things can be made easier. All the developer has to do is create a Tomcat configuration in IntelliJ IDEA and add the project that generated the war needed to be deployed. Here are the steps:

- Open the Run/Debug Configuration menu and select Edit Configurations (Figure 6-9).

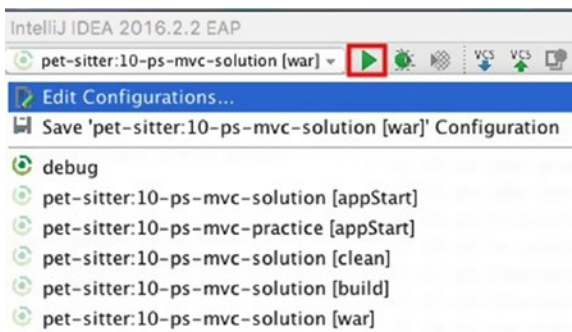


Figure 6-9. IntelliJ IDEA Run/Debug Configuration menu

- Click the + button and from the list select Tomcat Server -> Local , then click Ok (Figure 6-10).

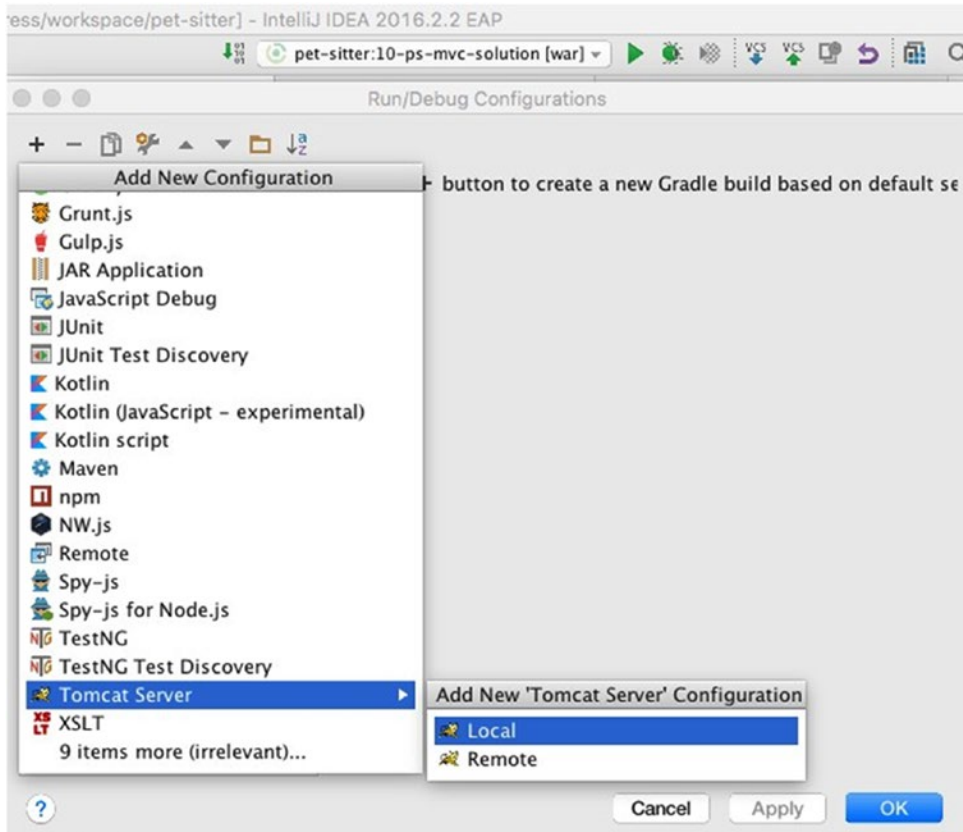


Figure 6-10. IntelliJ IDEA Run/Debug Configuration dialog window, before selecting configuration type

- In the following pop-up, click the Configure button (Figure 6-11).

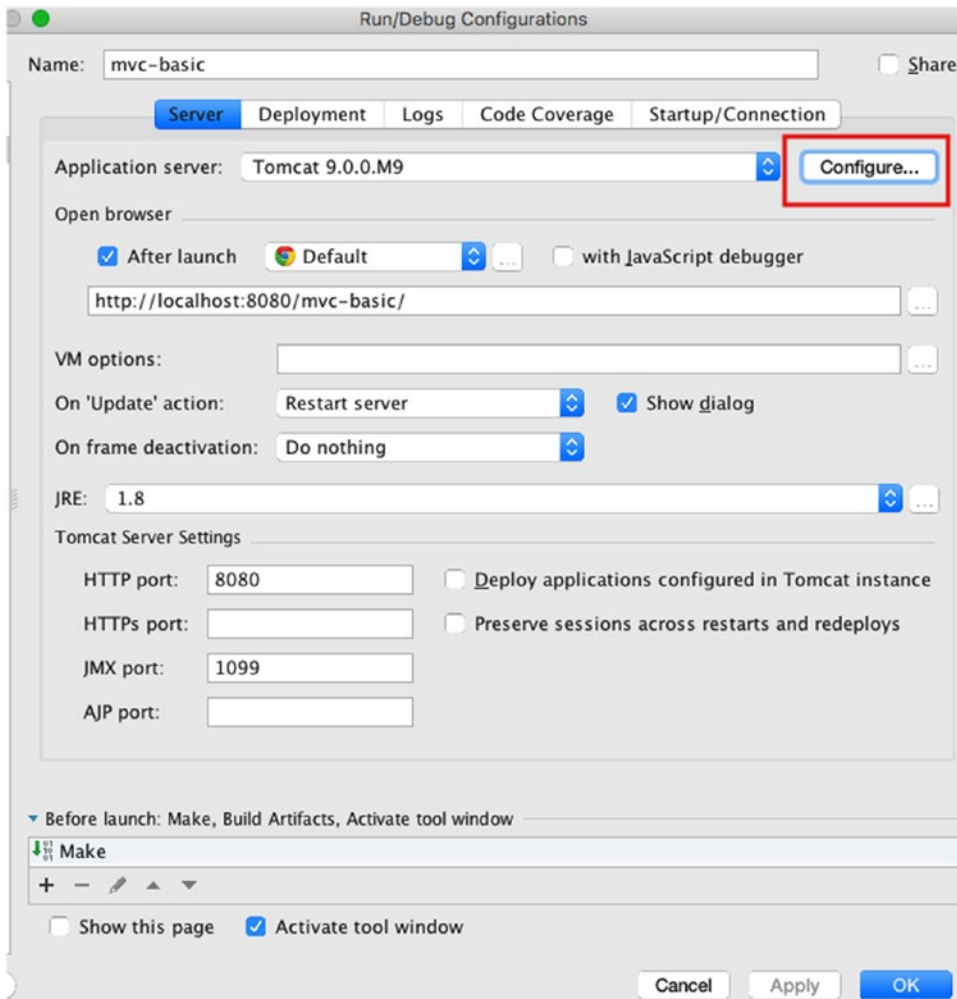


Figure 6-11. IntelliJ IDEA Run/Debug Configuration dialog window, after selecting configuration type

- In the Application Server configuration dialog window, click on the + button, add the Tomcat server you just installed and a name for it, and click Ok (Figure 6-12).

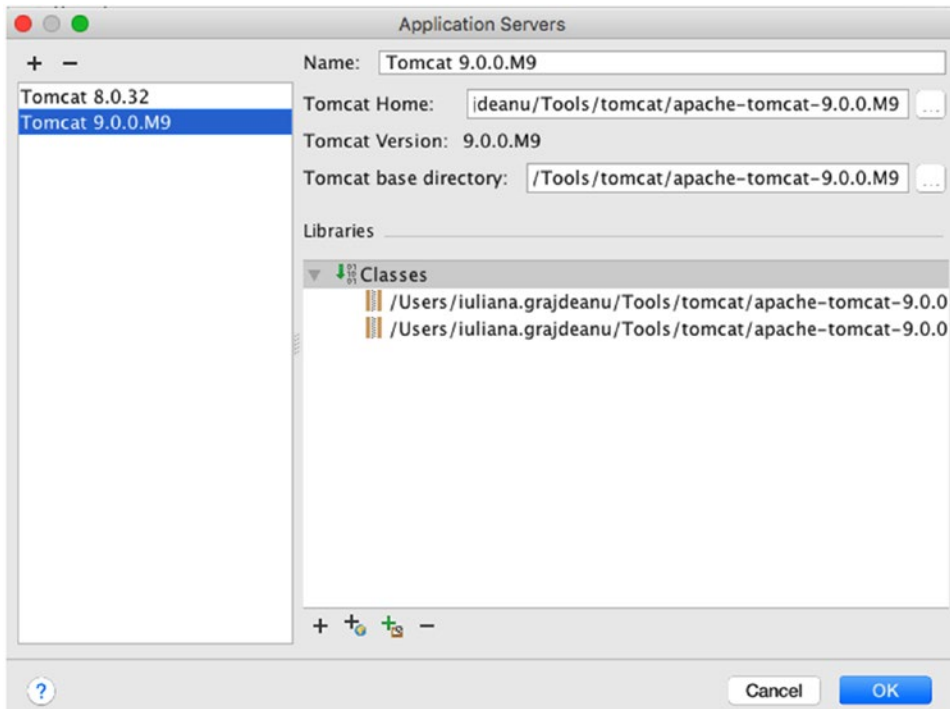


Figure 6-12. IntelliJ IDEA Run/Debug Application Server Configuration dialog window

- In the Run/Debug Configuration dialog window you should have now in the Application Server the Tomcat installation you set up previously. It is time to configure what to deploy on this instance. For this, go to the Deployment tab, and for each numbered item in Figure 6-13, do the following:
 1. Introduce a name for the run configuration; `mvc-basic` should be appropriate.
 2. Click on the + button.
 3. From the list of web artifacts that IntelliJ IDEA identifies, select `10-ps-mvc-practice-1.0.war`. Choose (exploded) if you are interested in running Tomcat in Debug mode, and use breakpoints.
 4. In the Application context section, introduce the name of the web application context, `mvc-basic` in this case.

Then click Ok in all dialog windows and start the server by selecting the `mvc-basic` configuration in the Run/Debug Configuration menu and clicking on the Run button that is evidenced in Figure 6-9. IntelliJ IDEA should automatically open the browser with the main page of the Pet Sitter application.

And this is how the Pet Sitter application can be run with Apache Tomcat 9. The steps are similar even if a different application server is used.

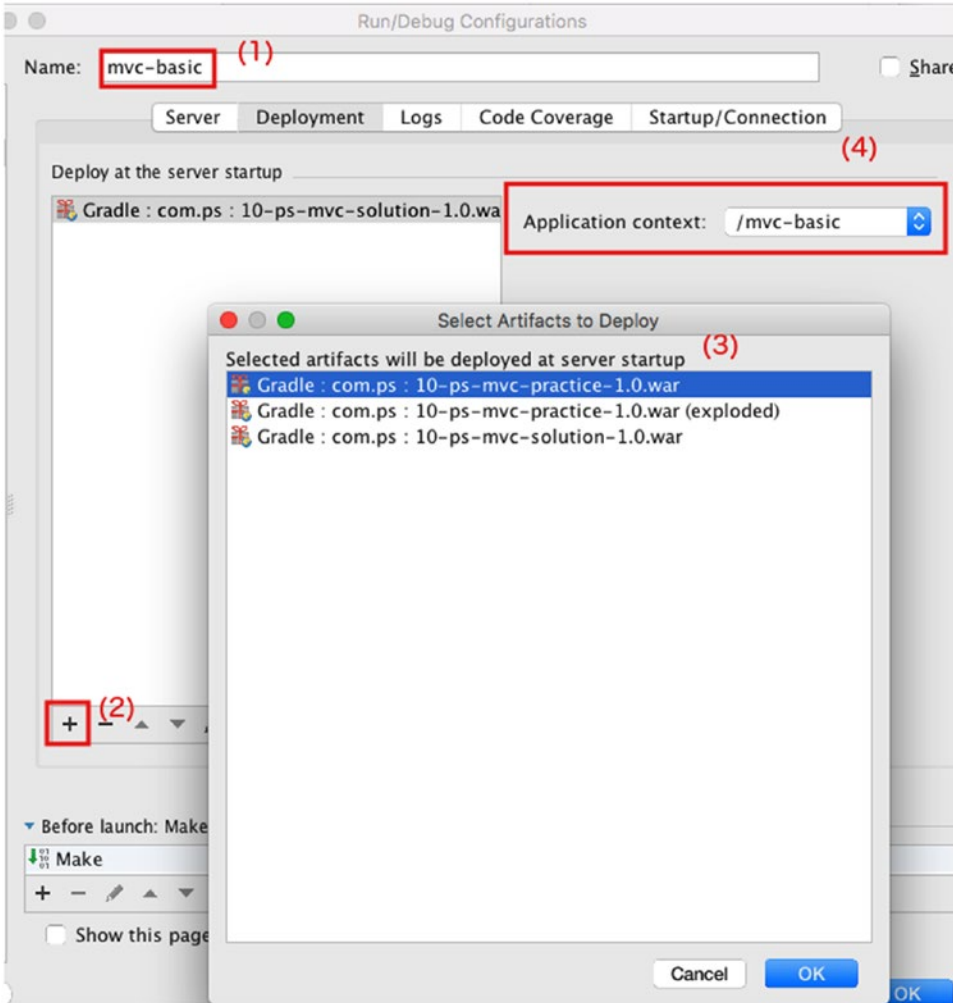


Figure 6-13. IntelliJ IDEA Run/Debug Configuration dialog window, the Deployment tab

Spring Security

Spring Security is yet another Spring Framework created to make a developer’s life easy and the work pleasant, since it is used to secure web applications. It is very easy to use and is highly customizable for providing access control over units of an application. For writing secure Spring web applications, this is the default tool developers go for, because configuration follows the same standard with all the Spring projects,

and infrastructure beans are provided out of the box for multiple types of authentication, all of which is obviously compatible with other Spring projects. Spring Security provides a wide set of capabilities that can be grouped into four areas of interest: authentication, authorizing web requests, authorizing methods calls, and authorizing access to individual domain objects.

In talking about securing an application, the following concepts are important:

- **Principal** is the term that signifies a user, device, or system that could perform an action within the application.
- **Credentials** are identification keys that a principal uses to confirm its identity.
- **Authentication** is the process of verifying the validity of the principal's credentials.
- **Authorization** is the process of making a decision whether an authenticated user is allowed to perform a certain action within the application
- **Secured item** is the term used to describe any resource that is being secured.

There are many ways of authentication, and Spring Security supports all of them: Basic, Form, OAuth, X.509, Cookies, Single-Sign-On. When it comes to where and how those credentials are stored, Spring Security is quite versatile, since it supports everything: LDAP, RDBMS, properties file, custom DAOs; even beans are supported, and many others.⁶

Authorization depends on authentication. A user has first to be authenticated in order for authorization to take place. The result of the authentication process is establishing whether the user has the right to access the application and what actions the user can perform based on roles. Most common user roles within an application are:

- **ADMIN** is a role used for full power. This kind of role is specific to users that have the right to access and manipulate all data, including that of other users.
- **MEMBER** is used for limited power. This kind of role is specific to users that can view data and manipulate only their own details.
- **GUEST** is used for restricted use of the application. This kind of role is used for users that can view only limited data.

Spring Security is preferred for developing web applications because it is **flexible** and because of its **portability**. Spring Security does not need a special container to run in; it can be deployed simply as a secured archive (WAR or EAR) and also can run in standalone environments. A web application secured with Spring Security and archived as a WAR can be deployed on a JBoss or an Apache Tomcat application server, and as long as the underlying method of storing credentials is configured, the application will run exactly the same in any of those application servers. Spring Security is very flexible because configuration of authentication and authorization are fully decoupled. Thus, the storage system of credentials can change without any action being needed to adapt the authorization configuration. This makes applications very consistent, because after all, the scopes of authentication and authorization are different, so it is only logical for them to be covered by different detachable components.

Spring Security is also quite extensible, since almost everything related to security can be extended and customized: how a principal is identified, where the credentials are stored, how authorization decisions are made, where security constraints are stored, etc.

⁶A full list of authentication technologies with which Spring Security integrates can be found here: <http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#what-is-acegi-security>

Security is a cross-cutting concern, so implementing authorization might lead to code cluttering and tangling. Spring Security is implemented using Spring AOP with separation of concerns in mind. Under the hood, Spring Security uses a few infrastructure beans to implement the two processes. In Figure 6-14, the process of authentication and authorization of a user is depicted in an abstract but accurate manner.

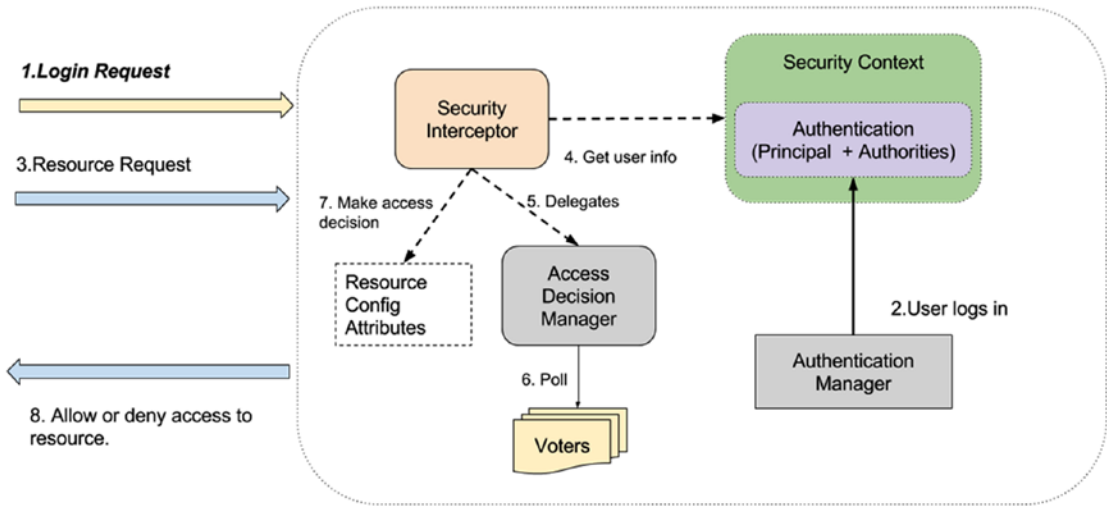


Figure 6-14. Spring Security under the hood

The flow depicted in the previous image can be explained as follows.

1. A user tries to access the application by making a request. The application requires the user to provide the credentials so it can be logged in.
2. The credentials are verified by the Authentication Manager and the user is granted access to the application. The authorization rights for this user are loaded into the Spring Security context.
3. The user makes a resource request (view, edit, insert, or delete information) and the Security Interceptor intercepts the request before the user accesses a protected/secured resource.
4. The Security Interceptor extracts the user authorization data from the security context and...
5. ...delegates the decision to the Access Decision Manager.
6. The Access Decision Manager polls a list of voters to return a decision regarding the rights of the authenticated user to system resources.
7. Access is granted or denied to the resource based on the user rights and the resource attributes.

Spring Security Configuration

To configure Spring security, the developer must take care of three things:

- declare the security filter for the application
- define the Spring Security context
- configure authentication and authorization

XML Configuration

To configure Spring Security, the `web.xml` must be modified to include the security filter and make the security context the root context of the application. The elements that need to be added/modified in the XML configuration of the web application are depicted in the following code snippet.

```
<web-app ...>
<!-- The root web application context is the security context-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/spring/security-config.xml
    </param-value>
  </context-param>
<!-- Bootstraps the root web application context before servlet initialization -->
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...

<!-- The front controller, the entry point for all requests -->
<servlet>
  <servlet-name>pet-dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
```

```

        /WEB-INF/spring/mvc-config.xml
        /WEB-INF/spring/app-config.xml
    </param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
</web-app>

```

■ ! The `springSecurityFilterChain` is a mandatory name that refers to a bean with the same name in the Spring root application context. The `DelegatingFilterProxy` bean delegates the calls to a list of chained security filter beans and acts as an interceptor for secured requests. In the case above, as the `<url-pattern .../>` element defines all requests will be intercepted (wildcard used: `/*`).

Under the hood, in a secured web environment the secured requests are handled by a chain of Spring-managed beans, which is why the proxy bean is named `springSecurityFilterChain`, because those filters are chained. This chain of filters has the following key responsibilities:

- driving authentication
- enforcing authorization
- managing logout
- maintaining `SecurityContext` in `HttpSession`

The Spring Security framework is built on the foundation of ACEGI Security 1.x. At the beginning, the security filter beans were manually configured and could be used individually, but this led to complex XML configurations that were verbose and prone to errors. Starting with Spring Security 2.0, the filter beans are created and initialized with default values, and manual configuration is not recommended unless it is necessary to change default Spring Security behavior. All filter beans implement the `javax.servlet.Filter` interface. Every bean of type implementing this interface has the purpose of performing tasks filtering on the request of a resource (a servlet or static content), on the response from a resource, or both. Although these filters work in the background, a little coverage is appropriate. In Table 6-1, the security filters are listed and the purpose of each is presented.

Table 6-1. *Spring Security chained filters*

Filter Class	Details
ChannelProcessingFilter	Used if redirection to another protocol is necessary
SecurityContextPersistenceFilter	Used to set up a security context and copy changes from it to HttpSession
ConcurrentSessionFilter	Used for concurrent session handling package
LogoutFilter	Used to log a principal out. After logout a redirect will take place to the configured view
BasicAuthenticationFilter	Used to store a valid Authentication token in the security context
JaasApiIntegrationFilter	This bean attempts to obtain a JAAS Subject and continue the FilterChain running as that Subject
RememberMeAuthenticationFilter	Is used to store a valid Authentication and use it if the security context did not change
AnonymousAuthenticationFilter	Is used to store an anonymous Authentication and use it if the security context did not change
ExceptionTranslationFilter	Is used to translate Spring Security exceptions in HTTP corresponding error responses
FilterSecurityInterceptor	Is used to protect URIs and raise access denied exceptions

Basically every time an HTTP request is received by the server, each of the filters performs its action if the situation applies. The fun part is that these filters can be replaced by custom implementations. Their position in the security chain is defined in Spring by a list of enumeration values, and using these values as parameters for the position attribute of the `<custom-filter .../>`, a different filter bean can be specified to use instead of the default one. In the following code snippet, the `ConcurrentSessionFilter` is replaced by a custom implementation.

```
<beans:beans ...>
  <http>
    <custom-filter position="CONCURRENT_SESSION_FILTER"
      ref="customConcurrencyFilter" />

    <beans:bean id="customConcurrencyFilter"
      class="com.ps.web.session.CustomConcurrentSessionFilter"/>
  </http>
</beans:beans>
```

The list of enumerated values can be found in the `spring-security.xsd`,⁷ and a match between the filters and the values is depicted in Table 6-2.

Table 6-2. *Spring Security chained filters*

Filter Class	Position Enumerated Value
<code>ChannelProcessingFilter</code>	<code>CHANNEL_FILTER</code>
<code>SecurityContextPersistenceFilter</code>	<code>SECURITY_CONTEXT_FILTER</code>
<code>ConcurrentSessionFilter</code>	<code>CONCURRENT_SESSION_FILTER</code>
<code>LogoutFilter</code>	<code>LOGOUT_FILTER</code>
<code>BasicAuthenticationFilter</code>	<code>BASIC_AUTH_FILTER</code>
<code>JaasApiIntegrationFilter</code>	<code>JAAS_API_SUPPORT_FILTER</code>
<code>RememberMeAuthenticationFilter</code>	<code>REMEMBER_ME_FILTER</code>
<code>AnonymousAuthenticationFilter</code>	<code>ANONYMOUS_FILTER</code>
<code>ExceptionTranslationFilter</code>	<code>EXCEPTION_TRANSLATION_FILTER</code>
<code>FilterSecurityInterceptor</code>	<code>FILTER_SECURITY_INTERCEPTOR</code>

The equivalent using Java Configuration does not use the position enum values, but a set of methods of the `HttpSecurity` object:

```
addFilterAfter(Filter filter, Class<? extends Filter> afterFilter)
```

and

`addFilterBefore(Filter filter, Class<? extends Filter> beforeFilter)`. They receive as a parameter the class of the filter, relative to which the custom implementation should be placed. In the previous example, the `customConcurrencyFilter` bean is placed in the chain after the `securityContextPersistenceFilter` bean. So the equivalent Java Configuration will look like the following code snippet. Do not focus on annotations, since they will be covered a little bit later in the section. Just pay attention to the custom filter definition and the `http.addFilterAfter(..)` call:

```
import com.ps.web.session.CustomConcurrentSessionFilter;
import org.springframework.security.web.context.SecurityContextPersistenceFilter;
...

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterAfter(
            SecurityContextPersistenceFilter.class, customConcurrencyFilter());
    }
}
```

⁷Available here: <http://www.springframework.org/schema/security/spring-security.xsd>.

```

    http
        .authorizeRequests()
        ...
        .logout()
        .logoutUrl("/logout")
        .logoutSuccessUrl("/");
    }
    ...
    @Bean
    CustomConcurrentSessionFilter customConcurrencyFilter(){
        return new CustomConcurrentSessionFilter();
    }
}

```

The URLs to intercept are defined in a Spring-specific XML file, and roles allowed to access them are defined in the Spring Context and loaded there at authentication time. In using XML, a separate file for security configuration is used called `security-config.xml` in this example. To make configuration easy to set up, Spring provides a special namespace named `spring-security` that contains strictly security configuration elements. The security configuration file can be written relative to this namespace, which removes the necessity to use a prefix for security elements. In the code snippet below, a simple version of a configuration file is depicted that restricts access to users' pages to only fully authenticated users. The login form location and logout paths are depicted as well.

```

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

    <http>
        <intercept-url pattern="/auth*"
            access="IS_AUTHENTICATED_ANONYMOUSLY"/>
        <intercept-url pattern="/users/**"
            access="IS_AUTHENTICATED_FULLY"/>
        <form-login login-page="/auth" />
        <logout logout-url="/logout" />
    </http>

</beans:beans>

```

The `IS_AUTHENTICATED_ANONYMOUSLY` value is processed under the hood by a bean known as a voter, of type `org.springframework.security.access.vote.AuthenticatedVoter`. In this class, three values for the access configuration attribute are defined. The voter will process this security configuration attribute and decide whether a resource should be available to an anonymous user. The anonymous user is used for public access (browsing) to a web application and is the type of user that accesses a site without logging in. The `IS_AUTHENTICATED_FULLY` configuration attribute is also named in a self-explanatory way: the resource that matches the associated URL can be accessed only by a user that was logged in, credentials verified, and

identity confirmed. Thus the user has been fully authenticated. Aside from the two configuration attributes, there is one more: `IS_AUTHENTICATED_REMEMBERED`. If a resource is protected by such a rule, it can be accessed only if the user is authenticated fully or authenticated using the remember-me method.⁸

The `<form-login .../>` configuration element is used to define the request URL for the login form where the user can provide its credentials. Using this configuration, when an anonymous user tries to access a secured resource, the application will direct him to the login form page, instead of just showing a page that will tell him he does not have access to that resource.

The `<logout .../>` configuration element is used to define the request URL for the logout form. When the user clicks on this link, the HTTP session will be invalidated, and then a redirect to `/login?success` will be done. This behavior can be configured and a different logout success URL can be defined, but most applications, redirect the user to the login page, and a message such as "You have successfully logged out" is displayed next to the login form, so the user can log in again if needed.

The paths defined as values for the pattern attribute are pieces of URLs defined using ANT style paths. The URLs that match them are secured and verified according to rules defined by the `<intercept-url .../>` elements. Wildcards can be used to specify a family of related URLs and apply the same security rule to all of them. The access attribute values are typical security token values used to specify what kind of access the user needs to have. Access is linked to user roles, so URLs can be configured to be accessed only by users with certain rights.

```
<beans:beans ...>
  <http>
    <intercept-url pattern="/users/edit"
      access="ROLE_ADMIN"/>
    <intercept-url pattern="/users/list"
      access="ROLE_USER"/>
    <intercept-url pattern="/users/**"
      access="IS_AUTHENTICATED_FULLY"/>
  </http>
</beans:beans>
```

■ ! The order of defining the URL patterns is important, and **the most restrictive must be on top**; otherwise, a more relaxed rule will apply, and some URL will be accessible to users that should not have access to them.

You might have noticed that writing configuration rules for specific URLs is quite annoying if you are using token values for security rights, since you have to check the documentation to see what you are allowed to use as a value for the access attribute. But Spring, being the flexible and beautiful technology that it is, starting with Spring Security 3.0 offers the possibility to use Spring EL expressions to declare access rules. So the file above can make use of security-specific methods as values for the access attributes, and all that is required is just to tell Spring Security that you want to use them, by configuring the `<http .../>` element accordingly, and setting the `use-expressions` attribute value to true.

```
<beans:beans ...>
  <http use-expressions="true">
    <intercept-url pattern="/auth*" access="permitAll"/>
    <intercept-url pattern="/users/edit"
      access="hasRole('ROLE_ADMIN')"/>
  </http>
```

⁸Remember-me or persistent-login authentication refers to web sites able to remember the identity of a principal between sessions.

```

<intercept-url pattern="/users/list"
  access="hasRole('ROLE_USER')"/>
<intercept-url pattern="/users/**"
  access="hasAnyRole('ROLE_USER, ROLE_ADMIN')"/>
</http>
</beans:beans>

```

The expressions have really obvious names regarding their purpose and can be combined to declare complex rules. The following listing depicts only a few of the possibilities:

- `hasRole('AnyRole')` checks whether the principal has the role given as argument.
- `hasAnyRole('[RoleList]')` checks whether the principal has any of the roles in the `RoleList`.
- `isAnonymous()` allows access for unauthenticated principals.
- `isAuthenticated()` allows access for authenticated and remembered principals.
- `isAuthenticated()` and `hasIpAddress('192.168.1.0/24')` allows access for authenticated and remembered principals in the network with this IP class: `192.168.1.0/24`.
- `hasRole('ROLE_ADMIN')` and `hasRole('ROLE_MANAGER')` allows access for principals that have role `ROLE_ADMIN` **and** `ROLE_MANAGER`.

■ ! Once the support for SpEL expression has been configured using `use-expressions="true"`, the previous syntax for access values cannot be used, so roles and configuration attributes cannot be used as values for the access attribute directly. So mixing the two ways is not possible.

Spring Security 4 introduced a simplification that allows for access expressions to be specified without the `ROLE_` prefix in front of them, and thus the configuration above becomes:

```

<http use-expressions="true">
  <csrf disabled="true"/>
  <intercept-url pattern="/auth*" access="permitAll"/>
  <intercept-url pattern="/users/edit"
    access="hasRole('ADMIN')"/>
  <intercept-url pattern="/users/list"
    access="hasRole('USER')"/>
  <intercept-url pattern="/users/**"
    access="hasAnyRole('USER, ADMIN')"/>
</http>

```

■ ! Also in Spring Security 4, the possibility of using CSRF tokens in Spring forms to prevent cross-site request forgery was introduced.⁹ A configuration without a `<csrf />` element configuration is invalid, and every login request will direct you to a 403 error page stating:

```
Invalid CSRF Token 'null' was found on the request parameter
'_csrf' or header 'X-CSRF-TOKEN'.
```

To migrate from Spring Security 3 to version 4, you have to add a configuration for that element, even if all you do is disable using CSRF tokens.

```
<http auto-config="true" use-expressions="true">
  <csrf disabled="true"/>
  <intercept-url pattern="/auth*" access="permitAll"/>
  ...
  <form-login login-page="/auth"
    authentication-failure-url="/auth?auth_error=1"
    default-target-url="/"/>
  <logout logout-url="/logout"
    logout-success-url="/"/>
</http>
```

The `authentication-failure-url` attribute is used to define where the user should be redirected when there is an authentication failure. This can be a special error view that depending on the parameter will show the user a different error message.

The `default-target-url` attribute is used to define where the user will be redirected after a successful authentication.

■ ! Other critical changes are related to the login form: **default** Spring resources, like the login url (URL that indicates an authentication request) and names of the request parameters (expected keys for generation of an authentication token).¹⁰ These were changed in order to match JavaConfig. 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`, and thus the login form in the `auth.jsp` view mapped to path `/auth` until Spring 3 looks like this:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
...
  <form action="<c:url value='j_spring_security_check'/">" method="post">
    <table>
      <tr>
        <td>
```

⁹This is a type of attack that consists in hacking an existing session in order to execute unauthorized commands in a web application. You can read more about it here: https://en.wikipedia.org/wiki/Cross-site_request_forgery.

¹⁰The full list of configuration details changes that were made to match Java Configuration can be found here: <https://jira.spring.io/browse/SEC-2783>.

```

        <label for="username">
            <spring:message code="login.username"/>
        </label>
    </td>
    <td>
        <input type='text' id='username' name='j_username'
            value='<c:out value="\${user}"/>' />
    </td>
</tr>
<tr>
    <td>
        <label for="password">
            <spring:message code="login.password"/>
        </label>
    </td>
    <td><input type='password' id='password' name='j_password' /></td>
</tr>
<tr>
    <td colspan="2">
        <button type="submit">
            <spring:message code="login.submit"/>
        </button>
    </td>
</tr>
</table>
</form>

```

Starting with Spring 4, the default login URL value is */login*, and the default names for the authentication keys are *username* and *password*, and thus the login form must be modified to:

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
...
<form action="<c:url value='/login'/>" method="post">
    <table>
        <tr>
            <td>
                <label for="username">
                    <spring:message code="login.username"/>
                </label>
            </td>
            <td>
                <input type='text' id='username' name='username'
                    value='<c:out value="\${user}"/>' />
            </td>
        </tr>
        <tr>
            <td>
                <label for="password">
                    <spring:message code="login.password"/>
                </label>
            </td>
            <td>
                <input type='password' id='password' name='password' />
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <button type="submit">
                    <spring:message code="login.submit"/>
                </button>
            </td>
        </tr>
    </table>
</form>

```

```

        </td>
        <td><input type='password' id='password' name='password' /></td>
    </tr>
    <tr>
        <td colspan="2">
            <button type="submit">
                <spring:message code="login.submit" />
            </button>
        </td>
    </tr>
</table>
</form>

```

■ ! All previous examples used default values for login URL and authentication key names: *j_spring_security_check*, *j_username*, *j_password* (in Spring Security 3), *login*, *username*, *password* (in Spring Security 4), but all these values can be redefined using Spring configuration. Just set the following attributes for the `<form-login ..>` element: `login-processing-url`, `username-parameter`, `password-parameter` with the desired values.

```

<form-login login-page="/auth"
    login-processing-url="/my-login"
    username-parameter="my-user"
    password-parameter="my-password"
    ...<!-- other attributes-->
/>

```

Configuring authentication is done in the Spring Security configuration file as well, and the default authentication provider is the DAO Authentication provider. A specific `UserDetailsService` implementation is used to provide credentials and authorities. In the examples in the book, the credentials are specified directly in the Spring Security Configuration file.

```

<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="john" password="doe" authorities="ROLE_USER" />
            <user name="jane" password="doe" authorities="ROLE_USER,ROLE_ADMIN" />
            <user name="admin" password="admin" authorities="ROLE_ADMIN" />
        </user-service>
    </authentication-provider>
</authentication-manager>

```

But there are other options. To increase application security a little more, the credentials could be read from a properties file, where the password is stored in encrypted form and a salt can be applied using configuration. **Password-salting** is an encryption method used to increase the security of passwords by adding a well-known string to them.

```

<!-- spring-config.xml -->
<authentication-manager>
  <authentication-provider>
    <password-encoder hash="md5" >
      <salt-source system-wide="SpringSalt"/>
    </password-encoder>
    <user-service properties="/WEB-INF/users.properties" />
  </authentication-provider>
</authentication-manager>

#/WEB-INF/users.properties
john=471540bd22898656564b9c85a18b3e80,ROLE_USER
#password: john

jane=1f533ad8d26c7bec84a291f62668a048,ROLE_USER,ROLE_ADMIN
#password: jane

admin=55c98bea671295de1e020621cc670ac4,ROLE_ADMIN
#password: admin

```

■ ! The passwords presented in the following code snippet were generated by postfixing the password value with the "SpringSalt" text and then applying an MD5 function on the resulting text value. Example:

```
MD5("john" + "SpringSalt") = "471540bd22898656564b9c85a18b3e80"
```

On Windows, the MD5 hash can be generated only for files using the FCIV command or the `Get-FileHash` command in the PowerShell. On MacOS there is an `md5` command, and on Linux systems there is an `md5sum` command. In the following code snippet, the call to generate the MD5 hash for a password equal to "john" with the salt "SpringSalt" for MacOS and Linux are depicted.

```

#MacOS
$ echo "johnSpringSalt" | md5
471540bd22898656564b9c85a18b3e80
#Linux system
$ echo "johnSpringSalt" | md5sum
471540bd22898656564b9c85a18b3e80

```

The credentials were decoupled from the configuration by isolating them in a property file. The file can be edited outside the application, and the properties can be reloaded using a property reader *refreshable bean*.¹¹

¹¹A refreshable bean is a dynamic-language-backed bean that with a small amount of configuration can monitor changes in its underlying source file resource and then reload itself when the dynamic language source file is changed (for example when a developer edits and saves changes to the file on the filesystem). Official documentation reference: <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#dynamic-language-refreshable-beans>.

The credentials property file has a specific syntax:

```
[username] = [password(encrypted)],[role1,role2...]
```

In the previous example, the passwords were added to the `SpringSalt` value and then were encrypted using the `md5`¹² algorithm. But they were all hashed in the exact same way and with the same salt. If someone could find out the salt that the server uses, they would become crackable. That is why the salt should be a random value, different for each user, for example a property unique to that user such as its id. The salt-source can be configured to use a constant user property such as the ID as salt value as well.

```
<authentication-manager>
  <authentication-provider>
    <password-encoder hash="md5">
      <!-- id property from class User -->
      <salt-source user-property="id" />
    </password-encoder>
  </authentication-provider>
</authentication-manager>
```

In the above examples, only the MD5 algorithm was used, but there are more of them supported in Spring Security, and a developer can use one of the supported ones (MD4, Bcrypt, SHA, SHA-256, etc.) or implement his own.

The in-memory, or directly in the configuration file, approach is useful for testing and development, but in a production scenario, a securer method of credential storage is usually used. In production environments, credentials are stored in a database or LDAP system. In this case, the service providing the credentials must be changed to a JDBC-based one:

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="authDataSource" />
  </authentication-provider>
</authentication-manager>
```

The authentication tables must be accessible using the `authDataSource` bean, and their structure must respect the following rules: one table is named `users`, one must be named `authorities`, and the following queries must be executed correctly:

```
SELECT username, password, enabled FROM users WHERE username = ?
SELECT username, authority FROM authorities WHERE username = ?
```

The Spring Security reference documentation includes some SQL scripts to create the security tables. The syntax is for HSQLDB, but the scripts can easily be adjusted for any SQL normalized database.¹³

A few details from this section are general and apply in a Java Configuration environment as well, but they were mentioned here to paint a full view of the Spring Security framework. And now it is time to see how Java Configuration can make all these configuration details more practical and quicker to set up.

¹²Read more about MD5 here: <https://en.wikipedia.org/wiki/MD5>

¹³Documentation reference for security table DDL scripts: <http://docs.spring.io/spring-security/site/docs/4.1.3.RELEASE/reference/htmlsingle/#appendix-schema>.

Spring XML Configuration without web.xml

If `web.xml` disappears, what happens with the `springSecurityFilterChain` filter? The security filter gets transformed into a class extending a Spring specialized class: `org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer`. And the class that matches the `DispatcherServlet` declaration must be made to extend the `org.springframework.web.servlet.support.AbstractDispatcherServletInitializer` so the root context can be set to be the security context. The following code snippet depicts the situation in which Spring Security is configured using XML and the web application is configured using a typical web initializer class.

```
import org.springframework.security.web.context.
    AbstractSecurityWebApplicationInitializer;
// Empty class needed to register the springSecurityFilterChain bean
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {
}

public class WebInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        XmlWebApplicationContext ctx = new XmlWebApplicationContext();
        ctx.setConfigLocation("/WEB-INF/spring/security-config.xml");
        return ctx;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext ctx = new XmlWebApplicationContext();
        ctx.setConfigLocations(
            // MVC configuration
            "/WEB-INF/spring/mvc-config.xml",
            // Service configuration
            "/WEB-INF/spring/app-config.xml");
        return ctx;
    }
    ...
}
```

Java Configuration

To develop a working configuration for a Spring Security web application, the XML configuration must be transformed into a security configuration class. The class that replaces the Spring XML configuration should extend `WebSecurityConfigurerAdapter`, so the amount of code needed to be written for a valid security configuration should be minimal. Thus, the XML configuration given as an example so far becomes:

```
package com.pr.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
    .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 {

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) {
        try {
            auth.inMemoryAuthentication()
                .withUser("john").password("doe").roles("USER").and()
                .withUser("jane").password("doe").roles("USER", "ADMIN").and()
                .withUser("admin").password("admin").roles("ADMIN");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/user/edit").hasRole("ADMIN")
            .antMatchers("/**").hasAnyRole("ADMIN", "USER")
            .anyRequest()
            .authenticated()
            .and()
            .formLogin()
            .usernameParameter("username") // customizable
            .passwordParameter("password") // customizable
            .loginProcessingUrl("/login") // customizable
            .loginPage("/auth")
            .failureUrl("/auth?auth_error=1")
            .defaultSuccessUrl("/home")
            .permitAll()
            .and()
            .logout()
            .logoutUrl("/logout")
            .logoutSuccessUrl("/")
            .and()
            .csrf().disable();
    }
}

```

The `@EnableWebSecurity` annotation must be used on Security configuration classes that must also extend `org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter` and provide implementation for the `configure(HttpSecurity http)` method. To simplify the configuration, the `configure(WebSecurity web)` method can also be overridden to specify resources that Spring Security should be ignoring, such as style files and images, for example, thus simplifying the implementation of the `configure(HttpSecurity http)` method and decoupling unsecured elements from secured ones.

```
package com.pr.config;
...
import org.springframework.security.config.annotation.web.
    builders.WebSecurity;
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/resources/**", "/images/**", "/styles/**");
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/user/edit").hasRole("ADMIN")
            .antMatchers("/**").hasAnyRole("ADMIN", "USER")
            .anyRequest()
            .authenticated()
            .and()
            .formLogin()
            .usernameParameter("username") // customizable
            .passwordParameter("password") // customizable
            .loginProcessingUrl("/login") // customizable
            .loginPage("/auth")
            .failureUrl("/auth?auth_error=1")
            .defaultSuccessUrl("/home")
            .permitAll()
            .and()
            .logout()
            .logoutUrl("/logout")
            .logoutSuccessUrl("/")
            .and()
            .csrf().disable();
    }
}
```

The `antMatcher(...)` method is the equivalent of the `<intercept-url.../>` element from XML, and equivalent methods are available to replace the configuration for the login form, logout URL configuration, and CSRF token support. To enable CSRF usage, the configuration above must also define a CSRF provider bean and use it in the configuration:

```
...
import org.springframework.security.web.csrf.CsrfTokenRepository;
import org.springframework.security.web.csrf.HttpSessionCsrfTokenRepository;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public CsrfTokenRepository repo() {
        HttpSessionCsrfTokenRepository repo = new HttpSessionCsrfTokenRepository();
        repo.setParameterName("_csrf");
        repo.setHeaderName("X-CSRF-TOKEN");
        return repo;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .csrf().csrfTokenRepository(repo());
    }
}
```

Out of the box, Spring provides three `CsrfTokenRepository` implementations that are listed and explained in Table 6-3.

Table 6-3. *Spring Security chained filters*

Filter Class	Position Enumerated Value
<code>CookieCsrfTokenRepository</code>	Persists the CSRF token in a cookie named "XSRFTOKEN" and reads from the header "X-XSRF-TOKEN" following the conventions of AngularJS
<code>HttpSessionCsrfTokenRepository</code>	Persists the CSRF token in the <code>HttpSession</code> in the parameter with the name set by calling method <code>setParameterName()</code> and reads from the header with the name set by calling the <code>setHeaderName()</code>
<code>LazyCsrfTokenRepository</code>	Delays saving new CSRF token until its generated attributes are accessed.

When CSRF support is used, logging out needs to be implemented accordingly, and make sure that the CSRF token is erased from existence and disabled so that it cannot be used by malevolent requests.¹⁴ So the simple logout link from Spring 3:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
  <a href="<spring:url value="/j_spring_security_logout"/>">
    <spring:message code="menu.logout"/>
  </a>
```

Becomes in Spring 4 a full fledged form that sends the CSRF token to the application to be erased:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
...
<spring:url value="/logout" var="logoutUrl" />
  <form action="{logoutUrl}" id="logout" method="post">
    <input type="hidden" name="{_csrf.parameterName}"
      value="{_csrf.token}"/>
  </form>
  <a href="#" onclick="document.getElementById('logout').submit();">
    <spring:message code="menu.logout"/>
  </a>
```

The `<spring:message .../>` element used above is a special Spring tag used for internationalization. The tag library directive must be present in the JSP file, and the values for the specific element are taken from properties internationalization files that are located under `WEB-INF`. Since internationalization is not a topic for the Spring Core certification exam, we won't go too deeply into this here.

And since the configuration section has covered everything needed, it is time to introduce how to use the security context and rules in the code.

Security Tag Library

In writing JSP pages, multiple tag libraries are available to make the work easier and less redundant and to provide functionality. By adding a tag library reference in the JSP page header, JSP elements defined in that library can be used. Spring provides a tag library that can be used to secure JSP elements.

In the configuration examples presented up to now, access to certain resources was managed via `<intercept-url .../>` elements or `antMatcher(...)` methods.

```
//mvc-security.xml
<http auto-config="true" use-expressions="true">
  <intercept-url pattern="/users/show/*"
    access="hasRole('ADMIN')"/>
...
</http>
```

```
//or SecurityConfig.java
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

¹⁴Cross-site request forgery or session-riding exploits the trust that a site has in a user's browser. When the CSRF token is stored in the session, it has a specific value for the duration of that session. So even if the session is intercepted and data from it is used by an attacker to access the site, by disabling the CSRF token at logout, sensitive requests that require the CSRF token are prohibited.

```

...
Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/users/show/*").hasRole("ADMIN")
            ...
    }
}

```

Each of the configurations above will deny any user with only USER role access to `/users/show/`. In Figure 6-15, the server reply is depicted when a user with USER role tries to access a `/users/show/` resource.



Figure 6-15. Server response when user is not authorized to view resource

But does it make sense to display on the page a link to a forbidden resource at all? Of course not. Also, production applications are usually quite big and contain many different URLs, so full configuration using an XML file or a security configuration class can become quite complex. To simplify this, starting with Spring Security version 2.0 a security tag library is provided that can be used to secure items at JSP level. In the above example, we can choose to display a column containing the link to the resource based on the security configuration for the user.

```

// <!-- /users/list.jsp -->
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
...
<table>
    <thead>
    <tr>
        <sec:authorize access='hasRole("ROLE_ADMIN")'>
            <td>
                <spring:message code="label.User.count"/>
            </td>
        </sec:authorize>
        <td>
            <spring:message code="label.User.username"/>
        </td>
        ... <!-- other cells-->
    </tr>
    </thead>
    <tbody>
    <tr>
        <td>
            <spring:message code="label.User.count"/>
        </td>
        <td>
            <spring:message code="label.User.username"/>
        </td>
        ... <!-- other cells-->
    </tr>
    </tbody>
</table>

```

```

</tr>
</thead>
<c:forEach var="user" items="{users}">
  <tr>
    <sec:authorize access='hasRole("ROLE_ADMIN")'>
      <td>
        <spring:url var="showUrl" value="show/{id}">
          <spring:param name="id" value="{user.id}"/>
        </spring:url>
        <a href="{showUrl}">{user.id}</a>
      </td>
    </sec:authorize>
    <td>
      {user.username}
    </td>
    ... <!-- other cells-->
  </tr>
</c:forEach>
</table>

```

The `<spring:url .. />` element is a tag from the Spring tag library that is used to compose a link dynamically based on a parameter. The definition in the previous code snippet will resolve to links like this: `http://localhost:8080/mvc-security/users/show/12`, where 12 is the id of the user.

The `<sec:authorize .. />` above has the effect that after a user has been authenticated, its roles are loaded into the security context, and when he accesses the `list.jsp` page, the response view is computed taking its roles into account. The security elements in the previous example state that only for users with ADMIN role should the `<td />` elements containing the `/users/show/*` URL should be part of the view. So the user `john`, for example, having only a USER role, will see a different view from that of a user with role ADMIN, as depicted in Figure 6-16.

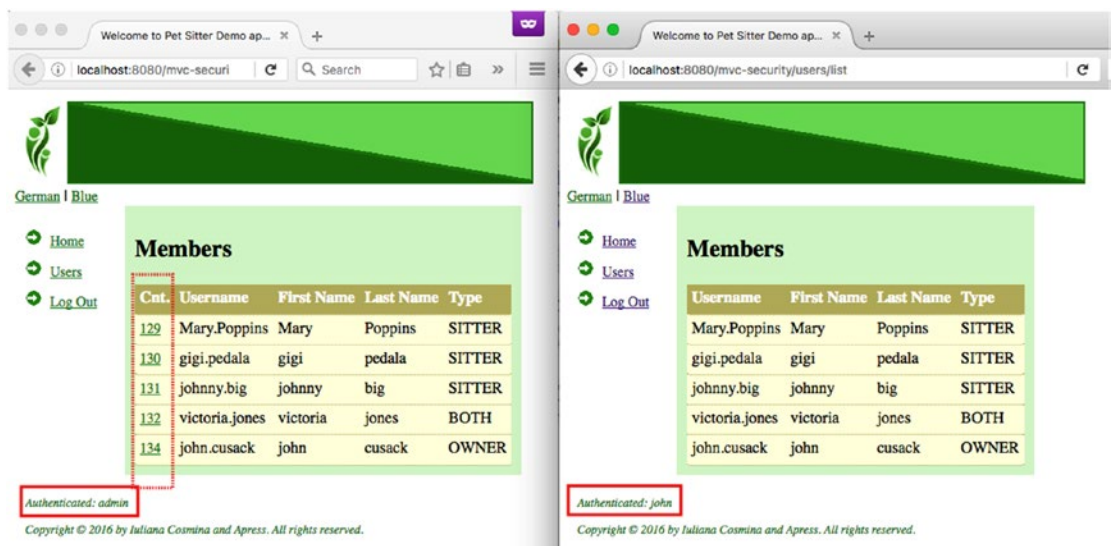


Figure 6-16. /users/show view for roles ADMIN and USER

According to what is said before, the link is not displayed, but what happens if the user enters the link manually in the browser? If there is no restriction for that link in the configuration, the view will be shown to the user regardless of its role. So the restriction from the configuration is needed as well. If the restriction is defined as depicted in the code snippet at the beginning of the section, the security rule can be used in the JSP code to avoid complicated expressions like `'hasRole("ROLE_ADMIN")'` by replacing the `access` attribute with `url` for the `authorize` element and replacing the `hasRole` expression with the URL defined in the security rule. This will tell Spring to check whether the user has access to that URL before displaying the view. And if it does not, the message in Figure 6-15 is displayed. This makes it possible to link a security rule to a url pattern that can be used as a security attribute value to secure resources with any URL path. So the JSP code above becomes:

```
// <!-- /users/list.jsp -->
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
...
<table>
    <thead>
    <tr>
        <sec:authorize url='/users/show/*'>
            <td>
                <spring:message code="label.User.count"/>
            </td>
        </sec:authorize>
        <td>
            <spring:message code="label.User.username"/>
        </td>
        ... <!-- other cells-->
    </tr>
    </thead>
    <c:forEach var="user" items="{users}">
        <tr>
            <sec:authorize url='/users/show/*'>
                <td>
                    <spring:url var="showUrl" value="show/{id}">
                        <spring:param name="id" value="{user.id}"/>
                    </spring:url>
                    <a href="{showUrl}">{user.id}</a>
                </td>
            </sec:authorize>
            <td>
                {user.username}
            </td>
            ... <!-- other cells-->
        </tr>
    </c:forEach>
</table>
```

But there is another security concern: the method that populates the view is defined in the service layer. If the application supports web services too, how can we prevent a user from calling that method directly and getting the data? Or by using a remote REST call? Because the security is currently defined only in the web layer. The next section is about how Spring Security can be used to secure items on lower levels too.

Method Security

To apply security to lower layers of an application, Spring Security uses AOP. The respective bean is wrapped in a proxy that before calling the target method, first checks the credentials of the user and calls the method only if the user is authorized to call it. In XML, the definition of such a proxy is done using the `global-method-security` configuration element provided by the security namespace to enable method-level security and by defining a secured pointcut and the rule that applies to it.

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

  <global-method-security>
    <protect-pointcut expression="execution(* com.ps.*.*Service.findById(*))"
      access="hasRole('ADMIN')" />
  </global-method-security>

</beans:beans>
```

But the same can be done more easily using the Spring Security namespace or Java Configuration and annotations. There are two alternatives for doing this:

1. Method-level security must be enabled by annotating a configuration class (good practice is to annotate the Security Configuration class to keep all configurations related to security in one place) with `@EnableGlobalMethodSecurity(securedEnabled = true)`. Methods must be secured by annotating them with Spring Security annotation `@Secured`

```
import org.springframework.security.config.annotation
    .method.configuration.EnableGlobalMethodSecurity;
...
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  ...
}

//UserServiceImpl.java service class
import org.springframework.security.access.annotation.Secured;
...
@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {

  @Secured("ROLE_ADMIN")
  public User findById(Long id) {
    return userRepo.findOne(id);
  }
}
```



```

    }
    ...
}

```

The equivalent configuration using XML makes use of the `<global-method-security ..>` element defined in the Spring Security namespace, which is the XML equivalent of annotation `@EnableGlobalMethodSecurity`. To enable method security, the `secured-annotations` attribute of this element must be configured to have the `enabled` value.

```

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

  <global-method-security secured-annotations="enabled" />

  <http ... />

</beans:beans>

```

2. Method-level security must be enabled by annotating a configuration class (good practice is to annotate the Security Configuration class to keep all configurations related to security in one place) with `@EnableGlobalMethodSecurity(jsr250Enabled = true)`. Methods must be secured by annotating them with JSR-250¹⁵ annotations.

```

import org.springframework.security.config.annotation
    .method.configuration.EnableGlobalMethodSecurity;
...
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    ...
}

//UserServiceImpl.java service class
import javax.annotation.security.RolesAllowed;
...
@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {

```

¹⁵JSR 250: Common annotations for the Java™ Platform <https://www.jcp.org/en/jsr/detail?id=250>.

```

@RolesAllowed("ROLE_ADMIN")
public User findById(Long id) {
    return userRepo.findOne(id);
}

...
}

```

The equivalent configuration using XML makes use of the `<global-method-security ..>` element defined in the Spring Security namespace, which is the XML equivalent of annotation `@EnableGlobalMethodSecurity`. To enable method security, the `jsr250-annotations` attribute of this element must be configured to have the `enabled` value.

```

<beans:beans ...>

    <global-method-security jsr250-annotations="enabled" />

    <http ... />

</beans:beans>

```

The JSR 250 annotations are standards-based and allow simple role-based constraints to be applied but do not have the power of Spring Security's native annotations.

Both approaches will lead to Spring Security wrapping the service class in a secure proxy. The abstract schema of how a secured method executes and the components involved is depicted in Figure 6-17.

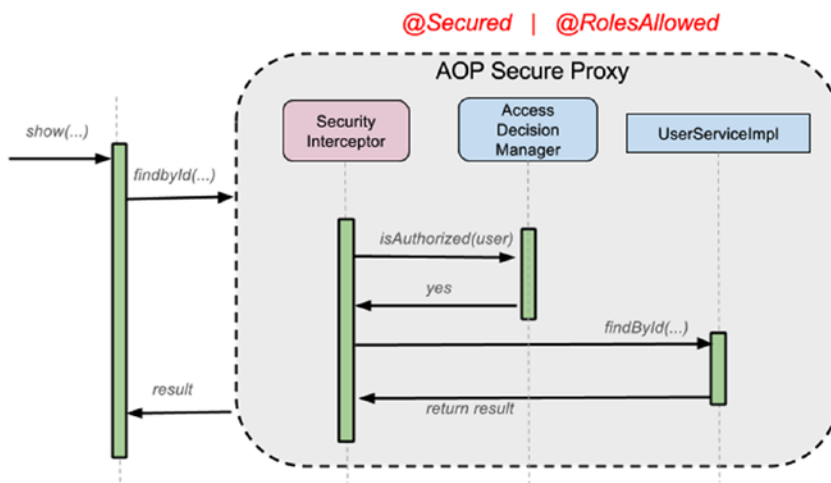


Figure 6-17. Abstract schema of a secured method execution

There are four annotations that support expression attributes that were introduced in Spring 3.0 to allow pre- and postinvocation authorization checks and also to support filtering of submitted collection arguments or return values: `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, and `@PostFilter`. They are enabled in XML by the `pre-post-annotations` attribute of the `<global-method-security ..>` element, which must be configured to have the `enabled` value.

```
<beans:beans ...>
    <global-method-security pre-post-annotations="enabled" />
    <http ... />
</beans:beans>
```

They are enabled using Java Configuration by the `prePostEnabled` attribute of the `@EnableGlobalMethodSecurity` element that must be configured to have the `true` value.

```
import org.springframework.security.config.annotation
    .method.configuration.EnableGlobalMethodSecurity;
...
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    ...
}
```

The annotation can be used as follows:

```
import org.springframework.security.access.prepost.PreAuthorize;

...
@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {

    @PreAuthorize("hasRole('USER')")
    public void create(User user){
        ...
    }
}
```

But the interesting thing about these annotations is that they can access method arguments. In the following snippet, the current logged-in user is verified if it has "admin" permission to delete the user given as argument. It does this using Spring ACL (Access Control List) classes.

```
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.acls.model.Permission;
import org.springframework.security.acls.model.Sid;
...
```

```

@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {

    @PreAuthorize("hasPermission(#user, 'admin')")
    public void delete(User user, Sid recipient, Permission permission){
        ...
    }
}

```

These annotations provide more granularity, since SpEL expressions can be used to restrict the domain that a user is allowed to affect with its actions. For example, in the following example, a user is allowed only to affect a user whose username matches that of the user argument.

```

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.acls.model.Permission;
import org.springframework.security.acls.model.Sid;
...
@Service
@Transactional(readOnly = true, propagation = Propagation.REQUIRED)
public class UserServiceImpl implements UserService {

    @PreAuthorize("#user.username == authentication.name")
    public void modifyProfile(User user){
        ...
    }
}

```

Spring Security ACL is very powerful and easy to implement when you can stick to their Spring database implementation. So if a project requires granulated control over resources, it is definitely the way to go.

■ ! This is all that can be said about Spring Security, and before jumping to the next section you can test your knowledge and understanding by solving the TODO task from project `11-ps-security-practice`. There are seven TODO tasks numbered from 48 to 54, and if you need inspiration or confirmation that your solution is correct, you can compare it with the proposed solution that you can find in project `11-ps-security-solution`.

Task TODO 48, located in the `SecurityWebApplicationInitializer` class, requires you to complete the configuration of this class to register the `DelegatingFilterProxy` bean.

Task TODO 49, located in the `SecurityConfig` class, requires you to complete the configuration of this class to enable support for Spring Security.

Task TODO 50, located in the `SecurityConfig` class, requires you to complete the configuration of this class to enable securing methods using annotations that express attributes.

Task TODO 51, located in the `SecurityConfig` class, requires you to complete the configuration of the `http` object and define the users in Table 6-4, which you can also find in the `auth.jsp` page.

Table 6-4. *Application users*

Username	Password	Role
john	doe	USER
jane	does	USER, ADMIN
admin	admin	ADMIN

Task TODO 52, located in the `SecurityConfig` class, requires you to complete the configuration of the `http` object and ensure that all URLs matching `/users/show/*` and `/users/delete/*` are available only to users with role ADMIN.

Task TODO 53, located in the `WebInitializer` class, requires you to complete the configuration of this class so that the spring context is registered as a root context.

Task TODO 54, located in the `list.jsp` file, requires you to complete the implementation of this JSP page to hide the column with members' IDs that contain the URL to the details view.

Spring Boot

Spring Boot is a set of preconfigured frameworks/technologies designed to reduce boilerplate configuration (infrastructure) and provide a quick way to have a Spring web application up and running. Its goal is to allow developers to focus on implementation of the actual required functionality instead of how to configure an application, by providing out of the box ready-to-use infrastructure beans. On the official Spring Boot page¹⁶ a REST application is given as an example and almost 20 lines of code are needed to have a runnable application. Impressive right? And this is only one of the advantages of using Spring Boot.

Spring Boot makes it easier for developers to build standalone production-level applications that are ready to run. Spring Boot can be used to create applications that are packed as a jar and can be run with `java -jar` or typical war packed projects that can be deployed on an application server. The new thing is that web archives can be executed with `java -jar` as well. The main focus of Spring Boot is on:

- Setting up the infrastructure for a Spring project in a really short time.
- Providing a default common infrastructure configuration but allowing easy divergence from the default as the application grows. So any default configuration can easily be overridden.
- provide a range of nonfunctional features common to a wide range of projects (embedded servers, security, metrics, health checks, externalized configuration).
- Remove the need for XML and code configuration.

The current stable version of Spring Boot as this book is being written is 1.4.0.RELEASE, which works with JDK 7+. In the book, source version 1.4.1.BUILD-SNAPSHOT will be used.

¹⁶Spring Boot official page: <http://projects.spring.io/spring-boot/>

Configuration

Spring Boot can be used like any Java library. It can be added as a dependency to the project, since it does not require any special integration tools and can be used in any IDE. Spring Boot provides ‘starter’ POMs to simplify the Maven configuration of a project. When Maven is used, the project must have as a parent the `spring-boot-starter-parent`, so that useful Maven defaults are provided. Since the sources attached to this book use Gradle, it is very suitable that Gradle is supported too. For Gradle, there is no need to specify a parent, but Spring provides starter dependencies specific to different types of applications. Thus in the `pet-sitter` parent configuration, all starter dependencies and versions will be declared, and subprojects will use only what is needed. There are different starter dependencies depending on the type of application; in the sample below, only a few are declared.¹⁷ To reuse configuration declarations from the parent project, all Spring Boot components and versions will be declared in the `pet-sitter/build.gradle` file.

```
//pet-sitter/build.gradle configuration file snippet
plugins {
    id 'com.gradle.build-receipt' version '1.0'
}
buildReceiptLicense {
    agreementUrl = 'https://gradle.com/terms-of-service'
    agree = 'yes'
}
//gradle build -Dreceipt
allprojects {
    version = '1.0'
    group = 'com.ps'

    ext {
        //spring libs
        springVersion = '4.3.2.RELEASE'
        springJpaVersion = '1.10.2.RELEASE'
        springSecurityVersion = '4.0.3.RELEASE'
        springBootVersion = '1.4.1.BUILD-SNAPSHOT'
        aspectjVersion = '1.8.4'
        yamlVersion='1.17'
        ...
        boot = [
            springBootPlugin:
                "org.springframework.boot:spring-boot-gradle-plugin:$springBootVersion",
            starterWeb      :
                "org.springframework.boot:spring-boot-starter-web:$springBootVersion",
            starterJetty    :
                "org.springframework.boot:spring-boot-starter-jetty:$springBootVersion",
            starterSecurity :
                "org.springframework.boot:spring-boot-starter-security:$springBootVersion",
            starterJpa      :
                "org.springframework.boot:spring-boot-starter-data-jpa:$springBootVersion",
            starterTomcat   :

```

¹⁷The full list is available in the Spring Boot Reference documentation <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter>

```

        "org.springframework.boot:spring-boot-starter-tomcat:$springBootVersion",
        starterTest      :
        "org.springframework.boot:spring-boot-starter-test:$springBootVersion",
        Actuator         :
        "org.springframework.boot:spring-boot-starter-actuator:$springBootVersion",
        Yaml              : "org.yaml:snakeyaml:$yamlVersion"
    ]
    ...
}
subprojects {
    apply plugin: 'java'
    sourceCompatibility = 1.8

    repositories {
        mavenLocal
        mavenCentral
        maven { url "https://oss.sonatype.org/content/repositories/snapshots/" }
        maven { url "http://repo.spring.io/snapshot/" }
        maven { url "http://repo.spring.io/milestone" }
    }
    ...
}

```

In the Gradle configuration file of the parent project `pet-sitter`, versions for main libraries or family of libraries are declared, as well as repositories from which dependencies are downloaded and the version of JAVA used for compiling and running the project. The `11-ps-boot-sample` is a submodule of the `pet-sitter` project. The `11-ps-boot-sample/11-ps-boot-sample.gradle` configuration file contains only the dependencies needed, without versions, because these are inherited from the parent configuration.

```
//11-ps-boot-sample/11-ps-boot-sample.gradle
apply plugin: 'spring-boot'
```

```

buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        classpath boot.springBootPlugin
    }
}

dependencies {
    compile boot.starterWeb
    testCompile boot.starterTest
}

jar {
    baseName = 'ps-boot'
}

```

Each release of Spring Boot provides a curated list of dependencies it supports. The versions of the necessary libraries are selected so that the API matches perfectly, and this is handled by Spring Boot. Therefore, manual configuration of dependencies versions is not necessary. Upgrading Spring Boot will ensure that those dependencies are upgraded as well. This can easily be proved by looking at the transitive dependencies of the `spring-boot-starter-web` in IntelliJ IDEA Gradle view, as depicted in Figure 6-18.

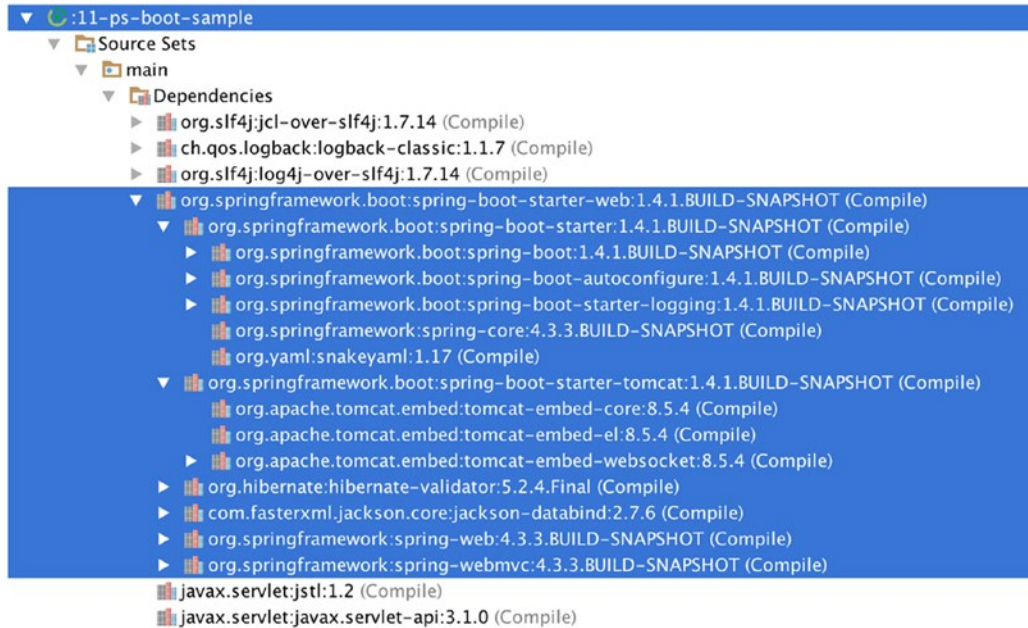


Figure 6-18. *11-ps-boot-sample: spring-boot-starter-web transitive dependencies in IntelliJ IDEA Gradle view*

The version for Spring Boot and project version are inherited from the `pet-sitter` project. To have this project built and running, all we need is one class annotated with `@SpringBootApplication`. This annotation is a top-level annotation designed to use only at class level. It is a convenience annotation that is equivalent to declaring the following three:

- `@Configuration`, because the class is a configuration class and can declare beans with `@Bean`.
- `@EnableAutoConfiguration` is a specific Spring Boot annotation from package `org.springframework.boot.autoconfigure` that has the purpose to enable Spring Application Context, 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, this will be used unless there is another `EmbeddedServletContainerFactory` configuration in the project.

- `@ComponentScan`, because the developer will 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 `scanBasePackages`.

```
package com.ps.start;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication(scanBasePackages =
    {"com.ps.web", "com.ps.start"})
public class Application {

    public static void main(String... args) {
        SpringApplication.run(Application.class, args);
        System.out.println("Started ...");
    }
}
```

The main method is the entry point of the application and it follows the Java convention for an application entry point. This method calls the static `run` method from the `org.springframework.boot.SpringApplication` class that will bootstrap the application and start the Spring IoC container, which will start the configured embedded web server. So if you run this class in IntelliJ IDEA or compile and build the application and execute the jar, the result will be the same: a Spring application will be started with a number of infrastructure beans already configured with the default, most common, configurations. Now that we have a Spring application context let's do something with it, like inspect all the beans. The easiest way to do this is to add a controller class that will display all those beans. But declaring a controller means views also have to be resolved, so the simplest way is to use a REST controller. A REST controller is a controller class annotated with `@RestController`. This annotation is a combination of `@Controller`, the typical stereotype annotation marking a bean as a web component and `@ResponseBody` an annotation that basically tells spring that the result returned by methods in this class do not need to be stored in a model and displayed in a view. The `CtxController` depicted in the following code snippet contains one method that returns a simple HTML code containing a list of all beans in the application context.

■ ! This `CtxController` is a special implementation designed to depict the beans names in the browser in a readable manner, as you will probably use a browser to test this project. The HTML format is not suitable for REST services, as you will be shown in **Chapter 7**.

To register this class, the package should be scanned with `@ComponentScan`. But this annotation is no longer needed as `@SpringBootApplication` has the behavior covered.

```
//CtxController.java
package com.ps.start;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Arrays;

@RestController
public class CtxController {

    @Autowired
    ApplicationContext ctx;

    @RequestMapping("/")
    public String index() {
        StringBuilder sb = new StringBuilder("<html><body>");

        sb.append("Hello there dear developer,
            here are the beans you were looking for: </br>");

        //method that returns all the bean names in the context of the application
        String beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
            sb.append("</br>").append(beanName);
        }
        sb.append("</body></html>");
        return sb.toString();
    }
}

// Application.java
@SpringBootApplication
@ComponentScan(basePackages = {"com.ps.web", "com.ps.start"})
public class Application {

    public static void main(String... args) {
        SpringApplication.run(Application.class, args);
        System.out.println("Started ...");
    }
}
```

If the application is made only from what was listed up to this point, when running the main method and accessing `http://localhost:8080`, the list of all the beans in the application context will be depicted as in Figure 6-19.

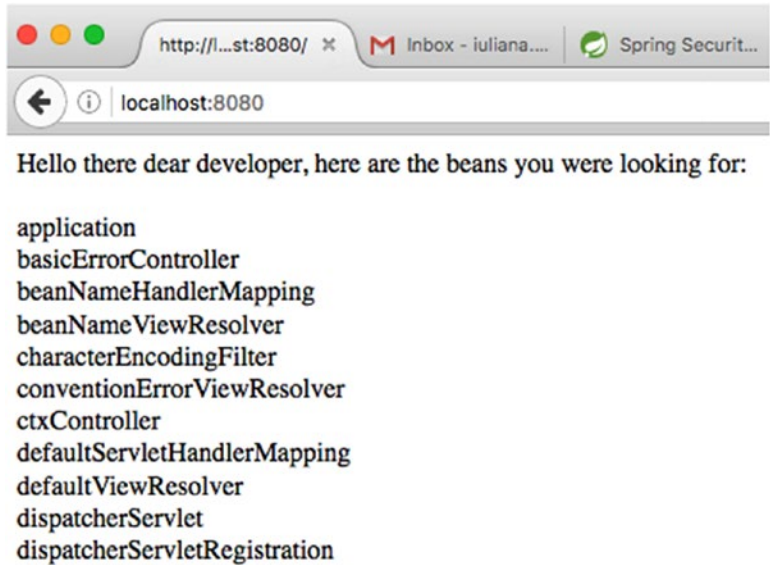


Figure 6-19. Infrastructure beans of an application context created by Spring Boot

The application will run on an embedded Tomcat container, which can be customized easily to declare port and context. There are multiple ways, but the easiest way that can also be externalized is to create a file named `application.properties` under `src/main/resources` and create a special bean implementing `org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer` to load those values and inject them in the embedded server configuration. The structure of the `11-ps-boot-sample` project is depicted in Figure 6-20.

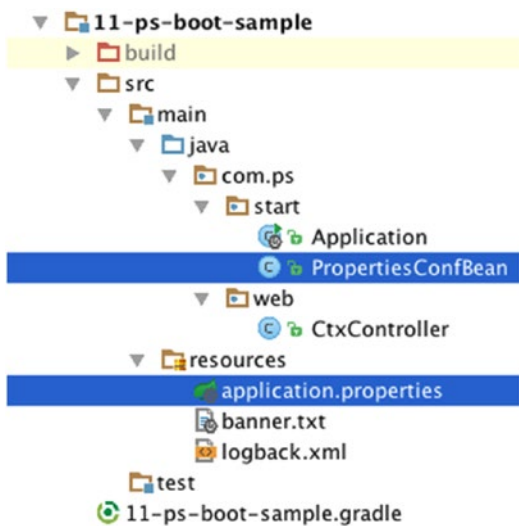


Figure 6-20. `11-ps-boot-sample`: Simple Spring Boot project structure

The contents of the `application.properties` are as follows:

```
app.port=8080
app.context=/ps-boot
```

The names of the properties used here are not reserved or fixed by Spring Boot. They can have any names, and their values will be injected in the bean that uses them in the `@Value` annotation.¹⁸

They are loaded by the following bean:

```
package com.ps.start;

import org.springframework.boot.context.embedded.ConfigurableEmbeddedServletContainer;
import org.springframework.boot.context.embedded.EmbeddedServletContainerCustomizer;
...
@Component
public class PropertiesConfBean implements EmbeddedServletContainerCustomizer {

    @Value("${app.port}")
    private Integer value;

    @Value("${app.context}")
    private String contextPath;

    @Override
    public void customize(ConfigurableEmbeddedServletContainer container) {
        container.setPort(value);
        container.setContextPath(contextPath);
    }
}
```

The `application.properties` can be placed in the following locations:

1. `/config` subdirectory of the current directory.
2. the current directory.
3. classpath `config` package.
4. the classpath root.

¹⁸There is a long list of Spring Boot common application properties, and you can inspect it in the Spring Boot Reference documentation: <http://docs.spring.io/spring-boot/docs/1.4.0.RELEASE/reference/htmlsingle/#common-application-properties>.

This list is ordered by precedence, and properties defined in upper positions override those in lower positions. The `application.properties` file can be provided when the application is executed from the command line using the `spring.config.location` argument:

```
java -jar ps-boot.jar --spring.config.location=
    /Users/iuliana.cosmina/temp/application.properties
```

The file name can be changed too, by using the `spring.config.name` in the command line:

```
java -jar ps-boot.jar --spring.config.name=my-boot.properties
```

With this configuration, the application can be accessed at `http://localhost:8080/ps-boot/`. The configuration above will work when embedded Jetty, Tomcat, or Undertow is used. If something specialized is needed—let's assume we are using Tomcat—a `TomcatEmbeddedServletContainerFactory` bean can be used. And because the `SpringBootApplication` also includes scanning capabilities, the whole application can be written in the `Application` class by annotating it with the `@RestController` annotation.

```
import org.springframework.boot.context.embedded.EmbeddedServletContainerFactory;
import org.springframework.boot.context.embedded
    .tomcat.TomcatEmbeddedServletContainerFactory;
...
@RestController
@SpringBootApplication(scanBasePackages = {"com.ps.start"})
public class Application {

    public static void main(String... args) {
        SpringApplication.run(Application.class, args);
        System.out.println("Started ...");
    }

    @Value("${app.port}")
    private Integer value;

    @Value("${app.context}")
    private String contextPath;

    @Bean
    public EmbeddedServletContainerFactory servletContainer() {
        TomcatEmbeddedServletContainerFactory factory =
            new TomcatEmbeddedServletContainerFactory();
        factory.setPort(value);
        factory.setSessionTimeout(10, TimeUnit.MINUTES);
        factory.setContextPath(contextPath);
        return factory;
    }
}
```

```

@Autowired
ApplicationContext ctx;

@RequestMapping("/")
public String index() {
    StringBuilder sb = new StringBuilder("<html><body>");

    sb.append("Hello there dear developer,
             here are the beans you were looking for: </br>");

    String beanNames = ctx.getBeanDefinitionNames();
    Arrays.sort(beanNames);
    for (String beanName : beanNames) {
        sb.append("</br>").append(beanName);
    }
    sb.append("</body></html>");
    return sb.toString();
}
}

```

Also, when a Spring Boot application starts, the Spring Boot banner is depicted in the console. This can be replaced by creating a file named `banner.txt` under `src/main/resources` containing the desired banner in ASCII format. The original Spring Boot banner and the Apress banner are depicted side by side in Figure 6-21.¹⁹



Figure 6-21. *Spring Boot and Apress banner*

There are quite a few ways to run this application:

1. In IntelliJ IDEA, right click on the Application class and select Run `Application.main()` from the menu that appears; basically, execute the class like any other Java class containing a main method.
2. If the jar plugin is configured in Gradle (and in our project it is so configured by default for all subprojects), after executing `gradle clean build`, the executable `ps-boot-1.0.jar` artifact is built and saved under `11-ps-boot-sample/build/libs`. The artifact can be executed with `java -jar ps-boot-1.0.jar`. The jar plugin can be configured in the module Gradle configuration file, and a different name can be set for the artifact.

¹⁹You can create your own ASCII banner using this site: <http://patorjk.com/software/taag/#p=display&f=Graffiti&t=Type%20Something%20>

```
//11-ps-boot-sample/11-ps-boot-sample.gradle
jar {
    baseName = 'ps-boot'
}
```

When this configuration snippet is present in the Gradle configuration file for a module, executing `gradle clean build` will create an executable artifact named `ps-boot.jar` and save it under `11-ps-boot-sample/build/libs`.

3. If the boot plugin is configured in Gradle, the application can be run by calling the boot Gradle task `gradle bootRun`.
4. If the war plugin is configured in Gradle, after executing `gradle clean war`, the resulting artifact will be a web archive named `ps-boot-1.0.war`, which can also be run with `java -jar ps-boot-1.0.war`. The war plugin can be configured in the module Gradle configuration file, and a different name can be set for the web artifact.

```
//11-ps-boot-sample/11-ps-boot-sample.gradle
apply plugin: 'war'
...
war {
    baseName = 'ps-boot'
}
```

When this configuration snippet is present in the Gradle configuration file for a module, executing `gradle clean war` will create an executable artifact named `ps-boot.war` and save it under `11-ps-boot-sample/build/libs`.

5. If the war plugin is configured in Gradle and we want to produce a deployable web archive that can be deployed on any application server, the embedded container dependencies need to be marked as provided. Since the project does not contain a `web.xml` file, it is mandatory to define a class extending `SpringBootServletInitializer` and override its `configure` method. The `SpringBootServletInitializer` class is a Boot-specific class implementing `WebApplicationInitializer`, which is needed to run a Spring application from a traditional WAR deployment. The developer must override the `configure()` method and provide the class needed to configure the Spring application, whence the class annotated with `@SpringBootApplication`. This will use Spring Framework's Servlet 3.0 support, and the application will be configured when it is run by the servlet container. The preferred practice is to make the class annotated with `@SpringBootApplication` extend this class.

```
//Application.java
import org.springframework.boot.web.support.SpringBootServletInitializer;
import org.springframework.boot.builder.SpringApplicationBuilder;
...
@RestController
@SpringBootApplication(scanBasePackages = {"com.ps.start"})
public class Application extends SpringBootServletInitializer {
```

```

@Override
protected SpringApplicationBuilder configure
    (SpringApplicationBuilder application) {
    return application.sources(Application.class);
}

public static void main(String... args) {
    SpringApplication.run(Application.class, args);
    System.out.println("Started ...");
}
...//etc
}

```

The web archive resulting in this case when `gradle clean war` is run must be deployed on an application server, and when the server starts, the application will be accessible at `http://localhost:8080/ps-boot/`.

When you run the application "in place" using the Gradle task `bootRun`, it would be useful to have the static resources (those under `/src/main/resources`) reloaded while the application is running, so development can take place without restarting the application. This can be done by customizing the `bootRun` task to use the effective contents of the `resources` directory instead of the compiled, processed ones.

```

//11-ps-boot-sample/11-ps-boot-sample.gradle
bootRun{
    addResources = true
}

```

For Spring Boot to be able to support this, another Boot-specific library must be added to the classpath: `spring-boot-devtools`. In our project setup, we need to add the library to the parent configuration file under `pet-sitter/build.gradle`:

```

//pet-sitter/build.gradle
allprojects {
    boot = [
        ... // other starter libs declarations
        starterWeb      :
            org.springframework.boot:spring-boot-starter-web:$springBootVersion",
        devtools        :
            "org.springframework.boot:spring-boot-devtools:$springBootVersion"
    ]
}

... // other configuration elements
}

```


Then the module configuration file `11-ps-boot-sample/11-ps-boot-sample.gradle` must be updated to add it as a compile dependency:

```
//11-ps-boot-sample/11-ps-boot-sample.gradle
apply plugin: 'spring-boot'

dependencies {
    compile boot.starterWeb, boot.devtools, misc.jstl, misc.servlet
    testCompile boot.starterTest
}

war {
    baseName = 'ps-boot'
}
... // other configuration elements
```

Configuration Using YAML

YAML is a superset of JSON and has a very convenient syntax for storing external properties in a hierarchical format. The `application.properties` contains only two properties, but for big production applications, that file can become quite complex, so it is useful to be able to group properties by their purposes and create hierarchies. An example of such a file:

```
spring:
  app:
    name: ps-boot
  datasource:
    driverClassName: org.h2.Driver

    url: jdbc:h2:sample;DB_CLOSE_ON_EXIT=TRUE
    username: sample
    password: sample
  Server:
    port: 9000
    context: /ps-boot
```

To use YAML, the `application.properties` must be replaced with `application.yml` file (the possible locations for this file are the same as for `application.properties`) and the `sneakyaml` dependency added to the project. A YAML file is transformed into a Java `Map<String, Object>`, and Spring Boot flattens the map so that it is one level deep and has period-separated keys. So the `application.properties` that corresponds to the `application.yml` depicted previously looks like this:

```
spring.app.name: ps-boot

spring.datasource.driverClassName: org.h2.Driver
spring.datasource.url: jdbc:h2:sample;DB_CLOSE_ON_EXIT=TRUE
spring.datasource.username: sample
spring.datasource.password: sample

server.port: 9000
server.context: /ps-boot
```

To configure the previous application using YAML, we do not have to add a dependency of `sakeyaml`, because it is a transitive dependency of the Spring Boot Web Starter. So the only thing to do is the following:

1. Convert `application.properties` to `application.yml`

```
app:
  port: 8084
  context: /ps-boot
  sessionTimeout:10
```

2. Create a class to load and hold the YAML values, and to make things interesting, the class should be able to work with external configuration files. This can be done using the Spring Boot specialized annotation `@ConfigurationProperties`.²⁰ This annotation also provides the possibility to validate the external properties and define a prefix for the properties. So if there are multiple prefixes, multiple classes can be defined. The name prefix is used by Spring Boot to identify the properties that are valid to bind to a configuration object and is defined as a value for the prefix attribute of the `@ConfigurationProperties` annotation.

```
import org.springframework.boot.context.properties.ConfigurationProperties;

import javax.annotation.PostConstruct;
import javax.validation.constraints.NotNull;

@ConfigurationProperties(prefix="app")
public class AppSettings {

    private static Logger logger = LoggerFactory.getLogger(AppSettings.class);

    @NotNull
    private Integer port;

    @NotNull
    private String context;

    @NotNull
    private Integer sessionTimeout;

    public AppSettings() {
    }

    @PostConstruct
    public void check() {
        logger.info("Initialized {} {}", port, context);
    }

    ...//getters & setters
}
```

²⁰The `@ConfigurationProperties` annotation can be used with `*.properties` files too, but it makes no sense to mention it twice.

■ ! Although setters and getters are not part of the previous code snippet, they are mandatory, because Spring Boot uses setters to populate the `AppSettings` bean properties, and getters are used to access the values so they can be used.

3. Enable support for beans annotated with `@ConfigurationProperties` by annotating the Configuration class with `@EnableConfigurationProperties` and specify the configuration bean class s argument for it.

```
import org.springframework.boot.context.properties.EnableConfigurationProperties;
...
@RestController
@SpringBootApplication(scanBasePackages = {"com.ps.start"})
@EnableConfigurationProperties(AppSettings.class)
public class Application extends SpringBootServletInitializer {
    ...
}
```

If this annotation is not used, then the `AppSettings` class can be annotated with `@Configuration`, and Spring Boot will properly create the bean and load the values from the file. This approach is practical when more than one configuration class is used.

4. Use the property values by injecting them where needed. In this case, the values need to be injected into a class implementing `EmbeddedServletContainerFactory`.

```
@RestController
@SpringBootApplication(scanBasePackages = {"com.ps.start"})
@EnableConfigurationProperties(AppSettings.class)
public class Application extends SpringBootServletInitializer {
    ...
    @Autowired
    private AppSettings appSettings;

    @Bean
    public EmbeddedServletContainerFactory servletContainer() {
        TomcatEmbeddedServletContainerFactory
            factory = new TomcatEmbeddedServletContainerFactory();
        factory.setPort(appSettings.getPort());
        factory.setSessionTimeout(
            appSettings.getSessionTimeout(), TimeUnit.MINUTES);
        factory.setContextPath(appSettings.getContext());
        return factory;
    }
    ....
}
```

■ ! Just keep in mind that YAML files can not be loaded via the `@PropertySource` annotation. This annotation is specific to properties files.

Logging

Spring Boot uses Commons Logging internally by default, but it leaves the underlying implementation open. Log4j2, Java Util, and Logback are supported. The application attached to this book uses Logback. The starters use Logback by default, and it is preconfigured to use the console as output, but it can be configured via the `logback.xml` file, which is located under `/src/main/resources`. By default, ERROR, WARN, and INFO level messages are logged. To modify this behavior and enable writing DEBUG messages for a category of core loggers (embedded container, Hibernate, and Spring Boot), the `application.properties` file must be edited, and this property must be added: `debug=true`.

Logging is initialized before the application context, so it is impossible to control logging from using `@PropertySources` in `@Configuration` classes. System properties and conventional Spring Boot external configuration files should be used. Depending on the logging system that is used, Spring Boot will look for the specific configuration files, in the order stated below:

- `logback-spring.xml`, `logback-spring.groovy`, `logback.xml`, `logback.groovy` for Log-back
- `log4j2-spring.xml`, `log4j2.xml` for Log4j2
- `logging.properties` for Java Util Logging

The logfile name to use by default by Spring Boot can be configured using the `logging.file` Spring Environment variable. There are more Spring Environment variables that can be used to configure Spring Boot logging, and the full list and purposes are available in the Spring Boot official reference documentation.²¹ Using filenames postfixes with `-spring` is recommended, since when standard configuration locations are used, Spring cannot completely control log initialization.

Testing with Spring Boot

The most practical way to test a Spring component or bean is not to involve Spring at all. If the class has a constructor, throw in some JUnit asserts and you are done. With Spring 4.3 it is really easy to test, since the need to use `@Autowired` has been removed. In a test environment, as long as there is one constructor, Spring will implicitly consider it an autowire target. In a more complex application in which integration tests are needed, testing can get complicated. That is why Spring Boot provides a helper annotation to configure a test environment.

Testing Spring web applications means testing that controllers work as expected, that they return the proper results, and that they use the proper views. This can be done outside the application server, but in case of integration tests, an embedded server would be useful to test them in a web environment. Spring

²¹Spring Environment logging variables: <http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-custom-log-configuration>

Boot provides the possibility to do just that by providing a specialized annotation `@SpringBootTest`. This annotation should be used on a test class that runs Spring Boot-based tests. It provides the following:

- If no `ContextLoader` is specified with `@ContextConfiguration`, it uses `org.springframework.boot.test.context.SpringBootTestContextLoader` by default.
- Automated search for a Spring Boot configuration when nested `@Configuration` classes are used.
- Loading environment-specific properties via the `properties` attribute. This attribute allows for specification of properties (key=value pairs) as values for the attribute.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    properties = {"app.port=9090"})
public class CtxControllerTest {
    ...
}
```

- Defining different web environment modes and starting a fully running container on a random port, using the `webEnvironment` attribute.
- Registering a `org.springframework.boot.test.web.client.TestRestTemplate` bean for use in web tests that use a fully running container.

Thus with the following controller defined,

```
@Controller
public class CtxController {

    public static final String INTRO = "Hello there dear developer,
        here are the beans you were looking for: </br>";

    @Autowired
    ApplicationContext ctx;

    @RequestMapping("/")
    @ResponseBody
    public String index() {
        StringBuilder sb = new StringBuilder("<html><body>");
        sb.append(INTRO);
        String beanNames = ctx.getBeanDefinitionNames();
        Arrays.sort(beanNames);
        for (String beanName : beanNames) {
            sb.append("</br>").append(beanName);
        }
        sb.append("</body></html>");
        return sb.toString();
    }
}
```

```

@RequestMapping("/home")
public String home(ModelMap model) {
    model.put("bogus", "data");
    return "home";
}
}

```

it can be easily tested with Spring Boot:

```

import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.ui.ModelMap;
...
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment =
    SpringBootTest.WebEnvironment.RANDOM_PORT)
public class CtxControllerTest {

    @Autowired
    CtxController ctxController;

    private ModelMap model = new ModelMap();

    @Test
    public void textIndex() {
        String result = ctxController.index();
        assertNotNull(result);
        assertTrue(result.contains(CtxController.INTRO));
    }

    @Test
    public void textHome() {
        String result = ctxController.home(model);
        assertEquals("home", result);

        String modelData = (String)model.get("bogus");
        assertEquals("data", modelData);
    }
}
}

```

The `SpringRunner` is an alias for `SpringJUnit4ClassRunner` introduced in Spring 4.3. More information on how testing was improved with Spring Boot 1.4 and Spring 4.3 is covered on the Spring official blog.²²

Spring Boot is quite a big topic, and it has two chapters in the official certification exam. But it is not covered fully in the official course. The complete documentation that you should read to be able to harness its power fully is the official reference documentation available publicly here: <http://docs.spring.io/spring-boot/docs/current/reference/>. Spring Boot is a great tool for speeding up development, and today, more and more companies that develop Spring applications are using it.

Summary

Below, a list of core concepts and important details related to Spring Web MVC, Spring Security, and Spring Boot has been compiled for you:

- Spring Web MVC was built to respect the MVC design pattern.
- The entry point in a Spring Web application is the `DispatcherServlet`, which is a front controller for the application.
- The `DispatcherServlet` assigns HTTP requests to method calls from classes called controllers using a collection of web-specific infrastructure beans.
- To create a Spring Web MVC application, the `DispatcherServlet` must be configured as the entry point of the application, and the application configuration must contain infrastructure MVC beans and custom controllers and view beans.
- The `@EnableWebMvc` annotation is used on a configuration class for a Spring Web MVC application.
- The MVC configuration class must implement `WebMvcConfigurer` or extend a class implementing this interface to customize the imported configuration.
- The `@MVC` model and an application server supporting Servlet 3.0+ allow for an application without any XML configuration.
- Spring Security allows for Authentication to be fully decoupled from Authorization.
- URLs can be protected using security rules, JSP elements can be protected using Security taglibs, and method access can be restricted using `@Secured` or `@RolesAllowed`.
- Method security is implemented using AOP using Spring Security-specific and JSR 250 annotations: `@Secured`, `@RolesAllowed`, `@PreAuthorize`, etc.
- Encrypting and salting passwords is supported.
- `@EnableWebSecurity` is the annotation used on a configuration class to have Spring Security enabled.
- This class must also implement `WebSecurityConfigurer` or extend a class implementing it to customize the imported configuration.
- Spring Boot is a set of preconfigured frameworks/technologies designed to reduce boilerplate configuration and provide a quick way to have a Spring application up and running.

²²Testing improvements in Spring Boot 1.4 <https://spring.io/blog/2016/04/15/testing-improvements-in-spring-boot-1-4>.

- Spring Boot default configurations can be easily customized.
- Spring Boot provides starter dependencies for a multitude of Spring applications.
- Spring Boot comes with a wide set of embedded servers, so developers do not have to download, install, and configure them in a development environment.
- Spring Boot does not generate code; it just dynamically wires up beans and settings and applies them to the application context when the application is started.
- Artifacts built with Spring Boot, whether jars or wars, are executable and can be executed with `java -jar`.

Quiz

Question 1: Which of the following statements regarding `DispatcherServlet` is true?

- It is used to enable web support for Spring applications.
- Intercepts all HTTP requests and delegates them to the appropriate handlers.
- It must be declared as a bean in the root context of the application.
- `DispatcherServlet` cannot be configured without the `web.xml` file.
- `DispatcherServlet` is the main servlet class of a Spring Web application.

Question 2: What does MVC stand for?

- Model View Controller
- Mapping View Controller
- Module View Control

Question 3: What annotation is used to map HTTP Requests to handler methods?

- `@HandlerMapping`
- `@RequestMapping`
- `@Mapping`

Question 4: Considering the following handler method definition:

```
@RequestMapping(value = "/showUser", method = RequestMethod.GET)
public String show(@RequestParam("userId") Long id, Model model) {
    ...
}
```

Which of the following requests are mapped to it?

- `http://localhost:8080/mvc-basic/showUser?userid=105`
- `http://localhost:8080/mvc-basic/showUser?userId=105`
- `http://localhost:8080/mvc-basic/showUser?id=105`
- `http://localhost:8080/mvc-basic/showUser?userId=2c`

Question 5: Which class in the following list is the default view resolver in Spring:

- A. JspResourceViewResolver
- B. ResourceViewResolver
- C. InternalResourceViewResolver

Question 6: What is authentication?

- A. the process of securing resources
- B. the process of making a decision whether a user should be allowed to access a resource
- C. the process of verifying the validity of the principal's credentials

Question 7: What is authorization?

- A. the process of verifying the validity of the principal's credentials
- B. the process of making a decision whether an authenticated user is allowed to perform a certain action within the application
- C. the process of generating credentials for a user

Question 8: What is a principal?

- A. an object storing the credentials for a user
- B. the term that signifies a user, device, or system that could perform an action within the application
- C. a term that signifies a secured resource

Question 9: What can be said about application security?

- A. is unnecessary
- B. should be provided by third party frameworks
- C. is a cross-cutting concern

Question 10: Consider the following XML configuration snippet:

```
<web-app ...>
<!-- The root web application context is the security context-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/spring/security-config.xml
    </param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
```

```

<filter>
  <filter-name>#####</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>#####</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

What should replace the '#####' pattern for the configuration above to be valid in a Spring Secure application?

- A. springSecurityFilterChain
- B. securityFilterChain
- C. securityHandlerChain

Question 11: Consider the following Java Configuration class snippet:

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    ...
  }
}

```

What is missing from the code above that will enable security at URL and at method level?

- A. @EnableWebSecurity
- B. @EnableWebMvc
- C. @EnableWebSecurity + @EnableWebMvc
- D. @EnableWebSecurity + @EnableGlobalMethodSecurity(jsr250Enabled = true)
- E. @EnableWebSecurity + @EnableGlobalMethodSecurity(secured = true)

Question 12: Which of the following Spring annotations taken together are the equivalent of the @SpringBootApplication?

- A. @Component
- B. @Configuration
- C. @ComponentScan
- D. @EnableAutoConfiguration
- E. @Controller

CHAPTER 7



Spring Advanced Topics

In the previous chapters, the objects handled by Spring were created on a JVM. These objects were accessed and manipulated indirectly by the user using the HTTP protocol. In this case, all that the end user needs to access the objects in the JVM is the web interface of the application.

Remoting and **Web Services** are ways of communicating between applications. The applications can run on the same computer, on different computers, on different networks, and can even be written in different languages (a Python application can consume a web service provided by a Java application, for example). In remoting, the applications communicating know about each other. There is a server application and a client application. Because of this, it supports state management options and can correlate multiple calls from the same client and support callbacks. On the client application, a proxy of the server target object is created and used to access the object on the server. Remoting can be used across any protocol, but it does not do well with firewalls. Remoting relies on the existence of common language runtime assemblies that contain information about data types. This limits the information that must be passed about an object and allows objects to be passed by value or by reference. The communication is done using a binary, XML, or JSON format. All these limitations make remoting quite deprecated at the moment this book is being written. Web services have completely replaced them.

Web Services constitute a cross-platform interprocess communication method using common standards and able to work through firewalls. They work with messages, not objects. So the client basically sends a message, and a reply is returned. Web services work in a stateless environment whereby each message results in a new object created to service the request. Web services support interoperability across platforms and are good for heterogeneous environments. They expose their own arbitrary sets of operations such as via WSDL (Web Services Description Language) and SOAP (Simple Object Access Protocol).

REST, or representational state transfer, also called RESTful web services, is currently the most popular way applications communicate with each other. REST services allow access and manipulation of textual representations of web resources using a uniform and predefined set of stateless operations. The most common protocol used with REST services is HTTP, so the HTTP methods map on REST operations such as GET, POST, PUT, DELETE. Initially, web resources were documents or files accessed using a URL (Uniform Resource Locator, also known as a web address), but recently, a web resource became able to be anything (object, entity) that can be accessed via the web and is identified by a URI (Uniform Resource Identifier).

The easiest way in which two heterogeneous systems can communicate is using a layer called **middleware**. Using this layer, software components like applications, enterprise Java beans, servlets, and other components that have been developed independently and that run on different networked platforms can interact with one another. There are three types of middleware:

- Remote Procedure Call, or RPC, which allows one application to call procedures from another application remotely as if they were local calls.
- Object Request Broker, or ORB-based, which enables an application's objects to be distributed and shared across heterogeneous networks. (Remoting falls in this category.)
- Message Oriented Middleware or MOM-based middleware, which allows distributed applications to communicate and exchange data by sending and receiving messages.

Spring provides support for the JMS (Java Messaging Service) API, which is an abstraction written in Java for accessing MOM middleware. Messaging is an implementation to handle the producer-consumer problem. An application produces messages, and a client application consumes them asynchronously. The application sending the messages has no knowledge of its client (or clients). Messaging is a form of loosely coupled distributed communication. The main advantages of using messaging for communication are the relaxed coupling between the producer and the consumer and the ability to integrate heterogeneous platforms, reduce system bottlenecks, increase scalability, and respond more quickly to change.

JMX (Java Management Extensions) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks. Those resources are represented by objects called **MBeans** (Managed Bean). The JMX support in Spring provides features to integrate a Spring application into a JMX infrastructure easily and transparently.

All of these topics will be superficially covered in this chapter, except for REST, which has its own chapter in the official documentation for the core certification exam.

Spring Remoting

Before this chapter, the topics presented were exemplified technically through applications executed on a single JVM. Service beans were calling repository beans methods, and tests classes were executed to check the correct communication between them. Figure 7-1 depicts the situation covered so far.

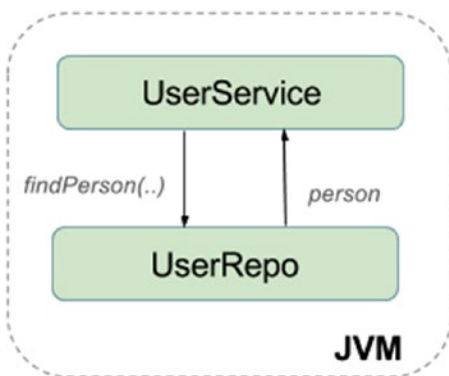


Figure 7-1. Client and Server applications run in the same JVM context

With a traditional remoting client, a `ClientService` class should be introduced to access remote objects that will exist in an application being run on a different JVM. Figure 7-2 abstractly depicts the situation covered in this chapter. The remote object is represented in this figure by the `RemoteUserService` class.

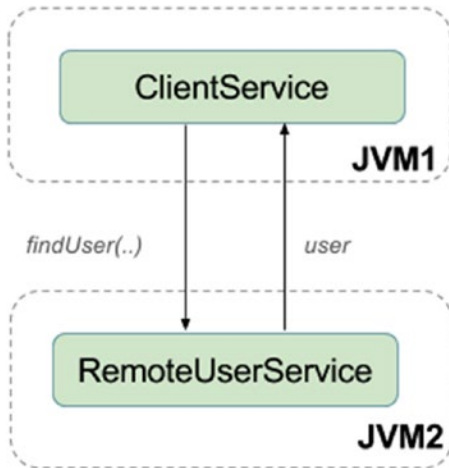


Figure 7-2. Client and Server applications run in the different JVM context

Java Remote Invocation allows for an object running in one JVM to invoke methods on an object running in another JVM. RMI (Remote Method Invocation) is Java's version of RPC (Remote Procedure Call). RMI applications often consist of two processes: a server and a client. The server creates the objects that will be accessed remotely, and it will expose a *skeleton* (the interface of the remote object). The client invokes methods on a *stub* (proxy object). The RMI model abstract functional schema is depicted in Figure 7-3.

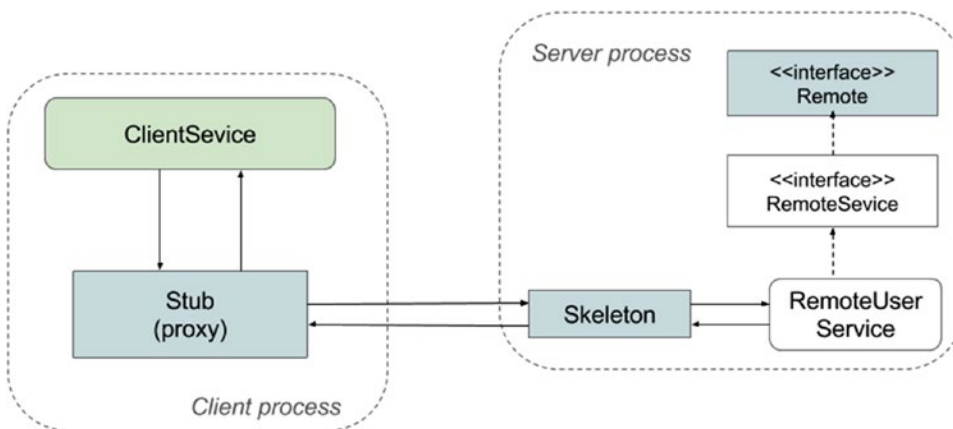


Figure 7-3. The RMI model abstract functional schema

The RMI model has a few disadvantages:

- The client and server implementations are coupled to the RMI framework.
 - The server interface must extend `java.rmi.Remote`.
 - The client must catch `java.rmi.RemoteException`, because it is a checked exception.¹
- RMI stub and skeleton have to be generated with the **rmic** compiler, which can be found in the `JAVA_HOME\bin` directory.²
- And extra code must be written to handle binding and retrieving objects from the remote server.

To pass objects between server and client, Java *marshaling* is used. Objects transferred using RMI must therefore be serializable, and they must implement `java.io.Serializable`. RMI uses the Java Remote Method Protocol (JRMP) for remote Java object communication. In the following remote example, the entity objects defined in `00-ps-core` are used, and they were defined as serializable so they could be handled by Spring JPA.

```
// User.java
import javax.persistence.*;
...
@Entity
@Table(name="P_USER")
public class User extends AbstractEntity {
    ...
}

// AbstractEntity.java
import java.io.Serializable;

@MappedSuperclass
public abstract class AbstractEntity
    implements Serializable {
    ...
}
```

Spring provides classes for writing remote applications that are designed to hide the “plumbing” details. Using class `org.springframework.remoting.rmi.RmiServiceExporter`, the interface `UserService` implemented by class `UserServiceImpl`, the service bean defined in `09-ps-data-jpa` can be exposed as an RMI object. This class can be used to bind to a registry or expose an endpoint. Using the class `org.springframework.remoting.rmi.RmiProxyFactoryBean`, the `UserService` interface can be accessed via proxies that are created to communicate with the server-side endpoint and to convert the remoting-specific exceptions to a runtime hierarchy, all extending the core `RemoteAccessException`. These two classes are basically utility classes to help developers write remote applications quickly and easily, because the code written by the developer is greatly reduced, by using configuration instead.³ The abstract schema of client and server applications remoting using Spring is depicted in Figure 7-4.

¹Oracle JavaDoc API: <http://docs.oracle.com/javase/8/docs/api/java/rmi/RemoteException.html>.

²<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/rmic.html>.

³This is an example of how RMI applications can be written without Spring: https://en.wikipedia.org/wiki/Java_remote_method_invocation.

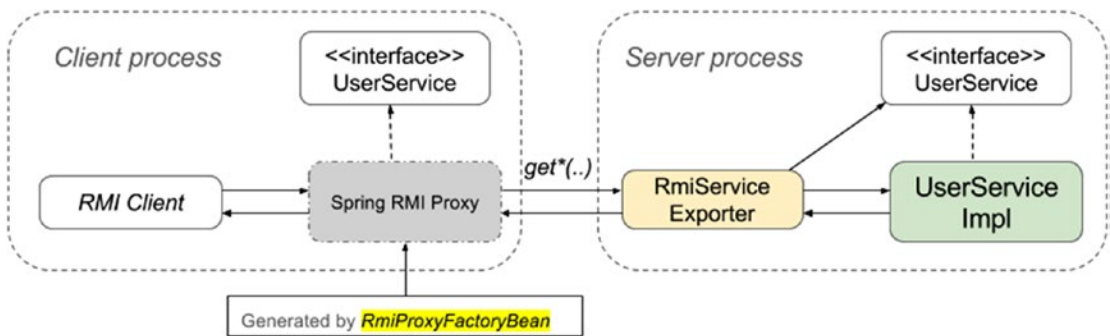


Figure 7-4. Spring remoting schema depicting the roles of *RmiServiceExporter* and *RmiProxyFactoryBean*

Spring Remote Configuration

Configuring the remote server and client can be done using XML or Java Configuration. The service that will be exposed remotely is a service bean defined in the 09-ps-data-jpa project that uses a Spring Repository to create and query users. In order to set up client and server remote applications using Spring that communicate over the JRMP protocol, the following steps have to be executed:

1. Configure a bean extending `org.springframework.remoting.rmi.RmiServiceExporter`. Using XML:

```

<!-- rmi-server-config.xml -->
<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 class="org.springframework.remoting.rmi.RmiServiceExporter"
    p:registryPort="1099"
    p:alwaysCreateRegistry="true"
    p:serviceName="userService"
    p:serviceInterface="com.ps.services.UserService"
    p:service-ref="userServiceImpl"/>

</beans>
  
```

Using Java Configuration:

```

//RmiServerConfig.java
import org.springframework.remoting.rmi.RmiServiceExporter;
...
@Configuration
public class RmiServerConfig {

    @Autowired
    @Qualifier("userServiceImpl")
    UserService userService;
  }
  
```

```

@Bean
public RmiServiceExporter userService() {
    RmiServiceExporter exporter = new RmiServiceExporter();
    exporter.setRegistryPort(1099);
    exporter.setAlwaysCreateRegistry(true);
    exporter.setServiceName("userService");
    exporter.setServiceInterface(UserService.class);
    exporter.setService(userService);
    return exporter;
}
}

```

Both of the beans defined previously have the following properties set:

- `registryPort` – the port of the registry for the exported RMI service, has the default value of 1099, so this property is not required to be set, and it is used in the example just for pedagogical purposes.
 - `alwaysCreateRegistry` – the default value for this property is `false`. When a client requests an RMI service, an existing registry is located, and if not found, it will be created. If this property is set to `true`, the registry will be created on client request, to avoid the overhead of locating an existing registry without any client needing it. Since this property has a default value of `false`, it is not required to be configured explicitly by the developer of the application.
 - `serviceName` – the exported RMI service is accessible by default to a location identified by the following template: `rmi://host:port/serviceName`. The service name is set by this property and must be set explicitly by the developer of the application.
 - `serviceInterface` – this property is used to set the interface of the bean type that will be exposed as an RMI service; it must be set explicitly by the developer of the application.
 - `service` – this property is used to set the bean that will be exposed as an RMI service; it must be set explicitly by the developer of the application.
2. Create a server application that will contain the remote exported bean and the application beans that need to be accessed remotely.

Using XML, the configuration looks like the following snippet.

```

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class RmiExporterBootstrap {

    public static void main(String args) throws Exception {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext(
                "spring/rmi-server-config.xml",
                "spring/app-config.xml");
    }
}

```



```

        System.out.println("RMI server started.");
        System.in.read();
        ctx.close();
    }
}

```

The `app-config.xml` contains all the service and repository beans needed to be executed on the remote server.

```

<beans ...">
  <!-- import service configurations -->
  <bean class="com.ps.config.ServiceConfig"/>
  <context:annotation-config/>
</beans>

```

Using Java Configuration, the context declared in the class `RmiExporterBootstrap` will have to be defined as in the following sample:

```

import com.ps.config.ServiceConfig;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class RmiExporterBootstrap {

    public static void main(String args) throws Exception {

        AnnotationConfigApplicationContext ctx =
            new AnnotationConfigApplicationContext(
                RmiServerConfig.class, ServiceConfig.class);
        System.out.println("RMI reward network server started.");
        System.in.read();
        ctx.close();
    }
}

```

By executing the `RmiExporterBootstrap` class, a Java process will start that will provide an RMI service named `userService` on port 1099. The `System.in.read()`; is used to keep the process running until the user presses a key.

3. Configure a bean of type `org.springframework.remoting.rmi.RmiProxyFactoryBean` that will take care of creating the proxy objects needed on the client side to access the RMI service. Using XML, the configuration looks like the following snippet.

```

<!-- rmi-client-config.xml -->
<beans ...>
  <bean id="userService"
    class="org.springframework.remoting.rmi.RmiProxyFactoryBean"
    p:serviceInterface="com.ps.services.UserService"
    p:serviceUrl="rmi://localhost:1099/userService"/>
</beans>

```

The same configuration using Java Configuration can be written like this:

```
//RmiClientConfig.java
import org.springframework.remoting.rmi.RmiProxyFactoryBean;

@Configuration
public class RmiClientConfig {

    @Bean
    public UserService userService() {
        RmiProxyFactoryBean factoryBean = new RmiProxyFactoryBean();
        factoryBean.setServiceInterface(UserService.class);
        factoryBean.setServiceUrl("rmi://localhost:1099/userService");
        factoryBean.afterPropertiesSet();
        return (UserService) factoryBean.getObject();
    }
}
```

Before creating the proxy bean, the factory bean must be initialized, which is why in the Java Configuration, the method `afterPropertiesSet()` must be called explicitly. The `serviceUrl` property will be set with the URL, where the server will provide the RMI service and the `serviceInterface` property should be set with the Interface that the RMI server bean is implementing. This interface is the one that the proxy created by the factory bean will implement to allow the client to call remote methods on the server `userService` bean. That is why in the last line of the method, the object returned by `factoryBean.getObject()` is cast to `UserService`.

4. Create a client application that will use the client factory bean. The easiest way to do this is to create a test class and inject the client proxy bean that the `RmiProxyFactoryBean` will create:

```
//RmiTests.java
@Configuration(locations = {"classpath:spring/rmi-client-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class RmiTests {

    @Autowired
    private UserService userService;

    public void setUp() {
        assertNotNull(userService);
    }

    @Test
    public void testRmiAll() {
        List<User> users = userService.findAll();
        assertEquals(5, users.size());
    }
}
```

```

@Test
public void testRmiJohn() {
    User user = userService.findByEmail("john.cusack@pet.com");
    assertNotNull(user);
}
}

```

To make sure that the client actually uses a proxy bean, you just have to run one of the test methods in debug, stop the execution at a breakpoint, and take a peek at the `userService` bean. In Figure 7-5 a screenshot from an IntelliJ IDEA test execution is depicted. You can clearly see that the `userService` is actually an RMI proxy created by the `RmiProxyFactoryBean`.

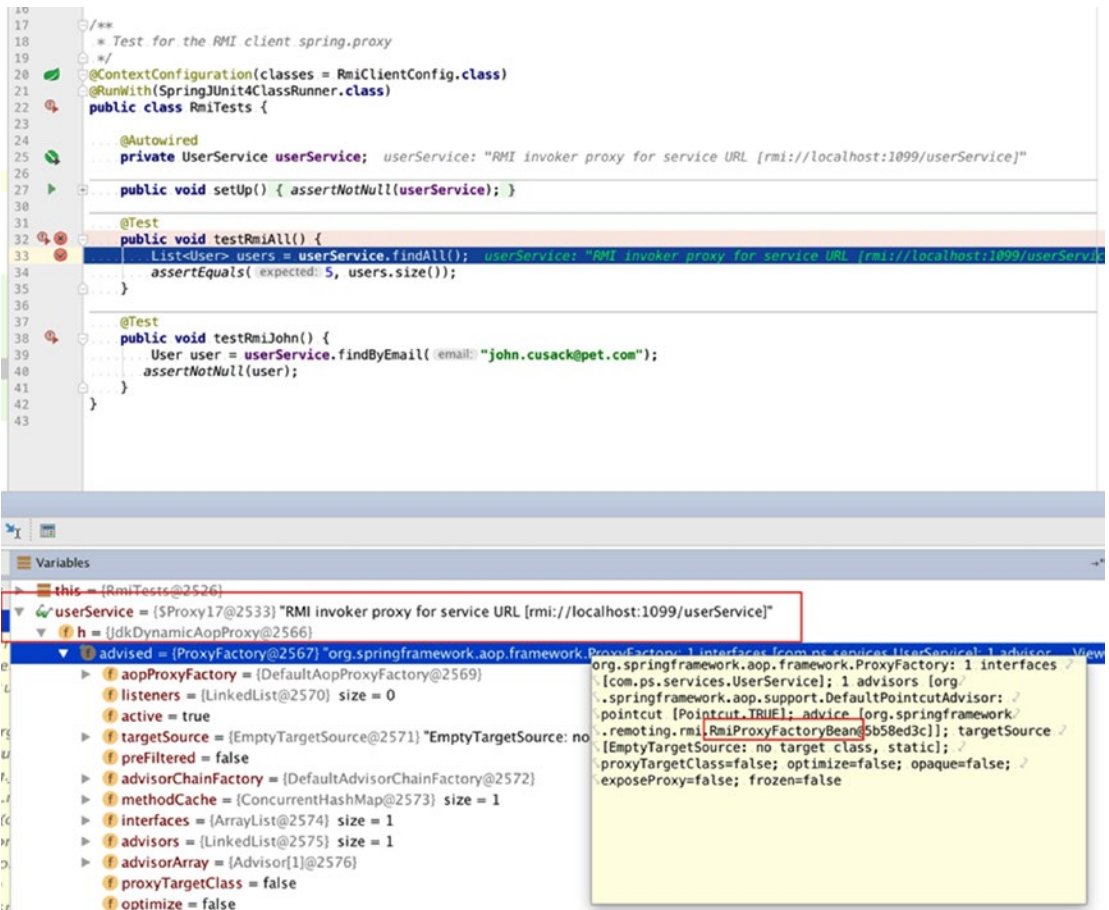


Figure 7-5. Screenshot of an RMI test executed in debug mode

If Java Configuration is used, the `@ContextConfiguration` declaration must be changed to the following:

```
@ContextConfiguration(classes = RmiClientConfig.class)
```

As you can notice when implementing and accessing RMI services, Spring can be used in the same declarative way as presented so far. This means that existing beans can be exposed as RMI services without modifying their code. Thus, the server interface is not required to implement the Remote interface. On the client side, using polymorphism, methods can be called directly on the injected bean, and the Spring remoting exceptions are unchecked (there is a hierarchy of remoting exceptions all extending `org.springframework.remoting.RemoteAccessException`), so the code to catch and handle them does not have to be written.

Spring `RmiServiceExporter` and `RmiProxyFactoryBean` can be used with multiple protocols. Spring provides classes that allow exposing RMI services over HTTP, using a lightweight binary HTTP-based protocol. The classes to use are `HttpInvokerProxyFactoryBean` and `HttpInvokerServiceExporter`. RMI methods will be converted to HTTP methods: GET and POST. The result of these methods is returned as an HTTP result. Method parameters and results are serialized using Java serialization, so transferred objects need to be serializable. Spring's `HttpInvoker` is another Java-to-Java binary remoting protocol; it requires that Spring be used on both the server and the client.

Aside from these two classes used with RMI over HTTP, there are two other protocols that Spring supports: the `Hessian` lightweight binary HTTP-based protocol provided by Caucho and `Burlap`, which is the XML-based alternative to `Hessian`. `Hessian` protocol is a slim binary cross-platform remoting protocol. Most cross-platform protocols are XML-based, and thus sacrifice a significant amount of performance to achieve interoperability. `Hessian` managed to achieve cross-platform interoperability with minimal performance degradation.

Classes for each of the protocols are provided by Spring: `HessianProxyFactoryBean` and the `HessianServiceExporter` for `Hessian` and `BurlapProxyFactoryBean` and `BurlapServiceExporter` for `Burlap`. Basically, to write a Spring RMI application over the HTTP protocol, the RMI specific classes for creating a proxy on the client side and for exposing a service on the server side must be replaced with HTTP Spring-specific classes, and the server must be run in a container, such as Apache Tomcat or Jetty. Figure 7-6 should look familiar; it describes the abstract schema for using the Spring `HttpInvoker` classes.

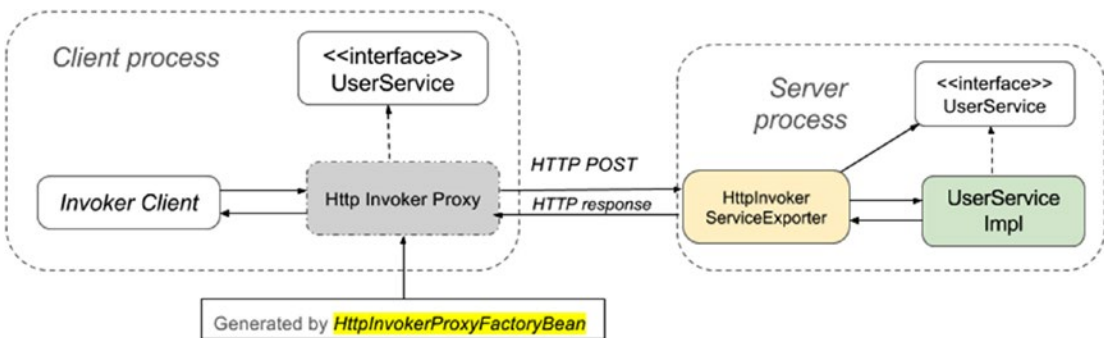


Figure 7-6. Spring remoting abstract schema

Whether the configuration is done in XML or using Java Configuration, on the server side, a bean of type `HttpInvokerServiceExporter` must be configured.

```
<!-- httpinvoker-server-config.xml -->
<beans ...>
  <bean name="/httpInvoker/userService"
        class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter"
        p:service-ref="userServiceImpl"
        p:serviceInterface="com.ps.services.UserService"/>
</beans>

// HttpInvokerConfig.java
@Configuration
public class HttpInvokerConfig {

    @Autowired
    @Qualifier("userServiceImpl")
    UserService userService;

    @Bean(name = "/httpInvoker/userService")
    HttpInvokerServiceExporter httpInvokerServiceExporter(){
        HttpInvokerServiceExporter invokerService = new HttpInvokerServiceExporter();
        invokerService.setService(userService);
        invokerService.setServiceInterface(UserService.class);
        return invokerService;
    }
}
```

The `/httpInvoker/userService` is the URL where the HTTP Service is exposed. On the client side, a bean of type `HttpInvokerProxyFactoryBean` must be configured.

```
<!-- httpinvoker-client-config.xml -->
<beans ...>
  <bean id="userService"
        class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean"
        p:serviceInterface="com.ps.services.UserService"
        p:serviceUrl="http://localhost:8080/invoker/httpInvoker/userService"/>
</beans>

//HttpInvokerClientConfig.java
@Configuration
public class HttpInvokerClientConfig {

    @Bean
    public UserService userService() {
        HttpInvokerProxyFactoryBean factoryBean = new HttpInvokerProxyFactoryBean();
        factoryBean.setServiceInterface(UserService.class);
        factoryBean.setServiceUrl
            ("http://localhost:8080/invoker/httpInvoker/userService");
        factoryBean.afterPropertiesSet();
        return (UserService) factoryBean.getObject();
    }
}
```

The properties for the Spring Http Invoker classes have the same meaning as for the RMI Spring classes; the only difference is that they apply to the HTTP protocol. As you probably noticed, the service URL is not an RMI URL, but an HTTP URL made from the web application URL [http://hostname:port/context] concatenated with the name of the invoker service bean. In the current example, the application context, the URL where the servlet handling HTTP requests is mapped is /invoker, and the invoker bean name is /httpInvoker/userService.

The web application that exposes the invoker service does not need a complex web interface, which is why the DispatcherServlet with a full-blown MVC configuration is not needed. All that is needed is a servlet to intercept requests, and `org.springframework.web.context.support.HttpServletRequestHandlerServlet` is suitable for the job. The configuration for a web application was already covered in **Chapter 6**.

```
<servlet>
  <servlet-name>invoker</servlet-name>
  <servlet-class>
    org.springframework.web.context.support.HttpServletRequestServlet
  </servlet-class>
  <init-param>
    <param-name>contextClass</param-name>
    <param-value>
org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </init-param>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
    com.ps.remoting.config.HttpInvokerConfig
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/*</url-pattern>
</servlet-mapping>
```

To test the service exposed over HTTP, the previous test class can be used. Only the configuration file or class must be changed. In the code snippet below you can see all configurations that can be used on the client test class. Since all the client needs is the interface type of the proxy, where the proxy points is decided by the configuration.

```
//@ContextConfiguration(locations = {"classpath:spring/rmi-client-config.xml"})
//@ContextConfiguration(classes = RmiClientConfig.class)

//@ContextConfiguration(locations = {"classpath:spring/httpinvoker-client-config.xml"})
@ContextConfiguration(classes = HttpInvokerClientConfig.class)
@RunWith(SpringJUnit4ClassRunner.class)
public class RmiTests {
  ...
}
```

To make sure the client actually uses an http invoker proxy bean, you just have to run one of the test methods in debug, stop the execution at a breakpoint, and take a peek at the `userService` bean. In Figure 7-7, a screenshot from an IntelliJ IDEA test execution is depicted. You can clearly see that the `userService` is actually an Http Invoker proxy created by the `HttpInvokerProxyFactoryBean`.

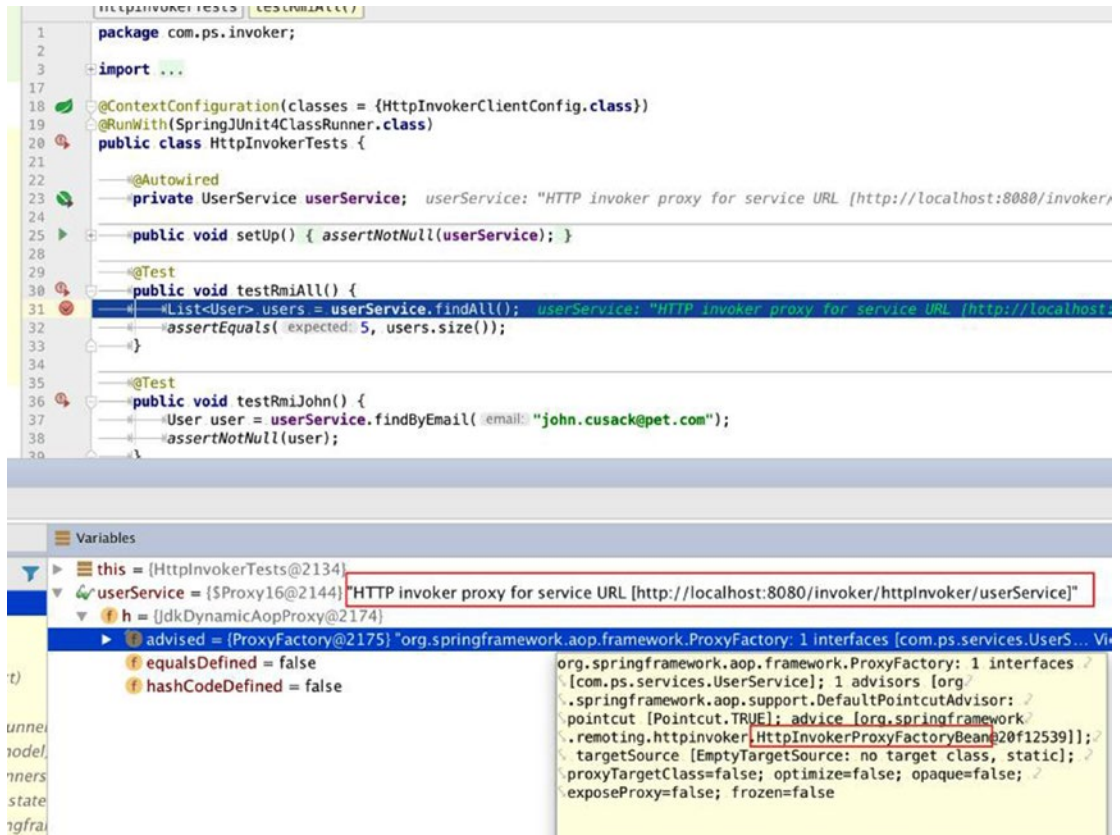


Figure 7-7. Screenshot of an HTTP Invoker test executed in debug mode

Hessian has been maintained and is still used with Spring 4. Currently in a draft state, Hessian 2 is the second incarnation of the Hessian protocol, but it is clearly not production ready. Burlap has not been maintained in a while, and in Spring 4, the Burlap Spring-specific classes `BurlapProxyFactoryBean` and `BurlapServiceExporter` are marked as deprecated and might be removed in Spring 5.

The Hessian-specific configuration is depicted in the following code snippet. (XML and Java Configuration).

```
<!-- hessian-service-config.xml -->
<beans ...>
  <bean name="/hessianInvoker/userService"
        class="org.springframework.remoting.caucho.HessianServiceExporter"
        p:service-ref="userServiceImpl"
        p:serviceInterface="com.ps.services.UserService"/>
</beans>
```

```
<!-- hessian-client-config.xml -->
<beans ...>
  <bean id="userServiceHessian"
        class="org.springframework.remoting.caucho.HessianProxyFactoryBean"
        p:serviceInterface="com.ps.services.UserService"
        p:serviceUrl="http://localhost:8080/invoker/hessianInvoker/userService"/>
</beans>
```

To test the Hessian remote service, just edit the `RmiTests` class and use the following configuration:

```
@ContextConfiguration(locations = {"classpath:spring/ hessian-client-config.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class RmiTests {
  ...
}
```

The Hessian client proxy factory bean and the server service exporter can be configured using Java Configuration. And if you feel confident enough, you are invited as a bonus exercise to do the implementation yourself. The source code used in this section can be found in the Pet Sitter project in modules `12-ps-remoting-sample` and `12-ps-remoting-practice`. The `12-ps-remoting-practice` projects contain a server- and client-side implementation for Http Invoker and Hessian remoting using XML configuration. The `12-ps-remoting-solution` is the proposed solution for HttpInvoker and Hessian using Java Configuration. You can compare this solution with the one you will implement in the `12-ps-remoting-practice`.

Spring JMS

JMS is part of the Java Platform Enterprise Edition and is defined by JSR 914.⁴ It is a messaging standard that allows applications with components based on JEE to create, send, receive, and read messages. The basic building blocks of a JMS applications are:

- messages
- message producers (publisher)
- messages consumers (subscriber)
- connections
- sessions
- connection factories
- destinations

In Figure 7-8, you can see all the previously listed components and how they fit into a JMS application.

⁴Java Community Process defining JMS: <https://jcp.org/en/jsr/detail?id=914>.

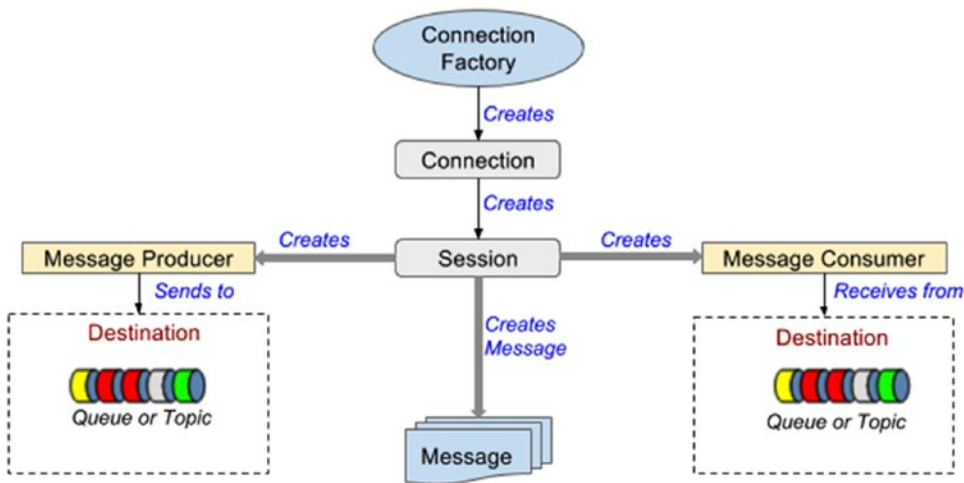


Figure 7-8. The JMS API Programming Model

In the introduction of this chapter it was mentioned that JMS is an abstraction for accessing Message Oriented Middleware, which is useful for avoiding vendor lock-in and for an increase in portability. In this section, the above components will be described briefly along with the process of publishing and subscribing to messages using a JMS broker using Spring, because as you probably suspect, Spring makes using JMS pretty easy too. Spring provides a JMS integration framework that makes using JMS API easy. The framework was developed using a similar approach to Spring JDBC.

JMS Connections and Sessions

The connection factories and destinations are the parts of a JMS application that are maintained administratively, rather than programmatically, since the technology providing the implementation must be decoupled from the implementation of JMS API used. JMS clients access these objects through JMS API standard interfaces. In an enterprise application, the connection factory can be bound to JNDI and managed externally by an enterprise application server. A JMS connection is obtained from a connection factory, which is injected into the client.

```
//XML standalone example
<bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:60006"/>
</bean>
```

```
//XML connection factory retrieved from JNDI
<jee:jndi-lookup id="connectionFactory"
    jndi-name="jms/ConnectionFactory" />
```

```
// using Java Configurations to declare a connection factory
//with JNDI name of jms/ConnectionFactory
@Resource(lookup = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

```
// standalone example
@Bean ConnectionFactory connectionFactory(){
    ActiveMQConnectionFactory connectionFactory =
        new ActiveMQConnectionFactory();
    connectionFactory.setBrokerURL("tcp://localhost:60006");
    return connectionFactory;
}

// Classic configuration
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.jms.*;
...
Properties properties = new Properties();
properties.put( /* JNDI properties */ );
Context context = new InitialContext(properties);
ConnectionFactory factory = (ConnectionFactory) context.lookup("jms/ConnectionFactory");
```

Once a connection factory is obtained all is left to do is get a `javax.jms.Connection` and then start a `javax.jms.Session` to start processing messages:

```
Connection connection = factory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
// process messages
...
```

The `createSession` method has two parameters: the first, of type `boolean`, is used to determine whether the session should be transacted (when `false` is used as argument, the resulting session is not transacted), and the second is used to determine whether the session automatically acknowledges messages when they have been received successfully. In the previous example, because the `Session.AUTO_ACKNOWLEDGE` is provided as an argument, the resulting session will do this.

The JMS session represents a unit of work. When processing messages is done, the session must be committed or rolled back, depending on the result of the processing.

```
session.commit(); // all ok
// or
session.rollback(); // something went wrong
```

The `Session` object is responsible for creating messages that will be sent to a client.

JMS Messages

The JMS API defines the standard form of a JMS message, which should be portable across all JMS providers. Although the JMS API was designed to accommodate many existing JMS providers, the message has a standard structure, being composed of a **header** and a **body**. The header contains system-level information common to all messages, such as the destination and the time it was sent and application-specific information, stored as keyword/value properties. The body contains the effective application data, and JMS defines five distinct message body types represented as Java interfaces extending `javax.jms.Message`. Thus a JMS message body is an instance of a class implementing any interface in the `javax.jms.Message` hierarchy. The five JMS API defined body types are listed in Table 7-1.

Messages are created by the Session object, and the following code snippet depicts how each type of message listed in the previous table is created:

```

TextMessage message = session.createTextMessage();
message.setText("this is a text message");

MapMessage mm = session.createMapMessage();
mm.set("key", "value");

session.createBytesMessage("bytes array".getBytes());

```

Table 7-1. JMS message formats

Message Type	Message Content
<code>javax.jms.TextMessage</code>	A Java String object
<code>javax.jms.MapMessage</code>	A set of key-value pairs with String keys and primitive values, without a fixed order.
<code>javax.jms.BytesMessage</code>	A stream of uninterpreted bytes.
<code>javax.jms.StreamMessage</code>	A stream of primitive values in the Java programming language, filled and read sequentially.
<code>javax.jms.ObjectMessage</code>	A Serializable object in the Java programming language

```

streamMessage = session.createStreamMessage();
//writing primitives on it
streamMessage.writeBoolean(false);
streamMessage.writeInt(223344);
streamMessage.writeChar('q');
//emptying the stream
streamMessage.reset();

//user must be serializable
session.createObjectMessage(user);

```

JMS Destinations

The `javax.jms.Destination` interface is implemented on the client side and is used to specify the target of the messages produced by the client and the source of messages the client consumes, because a client application can produce messages too, which can be sent as a reply to the JMS application with which it communicates. Imagine this scenario: the JMS server application sends a data message to the JMS client application and the client responds with a confirmation message that the data was received. Depending on the messaging domain, the implementation differs:

- a queue in point-to-point domains, where each message has one producer and one consumer.
- a topic in publisher/subscriber domains, where each message that is published is consumed by multiple subscribers.

The difference between Queues and Topics is depicted in Figure 7-9.

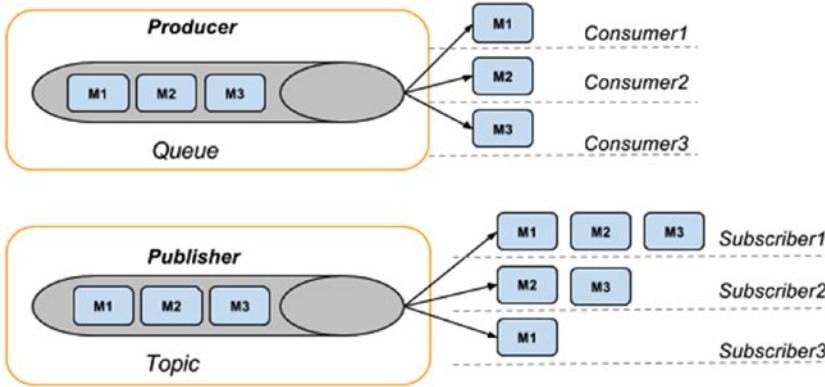


Figure 7-9. Difference between Queues and Topics

Destinations can be standalone or retrieved from JNDI:

```
// using Java Configurations to declare a queue
//with JNDI name of jms/UserQueue
@Resource(lookup = "jms/UserQueue")
private static Queue userQueue;

// using Java Configurations to declare a topic
//with JNDI name of jms/Topic
@Resource(lookup = "jms/Topic")
private static Topic topic;

// using XML to declare a queue
//with JNDI name of jms/Queue
<jee:jndi-lookup id="userQueue"
    jndi-name="jms/UserQueue" />

//XML standalone queue
<bean id="userQueue"
    class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="queue.users"/>
</bean>

//Java Configuration standalone Queue
@Bean
public Queue userQueue(){
    return new ActiveMQQueue("queues.users");
}
```

The `Session` object is responsible for creating producers and consumers using destination objects. Producer objects send messages, and consumer objects receive them.

```
MessageProducer producer = session.createProducer(userQueue);
ObjectMessage userMessage = session.createMessage(user);
producer.send(userMessage);
...
MessageConsumer consumer = session.createConsumer(userQueue);
Message message = consumer.receive();
```

So far, only core JMS components have been introduced, but before adding more, a JMS provider must be covered.

Apache ActiveMQ

There are quite a few JMS implementations out there. Most of them are provided by the Middle Oriented Middleware providers, example: WebSphere MQ, Oracle EMS, JBoss AP, SwiftMQ, TIBCO EMS, SonicMQ, ActiveMQ, WebLogic JMS. Some are open source, some are not. For some of them, a native C API also exists (TIBCO EMS and Sonic MQ). Some of them are implemented in Java itself.

Apache ActiveMQ, which was chosen as a JMS provider for this section, has the following characteristics:

- the most popular and powerful open source messaging and Integration Patterns server
- accepts non-Java clients
- can be used standalone in production environments
- supports pluggable transport protocols such as in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA transports
- can be used as an in memory JMS provider, using an embedded broker, to avoid the overhead of running separate processes when doing unit testing JMS⁵
- the `activemq` executable starts with a default configuration
- can also be used embedded in an application
- can be configured using ActiveMQ or Spring configuration (XML or Java Configuration)
- provides advanced messaging features such as message groups, virtual and composite destinations, and wildcards
- provides support for Enterprise Integration Patterns when used with Spring Integration or Apache Camel
- and many more.

⁵More details on the official Apache ActiveMQ site <http://activemq.apache.org/how-to-unit-test-jms-code.html>.

The current stable version as this chapter is being written is 5.14.1.⁶ It can be downloaded from the official site.⁷ To run the code sample attached to this chapter, you have to download the archive suitable for your system and follow the installation instructions on the official site. Basically, you have to unpack the archive somewhere and make sure it has the content depicted in Figure 7-10. You can see all the previously listed components and how they fit into a JMS client application.

```

/apache-activemq-5.14.1
LICENSE
NOTICE
README.txt
activemq-all-5.14.1.jar
bin
conf
data
docs
examples
lib
webapps
webapps-demo

```

Figure 7-10. The contents of the directory `apache-activemq-5.14.1`

In the `bin` directory there is an executable named `activemq` (or `activemq.bat` for Windows) that will start ActiveMQ on your computer. Just open a terminal (or a Command Prompt) and run the executable with parameter `start`.

```

$ ./activemq start
or
$ activemq.bat start (Windows)

```

ActiveMQ comes with a web interface and can be accessed at `http://127.0.0.1:8161/admin/`. It will request a user and a password. The official site says that you should use `admin/admin`. In Figure 7-11 you can see the Apache ActiveMQ web application, which can be used to monitor the JMS components mentioned previously: connection factories and destinations.

⁶Official ActiveMQ site: <http://activemq.apache.org/how-to-unit-test-jms-code.html>.

⁷Apache ActiveMQ <http://activemq.apache.org/activemq-5141-release.html>.

http://127.0.0.1:8161/admin

ActiveMQ™

The Apache Software Foundation
http://www.apache.org/

Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send | Support

Welcome!

Welcome to the Apache ActiveMQ Console of **localhost**
(ID:ROSBZM4044324X-63605-1476033511098-0:1)

You can find more information about Apache ActiveMQ on the [Apache ActiveMQ Site](#)

Broker

Name	localhost
Version	5.14.1
ID	ID:ROSBZM4044324X-63605-1476033511098-0:1
Uptime	1 hour 45 minutes
Store percent used	0
Memory percent used	0
Temp percent used	0

- Queue Views
 - Graph
 - XML
- Topic Views
 - XML
- Subscribers Views
 - XML
- Useful Links
 - Documentation
 - FAQ
 - Downloads
 - Forums

Copyright 2005-2015 The Apache Software Foundation.

Figure 7-11. The Apache ActiveMQ web application

The application covered by this section will use an ActiveMQ broker hosted at `tcp://localhost:61616`. The connection factory will use this broker to create connections. Aside from that, `ObjectMessage` objects depend on Java serialization of the marshal/unmarshal object payload. This process is generally considered unsafe, since malicious payload can exploit the host system. That's why starting with versions 5.12.2 and 5.13.0, ActiveMQ requires users to explicitly whitelist packages that can be exchanged using `ObjectMessages`.⁸ So the declaration of the `connectionFactory` bean will have to include packages that contain classes that can be serialized and used in JMS communication.

```
<!-- XML configuration -->
<beans ...>
  <bean id="connectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
    <property name="trustedPackages">
      <!-- List<String> argument-->
      <list>
        <value>com.ps</value>
      </list>
    </property>
  </bean>
</beans>
```

⁸More info here: <http://activemq.apache.org/objectmessage.html>.

```

        <!-- or general and unsafe -->
        <property name="trustAllPackages" value="true"/>
    </bean>
</beans>
//Java Configuration
@Configuration
public class JmsCommonConfig {

    List<String> packagesList= ...;

    @Bean
    public ConnectionFactory nativeConnectionFactory(){
        ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
        cf.setBrokerURL("tcp://localhost:61616");
        cf.setTrustedPackages(packagesList);
        // or or general and unsafe
        cf.setTrustAllPackages(true);
        return cf;
    }
    ...
}

```

The list of packages to be trusted for serialization can be quite long when objects are complex and contain fields of different types,⁹ so for teaching purposes, the `trustAllPackages` property can be used instead.

ActiveMQ destinations must be declared as beans of type `org.apache.activemq.command.ActiveMQQueue` for queues and `org.apache.activemq.command.ActiveMQTopic` for topics.

Spring JmsTemplate

Spring enables decoupling of the application from the backend infrastructure, but the application must be coded against the API. The `org.springframework.jms.core.JmsTemplate` is the central class of the Spring JMS core package. It is a helper class that is designed to simplify synchronous JMS message sending and receiving, by handling creation and release of resources. This template class provides the following development advantages:

- reduces boilerplate code, but the developer can implement callback interfaces, giving them a clearly defined high-level contract if the task requires it
- transparent management of JMS resources
- exposes a basic request-reply operation that makes it possible to send a message and wait for a reply on a temporary queue that is created as part of the operation
- provides practical exception handling, by converting JMS exceptions into Spring-specific JMS unchecked exceptions (extensions of `org.springframework.jms.JmsException`)

⁹Imagine an Object message class that contains three entities from three different packages; then the `packagesList` must contain ["package1", "package2", "package3"].

- supports custom message converters and destination resolvers
- provides convenience methods and callbacks

Starting with Spring Framework 4.1, the `org.springframework.jms.core.JmsMessagingTemplate` class was introduced, which is built on the `JmsTemplate` to provide an integration with the Spring messaging abstraction `org.springframework.messaging.Message<T>`, which allows generic message creation.

The `JmsTemplate` uses implementation of the `org.springframework.jms.support.converter.MessageConverter` implementation to convert between object and `Messages`. The `SimpleMessageConverter` handles basic types: text, serializable object types, maps, and `byte[]`. For more complex objects, developers can provide their own implementations or delegate to an OXM marshaller, which is available starting with Spring 3.

The following code snippet depicts the `MessageConverter` interface that a developer must implement to provide a custom message converter.

```
package org.springframework.jms.support.converter;
...
public interface MessageConverter {

    Message toMessage(Object object, Session session)
        throws JMSException, MessageConversionException;

    Object fromMessage(Message message)
        throws JMSException, MessageConversionException;
}
```

Out of the box, to make things easier, Spring provides a simple implementation for the `MessageConverter` class `org.springframework.jms.support.converter.SimpleMessageConverter` for default conversion of serializable objects into object messages, and a bean of this type is automatically created by Spring in JMS applications.

The `JmsTemplate` bean needs a connection factory and a destination to be initialized. Connection factories and destinations can be injected into a `JmsTemplate` bean using their IDs.

```
<!-- XML configuration -->
<beans ...>
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616"/>
        <property name="trustAllPackages" value="true"/>
    </bean>
    <bean id="confirmationQueue"
        class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="com.queue.confirmation"/>
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory"/>
        <property name="defaultDestination" ref="userQueue"/>
        <property name="pubSubNoLocal" value="false"/>
    </bean>
</beans>
```

```
//Java Configuration
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
...

@Bean
public ConnectionFactory connectionFactory(){
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    cf.setBrokerURL("tcp://localhost:61616");
    cf.setTrustAllPackages(true);
    return cf;
}

@Bean
public ActiveMQQueue confirmationQueue(){
    return new ActiveMQQueue("com.queue.confirmation");
}

@Bean
public JmsTemplate jmsTemplate(){
    JmsTemplate jmsTemplate = new JmsTemplate();
    jmsTemplate.setConnectionFactory(connectionFactory());
    jmsTemplate.setDefaultDestination(confirmationQueue());
    jmsTemplate.setPubSubNoLocal(false);
    return jmsTemplate;
}
```

The `pubSubNoLocal` property is used to set whether to inhibit the delivery of messages published by its own connection. Default value is `false`, but in the previous code samples, it is set explicitly for teaching purposes. When a message is received by the client (consumer), since the communication is asynchronous, a listener has to be set up to pick it up and do something with it. The interface to implement to provide a custom message listener is `javax.jms.MessageListener`, and the implementation for a single method: `onMessage(Message message)` must be provided. The message listener is managed by an implementation of the `org.springframework.jms.listener.MessageListenerContainer` in a Spring application, usually `DefaultMessageListenerContainer`. A bean of this type manages the JMS sessions with which the listener will be invoked. The `DefaultMessageListenerContainer` can be configured to set the consumer's concurrency limit for the `JmsTemplate` instances managed, by setting up the concurrency property (min-max interval as 5-10, or max value like 10). But each `JmsTemplate` bean can override this setting by setting up a different value for its own concurrency property. The implementation and configuration to set these two beans is depicted in the following code snippet. The `UserReceiver` class sends a confirmation that the user message was successfully received.

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;

public class UserReceiver implements MessageListener {

    private Logger logger = LoggerFactory.getLogger(UserReceiver.class);
    //used to generate unique IDs for Confirmation objects
    private static AtomicInteger id = new AtomicInteger();
```

```

@Autowired
MessageConverter messageConverter;

@Autowired
ConfirmationSender confirmationSender;

@Override
public void onMessage(Message message) {
    try {
        User receivedUser = (User) messageConverter.fromMessage(message);
        logger.info(" >> Received user: " + receivedUser);
        confirmationSender.sendMessage
            (new Confirmation(id.incrementAndGet(), "User "
                + receivedUser.getEmail() + " received."));
    } catch (JMSException e) {
        logger.error("Something went wrong ...", e);
    }
}

<!-- XML configuration -->
<bean id="containerListener"
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destination" ref="userQueue"/>
    <property name="messageListener" ref="userReceiver"/>
    <property name="concurrency" value="5-10"/>
</bean>

<bean id="userReceiver" class="com.ps.jms.UserReceiver"/>

//Java Configuration
@Bean
public UserReceiver userReceiver(){
    return new UserReceiver();
}

@Bean
public DefaultMessageListenerContainer containerListener() {
    DefaultMessageListenerContainer listener =
        new DefaultMessageListenerContainer();
    listener.setConnectionFactory(connectionFactory());
    listener.setDestination(userQueue());
    listener.setMessageListener(userReceiver());
    return listener;
}
}

```

On the server (producer) side, the message is sent, and different methods of the `JmsTemplate` bean can be used for this. The implementation that sends the message is a simple bean with an instance of `JmsTemplate` injected as dependency.

```

@Component
public class UserSender {

    @Autowired
    JmsTemplate jmsTemplate;
    public void sendMessage(final User user) {
        jmsTemplate.send ( (Session session) -> session.createObjectMessage(user));
    }
}

```

And the `JmsTemplate` used on the server(producer) side is defined as presented earlier. The queue used to send the messages is the `userQueue`.

```

@Bean
public ActiveMQQueue userQueue(){
    return new ActiveMQQueue("com.queue.user");
}

@Bean
public JmsTemplate jmsTemplate(){
    JmsTemplate jmsTemplate = new JmsTemplate();
    jmsTemplate.setConnectionFactory(connectionFactory());
    jmsTemplate.setDefaultDestination(userQueue());
    jmsTemplate.setPubSubNoLocal(false);
    return jmsTemplate;
}

```

The `JmsTemplate` provides methods that receive the destination as a parameter; such methods are useful when the application needs to send messages to multiple destinations. A custom message converter can be set on the `JmsTemplate`, and the `convertAndSend` methods can be used in this case.

```
convertAndSend(final Object message)
```

```
convertAndSend(Destination destination, final Object message)
```

```
convertAndSend(String destinationName, final Object message)
```

The project assigned to this section is `13-ps-jms-practice`. It contains two applications that communicate using ActiveMQ using two queues of objects. The producer application extracts users from the database and sends them as messages to the `userQueue` and then waits to receive a confirmation message on `confirmationQueue`. The consumer application receives users as messages on the `userQueue` and uses the `confirmationQueue` to send a confirmation for each user received. The configuration will be provided using Java Configuration classes. In Figure 7-12 you can see the design of the JMS application that will be used to test your understanding of JMS with Spring.

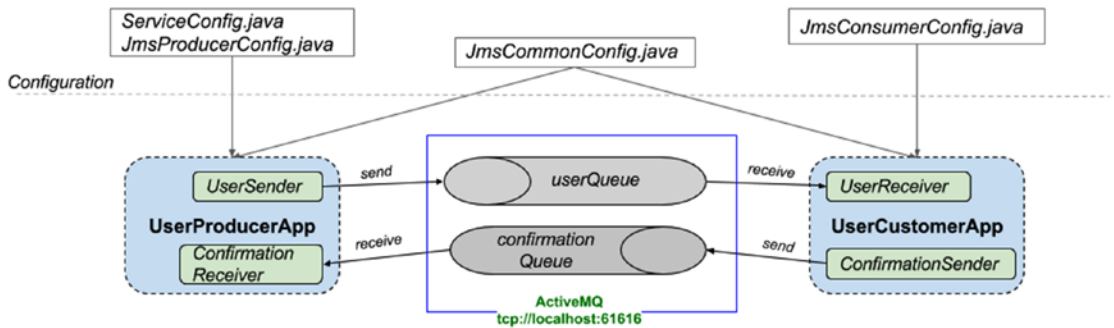


Figure 7-12. The Producer/Consumer JMS abstract design

The two applications have a common configuration class `JmsCommonConfig.java`. This class contains the configuration of the JMS infrastructure: the connection factory, the queues, and the message converter.

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.SimpleMessageConverter;
```

```
import javax.jms.ConnectionFactory;
```

```
@Configuration
public class JmsCommonConfig {

    @Bean
    public ConnectionFactory connectionFactory(){
        ActiveMQConnectionFactory cf =
            new ActiveMQConnectionFactory();
        cf.setBrokerURL("tcp://localhost:61616");
        cf.setTrustAllPackages(true);
        return cf;
    }

    @Bean
    public ActiveMQQueue userQueue(){
        return new ActiveMQQueue("com.queue.user");
    }

    @Bean
    public ActiveMQQueue confirmationQueue(){
        return new ActiveMQQueue("com.queue.confirmation");
    }
}
```

```

@Bean
public MessageConverter converter() {
    return new SimpleMessageConverter();
}
}

```

The `JmsTemplate` aggressively opens and closes resources such as connections and sessions, since it assumes that they are cached by the `connectionFactory`. For this reason, instead of using the `ActiveMQConnectionFactory` class as a type for the `connectionFactory` bean, the Spring-specific `CachingConnectionFactory` class should be used to wrap up the native implementation.

```

import org.springframework.jms.connection.CachingConnectionFactory;
...
@Configuration
public class JmsCommonConfig {

    @Bean
    public ConnectionFactory nativeConnectionFactory(){
        ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
        cf.setBrokerURL("tcp://localhost:61616");
        cf.setTrustAllPackages(true);
        return cf;
    }

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory cf = new CachingConnectionFactory();
        cf.setTargetConnectionFactory(nativeConnectionFactory());
        return cf;
    }
    ...
}

```

The classes to start the two applications are simple classes with a main method. They can be run from IntelliJ IDEA as Java applications. Each of them instantiates a context from the application classes and can be stopped from the terminal by pressing the <Enter> key. Before the applications can be run, the ActiveMQ broker must be started. The communication and the activity on the two queues can be monitored in the ActiveMQ web interface, in the Queues tab. The producer application should be started first (the `UserProducerApp` class), and in the ActiveMQ browser interface, the number of messages produced should be visible in the Message Enqueued column, as shown in Figure 7-13.

The screenshot shows the ActiveMQ web application interface. At the top, there is a navigation bar with links: Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send. Below the navigation bar, there is a search field for Queue Name and a Create button. The main content area is titled "Queues" and contains a table with the following data:

Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
com.queue.confirmation	0	1	0	0	Browse Active Consumers Active Producers 	Send To Purge Delete
com.queue.user	5	0	5	0	Browse Active Consumers Active Producers 	Send To Purge Delete

Figure 7-13. The ActiveMQ web application, the Queues tab depicting the two queues used by the application and messages already produced and stored in the userQueue

The queues are created the first time the producer application is started. The messages can be purged from the interface. The queues can be deleted from the web interface as well, as long as there is no application accessing them. After the consumer application (the UserConsumerApp class) is started, the activity on the second queue application should be visible as well in the Message Dequeued column. This situation is depicted in Figure 7-14.

The screenshot shows the ActiveMQ web application interface, similar to Figure 7-13. The navigation bar and search field are the same. The "Queues" table now shows updated data:

Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
com.queue.confirmation	0	1	5	5	Browse Active Consumers Active Producers 	Send To Purge Delete
com.queue.user	0	1	5	5	Browse Active Consumers Active Producers 	Send To Purge Delete

Figure 7-14. The ActiveMQ web application, the Queues tab depicting the two queues used by the application and messages already produced and consumed

The next step in creating Spring JMS applications is to add Spring Boot to the mix. In the `13-ps-jms-practice` project module you can find the two JMS applications described so far with a pre-Boot Java Configuration. In the `13-ps-jms-solution` you can find an equivalent Spring JMS application configured with Spring Boot.

JMS with Spring Boot

Writing Spring JMS applications is even easier with Spring Boot. The application in the project module `13-ps-jms-solution` contains a Spring Boot JMS application that sends `User` instances wrapped up in JMS Messages to the `userQueue`. A message listener is configured to process the message and send a confirmation message on the `confirmationQueue`. Another listener is defined that waits for the confirmation and prints its contents. To simplify the application even more, there is no need for a producer class and a consumer class. There is only one process that publishes and consumes messages from the two queues. The abstract schema of the Spring Boot JMS application is depicted in Figure 7-15.

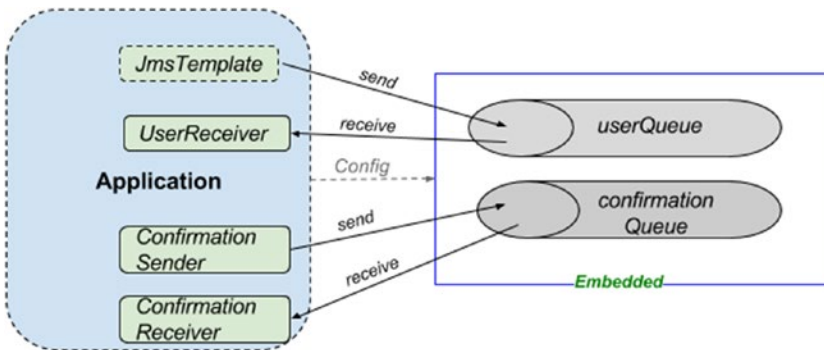


Figure 7-15. Spring Boot JMS application abstract schema

The application is made of only four classes, which have the following purpose:

- `UserReceiver` and `ConfirmationReceiver` have the responsibility of receiving the user and confirmation messages. The classes have a more practical implementation, making use of the `@JmsListener` annotation. Using this annotation on the methods that will process the messages allows the developer to escape the JMS restriction of implementing the `MessageListener`. This annotation marks the method on which it is used as a target of a JMS listener on the specified destination, and the connection factory attribute specifies the bean to use to build the JMS Listener Container. This annotation is processed by a special Spring bean post processor: `org.springframework.jms.annotation.JmsListenerAnnotationBeanPostProcessor`.

```
// UserReceiver.java
@Component
public class UserReceiver{

    private Logger logger = LoggerFactory.getLogger(UserReceiver.class);
    private static AtomicInteger id = new AtomicInteger();

    @Autowired
    ConfirmationSender confirmationSender;
```



```

    @JmsListener(destination = "userQueue",
        containerFactory = "connectionFactory")
    public void receiveMessage(User receivedUser) {
        logger.info(">> Received user: " + receivedUser);
        confirmationSender.sendMessage(
            new Confirmation(id.incrementAndGet(), "User "
                + receivedUser.getEmail() + " received."));
    }
}

//ConfirmationReceiver.java
@Component
public class ConfirmationReceiver {

    private Logger logger = LoggerFactory.getLogger(ConfirmationReceiver.class);

    @JmsListener(destination = "confirmationQueue",
        containerFactory = "connectionFactory")
    public void receiveConfirmation(Confirmation confirmation) {
        logger.info(">> Received confirmation: " + confirmation);
    }
}

```

- The ConfirmationSender class is just a bean with a JmsTemplate bean injected in it that is being used to send a confirmation object to the confirmationQueue.

```

@Component
public class ConfirmationSender {

    @Autowired
    JmsTemplate jmsTemplate;

    public void sendMessage(final Confirmation confirmation) {
        jmsTemplate.convertAndSend("confirmationQueue", confirmation);
    }
}

```

- The Application class is the configuration and starter of the application. It is annotated with the all-powerful @SpringBootApplication, which will make sure to scan for beans in the current package and will inject all infrastructure beans based on the classpath settings. The other annotation on this class is @EnableJms, and as you can probably speculate, this annotation enables the bean postprocessor that will process the @JmsListener annotations and will create the message listener container "under the hood". In this class, the connectionFactory bean was declared as well. In this example, the type of the bean is org.springframework.jms.config.JmsListenerContainerFactory, and the implementation used is the Spring Boot org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigurer, which will boot up all Spring Boot infrastructure beans necessary for a JMS application.

- There is also a converter bean defined of type `MappingJackson2MessageConverter` implementing the JMS `MessageConverter` interface that transforms the sent objects into text JSON format. Using a text converter is useful, especially when the messages to send are not of serializable types. And in this example, the `User` and the `Confirmation` classes were not declared to implement `Serializable` intentionally.
- The main method, aside from starting up the application, instantiates a `JmsTemplate` object that is used to send a user object.

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jms.
    DefaultJmsListenerContainerFactoryConfigurer;
import org.springframework.jms.annotation.EnableJms;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;

import org.springframework.jms.support.converter.MappingJackson2MessageConverter;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.MessageType;
...
@SpringBootApplication
@EnableJms
public class Application {
    private static Logger logger = LoggerFactory.getLogger(Application.class);

    public static void main(String args) throws Exception {
        ConfigurableApplicationContext context =
            SpringApplication.run(Application.class, args);
        JmsTemplate jmsTemplate = context.getBean(JmsTemplate.class);

        //Send an user
        System.out.println("Sending an user message.");
        jmsTemplate.convertAndSend("userQueue",
            new User("John.Cusack@pet.com", 5d, true));

        logger.info("Waiting for user and confirmation ...");
        System.in.read();
        context.close();
    }

    @Bean // Serialize message content to json using TextMessage
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter =
            new MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_type");
        return converter;
    }
}
```

```

@Bean
public JmsListenerContainerFactory<?> connectionFactory(
    ConnectionFactory connectionFactory,
    DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory factory =
        new DefaultJmsListenerContainerFactory();
    configurer.configure(factory, connectionFactory);
    return factory;
}
}

```

The `MappingJackson2MessageConverter` converter bean has two properties set in the previous code sample. The `targetType` is set with the JMS message type to which the information will be serialized. Default is `BytesMessage`. For JSON type, `TextMessage` is suitable, because in this way, the JMS messages become readable, and thus debugging in case of problems becomes easier. The `typeIdPropertyName` is used to name the JMS message property that carries the type ID for the contained object: either a mapped ID value or a raw Java class name. The default value is `NONE`, but this property must be set to allow converting from an incoming message to a Java object. In the previous example, the property is set to `_type`. Assuming that a `User` instance is sent, the JMS message in this instance is converted to look similar to what is depicted in the following code snippet:

```

ActiveMQTextMessage {
    commandId = 5,
    responseRequired = true,
    messageId = ID:ROSBZM4044324X-64390-1478342192637-4:3:1:1:1,
    destination = queue://userQueue,
    size = 1140,
    properties = {_type=com.ps.jms.User},
    text = {"email":"John.Cusack@pet.com", "rating":5.0, "active":true}
    ...
}

```

The `@JmsListener` was added in Spring version 4.1 and represents a really practical way to configure a message listener. Methods annotated with it have flexible signatures that can have arguments that the Spring container injects automatically, such as the JMS session, the exact message object, JMS headers, and others that you can find listed in the official documentation. The JSON snippet depicted earlier was retrieved by modifying the `UserReceiver` implementation and adding a parameter of type `Message`, which Spring will replace with the original JMS message at runtime.

```

@Component
public class UserReceiver{

    private Logger logger = LoggerFactory.getLogger(UserReceiver.class);
    private static AtomicInteger id = new AtomicInteger();

    @Autowired
    ConfirmationSender confirmationSender;

    @JmsListener(destination = "userQueue", containerFactory = "connectionFactory")
    public void receiveMessage(User receivedUser, Message message) {
        logger.info(">> Original received message: " + message);
    }
}

```

```

        logger.info(" >> Received user: " + receivedUser);
        confirmationSender.sendMessage(new Confirmation(id.incrementAndGet(), "User "
            + receivedUser.getEmail() + " received.));
    }
}

```

The `JmsTemplate` makes it very easy to send messages to destinations, and in this example is automatically created by Spring Boot. The `connectionFactory` is also created by Spring Boot and is backed up by ActiveMQ running in embedded mode.

JMS can be used as a mechanism to allow asynchronous request processing between backend components. It is quite useful when data replication is needed or in the execution of long-running processes of which the end user of an application has no knowledge and that should not impede the use of the application. Also, publish/subscribe is a good technique to decouple senders from receivers, thereby increasing the flexibility of an architecture. JMS is still used mostly in flexible processes where communication is asynchronous and messages can be lost and it is not a problem to resend them.

Spring Web Services

The term “web service” describes a standardized way of communication between web applications integration using XML, SOAP, WSDL, and UDDI open standards over an internet protocol backbone. This type of communication allows for a relaxed coupling of the applications doing the communication and is really useful when the applications are subject to frequent changes, since changes should not affect the compatibility between them. XML used to be the international language in the web services world, until JSON came into the picture. XML is a standardized way of organizing data that can be understood by all major platforms:

- Java, uses APIs as SAX, StAX, DOM, JAXB to manipulate XML
- .NET uses System.XML or .NET XML Parser
- Ruby uses Nokogiri, REXML or XMLSimple
- Perl uses LibXML, Perl-XML or XML::Simple
- etc.

SOAP is an acronym for **S**imple **O**bject **A**ccess **P**rotocol. It is a protocol specification for exchanging structured information in the implementation of web services in computer networks. SOAP is not necessarily linked to web services or RPC, and it encapsulates the key terms that should be respected in designing loosely coupled applications that need to exchange information: extensibility (security and WS-routing are among the extensions under development), neutrality (can operate over any protocol such as HTTP, SMTP, TCP, UDP, or JMS) and independence (allows for any programming model). Basically, SOAP is what allows communication between applications regardless of the implementation or the underlying operating system.

Web services implementation details are usually enclosed in the Web Services layer, which communicates directly with the service layer. The Web Service layer provides the logic to convert software objects into XML-based representations that can be sent to destination applications. Spring provides support for web services implementation through its Spring Web Services (Spring-WS), which is a product of the Spring community focused on creating document-driven Web services. Like all Spring products, its aim is to provide the infrastructure and support components that make writing web-services-based applications practical and pleasant. Below are listed a few key features of the Spring-WS.

- It makes the best practice of using XSD/WSDL first-contracts easy, which is quite important, since using WSDL is often seen as a hassle by developers, even if the main advantage is that it solves many interoperability issues.
- It provides XML API support: XML messages can be handled using JAXP APIs such as DOM, SAX, and StAX, but also JDOM, dom4j, XOM, or even marshalling technologies.
- It offers flexible XML Marshalling: the Object/XML Mapping module supports almost all known libraries for XML marshalling: OXM, Castor, JAXB 1 and 2, XStream, etc.
- WS-Security provides the tools to sign, encrypt, and decrypt SOAP messages and integrates with Spring Security.
- and many more.

Spring-WS is not a topic for the exam and is also a wide subject, so the surface will only be scratched in this section. The main points will, however, be covered to help you understand the general idea of designing and development of web-service-based applications with Spring. Here are the steps of designing a Contract-first:

- create sample messages. A simple service user message could look like this:

```
<userMessage xmlns="http://ws-boot.com/schemas/um" active="true">
  <email>John.Cusack@pet.com</email>
  <rating>5.0</rating>
</userMessage>
```

- define the XSD schema the service message must comply to:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://ws-boot.com/schemas/um">

  <xs:complexType name="userMessage">
    <xs:sequence>
      <xs:element name="email" type="xs:string"/>
      <xs:element name="rating" type="xs:double"/>
    </xs:sequence>
    <xs:attribute name="active" type="xs:boolean"/>
  </xs:complexType>
</xs:schema>
```

- restrict types and values, by enriching the XSD schema. In the next code snippet, a regular expression pattern is specify to validate the value of the email address.

```
<xs:element name="email">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="^[^@]+@[^\.]+\.\.+"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

SOAP Messages

SOAP messages have a structure similar to JMS Messages: they have a header and a body that are enclosed in a special SOAP envelope that identifies the XML document as a SOAP message. Additionally, a SOAP message can contain fault information regarding the errors that can occur during web service communication. So a SOAP message containing user information looks like the following XML snippet.

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:um="http://ws-boot.com/schemas/um">
<soapenv:Header/>
<soapenv:Body>
  <um:getUserRequest>
    <um:email>John.Cusack@pet.com</um:email>
  </um:getUserRequest>
</soapenv:Body>
</soapenv:Envelope>
```

When writing web-services applications there are four possible starting points:

- starting from an XML message: in this approach, development starts with an XML representation of the web-service message. Then a smart editor (like IntelliJ IDEA or Eclipse) is used to generate a simple XSD schema, which can be further customized. And from this point on, the development steps are the same as for the next option in the list. (This is the approach used to develop the web service covered by this section.)
- starting from an XML Schema: in this approach, development begins with an XML schema (XSD file) that defines XML data structures to be used as parameters and return types in the web service operations. A smart editor should provide you with an option to generate Java classes that are needed to represent objects to be handled by the web service application. Or the Java `xjc` utility can be used, which will be introduced soon. In IntelliJ IDEA there are two options for code generation under the **WebServices** menu that appears on right-clicking the `*.xsd` file: generation with JAXB¹⁰ and generation with XMLBeans.¹¹
- starting from a Java Class: in this approach, XSD schemas and WSDL can be generated from existing code using a smart editor or the Java `xjc` utility, which will be introduced soon.
- starting from a WSDL: this is also called the “contract-first” or “top-down” method, and in this approach, you begin by defining the web service contract or use a preexisting one. If you have a `*.wsdl` file, you can use a smart editor to generate Java code from WSDL or the Java `xjc` utility, which will be introduced soon. (IntelliJ IDEA has this option under the WebServices menu item.)

¹⁰JAXB is one part of the JavaEE standards and refers to Java Architecture for XML Binding. The use of JAXB is preferred, since it offers an alternative for the processing of XML documents in Java without necessarily having to understand all the details of XML technology.

¹¹XMLBeans is an open-source project developed by BEA systems, providing similar functionality to JAXB in allowing XML documents to be accessed. A good background knowledge in XML is, however, required for dealing with XMLBeans.

The XSD schema defines the web service domain and the operations that can be performed using web services. Spring-WS will export this schema as a WSDL (Web Service Definition Language) automatically. For the application in `14-ps-ws-sample` the XSD schema will contain an operation that uses a user's email to extract the user type. The XSD schema is located under `src/main/resources/sample` and is named `userMessage`. This project is created with Spring Boot to set the focus on the implementation of the contract and not on the application infrastructure.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:um="http://ws-boot.com/schemas/um"
  elementFormDefault="qualified"
  targetNamespace="http://ws-boot.com/schemas/um">

  <xs:element name="getUserRequest">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="email" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="getUserResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="userType" type="um:userType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="userMessage">
    <xs:sequence>
      <xs:element name="email">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:pattern value="^+@+@\.\.+"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
      <xs:element name="rating" type="xs:double"/>
      <xs:element name="userType" type="um:userType"/>
    </xs:sequence>
    <xs:attribute name="active" type="xs:boolean"/>
  </xs:complexType>

  <xs:simpleType name="userType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="OWNER"/>
      <xs:enumeration value="SITTER"/>
      <xs:enumeration value="BOTH"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

In Figure 7-16, a client and server WS application and the way they communicate are depicted.

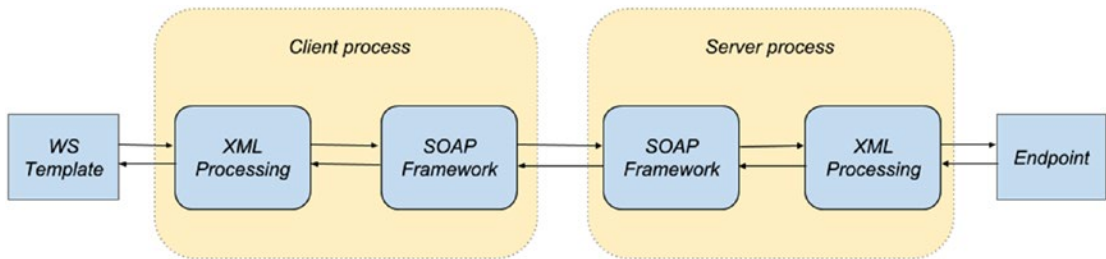


Figure 7-16. Spring WS Client/Server applications

Now that the message structure has been defined, the next step is to generate the backing classes for it.

Generating Java Code with XJC

The JDK comes with a utility executable called `xjc`. This executable can be used to generate classes to use in a web service application using the XSD schema. This can be done by running the executable manually or by creating a Gradle task to do that for you. In the `14-ps-ws-sample` project, there is a `README.adoc` file that contains the `xjc` command to generate the required Java classes and save them in an additional source directory named `jaxb`. The classes are by default generated with JAXB and thus are annotated with JAXB annotations.

```
xjc -d src/main/jaxb -p com.ps.ws src/main/resources/sample/userMessage.xsd
```

Smart editors like IntelliJ IDEA have the capability of generating JAVA code from a built-in XSD schema. Just select the `userMessages.xsd` file and right click. In the menu that appears, there is a `WebServices` option. Click on it, and it will expand. In the submenu there is a `Generate Java Code from XML Schema using JAXB`. By selecting this, a pop-up will appear that will require the package name and location where the classes should be saved. If you open the `UserMessage` class, you will notice the specific JAXB annotation everywhere, which tell how objects of that type will be serialized and deserialized:

```
import javax.annotation.Generated;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlSchemaType;
import javax.xml.bind.annotation.XmlType;
```

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "userMessage",
    namespace = "http://ws-boot.com/schemas/um",
    propOrder = {
        "email",
        "rating",
        "userType"
    })
```



```

@Generated(value = "com.sun.tools.internal.xjc.Driver",
    date = "2016-10-16T08:23:57+03:00",
    comments = "JAXB RI v2.2.8-b130911.1802")
public class UserMessage {

    @XmlElement(namespace = "http://ws-boot.com/schemas/um",
        required = true)
    @Generated(value = "com.sun.tools.internal.xjc.Driver",
        date = "2016-10-16T08:23:57+03:00",
        comments = "JAXB RI v2.2.8-b130911.1802")
    protected String email;
    @XmlElement(namespace = "http://ws-boot.com/schemas/um")
    @Generated(value = "com.sun.tools.internal.xjc.Driver",
        date = "2016-10-16T08:23:57+03:00",
        comments = "JAXB RI v2.2.8-b130911.1802")
    protected double rating;
    ...
}

```

More information about JAXB can be found on the official JAXB site <https://jaxb.java.net/>.

Spring Boot WS Application

In Figure 7-17 you can see the structure of the 14-ps-ws-sample application.

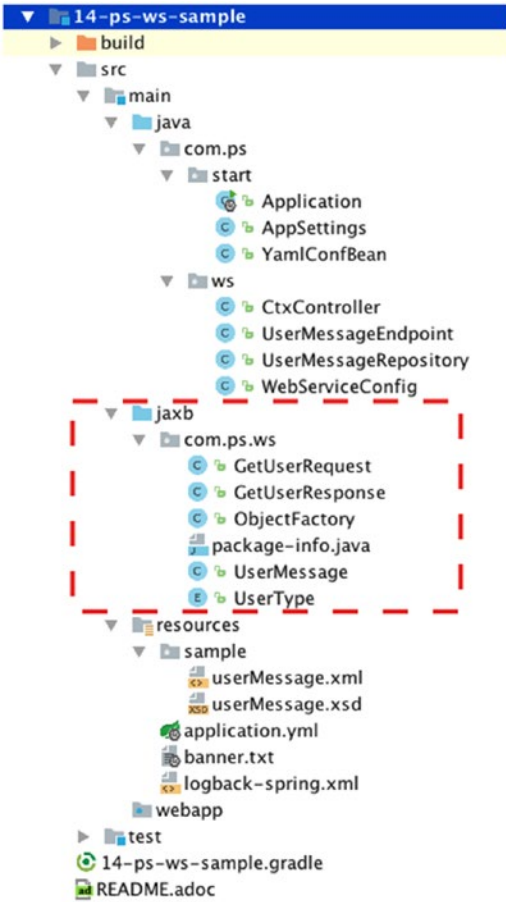


Figure 7-17. The structure of the Spring WS application

The `com.ps.start` groups together Spring Boot components.

The `com.ps.ws` under the `src/main/java` directory groups together classes related to WS communication that are written by the developer:

- the configuration of WS application, the `WebServiceConfig`. This class extends the Spring WS `org.springframework.ws.config.annotation.WsConfigurerAdapter`, which is a Spring naked (empty methods) implementation of the web services core configuration interface `org.springframework.ws.config.annotation.WsConfigurer`. This interface defines callback methods to customize the Java-based configuration for Spring Web Services enabled using the `@EnableWs`.

```
import org.springframework.boot.web.servlet.ServletRegistrationBean;
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.ws.config.annotation.EnableWs;
```

```

import org.springframework.ws.config.annotation.WsConfigurerAdapter;
import org.springframework.ws.transport.http.MessageDispatcherServlet;
import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
import org.springframework.xml.xsd.SimpleXsdSchema;
import org.springframework.xml.xsd.XsdSchema;

@Configuration
@EnableWs
public class WebServiceConfig extends WsConfigurerAdapter {

    @Bean
    public ServletRegistrationBean messageDispatcherServlet
        (ApplicationContext applicationContext) {
        MessageDispatcherServlet servlet = new MessageDispatcherServlet();
        servlet.setApplicationContext(applicationContext);
        servlet.setTransformWsdlLocations(true);
        return new ServletRegistrationBean(servlet, "/ws/*");
    }

    @Bean(name = "userMessage")
    public DefaultWsdl11Definition defaultWsdl11Definition
        (XsdSchema userMessageSchema) {
        DefaultWsdl11Definition wsdl11Definition = new DefaultWsdl11Definition();
        wsdl11Definition.setPortTypeName("UsersPort");
        wsdl11Definition.setLocationUri("/ws");
        wsdl11Definition.setTargetNamespace(UserMessageEndpoint.NAMESPACE_URI);
        wsdl11Definition.setSchema(userMessageSchema);
        return wsdl11Definition;
    }

    @Bean
    public XsdSchema userMessageSchema() {
        return new SimpleXsdSchema(new ClassPathResource("sample/userMessage.
xsd"));
    }
}

```

- The `org.springframework.ws.server.endpoint.adapter.DefaultMethodEndpointAdapter` bean is used to enable the postprocessor beans that will process the `@Endpoint`, `@RequestPayload`, `@PayloadRoot`, and `@ResponsePayload` annotations that are used to configure a WS endpoint. The `messageDispatcherServlet` bean is used to intercept SOAP requests. The application context must be injected into this bean in order to be able to detect Spring beans automatically. This bean does not override the default `dispatcherServlet` bean, so a web interface can be set up for this application as well; with both servlets configured, this web application could handle normal web requests and web services requests. The `DefaultWsdl11Definition` exposes a WSDL 1.1 using an instance of `SimpleXsdSchema` instantiated with the `userMessage.xsd` schema.

- The `UserMessageEndpoint` class is annotated with the Spring specialized annotation `@Endpoint`, which marks this class as a web service endpoint that will process SOAP messages. Classes annotated with `@Endpoint` handle WS requests in a similar manner, while classes annotated with `@Controller` handle web requests.

```
import com.ps.ws.GetUserRequest;
import com.ps.ws.GetUserResponse;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;

@Endpoint
public class UserMessageEndpoint {

    public static final String NAMESPACE_URI = "http://ws-boot.com/schemas/um";

    @Autowired
    private UserMessageRepository userMessageRepository;

    @Autowired
    public UserMessageEndpoint(UserMessageRepository userMessageRepository) {
        this.userMessageRepository = userMessageRepository;
    }

    @PayloadRoot(namespace = NAMESPACE_URI, localPart = "getUserRequest")
    @ResponsePayload
    public GetUserResponse getUser(@RequestPayload GetUserRequest request) {
        GetUserResponse response = new GetUserResponse();
        response.setUserType(userMessageRepository.findUserType(request.getEmail()));
        return response;
    }
}
```

- The `@Payload` annotation marks an endpoint method as the handler for an incoming request and is identified by Spring WS based on the message's namespace and the localPart.
- The `@ResponsePayload` annotation specifies that Spring WS should map the returned value to the response payload.
- The `@RequestPayload` annotation specifies that the incoming message will be mapped to the request parameter of the method.

The `com.ps.ws` package under `src/main/jaxb`, which is circled with red in Figure 7-17, contains the classes generated with `xjc`. They have to be regenerated every time the `userMessage.xsd` schema is modified.

Publishing WSDL

WSDL defines a network interface that consists of endpoints that get messages and then sometimes reply with messages. WSDL describes the endpoints, and the request and reply messages. Spring-WS automatically generates WSDL from XSD when using a `DefaultWsd11Definition` bean, which can be accessed at the URL `http://<host>:<port>/ws/userMessage.wsdl` and is generated dynamically at runtime. The `DefaultWsd11Definition` builds a WSDL from an XSD schema. This definition iterates over all element elements found in the schema, and creates a message for all elements. Next, it creates a WSDL operation for all messages that end with the defined request or response suffix. The recommendation is to use a static WSDL file on production applications and expose it using a bean of type `org.springframework.ws.wsdl.wsdl11.SimpleWsd11Definition`, since generating a WSDL file might take a long time. During development, having it generated at runtime is useful, because if the contract changes, the WSDL must adapt.

■ ! The location where the WSDL information can be accessed depends on the `DefaultWsd11Definition` bean. Basically, the WSDL information can be found at the location defined like this:

```
http://<host>:<port>/[locationUri]/[DefaultWsd11Definition_bean_id].wsdl
```

For the application described so far, the WSDL information can be found at `http://localhost:8080/ws/userMessage.wsdl`, and when opened in a browser, all the information in the XSD schema and some other information related to the server, such as the SOAP endpoint address, are shown. The address is generated relative to the URL used to access the WSDL information, because the `transformWsd11Locations` property of the `DefaultWsd11Definition` bean is set to `true`. The next XML snippet is a piece of the WSDL generated by Spring WS for the `14-ps-ws-sample` application.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://ws-boot.com/schemas/um"
  targetNamespace="http://ws-boot.com/schemas/um">
  <wsdl:types>
    <xs:schema .../>
    <wsdl:message name="getUserResponse">
      <wsdl:part element="tns:getUserResponse" name="getUserResponse">
        </wsdl:part>
      </wsdl:message>
    <wsdl:message name="getUserRequest">
      <wsdl:part element="tns:getUserRequest" name="getUserRequest">
        </wsdl:part>
      </wsdl:message>
    ...
  <wsdl:service name="UsersPortService">
    <wsdl:port binding="tns:UsersPortSoap11" name="UsersPortSoap11">
      <soap:address location="http://localhost:8080/ws"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Testing Web Services applications

To test the application, a test class can be written, and Spring web-services-specific components such as the `WebTemplate` can be used to make development practical. The application is written using Spring Boot, so it is only logical to use Spring Boot to test it. The `SpringBootTest` annotation is used on the test class that runs Spring-Boot-based tests and searches for Spring Boot configuration classes that will be used to create a Spring boot test context. The `@LocalServerPort` is a Spring Boot annotation that can be used at the field or method/constructor parameter level to inject the HTTP port that was allocated at runtime. This value is needed in the test method to create the URL where the web service to be tested is exposed.

```
import org.springframework.boot.context.embedded.LocalServerPort;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.util.ClassUtils;
import org.springframework.ws.client.core.WebServiceTemplate;

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ApplicationTest {

    private static final Logger logger = LoggerFactory.getLogger(Application.class);

    //used to deserialize Web Service messages
    private static final Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    private static final WebServiceTemplate ws = new WebServiceTemplate(marshaller);

    @LocalServerPort
    private int port;

    @BeforeClass
    public void init() throws Exception {
        marshaller.setPackagesToScan(
            ClassUtils.getPackageName(GetUserRequest.class));
        marshaller.afterPropertiesSet();
    }

    @Test
    public void testSendAndReceive() {
        GetUserRequest request = new GetUserRequest();
        request.setEmail("John.Cusack@pet.com");

        Object responseObject = ws.marshalSendAndReceive(
            "http://localhost:" + port + "/ws", request);
        assertThat(responseObject).isNotNull();
        assertThat(responseObject instanceof GetUserResponse).isTrue();

        GetUserResponse response = (GetUserResponse) responseObject;
        assertThat(response.getUserType()).isEqualTo(UserType.BOTH);
    }
}
```

The `WebServiceTemplate` class is the equivalent of `RestTemplate` and `JdbcTemplate` for Web Services. It is the central class for client-side Web services and provides a message-driven approach to sending and receiving `WebServiceMessage` messages. It simplifies access to web services, since it extracts the object from the SOAP message body usingmarshallers/unmarshallers. The `WebServiceTemplate` object needs to serialize/deserialize messages and needs a `JAXB2Marshaller` to transform the messages from XML to Java Objects and vice versa. To test a SOAP interface and actually see the messages in XML form, the SoapUI¹² application can be used as a client or the `curl` command in Unix-based systems. The SoapUI team also developed a plugin for IntelliJ IDEA, which can be installed from the IntelliJ IDEA Preferences menu. The interface of the standalone SoapUI is very similar to the plugin interface, so the images in this section will also be helpful in case you do not use IntelliJ IDEA for development. In Figure 7-18, the IntelliJ IDEA Plugin section from the Preferences menu is depicted. Just insert SoapUI in the search box, and when the plugin appears in the list, click the Install button.

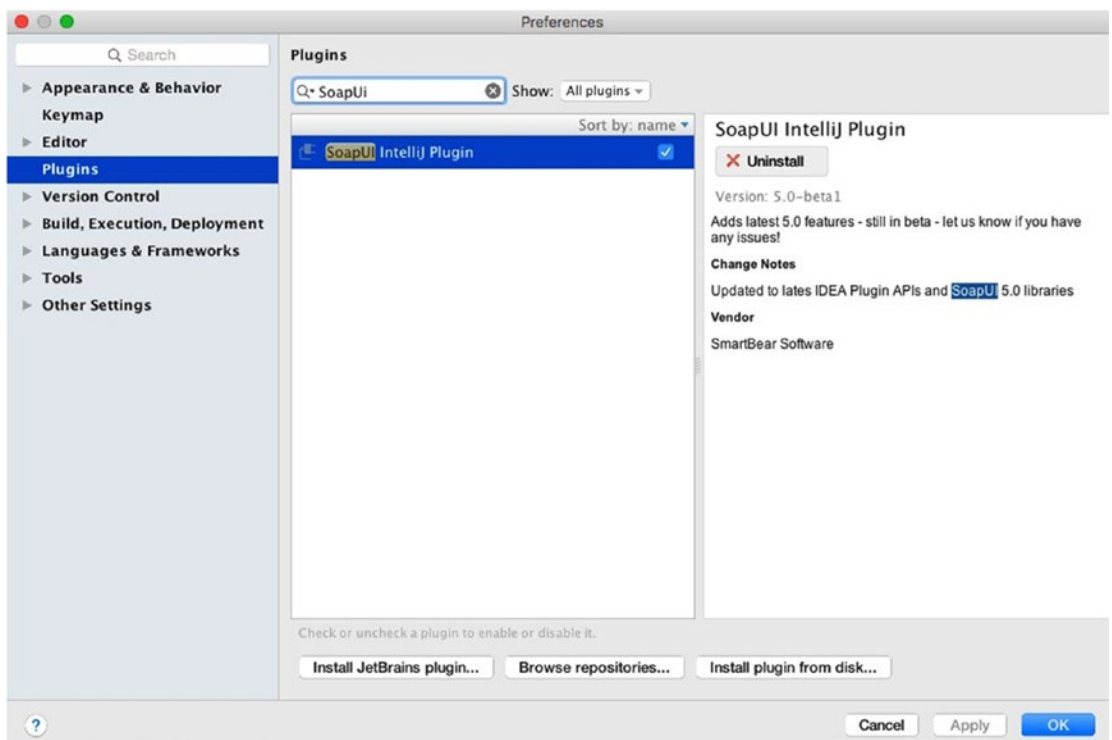


Figure 7-18. The plugins sections of IntelliJ IDEA

The installation of the plugin will require a restart of IntelliJ IDEA. After the restart, two SoapUI tabs will appear. Start SoapUI from the **Tools ► SoapUI ► Start SoapUI menu**, and then open the SoapUI Navigator tab and create a SOAP project. The WSDL location of the SOAP interface is required as is a name for the project. The dialog to create a new project is depicted in Figure 7-19.

¹²<http://www.soapui.org/>.

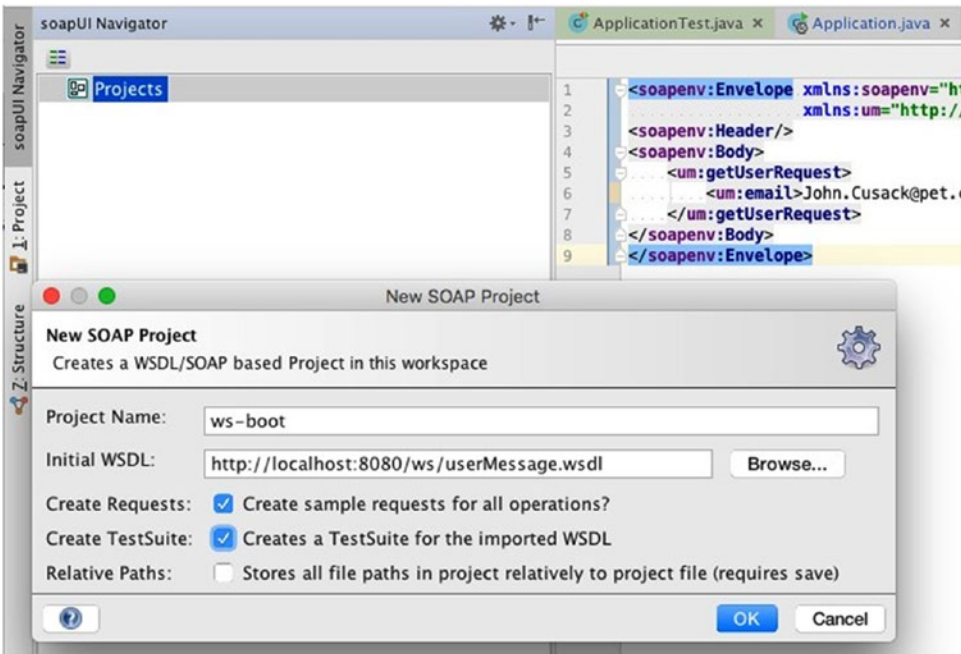


Figure 7-19. The SoapUI plugin interface for creating projects

The project will use the WSDL information to generate a skeleton message requesting information about a user. Since the application uses an email address as a criterion when requesting a user, SoapUI does not know what data to put in the skeleton message, so it just replaces the missing email with a question mark. The developer must replace that question mark with an email (e.g., John.Cusack@pet.com) before sending the request. The application receiving a message containing the user email extracts information about the user and sends the user type back. So replace the question mark with a valid email and click the green arrow button surrounded with red in Figure 7-20, and the returned message will be displayed in the text area on the right.

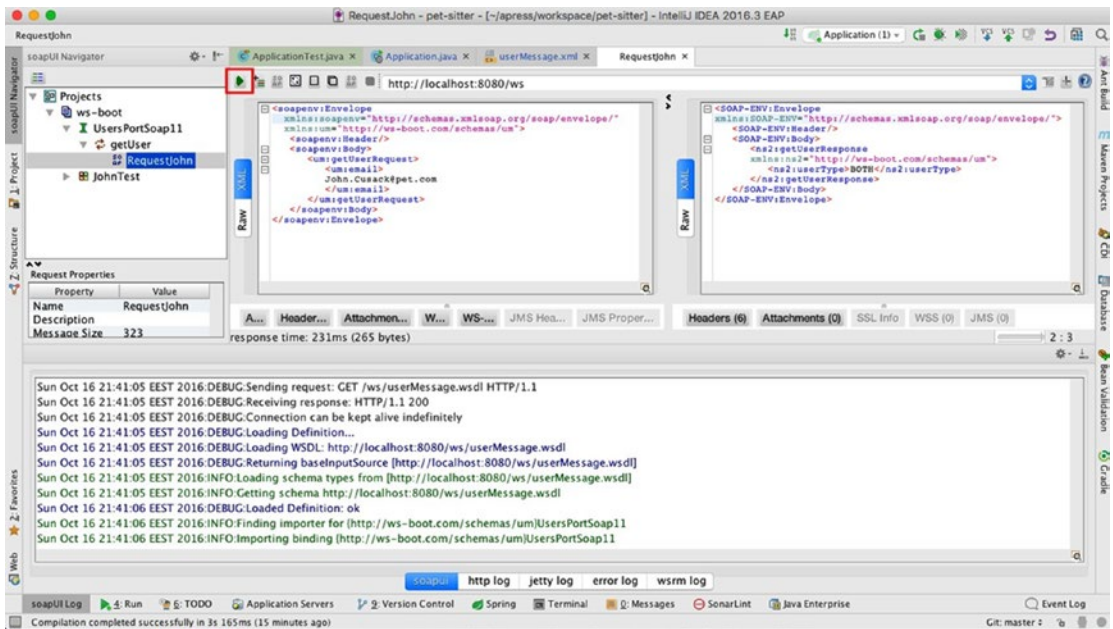


Figure 7-20. The SoapUI Navigator interface for sending a SOAP message

SOAP is the precursor of all Web services interfaces, and although REST is rapidly gaining ground, it is not going away anytime soon, since SOAP 1.2, which was recently introduced, has fixed many of the perceived shortcomings of the technology and is pushing it to new levels of both adoption and ease of use. Also, starting with version 1.2, the acronym SOAP no longer stands for Simple Object Access Protocol from the W3C organization; it is now just the name of the specification. REST uses HTTP/HTTPs protocols, but SOAP has the main advantage of being more flexible than that, since it can use almost every transport protocol to send a message, even JMS. The main disadvantage of SOAP is only the use of XML, which is verbose and takes a lot of time to be parsed. Therefore, even if it is not part of the official exam, this section might be useful to you in your future projects. There is no practical application for this section, but you are welcome to modify the `userMessage.xsd` schema and generate a more complex `UserMessage` object and add some more requests in the `UserMessageEndpoint` class.

Spring REST

REST is an acronym for **RE**presentational **S**tate **T**ransfer. It was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. REST is a lightweight alternative to mechanisms like RPC (Remote Procedure Calls) and Web Services (SOAP, WSDL, etc). REST is an *architecture style* for designing networked (distributed) applications. The idea is that rather than using complex mechanisms such as CORBA, RPC, or SOAP to connect between machines, simple HTTP is used to make calls between machines. RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.

Web applications are not used only by browser clients. Programmatic clients can connect using HTTP (e.g., mobile applications and basically any application that was developed to be able to request and receive data using HTTP as communication protocol). The REST architectural style describes best practices

to expose web services over HTTP; it is used not only as a transport but as an application protocol. The following HTTP specifications are used:

- HTTP verbs are used as actions to execute on the resources (GET,PUT,PATCH, POST, DELETE, HEAD, OPTIONS).¹³ The main REST HTTP methods with their actions are presented in Table 7-2.

Table 7-2. *Message Converters Table*

HTTP method	Purpose	Observation
GET	Read	Reads a resource, does not change it. Therefore can be considered safe . Reading the same resource always returns the same result. Therefore can be considered idempotent .
POST	Create	Used to create a new resource. Neither safe nor idempotent . Two identical POST requests will result in two identical resources being created or errors at application level.
PUT	Update	Most often used for update capabilities. It is not safe , since it modifies the state on the server, but is idempotent (unless subsequent calls of the same PUT request increments a counter within the resource, for example).
DELETE	Delete	Used to delete resources. Not safe, but can be considered idempotent . Because requests to delete a resource that no longer exists will always return a 404 (not found).

- URIs (Uniform Resource Identifiers) are used to identify resources. The resources are conceptually separate from representations. Representations of the resources are returned from the server to the client after a client request (typically JSON or XML). Representations contain metadata information that can be used by the client in order to modify or delete the resource on the server, provided it has permission to do so.
- HTTP response: response codes, body, and headers are used to deliver the state to clients. Clients deliver the state using body contents, request headers, and the URI.

A RESTful architecture is a stateless client-server architecture, so the system is disconnected (loosely coupled). The server might not be available all the time, so operations are asynchronous, since client cannot assume direct connection to the server. Sometimes, requested resources can be cached and some other unknown software and hardware layers can be interposed between client and server. Intermediary servers may improve system scalability and/or security by enabling load-balancing, providing shared caches, and enforcing security policies. In REST communication the objects handled are representations of resources. Representations can link to other resources, thus providing the opportunity of extension and discovery. HTTP headers and status codes are used to communicate results to clients. When it is working with a specific container, a session is used to preserve the state across multiple HTTP requests. Using REST, there is no need for this, because REST communication is stateless, which increases the scalability, since the

¹³Although REST seems strongly connected to HTTP, REST principles can be followed using other protocols too, for example POP, IMAP, and any protocol that uses URL-like paths and supports GET and POST methods.

server does not have to maintain, update, or communicate the session state. Since HTTP is supported by every platform/language, this means that REST communication can be done between quite a wide range of different systems, and thus it is very scalable and definitely interoperable.¹⁴

To analyze contents of the REST requests and responses, the SoapUI plugin (or application) that was introduced in the web services section can be used. Start SoapUI from the **Tools > SoapUI > Start SoapUI menu**, and then open the SoapUI Navigator tab and create a REST project. A pop-up will require a URL matching a REST request. After the Request object is created, the green arrow button can be clicked to send the request, and the server will respond with the representation matching the request. But more about this after Spring Support for REST is covered. In Figure 7-21 you can see how the SoapUI IntelliJ IDEA plugin can be used to create a Rest project.

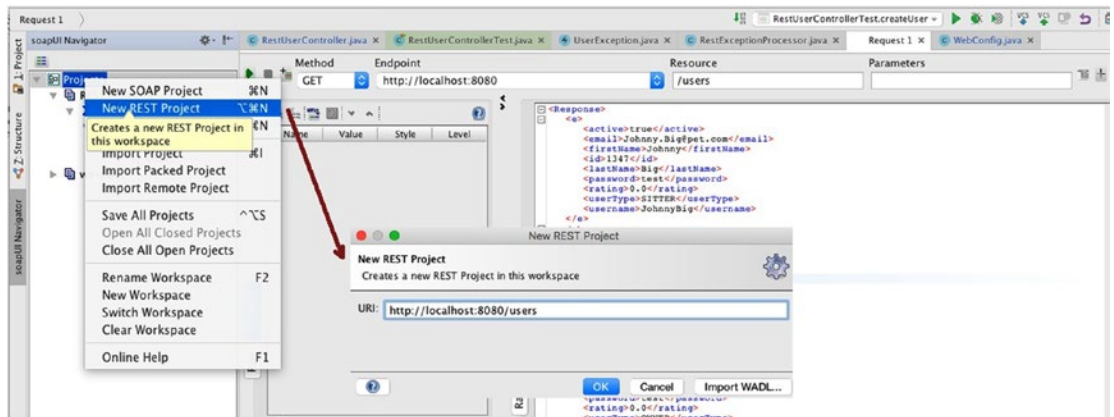


Figure 7-21. The SoapUI Navigator interface for sending a REST request

Spring Support for REST

JAX-RS 2.0 is the latest major release of JAX-RS, which is a Java EE standard for building RESTful applications. JAX-RS focuses on programmatic clients, and there are various implementations (Jersey(RI), RESTEasy, Restlet, CXF). Spring MVC does not implement JAX-RS, but it does provide REST Support starting with version 3.0, offering programmatic client support, but also browser support. The core class for building programmatic clients in Java is the `RestTemplate`. On the server side, the core component for REST support is Spring MVC. Using the `@RequestMapping` annotation introduced in the previous chapter, HTTP REST requests can be mapped to controllers. The REST controllers are special. Although annotated with the stereotype annotation `@Controller`, their methods do not return results that can be mapped to a view, but to a representation. The `DispatcherServlet` is notified that the result of the execution of a controller does not have to be mapped to a view by annotating the method with `@ResponseBody`.

¹⁴See footnote 23.

```

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class RestUserController {

    @Autowired
    UserService userService;

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public @ResponseBody List<User> all() {
        return userService.findAll();
    }
    ...
}

```

In Spring 4.0, the `@RestController` annotation was introduced. This annotation is basically a combination of `@Controller` and `@ResponseBody`, and when used on a class, it assumes that all values returned by the methods in this controller should be bound to the web response body. The `@RestController` annotation is a specialized version of the `@Controller` that does more than just mark a component as being used for REST requests; thus it is not considered a stereotype annotation, and was declared in a different package: `org.springframework.web.bind.annotation`.

```

import org.springframework.web.bind.annotation.RestController;

@RestController
public class RestUserController {

    @Autowired
    UserService userService;

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> all() {
        return userService.findAll();
    }
    ...
}

```

A URL can be mapped to multiple methods if the method attribute is set with a different HTTP method. The example above declared the HTTP method type to be GET. The same URL can be used with a POST method and used to create a user, for example.

```

import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.http.HttpStatus;
@RestController
public class RestUserController {
    ...

    @ResponseStatus(HttpStatus.CREATED)
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public void create(@RequestBody User newUser,

```

```

        @Value("#{request.requestURL}")
        StringBuffer originalUrl, HttpServletResponse response)
            throws Exception {
    if (newUser.getId() != null) {
        throw new UserException("User found with email "
            + newUser.getEmail() + ". Cannot create!");
    }
    User user = userService.create(newUser);
    response.setHeader("Location",
        getLocationForUser(originalUrl, user.getUsername()));
}

protected static String getLocationForUser(StringBuffer url, Object childIdentifier) {
    UriTemplate template = new UriTemplate(url.toString() + "/${username}");
    return template.expand(childIdentifier).toASCIIString();
}
}

```

The `getLocationForUser` method is a utility method used to create the URI of the created resource, which is then placed in the HTTP Response `Location` header. This is really useful, because it generates a URI that is relative to the application context, which is automatically injected by Spring, so no hard-coding is needed. In the previous example, the original request URI could be `localhost:8080/users`, and the `Location` header value would be `localhost:8080/users/johncusack`, which is created by concatenating the original URL with the username value. The URL parameter `$username`¹⁵ is replaced by calling the `template.expand(...)` method. This URI can be used further by the client to request the newly created resource representation.

■ ! You may have noticed in the previous example that the identification key for a user resource is its `username`. Since the `username` is unique in our database and the database ID should not be exposed for security reasons, the `username` is a more suitable choice for an identifier in this case. Also keep in mind that case-sensitive values in the URL might cause trouble, especially when the server application is installed on a case-insensitive operating system like Windows.

Web applications use a collection of status codes to let the client know what happened to a request, whether it was successfully processed and the state of the resource. The most familiar to you is probably the 404 `Not Found` status code that is returned when a requested resource cannot be found. A full list of HTTP status codes can be found on Wikipedia, and you can look at it if you are curious and unfamiliar with HTTP status codes.¹⁶

¹⁵The `$` sign was used as a prefix in the path variable name, to make it obvious that this piece of URI is a variable that will be replaced with an actual value.

¹⁶http://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

With Spring MVC, the status code of a response can be set easily using the `@ResponseStatus` annotation on controller methods. This annotation can receive as value any of the constants defined in the Spring class `HttpStatus`. Table 7-3 contains the most common response statuses used in RESTful applications.

Table 7-3. HTTP status codes table

HTTP Status	HttpStatus constant	Observation
200	OK	Successful GET with returned content
201	CREATED	Successful PUT or POST; location header should contain URI or new resource.
204	NO_CONTENT	Empty response, after successful PUT or DELETE
404	NOT_FOUND	Resource was not found.
403	FORBIDDEN	Server is refusing to respond to the request, because the response is not authorized.
405	METHOD_NOT_ALLOWED	Http method is not supported for the resource identified by the Request URI.
409	CONFLICT	Problems when making changes, when PUT or POST try to save data that already exists and is marked as unique.
415	UNSUPPORTED_MEDIA_TYPE	The server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

Having a controller method annotated with `@ResponseStatus` will stop the `DispatcherServlet` from trying to find a view to render. Thus controller methods that return void will just return a response with an empty body. The methods that return an empty body are usually the DELETE methods:

```
@RestController
public class RestUserController {
    ...
    @ResponseStatus(HttpStatus.NO_CONTENT)
    @RequestMapping(value = "/users/{username}", method = RequestMethod.DELETE)
    public void delete(@PathVariable("username") String username)
        throws UserException {
        User user = userService.findByUsername(username);
        if (user == null ) {
            throw new UserException("No user found for username " + username);
        }

        userService.deleteById(user.getId());
    }
}
```

Starting with Spring 4.3, composed annotations have been introduced that can replace the `@RequestMapping` annotation with a certain HTTP method. They are grouped in the `org.springframework.web.bind.annotation` package. In Table 7-4, Spring annotations and other equivalents are listed.

Table 7-4. HTTP status codes Table

HTTP Method	@RequestMapping	Spring 4.3 Annotation
GET	@RequestMapping(method=RequestMethod.DELETE)	@GetMapping
POST	@RequestMapping(method=RequestMethod.POST)	@PostMapping
DELETE	@RequestMapping(method=RequestMethod.DELETE)	@DeleteMapping
PUT	@RequestMapping(method=RequestMethod.PUT)	@PutMapping

■ ! The difference between @GetMapping and @RequestMapping is that the former does not support the consumes attribute of @RequestMapping. (This attribute will be introduced a little bit later in this section.)

If for some reason @RestController cannot be used, in Spring MVC, type org.springframework.http.ResponseEntity can be used as the return value from a @Controller method to tell the dispatcher that the response is to be serialized and forwarded to the client instead of being resolved to a view:

```
@Controller
public class UserController {

    @Autowired
    UserService userService;

    @Autowired
    PetService petService;

    @RequestMapping(value = "/pets/{username}", method = RequestMethod.POST)
    public ResponseEntity<String> createPet(@PathVariable("username") String username,
        @RequestBody Pet pet,
        @Value("#{request.requestURL}") StringBuffer url)
        throws UserException {
        User owner = userService.findByUsername(username);
        if (owner == null) {
            throw new UserException("User not found with username " + username);
        }
        petService.save(pet);
        HttpHeaders headers = new HttpHeaders();
        headers.add("Location", getLocationForPet(url, pet.getRfid()));
        return new ResponseEntity<>(headers, HttpStatus.CREATED);
    }

    /**
     * Determines URL of pet resource based on the full URL of the given request,
     * appending the path info with the given childIdentifier using a UriTemplate.
     */
    protected static String getLocationForPet(StringBuffer url, Object petRfid) {
        UriTemplate template = new UriTemplate(url.append("/{petid}").toString());
        return template.expand(petRfid).toASCIIString();
    }
}
```

In the previous code snippet, a Pet record is created for user identified by \$username path variable. After the object is created, a response is sent with the location where the Pet record can be accessed via REST requests. So if the original URI was `http://localhost:8080/pets/johncusack`, the returned result is a `ResponseEntity` that contains the following URI: `http://localhost:8080/pets/johncusack/11223344`, with `11223344` being the Pet RFID value.

Exception Handling

Sometimes REST requests cannot be resolved successfully. Perhaps the client requested a resource no longer available, or some action requested by the client generated an error on the server. In this case, the `@ResponseStatus` can be used on controller exceptions handler methods to give the client a basic idea of why the request was not processed correctly.

```
@RestController
public class RestUserController {
...
    /**
     * Maps IllegalArgumentException to a 404 Not Found HTTP status code.
     * Requested resource was not found
     */
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler({IllegalArgumentException.class})
    public void handleNotFound() {
        // just return empty 404
    }

    /**
     * Maps DataIntegrityViolationException to a 409 Conflict HTTP status code.
     * Creating a resource was not possible
     */
    @ResponseStatus(HttpStatus.CONFLICT)
    @ExceptionHandler({DataIntegrityViolationException.class})
    public void handleAlreadyExists() {
        // just return empty 409
    }
...
}
```

Another method to process REST request-related exceptions is to define a specialized exception class for a REST controller and a specialized component to intercept those types of exceptions and treat them in a certain way. For example, considering a class called `UserException` that is thrown every time a method in the `RestUserController` is not executed correctly, the class defined in the following code snippet will intercept such exceptions and transform them into JSON error objects that the client can interpret easily. This behavior is configured using the `@ControllerAdvice` annotation, which is a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning. Classes annotated with this annotation typically declare one or more methods annotated with `@ExceptionHandler` that will take care of the exception handling.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
```



```
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class RestExceptionHandler {

    @ExceptionHandler
    @ResponseStatus(value = HttpStatus.NOT_FOUND)
    @ResponseBody
    public JsonError exception(UserException ex) {
        return new JsonError(ex.getMessage());
    }
}
```

In the previous code sample, the exception type is used as a parameter in the handler method, so that only exceptions of that type will be intercepted and treated. Also, catching custom exceptions is useful, because they usually contain specific information related to what went wrong. Similar restrictions can be implemented also by configuring the `@ControllerAdvice` annotation. The scope for a controller advice can be limited to exceptions thrown by methods of controllers declared in a specific set of packages by setting the `basePackages` attribute.

```
...
@ControllerAdvice(basePackages = {"com.ps.web", "com.ps.rest"})
public class RestExceptionHandler {

    @ExceptionHandler
    @ResponseStatus(value = HttpStatus.NOT_FOUND)
    @ResponseBody
    public JsonError exception(Exception ex) {
        return new JsonError(ex.getMessage());
    }
}
```

The scope of a controller advice can also be limited to a class or hierarchy of classes, by setting the `basePackageClasses` attribute.

```
...
@ControllerAdvice(basePackageClasses = RestExceptionHandler.class)
public class RestExceptionHandler {

    @ExceptionHandler
    @ResponseStatus(value = HttpStatus.NOT_FOUND)
    @ResponseBody
    public JsonError exception(UserException ex) {
        return new JsonError(ex.getMessage());
    }
}
```

HTTP Message Converters

The `@ResponseBody` introduced earlier is also used to facilitate understanding of the REST message format between client and server. As mentioned at the beginning of the section, the resource representation can have different formats: XML, JSON, HTML, etc. The client must know the format to use, or request a resource with a representation it understands from the server. Representations are converted to HTTP requests and from HTTP responses by implementations of the `org.springframework.http.converter.HttpMessageConverter<T>` interface. Message converters are automatically detected and used by Spring in applications configured with `<mvc:annotation-driven/>` or `@EnableWebMvc`. In the code sample for this chapter, the representations are in JSON format, so `MappingJackson2HttpMessageConverter` is used. Spring comes with a default wide list of supported message converters, but in case of need, a developer can provide his own implementation of the `HttpMessageConverter<T>`. Here is a list of the most commonly used message converters and the datatype handled:

Table 7-5. *Message Converters Table*

Message Converter	Data Type	Observation
<code>StringHttpMessageConverter</code>	<code>text/plain</code>	
<code>MappingJackson2HttpMessageConverter</code>	<code>application/*+json</code>	Only if Jackson 2 is present on the classpath
<code>AtomFeedHttpMessageConverter</code>	<code>application/atom+xml</code>	Only if Rome is present on the classpath
<code>RssChannelHttpMessageConverter</code>	<code>application/rss+xml</code>	Only if Rome is present on the classpath
<code>MappingJackson2XmlHttpMessageConverter</code>	<code>application/*+xml</code>	Only if Jackson 2 is present on the classpath

As the name of the annotation so obviously implies, `@ResponseBody` is applied to the response, and its presence on a controller method tells Spring that the response should be processed with an HTTP message converter before being sent to the client. If the method requires an argument that needs to be processed by the same HTTP converters, then the parameter must be annotated with `RequestBody`. Take a controller method resolving a PUT request. This method receives a user representation from the client that must be converted to a `User` instance before its state is updated into the database.¹⁷ This method also returns the representation of the updated instance, so it needs both of the annotations present.

```
import org.springframework.http.MediaType;
...
@RestController
public class RestUserController {
    @ResponseStatus(HttpStatus.OK)
    @RequestMapping(value = "/users/{username}",
        method = RequestMethod.PUT,
        consumes = MediaType.APPLICATION_JSON_UTF8_VALUE
```

¹⁷Of course, the preferred approach is to do an update operation, then do a GET request to extract the saved data, but for teaching purposes, the two were combined.

```

        produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
public User updateByEmail(
    @PathVariable("$username") String username,
    @RequestBody User newUser) throws UserException {
    User user = userService.findByUsername(username);
    if (user == null) {
        throw new UserException("User not found with username " + username);
    }
    copyProperties(newUser, user);
    userService.update(user);
    return user;
}
...
}

```

In the previous example, the `consumes` and `produces` annotation attributes of the `@RequestMapping` were used. These two attributes are used to narrow the primary mapping for a request.

The `consumes` attribute defines the consumable media types of the mapped request (defined on the server), and the value of the `Content-Type` header (defined on the client side) must match at least one of the values of this property in order for a method to handle a specific REST request.

The `produces` attribute defines the producible media types of the mapped request, narrowing the primary mapping, and the value of the `Accept` header (on the client side) must match at least one of the values of this property in order for a method to handle a specific REST request.

■ ! The approach presented above to update a user is far from what you would use in practice. It was designed especially to prove how the two attributes work and how they can be used together. The example works, though, and you can test it by running the `15-ps-ws-rest-practice` project, which is to be used when testing knowledge acquired after traversing this section.

Spring MVC Configuration for RESTful Applications

A Spring Web RESTful application is basically a Spring MVC application, so the configuration is the same as presented in **Chapter 6: Spring Web**, no matter whether XML or Java Configuration is used. To support a custom HTTP message converter implementation, the dependency must be added to the classpath of the application, and a special bean must be declared in the configuration. In the following code snippet, the configuration for a Spring Web RESTful application is depicted, and it contains a bean of type `MappingJackson2HttpMessageConverter` that will be automatically used by Spring to convert objects to JSON representations and vice versa.

```

Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.ps.web", "com.ps.exs"})
public class WebConfig extends WebMvcConfigurerAdapter {

    //Declare our static resources.
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/images/**")

```

```

        .addResourceLocations("/images/")
        // cache them to speed up page loading
        .setCachePeriod(31556926);
    }
    @Override
    public void configureDefaultServletHandling(
        DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

    //One simple view, to confirm the application is up
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("welcome");
    }

    @Bean
    InternalResourceViewResolver getViewResolver(){
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/");
        resolver.setSuffix(".jsp" );
        return resolver;
    }

    @Bean
    public MappingJackson2HttpMessageConverter
        mappingJackson2HttpMessageConverter() {
        MappingJackson2HttpMessageConverter
            mappingJackson2HttpMessageConverter =
                new MappingJackson2HttpMessageConverter();
        // when client is a browser JSON response is displayed indented
        mappingJackson2HttpMessageConverter.setPrettyPrint(true);
        //set encoding of the response
        mappingJackson2HttpMessageConverter.setDefaultCharset
            (StandardCharsets.UTF_8);
        mappingJackson2HttpMessageConverter.setObjectMapper(objectMapper());
        return mappingJackson2HttpMessageConverter;
    }

    @Bean
    public ObjectMapper objectMapper() {
        ObjectMapper objMapper = new ObjectMapper();
        objMapper.enable(SerializationFeature.INDENT_OUTPUT);
        objMapper.setSerializationInclusion
            (JsonInclude.Include.NON_NULL);
        return objMapper;
    }
}

```

```

@Override
public void configureMessageConverters(
    List<HttpMessageConverter<?>> converters) {
    super.configureMessageConverters(converters);
    converters.add(mappingJackson2HttpMessageConverter());
}
}

```

Using RestTemplate to Test RESTful Applications

A RESTful application can be accessed by any type of client that supports creating the type of request supported by the application:¹⁸ browsers, mobile applications, desktop applications, etc. The `RestTemplate`¹⁹ is Spring's central class for synchronous client-side HTTP access. This class provides a wide set of methods for each HTTP method that can be used to access RESTful services. A correspondence between HTTP methods and `RestTemplate` methods that can be used to access REST Services is depicted in Figure 7-22.

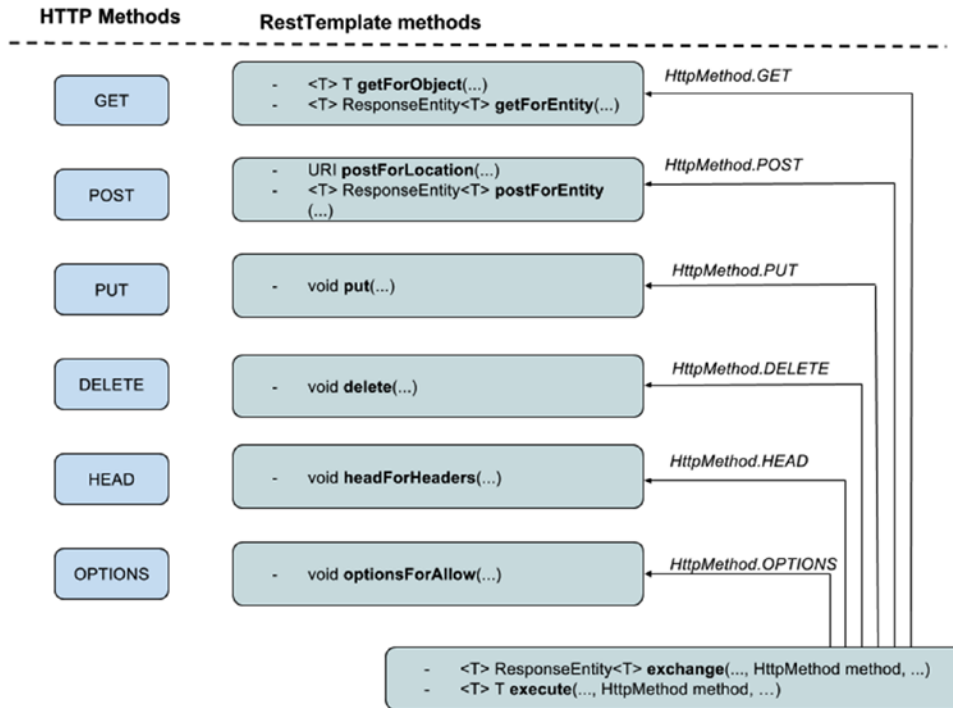


Figure 7-22. *RestTemplate API to HTTP methods correspondence*

¹⁸HTTP requests in our case, but as mentioned previously, REST can be used with other protocols as well.

¹⁹JavaDoc for this class can be found here: <http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>.

As you can see, the `execute` and `exchange` methods can be used for any type of REST calls as long as the HTTP method is given as a parameter for the methods. All methods are polymorphic,²⁰ and using one or another depends on the requirements and the developer's preferences. URI instances are returned to identify resources, and `RestTemplate` methods support URI templates. So the following two calls are identical:

```
//using URI Template
String url = "http://localhost:8080/users/{username}";
User user = restTemplate.getForObject(url, User.class, "johncusack");

// using URI
String url = "http://localhost:8080/users/johncusack";
User user = restTemplate.getForObject(url, User.class);
```

■ ! Just as a reminder, the path variable `username` is valid, and it works as expected: the `$` character is allowed to be part of a path variable name. This peculiar name for path variables was chosen to make it really obvious to you, the developer, that this part of the URL is a variable that must be replaced with an actual value when making a REST request.

The `username` was chosen as a key to identify users, because it is unique in the database, and even if the database is truncated and reloaded, the username will not change. The database ID should not be exposed for security reasons, and it is also subject to change in case the database is truncated and reloaded while the application is running.

The `execute` method can also be given an `org.springframework.web.client.RequestCallback` implementation as a parameter that will tell the `RestTemplate` what to do with the request before sending it to the server. Considering this, a GET Request for a `User` instance with `username=johncusack` could be written with the `exchange` method like this:

```
String url = "http://localhost:8080/users/{username}";
Person person = restTemplate.execute(url, HttpMethod.GET,
    new RequestCallback() {
        @Override
        public void doWithRequest(ClientHttpRequest request)
            throws IOException {
            HttpHeaders headers = request.getHeaders();
            headers.add("Accept", MediaType.APPLICATION_JSON_UTF8_VALUE);
            System.out.println("Request headers = " + headers);
        }
    }, new HttpResponseMessageExtractor<User>(User.class,
    restTemplate.getMessageConverters())
    , new HashMap<String, Object>() {{
    put("username", "johncusack");
    }});
```

²⁰Multiple methods with the same name but different signatures are provided. `RestTemplate` <http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html>.

Objects passed to and returned from the methods `getForObject()`, `postForLocation()`, and `put()` are converted to HTTP requests and from HTTP responses by `HttpMessageConverters`.

The exchange method uses an `HttpEntity` object to encapsulate request headers and use it as a parameter. The method returns a `ResponseEntity` object containing the result of a REST request, and its body is automatically converted using the registered `HttpMessageConverter` implementation.

```
import org.springframework.http.ResponseEntity;
...

public class RestUserControllerTest {
    public void editUser() {
        String url = "http://localhost:8080/users/{username}";
        User user = new User();
        user.setEmail("MissJones@pet.com");
        user.setUsername("jessicajones");
        //set other properties
        ..

        final HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON_UTF8);
        final HttpEntity<User> userRequest = new HttpEntity<>(user, headers);
        ResponseEntity<User> responseEntity = restTemplate.
            exchange(url, HttpMethod.PUT, userRequest, User.class,
                "jessicajones");

        User editedUser = responseEntity.getBody();
        assertNotNull(editedUser);
        assertEquals("MissJones@pet.com", editedUser.getEmail());
    }
}
```

Message converters are automatically detected and used by Spring in applications configured with `<mvc:annotation-driven/>` or `@EnableWebMvc`. In the code sample for this chapter, the representations are in JSON format, so `MappingJackson2HttpMessageConverter` is used. And because the message is supported by default, the `HttpMessageConverterExtractor<T>` is not necessary in the previous example. Also, if no `Accept` header is specified, all those supported by Spring will be considered, so in this case, `RequestCallback` becomes unnecessary too, and we can stick to the simpler `restTemplate.getForObject` method mentioned in the previous code snippet, because even in production applications, you will rarely need anything else.

The application contained in the `15-ps-ws-rest-practice` is a Spring Web RESTful application that can be used to test your understanding of Spring support components for REST applications. The `RestUserController` class provides controller methods that resolve REST requests. The client is represented by the `RestUserControllerTest` class. This test class has a `RestTemplate` injected in it that can be used to test all REST operations with users exposed by the `RestUserController` controller. The web application

has no context, and the REST Services exposed by the controller are available using URIs like `http://localhost:8080/users/*`. The `RestUserControllerTest` class contains methods to test REST requests using the four main HTTP methods:

- **GET:** method used to retrieve a representation of a resource; might have length restrictions depending on server settings and client used.²¹ When resource is not found, a **404 (Not found)** status code is returned, **200(OK)** otherwise. In Figure 7-23, the GET request and response contents are depicted. A REST GET request can be sent with a `RestTemplate` instance: all that is needed is the REST URI and a username to identify the resource. The path variable was named `$username`, prefixed with `$` to make the parametrized piece of the URI really obvious. The following code snippet depicts the test method performing a GET request for a user resource that has username equal to “`johncusack`”.

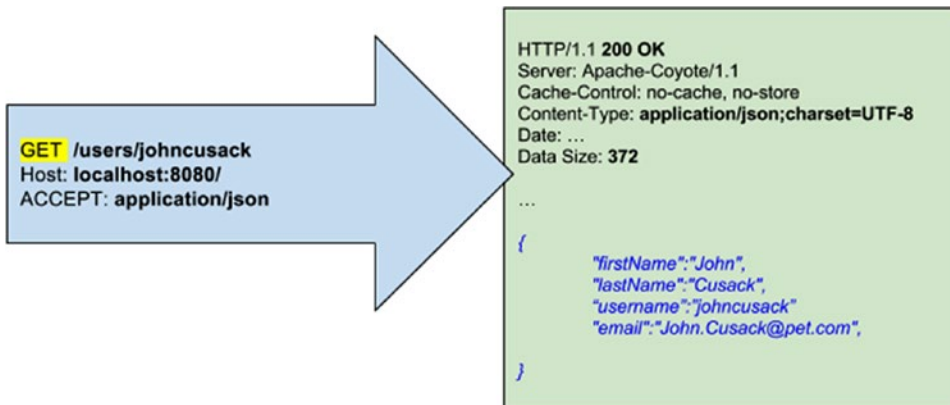


Figure 7-23. GET request and response example

```
public class RestUserControllerTest {
    private static final String GET_PUT_DEL_URL =
        "http://localhost:8080/users/{$username}";
    private RestTemplate restTemplate = null;
    @Before
    public void setUp() {
        restTemplate = new RestTemplate();
    }

    //Test GET by username
    @Test
    public void findByUsername() {
        User user = restTemplate.getForObject(GET_PUT_DEL_URL,
            User.class, "johncusack");
        assertNotNull(user);
        assertEquals("John.Cusack@pet.com", user.getEmail());
    }
}
```

²¹Most web servers have a limit of 8192 bytes (8KB), which is usually configurable in the server configuration. As for the client, the HTTP 1.1 specification warns about URI lengths bigger than 255 bytes, because older client or proxy implementations might not properly support them.

- **POST**: method used to create a new resource. When the resource being created requires a parent that does not exist, a **404 (Not found)** status code is returned.²² When an identical resource already exists, a **409 (Conflict)** status code is returned. When the resource was created correctly, a **201 (Created)** status code is returned. In Figure 7-24, the POST request and response contents are depicted.

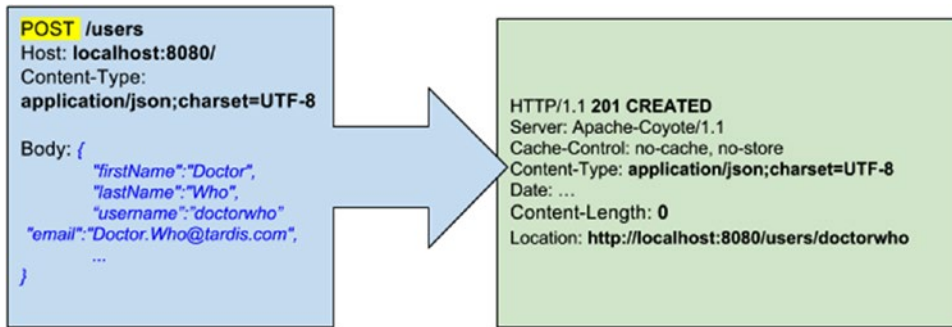


Figure 7-24. POST request and response example

A REST POST request can be sent with a `RestTemplate` instance: all that is needed is the REST URI and user object. The URI is identical to the one used to retrieve a user resource, but the HTTP method is different.

```
import org.springframework.http.*;
...
public class RestUserControllerTest {
    private static final String GET_PUT_DEL_URL =
        "http://localhost:8080/users/{username}";
    private static final String GET_POST_URL = "http://localhost:8080/users";
    private RestTemplate restTemplate = null;
    @Before
    public void setUp() {
        restTemplate = new RestTemplate();
    }

    //Test POST
    @Test
    public void createUser() {
        User user = new User();
        user.setEmail("Doctor.Who@tardis.com");
        user.setUsername("DoctorWho");
        ...
        final HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
    }
}
```

²²e.g., a POST request to create a pet contains a username of the owner. If an owner with the specified username does not exist in the database, a 404 should be used to notify the client of the failure.

```

        final HttpEntity<User> crRequest = new HttpEntity<>(user, headers);
        URI uri = restTemplate.postForLocation(GET_POST_URL,
            crRequest, User.class);
        assertNotNull(uri);
        assertTrue(uri.toString().contains("doctorwho"));

        // test insertion
        User newUser = restTemplate.getForObject(uri, User.class);
        assertNotNull(newUser);
        assertNotNull(newUser.getId());
    }
}

```

In this case, the method `postForLocation` is used, because the successful creation of the resource must be tested afterward. The `HttpHeaders` object is used to set headers of the request, such as the media type of the message, so Spring will know what message converter to use to convert the JSON representation into the user instance that needs to be saved.

- **PUT**: updates an existing resource or creates it with a known destination URI. The URI of a resource contains an identifier for that resource. When a PUT request refers to an existing resource, the resource is updated; otherwise, a new resource with the identifier from the URI and the contents in the request body is created. When the resource being updated requires a parent that does not exist,²³ or if the resource requested to be updated does not exist, a **404 (Not found)** status code is returned.

When the resource is updated correctly and nothing is returned as a response body, a **204 (No content)** response status is returned.

When the resource does not exist, PUT acts like a POST and creates it, and the location of the resource is returned. The response status in this case is **201 (CREATED)**.

²³A REST service that allows you to change the owner of a pet needs the username of that owner. If a username that is not present in the database is specified, the user should be notified that the resource does not exist by returning a 404 HTTP response code.

In Figure 7-25, the PUT request and response contents are depicted.

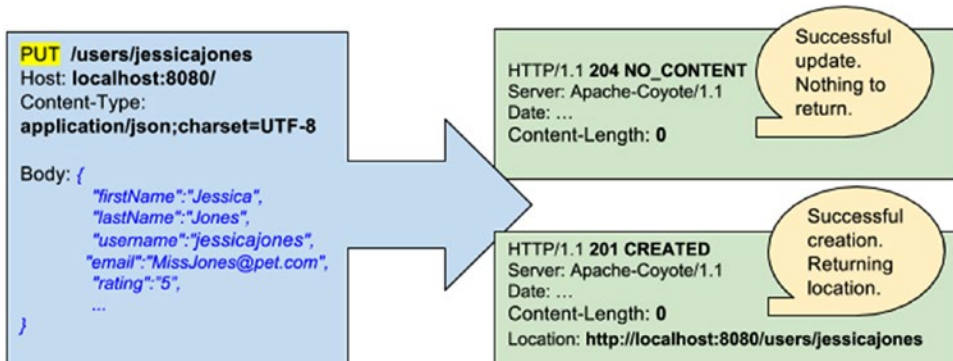


Figure 7-25. PUT request and response example

A REST PUT request can be sent with a RestTemplate instance: all that is needed is the REST URI and user object. The URI is identical to the one used to retrieve a user resource, but the HTTP method is different.

```

import org.springframework.http.*;
...
public class RestUserControllerTest {
    private static final String GET_PUT_DEL_URL =
        "http://localhost:8080/users/{username}";
    private static final String GET_POST_URL = "http://localhost:8080/users";
    private RestTemplate restTemplate = null;
    @Before
    public void setUp() {
        restTemplate = new RestTemplate();
    }

    // Test PUT
    @Test
    public void editUser() {
        User user = new User();
        user.setEmail("MissJones@pet.com");
        user.setUsername("JessicaJones");
        ...

        final HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON_UTF8);
  
```

```

        final HttpEntity<User> userRequest = new HttpEntity<>(user, headers);
        ResponseEntity<User> responseEntity = restTemplate.exchange
            (GET_PUT_DEL_URL, HttpMethod.PUT, userRequest, User.class,
             "JessicaJones");

        User editedUser = responseEntity.getBody();
        assertNotNull(editedUser);
        assertEquals("MissJones@pet.com", editedUser.getEmail());
    }
}

```

The exchange method was used here just for teaching purposes, but RestTemplate provides put methods with various signatures that are much easier to use.

- **DELETE:** deletes a resource. When the resource being deleted does not exist, a **404 (Not found)** status code is returned. When the resource was deleted correctly, a **200 (OK)** status code is returned. In Figure 7-26, the DELETE request and response contents are depicted.

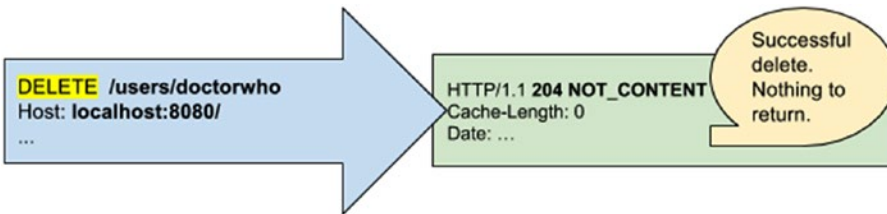


Figure 7-26. DELETE request and response example

A REST DELETE request can be sent with a RestTemplate instance: all that is needed is the REST URI and username. The URI is identical to the one used to retrieve a user resource, but the HTTP method is different.

```

import org.springframework.http.*;
...
public class RestUserControllerTest {
    private static final String GET_PUT_DEL_URL =
        "http://localhost:8080/users/{username}";
    private static final String GET_POST_URL = "http://localhost:8080/users";
    private RestTemplate restTemplate = null;
    @Before
    public void setUp() {
        restTemplate = new RestTemplate();
    }

    // Test DELETE
    @Test
    public void deleteUser() {
        restTemplate.delete(GET_PUT_DEL_URL, "doctorwho");
    }
}

```

The delete response has an empty body if the operation succeeds. The only way to test the success of a deletion is to request the resource with URI equal to `http://localhost:8080/users/doctorwho` and expect a failure or a response with 404 (Not Found) status to be returned, because the resource has already been deleted.

A Spring RESTful application can be tested without deploying it on a server by declaring a mock restful server and using mock dependencies, so the REST requests can be tested in isolation. The class that provides this is part of the Spring Test library and is called `org.springframework.test.web.client.MockRestServiceServer`. A `RestTemplate` will be bound to this mock server, and this is where all requests will be sent. Not being an actual server, the full URL for a request is unnecessary, only the request mapping defined on the controller method. In every test method, the following steps are traversed:

1. The mock server first defines what request it expects to receive, how many times, and how it will answer.
2. The `RestTemplate` method will be called with the request mapping URL chunk (e.g., `"/users"`)
3. The `asserts` method is performed to make sure the `RestTemplate` request actually received the response defined at step 1.

The `StandaloneRestUserControllerTest` class in project `15-ps-ws-rest-solution` can be used to test a RESTful application in a test mock context. A static internal class named `TestConfig` is defined to provide mock dependencies to the test context. `JMock` and `Mockito` components are used to provide mock objects so that the REST communication (request and response) can be tested in isolation.

```
import org.mockito.Mockito;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.client.MockRestServiceServer;
import org.springframework.web.client.RestTemplate;

import static org.junit.Assert.*;
import static org.springframework.test.web.client.match.
    MockRestRequestMatchers.method;
import static org.springframework.test.web.client.match.
    MockRestRequestMatchers.requestTo;
import static org.springframework.test.web.client.ExpectedCount.once;
import static org.springframework.test.web.client.response.
    MockRestResponseCreators.withSuccess;
...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {WebConfig.class,
    StandaloneRestUserControllerTest.TestConfig.class})
@WebAppConfiguration
public class StandaloneRestUserControllerTest {
```

```

@Configuration
static class TestConfig {
    @Bean
    UserService userService() {
        return Mockito.mock(UserService.class);
    }
}

private RestTemplate restTemplate = null;

private MockRestServiceServer server = null;

@Before
public void setup() {
    restTemplate = new RestTemplate();
    server = MockRestServiceServer.bindTo(restTemplate).build();
}

@Test
public void findByUsername() {
    //step 1
    server.expect(once(), requestTo("/users/johncusack"))
        .andExpect(method(HttpMethod.GET))
        .andRespond(withSuccess("{\n" +
            "  \"id\" : 1575,\n" +
            "  \"username\" : \"johncusack\",\n" +
            "  \"firstName\" : \"John\",\n" +
            "  \"lastName\" : \"Cusack\",\n" +
            "  \"password\" : \"test\",\n" +
            "  \"userType\" : \"OWNER\",\n" +
            "  \"email\" : \"John.Cusack@pet.com\",\n" +
            "  \"rating\" : 0.0,\n" +
            "  \"active\" : true\n" +
            "}", MediaType.APPLICATION_JSON_UTF8));

    //step 2
    User user = restTemplate.getForObject("/users/{username}",
        User.class, "johncusack");
    //step 3 assertNotNull(user);
    assertEquals("John.Cusack@pet.com", user.getEmail());
    assertEquals(UserType.OWNER, user.getUserType());
}
...
}

```

Advantages of REST

- REST is simple.
- REST is widely supported.
- Resources can be represented in a wide variety of data formats (JSON, XML, etc.).

- You can make good use of HTTP cache and proxy server to help you handle high load and improve performance.
- It reduces client/server coupling.
- Browsers can interpret representations.
- Javascript can use representations.
- A rest service can be consumed by applications written in different languages.
- It makes it easy for new clients to use a RESTful application, even if the application was not designed specifically for them.
- Because of statelessness of REST systems, multiple servers can be behind a load-balancer and provide services transparently, which means increased scalability.
- Because of the uniform interface, little or no documentation of the resources and basic operations API is necessary.
- Using REST does not imply specific libraries at client level in order to communicate with the server. With REST, all that is needed is a network connection.

REST services can be secured, but since the interaction between client and server is stateless, credentials have to be embedded in every request header. Basic authentication is the easiest to implement without additional libraries (HTTP Basic, HTTP Digest, or XML-DSIG or XML-Encryption), but it guarantees the lowest level of security. Basic authentication should never be used without TLS (formerly known as SSL) encryption, because the credentials can be easily decoded otherwise. In Figure 7-27, you can see how basic authentication is used when a client communicates with a RESTful application that requires basic authentication.

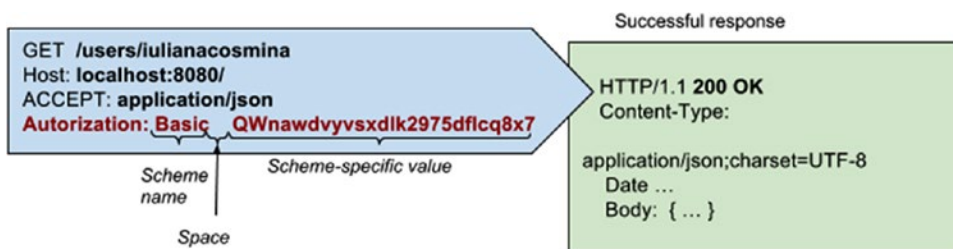


Figure 7-27. Basic authentication when RESTful systems are used

Practice Section

The `15-ps-ws-rest-practice` is an incomplete project; there are annotations and method bodies missing. To test your understanding of implementing Spring RESTful applications and testing them, you can try completing it. There are seven TODO tasks numbered from 55 to 61, and if you need inspiration or a confirmation that your solution is correct, you can compare it with the proposed solution that you can find in project `15-ps-ws-rest-solution`.

Task TODO 55, located in the `RestUserController` class, requires you to complete the configuration of this class to register this class as a controller that handles REST requests.

Task TODO 56, located in the `RestUserController` class, requires you to complete the configuration of the `create` method, so that it can handle an HTTP POST request and create a `User` instance. If you want to test the method right away, start the application with `gradle appRun` in the command line or by

executing the `appStart` from the Gradle Projects tab under this module node and solve task `TODO 59` in the `RestUserControllerTest` class to use the `restTemplate` object to send an HTTP POST request.

Task `TODO 57`, located in the `RestUserController` class, requires you to complete the configuration of the `update` method so that it can handle an HTTP PUT request and update a `User` instance. If you want to test the method right away, start the application with `gradle appRun` in the command line or by executing the `appStart` from the Gradle Projects tab under this module node and solve task `TODO 60` in the `RestUserControllerTest` class to use the `restTemplate` object to send an HTTP post request.

Task `TODO 58`, located in the `RestUserControllerTest` class, requires you to complete the test method body to make a REST call using `restTemplate` to request a user instance with `username="johncusack"`.

Task `TODO 61`, located in the `RestUserControllerTest` class, requires you to complete the test method body to make a REST call using `restTemplate` to request deletion of the user instance with `username="doctorwho"`.

To test the result of your operations, you can run the test method `getAll()` from class `RestUserControllerTest`, which will print in the console all the users in the database. Or you can access the URIs right from the browser, since the browser knows how to render the JSON response properly. Another solution if you do not want to leave the editor is to use the SoapUI IntelliJ Idea plugin mentioned at the beginning of the section. Just create a project and add requests with the proper URIs. In Figure 7-28,

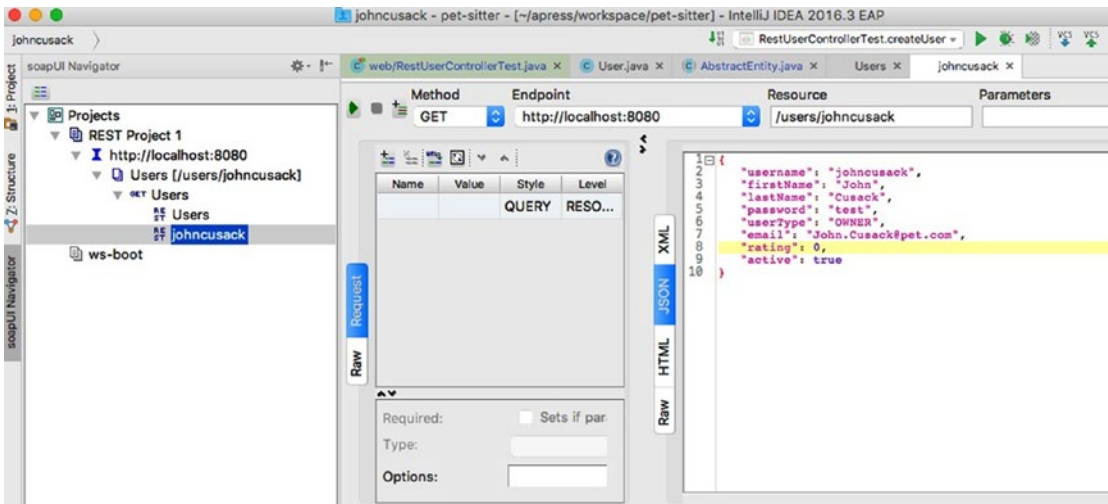


Figure 7-28. The SoapUI `http://localhost:8080/users/johncusack` REST request and response

the request for the user with username `johncusack` is depicted. After clicking the green arrow button, the response is depicted in the textfield on the right.

RESTful Spring Application with Spring Boot

The `15-ps-ws-rest-practice` application is configured using plain Spring MVC and has even a view resolver configured, so that if the application is started successfully, a web page becomes accessible at location `http://localhost:8080/`. The simplest way to configure a RESTful Spring application is using Spring Boot, because the simplest RESTful application will need fewer than 20 lines of code. The module `15-ps-rest-boot-sample` contains a Spring Boot RESTful application that provides the same HTTP methods as the Spring MVC application. The only difference is, to keep things simple, a simple version of the `User` class is used, and no JPA is present. In Figure 7-29, the `15-ps-rest-boot-sample` application structure is depicted.

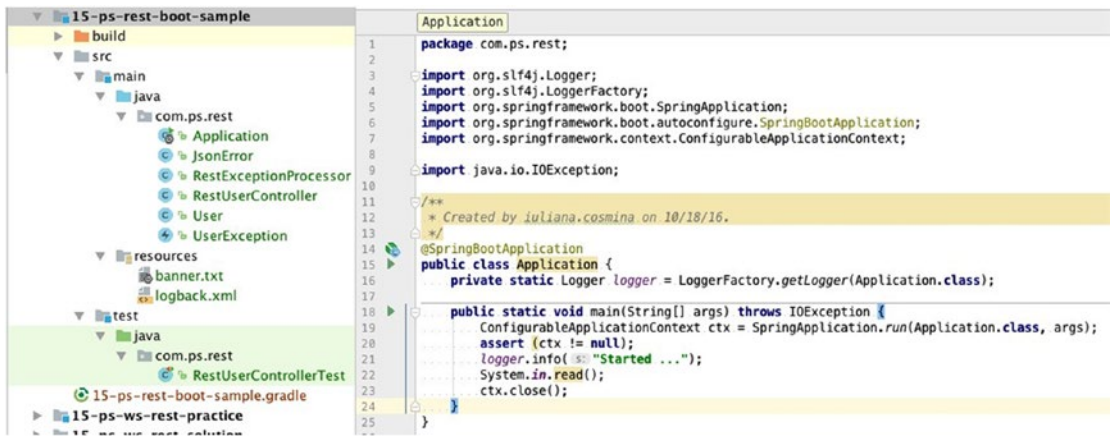


Figure 7-29. The 15-ps-rest-boot-sample Spring RESTful Boot application structure

The Application class is the entry point of the application. Run this class as an application in IntelliJ IDEA, and the REST services will become available.

```
package com.ps.rest;

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
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("Started ...");
        System.in.read();
        ctx.close();
    }
}
```

The `RestUserControllerTest` contains methods to test all the REST requests exposed by the `RestUserController` class. The `RestUserController` class is identical to the one introduced earlier; the URIs are the same. The only difference is that it gets the user instances from a map structure.

...

```
@RestController
public class RestUserController {

    // replacement for JPA UserService
    private Map<String, User> userService = new HashMap<>();

    // create some users so we can test the controller
    @PostConstruct
    public void init() {
        User user = new User("johncusack", "John.Cusack@pet.com", 5d, true);
        userService.put(user.getUsername(), user);
        user = new User("jessicajones", "Jessica.Jones@pet.com", 4d, true);
        userService.put(user.getUsername(), user);
    }

    @RequestMapping(value = "/users", method = RequestMethod.GET)
    public List<User> all() {
        return new ArrayList<>(userService.values());
    }

    @ResponseStatus(HttpStatus.CREATED)
    @RequestMapping(value = "/users", method = RequestMethod.POST)
    public void create(@RequestBody @Valid User newUser,
        @Value("#{request.requestURL}")
        StringBuffer originalUrl, HttpServletResponse response)
        throws UserException {
        if (userService.containsKey(newUser.getUsername())) {
            throw new UserException("User found with email "
                + newUser.getEmail() + ". Cannot create!");
        }
        userService.put(newUser.getUsername(), newUser);
        response.setHeader("Location", getLocationForUser(originalUrl,
            newUser.getUsername()));
    }
    ...
}
```

Spring Boot is very practical for writing RESTful applications, because it automatically configures the infrastructure beans necessary in the background, and all that is left for the developer to do is to create REST controller classes. This is all that can be said about REST support in Spring, and it is time for the last advanced topic in this chapter, which it is not part of the official certification exam, but you might consider it useful.

Spring JMX

JMX is an acronym for Java Management Extensions, and this technology provides the tools for building distributed, Web-based, modular, and dynamic solutions for managing and monitoring devices, applications, and service-driven networks.²⁴ JMX technology can be used to monitor and manage the Java Virtual Machine (Java VM) as well. The JMX specification defines the architecture, design patterns, APIs, and services in the Java programming language for management and monitoring of applications and networks. JMX allows configuration properties to be changed at runtime, can report cache hit/miss ratios at runtime, and can even invoke or simulate a client call.

In JDK versions 5.0 and 6, the JMX technology was developed as Java Specification Request JSR 3,²⁵ Java Management Extensions, and JSR 160.²⁶ JMX Remote API. Evolutions of the JMX API and JMX Remote API in JDK version 7 are covered by JSR 255, JMX API 2.0. JMX is dynamic, so information about an application's internal status can be gathered during runtime. So basically, the JDK comes with JMX out of the box. And it is not hard to use. You must create an interface that ends with MXBean or use the annotation. Of course, there must be an implementation as well. The implementation is then used to manage facets of the application.

Using JMX, a given resource can be instrumented by one or more components called **Management Beans**, or simply **MBeans**. These are objects containing management metadata. The MBeans are registered in a core-managed object server, known as the MBean server. The **MBean server** acts as a management agent and can run on most devices that have been enabled for the Java programming language.

JMX Architecture

The JMX architecture is depicted in Figure 7-30. The JMX architecture is structured into three layers:

- **Instrumentation layer:** where resources are wrapped in MBeans.
- **Agent layer:** the management infrastructure consisting of the MBean Server and agents that provide the following JMX services:
 - Monitoring
 - Event notification
 - Timers

²⁴Oracle official definition found here: <http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html>.

²⁵Official page: <https://jcp.org/en/jsr/detail?id=3>.

²⁶Official documentation here: <http://www.jcp.org/en/jsr/detail?id=160>.

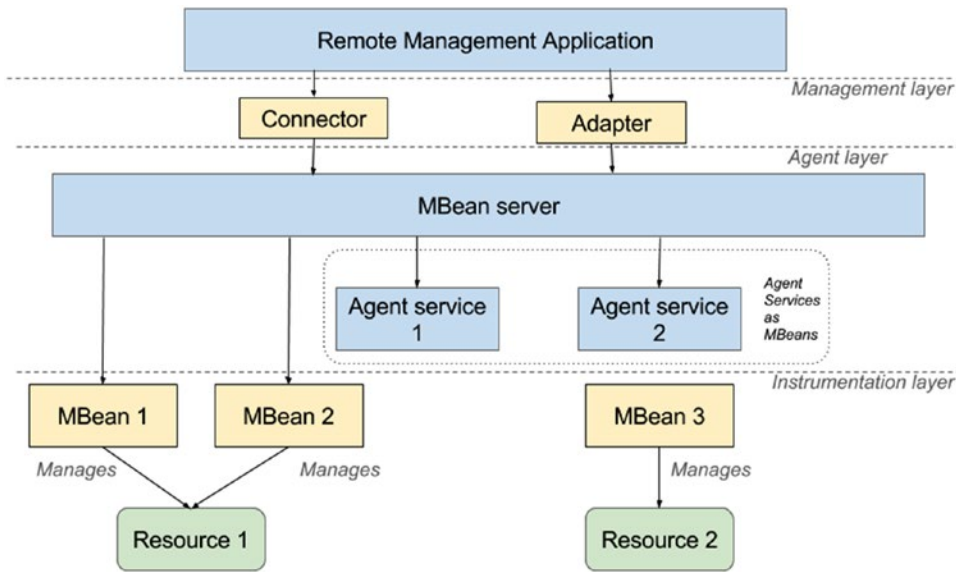


Figure 7-30. JMX architecture

- Management layer:** Defines how external management applications can interact with the underlying layers in terms of protocols, APIs, and so on. This layer uses an implementation of the distributed services specification (JSR-077), which is part of the Java 2 Platform, Enterprise Edition (J2EE) specification.²⁷

The JMX technology defines standard connectors (known as JMX connectors) that enable you to access JMX agents from remote management applications. The MBean server acts as a broker for communication between local MBeans and agents and between MBeans and remote clients. The MBean server keeps a keyed reference as evidence of all registered MBeans. Common and accessible clients for the MBean Server are provided within the JDK: `jconsole`, `jvisualvm`, and `jmc`. And there is also an open source tool named `VisualVM`.²⁸

An MBean is a managed Java object that follows the design patterns set in the JMX specification. An MBean can represent a device, an application, or any resource that needs to be managed. MBeans expose a management interface that consists of the following:

- attributes** (properties), which can be readable, writable, or both
- operations** (methods)
- self description**

²⁷J2EE TM management official page: <https://jcp.org/en/jsr/detail?id=077>.

²⁸VisualVM official page <https://visualvm.github.io/>.

There are five type of MBeans defined by the JMX specification:

- Standard MBeans, also known as Simple MBeans
- Dynamic MBeans
- Open MBeans
- Model MBeans
- MXBeans

The management metadata can be defined statically by implementing a Java interface or by annotating the implementation, and it will automatically be generated at runtime.

Plain JMX

The plainest example of a JMX MBean interface can look like this.

```
package com.ps.jmx;

public interface JmxCounterMBean {
    int getCount(); // attribute "count"
    void add(); // operation "add"
}

public class JmxCounter implements JmxCounterMBean{
    public int getCount() {
        ...
    }

    public void void add() {
        ...
    }
}
```

■ **CC** By convention, an MBean interface takes the name of the Java class that implements it, with the suffix MBean added.

Once a resource has been instrumented by MBeans, the management of that resource is performed by a JMX agent.

```
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
ObjectName name = new ObjectName("com.ps.jmx:type=UserCounter");
UserCounter mbean = new UserCounter();
mbs.registerMBean(mbean, name);
```

Spring JMX

As usual, using native APIs is cumbersome, so Spring JMX was introduced to make developers' lives easy by providing a practical way to integrate a Spring application into a JMX infrastructure. Spring's JMX support provides four core features:

- The automatic registration of any Spring bean as a JMX MBean.
- A flexible mechanism for controlling the management interface of your beans.
- The declarative exposure of MBeans over remote JSR-160 connectors.
- The simple proxying of both local and remote MBean resources.

These features are provided without the need for writing complex code and without coupling the application to Spring or JMX classes or interfaces. The JMX infrastructure can be configured using the context namespace or using Java Configuration. Spring beans can be exposed as MBeans using annotations or XML. JMX beans can be consumed using a proxying mechanism transparently. Until now there is nothing new; this is what Spring provides for most commonly used APIs.

The core class in Spring's JMX framework is the `MBeanExporter`, responsible for taking your Spring beans and registering them with a JMX `MBeanServer`. Declaring the server and the exporter beans is done in a typical Spring manner.

```
<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/context
        http://www.springframework.org/schema/context/spring-context.xsd">

  <!-- Configuration for JMX exposure in the application -->
  <context:mbean-server />
  <context:mbean-export />
</beans>
```

The `<context:mbean-server/>` declares a bean of type `org.springframework.jmx.support.MBeanServerFactoryBean`. This bean has the responsibility of obtaining a `javax.management.MBeanServer` reference from the standard JMX 1.2 `javax.management.MBeanServerFactory` API.

The `<context:mbean-export/>` declares a bean of type `org.springframework.jmx.export.MBeanExporter`, and this is the bean that allows exposing any Spring-managed bean to a `MBeanServer` without the need to define any Spring-managed bean to a `MBeanServer` without the need to define any JMX-specific information in the bean classes.

An equivalent Java Configuration is depicted in the following code snippet. Basically, the two context elements from XML become two beans in a configuration class:

```
import org.springframework.jmx.export.MBeanExporter;
import org.springframework.jmx.support.MBeanServerFactoryBean;
...
@Configuration
public class JmxConfig(){
```

```

// equivalent of <context:mbean-server />
@Bean
MBeanServerFactoryBean mbeanServer(){
    return new MBeanServerFactoryBean();
}

// equivalent of <context:mbean-export />
@Bean
MBeanExporter exporter(){
    MBeanExporter exporter = new MBeanExporter();
    exporter.setAutodetect(true);
    exporter.setBeans(map);
    return exporter;
}
}

```

The project `16-ps-jmx-sample` is assigned to this section. It contains the implementation with Spring Boot, because in this way, the setup becomes even easier, since all that is needed is one annotation: `@EnableMBeanExport`.

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import org.springframework.context.annotation.EnableMBeanExport;

@SpringBootApplication
@EnableMBeanExport
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("Started ...");
        System.in.read();
        ctx.close();
    }
}

```

The `@EnableMBeanExport` is very important, because it enables default exporting of all standard MBeans from the Spring context, as well as all `@ManagedResource` annotated beans. Behind the scenes, it basically declares and registers a bean of type `JmxMBeanServer` and a bean of type `AnnotationMBeanExporter` for you that will take care of registering and will expose your Spring managed beans as MBean, so that you don't have to.

After starting the application, you can check that the two beans have been created for you by analyzing the console log. You should find some log entries like these:

```
...
INFO o.s.j.e.a.AnnotationMBeanExporter - Registering beans for JMX exposure on startup
DEBUG o.s.j.e.a.AnnotationMBeanExporter - Autodetecting user-defined JMX MBeans
...
DEBUG o.s.j.s.JmxUtils - Found MBeanServer:
    com.sun.jmx.mbeanserver.JmxMBeanServer@7c05a4af
DEBUG o.s.b.f.s.DefaultListableBeanFactory -
    Creating instance of bean 'mbeanExporter'
DEBUG o.s.b.f.s.DefaultListableBeanFactory -
    Creating shared instance of singleton bean 'mbeanServer'
...
```

Spring `MBeanExporter` exposes existing POJO beans to the `MBeanServer` without the need to write the registration code. Spring beans must be annotated with the proper annotations.

```
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;

import javax.annotation.ManagedBean;

@ManagedResource(description = "sample JMX managed resource"
    , objectName="bean:name=jmxCounter")
public class JmxCounterImpl implements JmxCounter {

    private int counter =0;

    @ManagedOperation(description = "Increment the counter")
    @Override
    public int add() {
        return ++counter;
    }

    @ManagedAttribute(description = "The counter")
    @Override
    public int getCount() {
        return counter;
    }
}
```

The `ObjectName` of the managed bean is either derived from the fully qualified class name or passed as attribute to the `@ManagedResource`. The `@ManagedResource` marks all instances of a `JmxCounterImpl` as JMX managed resources. `@ManagedOperation` marks the method as a JMX operation. `@ManagedAttribute` marks a getter or setter as one half of a JMX attribute.

The MBeanExporter receives the managed resources as arguments in a map:

```
@Configuration
public class JmxConfig {

    @Bean
    MBeanExporter exporter(){
        MBeanExporter exporter = new MBeanExporter();
        Map<String, Object> map = new HashMap<>();
        map.put("bean:name=jmxCounter1", jmxCounter());
        exporter.setBeans(map);
        return exporter;
    }
    ...
}

// in this case the JMX bean must be annotated with @Component
@Component
@ManagedResource(description = "Sample JMX managed resource",
    objectName="bean:name=jmxCounter")
public class JmxCounterImpl implements JmxCounter {
    ...
}
```

But the MBeanExporter bean can detect all JMX beans if configured so: set the autodetect property to true.

```
@Configuration
public class JmxConfig {

    @Bean
    MBeanExporter exporter(){
        MBeanExporter exporter = new MBeanExporter();
        exporter.setAutodetect(true);
        return exporter;
    }
    ...
}

// in this case the JMX bean must be annotated with @Component
@Component
@ManagedResource(description = "Sample JMX managed resource",
    objectName="bean:name=jmxCounter")
public class JmxCounterImpl implements JmxCounter {
    ...
}
```

But when the `@EnableMBeanExport` annotation is used, there is no need to configure the exporter bean explicitly.

The `objectName` attribute contains the name of the resulting MBean, so any client, local or remote, will have to look for `jmxCounter`. Now that we have the JMX configuration and a Spring bean exposed as an MBean, the application can be started and the search for the Spring managed bean can begin.

As a client, Java Mission Control from Oracle will be used for the current example.²⁹ If you have the `JAVA_HOME` variable set in your system and the `JAVA_HOME\bin` on the classpath, open a terminal (or a Command Prompt on Windows) and type `jmc`. In Figure 7-31, you can see the JMC window.

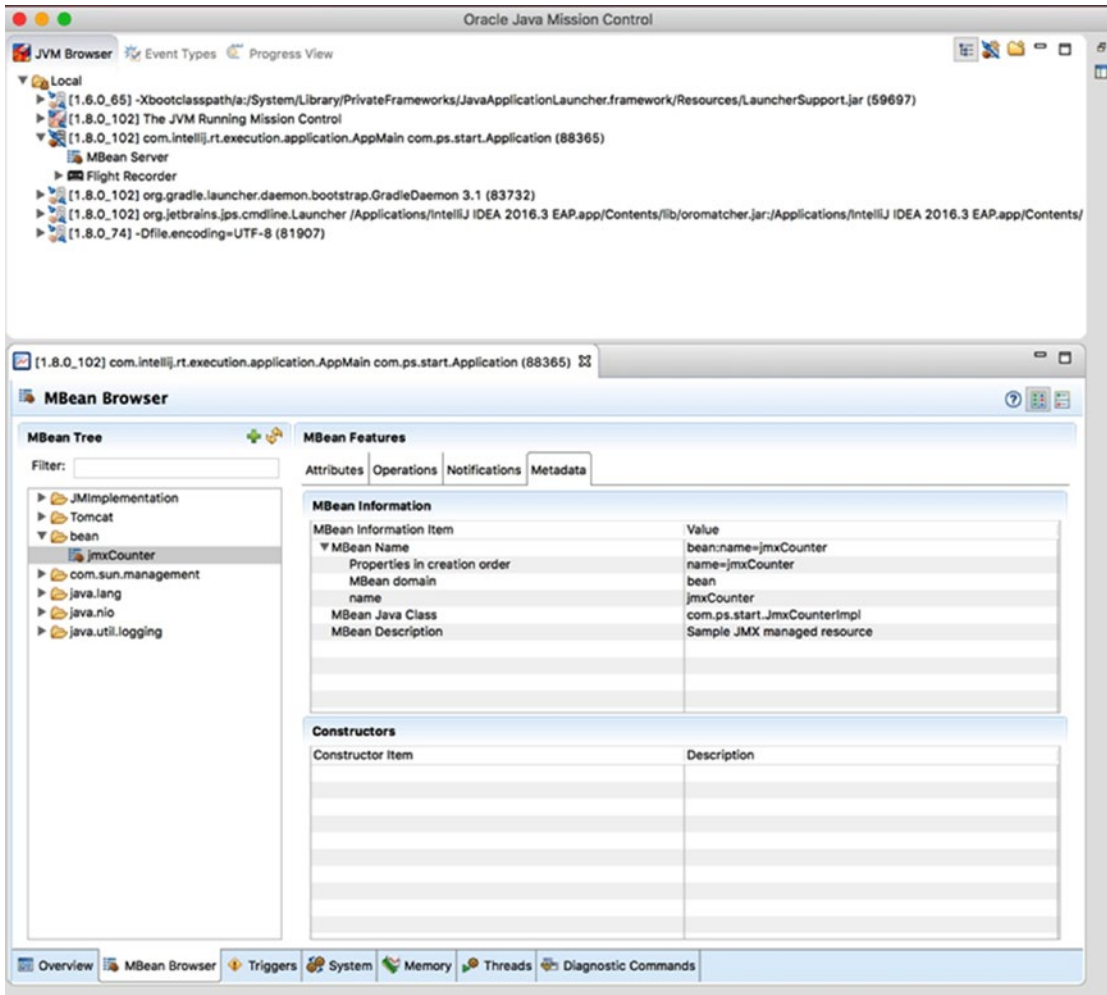


Figure 7-31. Java Mission Control connected to the `16-ps-jmx-sample` application and depicting the `jmxCounter` MBean

²⁹You can also use `jconsole`, `jvisualvm`, or the outsource `visualvm`. In this book, JMC is covered, because it has recently become one of the most competent monitoring and analysis tools for Java Applications. It can be used to generate and analyze full or partial thread and memory dumps.

In the JMC, you will see a tab called JVM Browser that displays all the JAVA processes running on the system. You can recognize the Java process corresponding to the 16-ps-jmx-sample by the name of the main class `com.ps.start.Application`. Double click and expand the node; then double click on the MBeanServer node. A tab named MBean Browser will be opened, and in it under the bean node, on expanding it, you will see the `jmxCounter`, which can be inspected by clicking on the tabs on the right. If you click on the operations tab, you will see the two operations and a button named Execute that you can use to execute the `jmxCounter` MBean methods.

It was mentioned before that client applications can access the MBeans through proxies. The Spring JMX architecture allows this, since it is the standard remoting communication between Spring applications. In Figure 7-32 you can see the Spring JMX architecture.

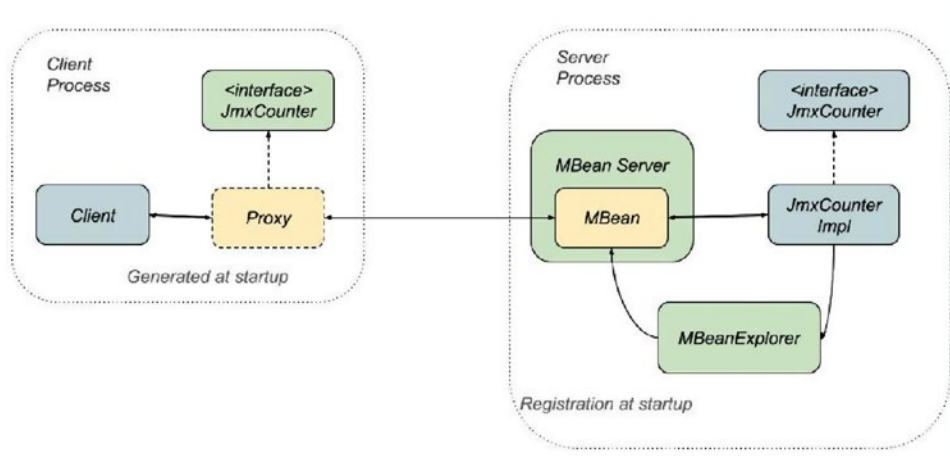


Figure 7-32. Spring JMX Architecture

So basically, the server exposes Spring beans as MBeans managed by the MBeanServer, and the client accesses them using proxies. Typical Spring, right?

To be able to create a Spring client for the MBean, a connector that the client will use to access the MBean must be defined for it on the server side. For this, we need a bean of type `ConnectorServerFactoryBean` to be configured.

```
@SpringBootApplication
@EnableMBeanExport
public class Application {
    private static Logger logger = LoggerFactory.getLogger(Application.class);

    @Bean
    ConnectorServerFactoryBean connector(){
        return new ConnectorServerFactoryBean();
    }

    ...
}
```

By default, `ConnectorServerFactoryBean` creates a `JMXConnectorServer` bound to `"service:jmx:jmxmp://localhost:9875"`, and this is the URL that the client will use to access the MBean. The client in this case will be a test class with a bean of type `MBeanServerConnectionFactoryBean` injected into it that will be used to access the MBean remotely, through a proxy object of course. The JMXMP protocol is marked as optional by the JSR 160 specification: currently, the main open-source JMX implementation, MX4J, and the one provided with the JDK do not support JMXMP. So the easier way is to expose the MBean through RMI. For this, the server configuration must be changed to include an RMI registry and connector beans.

```
import org.springframework.jmx.support.ConnectorServerFactoryBean;
import org.springframework.jmx.support.MBeanServerFactoryBean;
import org.springframework.remoting.rmi.RmiRegistryFactoryBean;
....
@SpringBootApplication
@EnableMBeanExport
public class Application {
    private static Logger logger = LoggerFactory.getLogger(Application.class);

    @Bean
    @DependsOn("rmiRegistry")
    ConnectorServerFactoryBean connector() {
        ConnectorServerFactoryBean cf = new ConnectorServerFactoryBean();
        try {
            cf.setObjectName("connector:name=rmi");
        } catch (MalformedObjectNameException e) {
            return null;
        }
        cf.setServiceUrl(
            "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi");
        return cf;
    }

    @Bean
    RmiRegistryFactoryBean rmiRegistry() {
        RmiRegistryFactoryBean rmiRegistryFactoryBean = new RmiRegistryFactoryBean();
        rmiRegistryFactoryBean.setAlwaysCreate(true);
        rmiRegistryFactoryBean.setPort(1099);
        return rmiRegistryFactoryBean;
    }
    ...
}
```

On the client side we need a `MBeanServerConnection` bean to be able to connect to the RMI server, which will be created by the `MBeanServerConnectionFactoryBean` factory bean, and a `MBeanProxyFactoryBean` that will take care of creating the `JmxCounter` proxy.

```
import com.ps.start.JmxCounter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jmx.access.MBeanProxyFactoryBean;
import org.springframework.jmx.support.MBeanServerConnectionFactoryBean;
```

```

import javax.management.MBeanServerConnection;

@Configuration
public class TestConfig {

    @Bean
    MBeanServerConnection connection() {
        MBeanServerConnectionFactoryBean conn = new MBeanServerConnectionFactoryBean();
        try {
            conn.setServiceUrl(
                "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/jmxrmi");
            conn.afterPropertiesSet();
            return conn.getObject();
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    @Bean
    MBeanProxyFactoryBean proxy() throws MalformedObjectNameException {
        MBeanProxyFactoryBean proxy = new MBeanProxyFactoryBean();
        proxy.setObjectName("bean:name=jmxCounter");
        proxy.setProxyInterface(JmxCounter.class);
        proxy.setServer(connection());
        return proxy;
    }
}

```

The test class just has to retrieve the proxy and do some actions on it:

```

...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {TestConfig.class})
public class TestClient {
    private static Logger logger = LoggerFactory.getLogger(TestClient.class);

    @Autowired
    JmxCounter counter;

    @Test
    public void test() {
        assertNotNull(counter);
        counter.add();
        assertEquals(1, counter.getCount());
    }
}

```

Summary

After reading this chapter you should have a proper understanding of how Spring can be used to provide and consume quite a few types of services. Below, you can find a simple list of topics, which you should keep handy when reviewing the acquired knowledge, but keep in mind that only Spring REST is part of the official certification exam.

- What does RMI stand for?
- What are the disadvantages of the RMI model?
- What is the `HttpInvoker` used for?
- Explain JMS core concepts like publisher, subscriber, producer, consumer, destinations.
- How many types of JMS messages are there?
- What is `JmsTemplate` used for?
- What does JMS stand for?
- How many types of Destinations are there?
- What is the structure of a JMS message?
- What is SOAP?
- Advantages and disadvantages of SOAP.
- `@EnableWs` is the key configuration annotation for a Spring WS application.
- How can classes for a WS implementation be generated?
- `@Endpoint` is the annotation that marks a class as a web service endpoint.
- What is REST?
- What type of clients can access a web application?
- How are resources exposed to the client?
- How many types of representations are supported?
- What is the difference between `@Controller` and `@RestController`?
- Make sure you can describe Spring MVC support for RESTful applications.
- Understand how to access Request/Response Data.
- What are `@ResponseBody` and `@RequestBody` used for?
- Use `MessageConverters`.
- Is Spring MVC needed in the classpath in the configuration of a RESTful application?
- `RestTemplate` is the core Spring class for creating clients for REST applications.
- Spring JMX allows practical export of Spring managed Beans as MBeans to a JMX `MBeanServer`.
- Spring JMX provides an easy way to configure JMX applications, so that the developer can focus on the Spring Bean implementation.
- `@EnableMBeanExport` is the key configuration annotation for a Spring JMX application.

Quick Quiz

Question 1: What is REST?

- A. a software design pattern
- B. a framework
- C. an architecture style

Question 2: Which of the methods below are HTTP methods?

- A. PUT
- B. GET
- C. SUBMIT
- D. OPTIONS

Question 3: What Spring class can be used to access and test REST services?

- A. RestTemplate
- B. RmiTemplate
- C. Both
- D. None

Question 4: What does the RestTemplate handle?

- A. Resources
- B. Representations
- C. Both

Question 5: What can be said about the `@RestController` annotation?

- A. It is used to declare a controller providing REST Services.
- B. It is annotated with `@Controller` and `@ResponseBody`.
- C. In controllers annotated with this annotation `@RequestMapping` methods assume `@ResponseStatus` semantics by default.

Question 6: What is the effect of annotating a method with `@ResponseStatus`?

- A. The default behavior for resolving to a view for methods returning void or null is overridden.
- B. The HTTP Status code matching the `@ResponseStatus` is added to the response body.
- C. It forces use of HTTP message converters.

Question 7: Which of the following HTTP message converters are supported by Spring MVC?

- A. StringHttpMessageConverter
- B. MappingJackson2HttpMessageConverter, but Jackson2 must be in the classpath
- C. YamlMessageConverter

Question 8: Which of the following RestTemplates can be used to make a GET REST call to a URL?

- A. restTemplate.getForObject(...)
- B. optionsForAllow(...)
- C. getForEntity(...)
- D. exchange(..., HttpMethod.GET,...)

Question 9: Does the following REST handler method comply with the HATEOAS constraint?

```

@ResponseStatus(HttpStatus.CREATED)
@RequestMapping(value = "/create", method = RequestMethod.POST,
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE,
    consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
public Person createPerson(@RequestBody @Valid Person newPerson) {
    logger.info("-----> CREATE");
    Hospital hospital = hospitalManager.findByCode(
        newPerson.getHospital().getCode());
    newPerson.setHospital(hospital);
    Person person = personManager.save(newPerson);
    logger.info("-----> PERSON: " + person);
    return person;
}

```

- A. Yes, because it returns a representation of the object that was created.
- B. No, because it does not set the location header to the URI of the created resource.
- C. This is not a REST handler method
- D. No, because no Link object is added to the returned resource.

CHAPTER 8



Spring Microservices with Spring Cloud

Microservices are a specialization and implementation approach for service-oriented architectures (SOA), and they are used to build flexible, independently deployable services. Microservices is a paradigm that requires for services to be broken down into highly specialized instances as functionality and be interconnected through agnostic communication protocols (like REST, for example) that work together to accomplish a common business goal. Each microservice is a really small unit of stateless functionality, a process that does not care where the input is coming from and does not know where its output is going. It has no idea what the big picture is. Because it is specialized and decoupled, each problem can be identified, the cause localized and fixed, and implementation redeployed without affecting other microservices. This means that microservices systems have high cohesion of responsibilities and really low coupling, and these qualities allow the architecture of an individual service to evolve through continuous refactoring, reduce the necessity of a big up-front design, and allow for software to be released earlier and continuously.

Microservices have grown in popularity in recent years, and because of the small granularity and lightweight communication protocols, they have become the preferred way to build enterprise applications. Microservices' modular architectural style seems particularly well suited to cloud-based environments. This architectural method is quite scalable and is considered ideal when multiple platforms and devices must be supported. Consider the biggest players on the Web right now: Twitter, Netflix, Amazon, PayPal, Soundcloud, and others. They have large-scale websites and applications that have evolved from monolithic architecture to microservices so that they can be accessible from every kind of device.

Classic application development is characterized by multiple types of services being developed as a single monolithic, autonomous unit and deployed on a server for clients to access. Usually, these applications are multilayered, and each layer corresponds to different functional areas of the application. The monolithic application would handle HTTP requests, execute business logic, and also handle database operations. In a monolithic infrastructure, the services that make the system are organized logically within the same code base and unit of deployment. The disadvantage of monolithic applications is that changes end up being tied to other changes, and sometimes problems are harder to identify when there are multiple layers involved. A modification made to a small section of an application might require building and deploying an entirely new version. And obviously, regression tests to verify that code previously developed and tested still performs correctly even after it was changed are mandatory. Thus, underlining advantages and disadvantages of microservices can be done only by comparing them to old style monolithic architecture.

A summary of microservices' advantages is listed below:

- increased granularity
- increased scalability
- easy to automate deployment and testing

- easy to test
- increased decoupling
- enhanced cohesion
- suitable for continuous refactoring, integration, and delivery
- increased module independence
- organized around capabilities
- improved agility and velocity, because when a system is correctly decomposed into microservices, each service can be developed and deployed independently and in parallel with the others
- each service is elastic,¹ resilient, composable, minimal, and complete
- improvement of fault isolation
- elimination of long-term commitment to a single technology stack, because microservices can be written in different programming languages
- makes it easier to integrate new developers in a team

A summary of microservices' disadvantages is listed below:

- Microservices introduce additional complexity and the necessity of careful handling of requests between modules.
- Handling multiple databases and transactions can be painful.
- Testing microservices can be cumbersome, since each dependency of a microservice must be confirmed valid before the service can be actually tested.
- Deployment becomes complex as well, requiring coordination among modules.

The conclusion so far is that the new-age titan applications that need to support multiple clients and platforms are very good candidates for microservices. Microservices help break up monolithic applications into individual units of deployment, which are able to evolve their own scaling requirements irrespective of the other subsystems. Let's see whether Spring can make development of microservices-based applications as practical as it has made monolithic applications development.

Microservices with Spring

Microservices architectural idea is similar to how beans are managed by the Spring container and how they communicate in a Spring application context. Imagine it like this: if a Spring Application Context is a forest, then each bean is a tree. Microservices are the full-blown ecosystem. This example was given to you because programming is nothing other than modeling the real world using software components, and it might make the idea of microservices more approachable. The demo application built throughout this book is basically a site where people register themselves and their pets so they can provide pet-sitting services. If it were built with microservices, separate microservices would have to be developed for user, pet, request, reply, and reviews management. This also means that a few moving components would be needed to set up such a

¹A microservice must be able to scale, up or down, independently of other services in the same application.

system, because communication between microservices must be covered too. Figure 8-1 depicts the classic monolith architecture and the microservices architecture side by side.

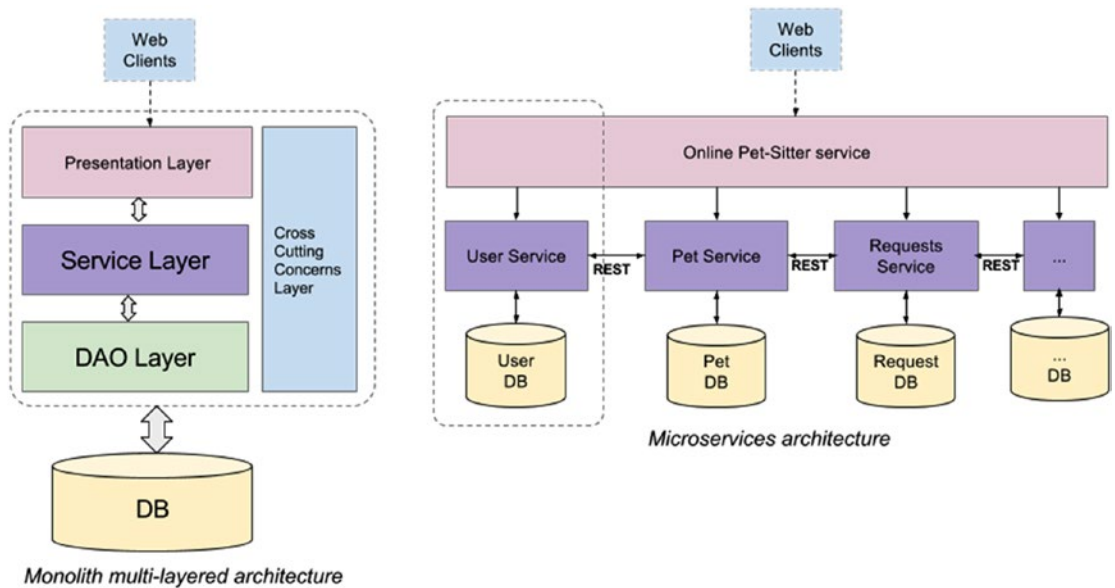


Figure 8-1. Monolith and Microservices architecture

The figure is quite generic, and it might lead to the idea that each microservice has its own database, which is correct, but there is not actually a need to provision a database server for each service. The database used in development can be relational (SQL Based) or nonrelational (NoSQL). If a relational database is used, there are three options of database implementation strategies:

- **private-tables-per-service:** each service owns a set of tables that must be accessed only by that service
- **schema-per-service:** each service has a database schema that is private to that service
- **database-server-per-service:** each service has its own database server

The chosen microservice database strategy implementation influences the type of database that is most suitable for the application. Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing, since it makes ownership clearer. Also, if there is one database server that is shared among microservices, a relational database is suitable, since all the data is in one place and has been organized properly by following normalization standards. Having a different database for each service might make sharing data among services difficult and introduce data redundancy. NoSQL databases are more suitable for this type of implementation, since they are better at handling redundant data. Also, keep in mind that a downside of not sharing databases is that maintaining data consistency and implementing queries is more challenging.

Spring is the best framework for developing advanced applications such as those based on the microservices architecture, because of the following characteristics:

- dependency injection and integration
- super-practical configuration and service creation
- straightforward service discovery registry

To develop a microservices application with Spring components, good knowledge of the following Spring technologies is needed:

- a service registration and discovery technology like Netflix’s OSS Eureka
- Spring Cloud projects like Eureka or Consul
- REST concepts

Spring Boot makes it easy to create standalone, production-grade Spring-based applications that you can “just run.” This has been already covered in the book, in **Chapter 6**, and it is the only way to develop Spring microservices applications, since Spring Boot is designed for developer-heightened productivity by making common concepts, such as RESTful HTTP and embedded web application runtimes, easy to wire up and use. It is flexible, allowing the developer to select only the modules they want to use, removing overwhelming or bulky configurations and runtime dependencies.

Spring Cloud is a big umbrella project that provides development tools designed to ease the development of distributed applications. It contains components designed to be used to build common patterns in distributed systems:

- configuration management (Spring Cloud Config provides centralized external configuration backed by a Git repository)
- service discovery (Eureka is an AWS Service registry for resilient midtier load balancing and failover and is supported by Spring Cloud)
- circuit breakers (Spring Cloud supports Netflix’s Hystrix, which is a library that provides components that stop calling services when a response is not received by a predefined threshold)
- intelligent routing (Zuul, used to forward and distribute calls to services)
- micro-proxy (client-side proxies to midtier services)
- control bus (a messaging system can be used for monitoring and managing the components within the framework, as is used for “application-level” messaging)
- one-time tokens (used for data access only once with Spring Vault)
- global locks (used to coordinate, prioritize, or restrict access to resources)
- leadership election (the process of designating a single process as the organizer of some task distributed among several nodes)
- distributed sessions (sessions distributed across multiple servlet servers)
- cluster state (cluster state request is routed to the master node, to ensure that the latest cluster state is returned)
- client-side load balancing

In case you are interested in building microservices applications with Spring Cloud, the documentation provided by the Spring team is available here: <http://projects.spring.io/spring-cloud/spring-cloud.html>. It covers every topic in the previous list and more.

Coordinating distributed systems is not easy and can lead to boilerplate code. Spring Cloud just makes it easier for developers to write this type of management code, and the result works in any distributed environment including a development station, data centers, and managed platforms such as Cloud Foundry.² Spring Cloud builds on Spring Boot, and it comes with the typical Spring Boot advantages: out-of-the-box preconfigured infrastructure beans, which can be further configured or extended to create a custom solution. It follows the same Spring declarative approach, relying on annotations and property files.

Spring Cloud Netflix provides integration with Netflix OSS (Open Source Software). Tofficial GitHub page <https://netflix.github.io/> is basically a collection of open-source libraries that their developers wrote to solve some distributed-systems problems at scale. Written in Java, it is now among the most frequently used software solutions for writing microservices applications in Java.

Registration and Discovery Server

The microservices architecture ensures that a set of processes will work together toward a common goal, providing the end user a competent and reliable service. For this to work, the processes must communicate efficiently. To communicate with each other, they first have to know “of” each other. This is where the **Netflix Eureka** registration server comes in. And because it is open-source, it was incorporated in Spring Cloud, and the simplicity principles of Spring now apply. A registration or discovery server that will be used by the processes of a microservices application to register and discover each other can be declared as in the following snippet, which is part of the `17-ps-micro-sample`.

```
package com.ps.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
import java.io.IOException;

@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServer {

    private static Logger logger =
        LoggerFactory.getLogger(DiscoveryServer.class);

    public static void main(String args) throws IOException {
        // Tell server to look for discovery.properties or discovery.yml
        System.setProperty("spring.config.name", "discovery");
    }
}
```

²Spring Cloud projects are publicly available on Github: <https://github.com/spring-cloud>.

```

        ConfigurableApplicationContext ctx =
            SpringApplication.run(DiscoveryServer.class, args);
        assert (ctx != null);
        logger.info("Started ...");
        System.in.read();
        ctx.close();
    }
}

```

■ ** The `System.in.read()`; call is used so that you can stop the application gracefully by pressing the <ENTER> key.

To use the Eureka Server in a project, the `spring-cloud-starter-eureka-server` starter project must be included as a dependency of the project. Notice the `@EnableEurekaServer` annotation, which is responsible for injecting a Eureka server instance into your project. The server has a home page with a UI and HTTP API end- points according to the normal Eureka functionality under `/eureka/`.

The `discovery.yml` file contains settings for this server. By default, Spring Boot looks for a file named `application` from which to read the configuration. To specify that the configuration should be read from a different file, the name of the file must be added as a value for the system property `spring.config.name`, by calling `System.setProperty("spring.config.name", "discovery")`; before creating the Spring context.

```

eureka:
  instance:
    hostname: localhost
  client:
    registerWithEureka: false # do not auto-register as client
    fetchRegistry: false
server:
  port: 3000 # where this discovery server is accessible
  waitTimeInMsWhenSyncEmpty: 0

```

■ ! If the port value is not specified, the default value for the port is implicitly set to 8761. (In the above example, the port is set to 3000 just as an example.)

The server will be available on the 3000 port, and when the above class is run, the server interface depicting a few stats and registered microservices becomes accessible from the browser at location `http://localhost:3000/`. Figure 8-2 depicts the interface of a naked Eureka Server, with no microservices registered yet.

The screenshot shows the Spring Eureka web interface in a browser window at localhost:3000. The interface includes a search bar, navigation links (HOME, LAST 1000 SINCE STARTUP), and several sections:

- System Status:** A table showing environment (test), data center (default), current time (2016-10-27T19:04:47 +0300), uptime (00:09), lease expiration enabled (true), renew threshold (1), and renew interval (16).
- DS Replicas:** A section titled "Instances currently registered with Eureka" showing a table with columns for Application, AMIs, Availability Zones, and Status. It currently displays "No instances available".
- General Info:** A table listing various system metrics such as total-avail-memory (1080mb), environment (test), num-of-cpus (8), current-memory-usage (504mb (46%)), server-uptime (00:09), registered-replicas, unavailable-replicas, and available-replicas.
- Instance Info:** A table showing instance details like ipAddr (192.168.56.1) and status (UP).

Figure 8-2. Spring Eureka web interface

Netflix's original version of the Eureka Server avoids answering clients for a configurable period of time if it starts with an empty registry. The `waitTimeInMsWhenSyncEmpty` property controls this behavior, and it was designed so that clients would not get partial/empty registry information until the server has had enough time to build the registry. In the previous example, it is set to zero, to start answering clients as soon as possible.

! If not set, the default value for the `server.waitTimeInMsWhenSyncEmpty` is **5 minutes**.

The `eureka.client.registerWithEureka` property has "true" as a default value and is used to register Eureka clients. Since the application above registers a server, it must be explicitly set to "false" to stop the server from trying to register itself.

Now that we have the server, the microservices must be developed.

Microservices Development

To create a microservice, a service implementation and a class that will interact with the registration and discovery server to register itself as client are needed. The implementation of the service is a typical Spring service, and in our example we'll have three microservices: one that will handle users, one that will handle pets, and one that will connect the functionality of the two. In Figure 8-3, a schema of how these microservices interact is depicted.

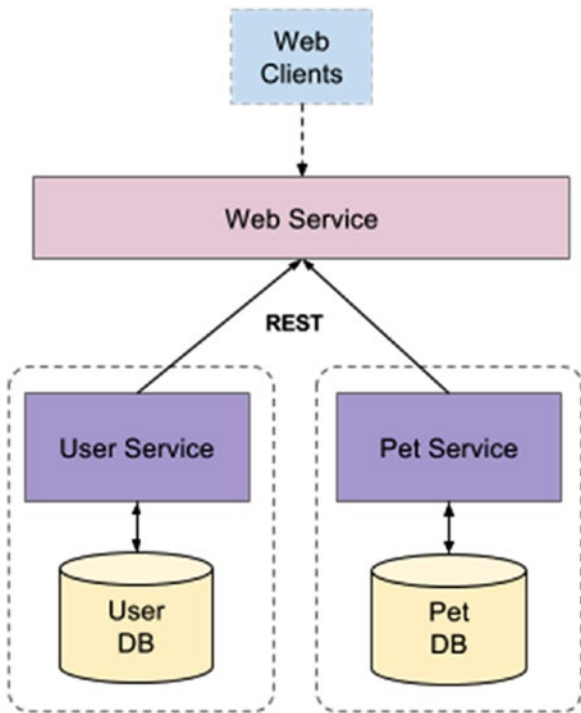


Figure 8-3. *17-ps-micro-sample* project structure

A microservice is a standalone process that handles a well-defined requirement. When creating a distributed application that is based on microservices, each microservice component should be wrapped up in packages based on their purpose, and the overall implementation should be very loosely coupled, but very cohesive. The three microservices will have separate packages in the project structure, and they will communicate with each other using RESTful requests. In Figure 8-4, the *17-ps-micro-sample* is depicted.

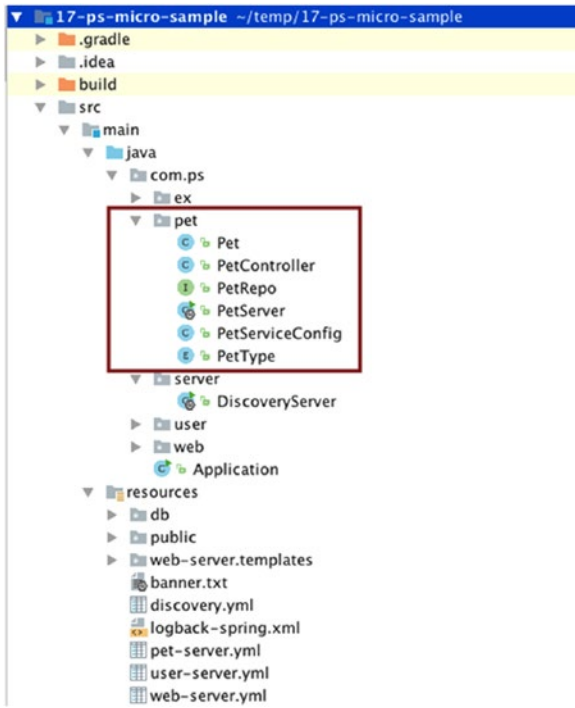


Figure 8-4. 17-ps-micro-sample project structure

In Figure 8-5, all the beans making up the microservices and their interactions are depicted:

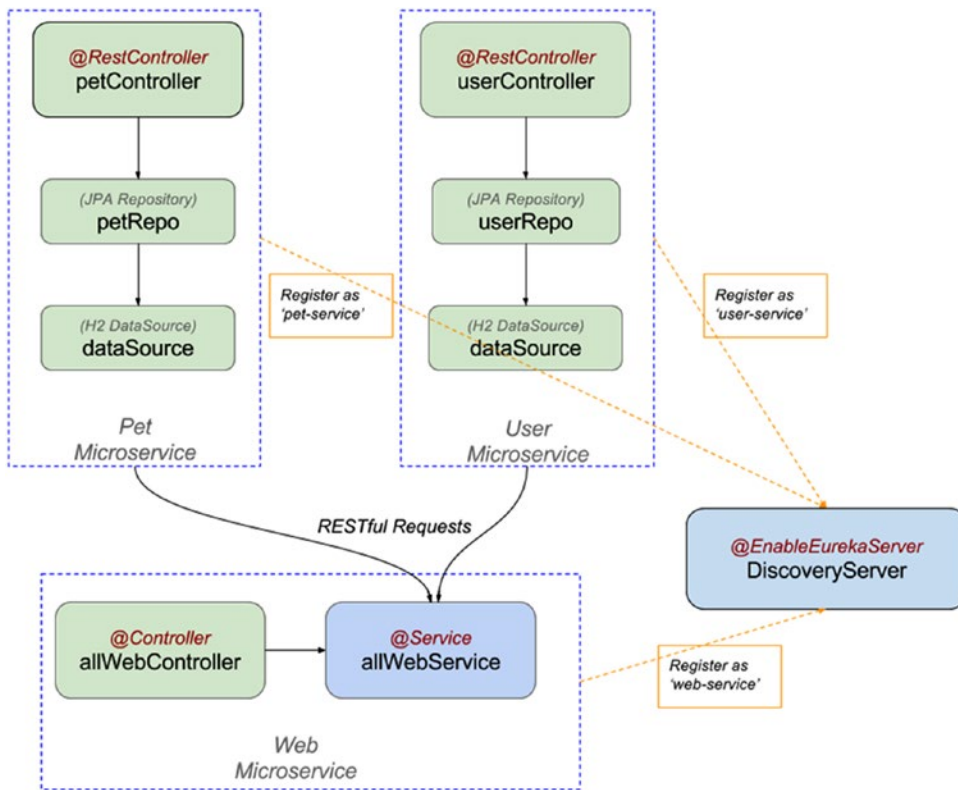


Figure 8-5. 17-ps-micro-sample project architecture

The pet-service and user-service microservices implementations are similar. Each of them uses a Spring Data JPA repository bean (JpaRepository implementation) and a Spring REST controller to provide a RESTful interface to expose their data.

```

@RestController
@RequestMapping("/pets")
public class PetController {

    @RequestMapping("/")
    public List<Pet> all() {
        ... // gell all pets
    }

    @RequestMapping("/owner/{id}")
    public List<Pet> byOwner(@PathVariable("id") Long ownerId) {
        // get all pets for owner with id = ownerId
    }
}
...
}
    
```

What is special about this implementation is the `PetServer` class, which is a SpringBoot special class that is used to register itself with the Eureka discovery and registration server configured in the previous section.

```
package com.ps.pet;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.context.ConfigurableApplicationContext;

@EnableAutoConfiguration
@EnableDiscoveryClient
@Import(PetServiceConfig.class)
public class PetServer {

    private static Logger logger = LoggerFactory.getLogger(PetServer.class);

    public static void main(String args) throws IOException {
        // Tell server to look for pet-server.properties or pet-server.yml
        System.setProperty("spring.config.name", "pet-server");
        ConfigurableApplicationContext ctx =
            SpringApplication.run(PetServer.class, args);
        assert (ctx != null);
        logger.info("Started ...");
        System.in.read();
        ctx.close();
    }
}
```

The `@EnableAutoConfiguration` annotation is used instead of `@SpringBootApplication`, because some custom configuration comes from the `PetServiceConfig` class, and we want control of that. But the application is still a Spring Boot application.

Notice the `@EnableDiscoveryClient` annotation, which is the key component that transforms this application into a microservice, because it enables service registration and discovery. The process registers itself with the application name `pets-service` and specifies that it will be available on port 4000 in the `pet-server.yml` configuration file. Its contents are depicted in the following configuration snippet.

```
# Spring properties
spring:
  application:
    name: pets-service # Service registers under this name

# HTTP Server
server:
  port: 4000 # HTTP (Tomcat) port
```

Discovery Server Access

```
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:3000/eureka/

  instance:
    leaseRenewalIntervalInSeconds: 5
    preferIpAddress: false
```

The previous configuration snippet contains three sections:

- The **Spring section** defines the application name as `pets-service`. The microservice will register itself with this name with the Eureka server.
- The **Server section** defines the custom port to listen on. All the microservice processes in the application use Tomcat by default, because it is the implicit dependency of the `spring-boot-starter-web` and by default will try to use the 8080 port, and only one process can listen on a port at one time. So each microservice will have a different port assigned via configuration. `pets-service` has 4000, `user-service` has 4001, and `web-service` has 4002.
- The **Eureka section** defines the URL where the server to register to is located using the `eureka.client.serviceUrl.defaultZone`. Other properties can be set to customize the interaction with the Eureka server.

■ ! The `eureka.client.registerWithEureka` property has "true" as a default value and is used to register Eureka clients. It is used here explicitly for teaching purposes, to make it more obvious that the `pets-service` microservice is a client.

- Eureka clients fetch the registry information from the server and cache it locally. After that, the clients use that information to find other services. Since the `pets-service` does not need to communicate with other microservices, it has no use for the registry information. So the `eureka.client.fetchRegistry` is set to `false` to avoid wasting time and resources on a useless operation.
- Eureka clients need to tell the server that they are still active by sending a signal called heartbeat. By default, the interval is 30 seconds. But it can be set to smaller intervals by customizing the value of the `eureka.instance.leaseRenewalIntervalInSeconds` property. During development, it can be set to a smaller value, which will speed up registration, but on production, this will generate extra communication with the server that might cause service lag. For production, the default value should not be modified.
- The `eureka.instance.preferIpAddress` is used to tell the Eureka server whether it should use the domain name or an IP. In our case, because everything is working on the same machine, we prefer the domain name (`localhost`), which is why the property is set to `false`.

All these details and much more are available on the Netflix GitHub page <https://github.com/Netflix/eureka/wiki/Understanding-eureka-client-server-communication>. Only those important for the implementation given as an example were quoted and explained here.

The `PetServer` class introduced a new Spring Boot main class, so the Gradle build will fail now, because it does not know what main class to execute. So the Gradle configuration must be modified to specify the main class of the application. But since `17-ps-micro-sample` is a distributed application, the purpose is to be able to start it multiple times:

- once with the role of the discovery and registration server, listening on the 3000 port
- once with the role of `pets-service` microservice, listening on the 4000 port
- once with the roles of `users-service` microservice, listening on the 4001 port
- once with the role of `web-service` microservice, listening on the 4002 port

This means that the main class should start a different Spring application based on the argument given to it. For the application given as example, the main class looks like this:

```
package com.ps;

import com.ps.pet.PetServer;
import com.ps.server.DiscoveryServer;
import com.ps.user.UserServer;
import com.ps.web.WebServer;

import java.io.IOException;
public class Application {

    public static void main(String args) throws IOException {
        if (args.length == 0) {
            System.out.println("Specify application to start!
(Options: reg, user, pet, web)");
        } else {
            switch (args[0]) {
                case "reg":
                    DiscoveryServer.main(args);
                    break;
                case "user":
                    UserServer.main(args);
                    break;
                case "pet":
                    PetServer.main(args);
                    break;
                case "web":
                    WebServer.main(args);
                    break;
                default:
                    System.out.println("Specify application to start!
(Options: reg, user, pet, web)");
            }
        }
    }
}
```

To tell the Spring Boot plugin that this is the main class of our application, we add the following configuration snippet to the Gradle configuration file:

```
springBoot {
    mainClass = "com.ps.Application"
}
```

Now the Gradle application can be built again with the `gradle clean build`. The result of this build will be the `ps-micro-1.1.0.RELEASE.jar`, located under `17-ps-micro-sample/build/libs`. The easiest way to run multiple instances of the application is to open multiple terminals (command prompt instances in Windows) and execute `java -jar ps-micro-1.1.0.RELEASE.jar` with all the different arguments available in the application. This is also practical for shutting them down gracefully, or when only one of the services must be stopped. In Figure 8-6, four terminals for starting different instances of the application are depicted.

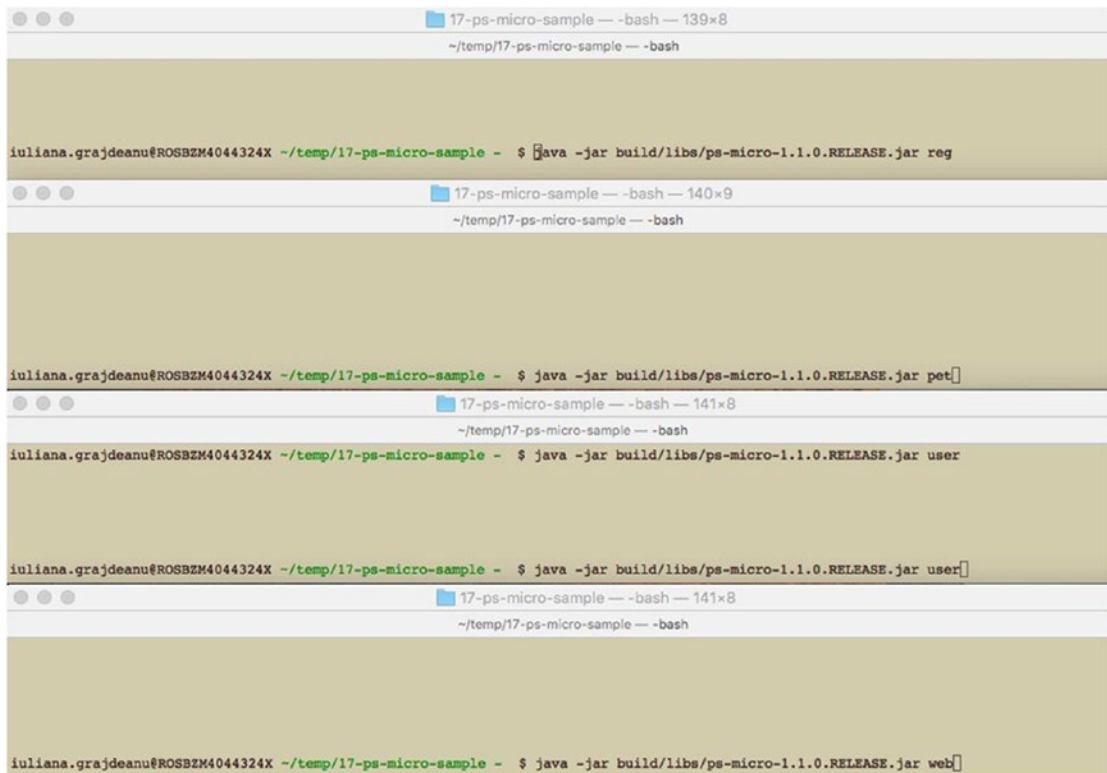


Figure 8-6. Starting different instances of the 17-ps-micro-sample application

Registration of a service takes up to 30 seconds, if not set otherwise through a smaller value for the `eureka.instance.leaseRenewalIntervalInSeconds` property, so be patient and watch the console log of the registration server. All the main classes end with the `Started ...` text being displayed. If this text is displayed and no exceptions can be seen in any of the consoles, this means that all instances were started correctly. After registration, when accessing the `http://localhost:3000/` interface for the Eureka server, you should see all the client services that were registered in the instances tab, as depicted in Figure 8-7.

The screenshot shows the Spring Eureka web interface at localhost:3000. The page title is "spring Eureka" and it includes a navigation bar with "HOME" and "LAST 1000 SINCE STARTUP".

System Status

Environment	test	Current time	2016-10-27T21:26:30 +0300
Data center	default	Uptime	00:01
		Lease expiration enabled	false
		Renews threshold	6
		Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PETS-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.1:pets-service:4000
USERS-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.1:users-service:4001
WEB-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.1:web-service:4002

General Info

Name	Value
total-avail-memory	1325mb
environment	test
num-of-cpus	8
current-memory-usage	427mb (32%)
server-uptime	00:01
registered-replicas	
unavailable-replicas	
available-replicas	

Instance Info

Name	Value
ipAddr	192.168.56.1
status	UP

Figure 8-7. Client microservices registered with the Eureka Server

You can see more details about the registered microservices in the `http://localhost:3000/eureka/apps/`.

```
<applications>
  <versions_delta>1</versions_delta>
  <!-- how many instances are up: 3 -->
  <apps_hashcode>UP_3</apps_hashcode>

  <application>
    <name>PETS-SERVICE</name>
    <instance>
      <instanceId>192.168.56.1:pets-service:4000</instanceId>
      <hostName>192.168.56.1</hostName>
```

```

<app>PETS-SERVICE</app>
<ipAddr>192.168.56.1</ipAddr>
<status>UP</status>
<overriddenstatus>UNKNOWN</overriddenstatus>
<port enabled="true">4000</port>
<healthCheckUrl>http://192.168.56.1:4000/health</healthCheckUrl>
...
</application>
<!-- the others users-services and web-services -->
...
</applications>

```

The same information is available at <http://192.168.56.1:3000/eureka/apps/PETS-SERVICE> if the service was started correctly. Otherwise, a 404 error will be displayed.

At registration time, each microservice gets a unique registration identifier from the server. If another process registers with the same ID, the server treats it as a restart, so the first process is discarded. To run multiple instances of the same process—for reasons of load balancing and resilience and because, after all, it is a distributed application and it should be possible to do this—we have to make sure that the server generates a different registration ID. The simplest way in which this can be done is by providing the option of specifying a different port for the microservice. The registration ID with the configuration used so far, the one in the `<instanceId>` element, is the default naming template; it appears as follows:

```

${ipAddress}:${spring.application.name}:${server.port}

```

The registration Id can be set using Eureka property configurations:

```

eureka:
  instance:
    metadataMap:
      instanceId:
        ${spring.application.name}:${spring.application.instance_id:${server.port}}

```

If the `spring.application.instance_id` is not defined, it will revert to the default format mentioned earlier. The `spring.application.instance_id` is set only when Cloud Foundry is used,³ and it conveniently provides a unique ID number for each instance of the same application.

For running a microservices application locally, the method with a parametrizable port is more practical. This can be done easily by setting the `server.port` property before starting the application.

```

// in the Application.main(...) method
...
case "pet":
  if (args.length == 2) {
    System.setProperty("server.port", args[1]);
  }
  PetServer.main(args);
  break;
...

```

³Cloud Foundry is an open-source cloud platform as a service (PaaS) on which developers can build, deploy, run, and scale applications on public and private cloud models. It was created by VMware and kept by Pivotal after the takeover. There is a Spring Cloud project that makes it easy to run Spring Cloud apps in Cloud Foundry <https://cloud.spring.io/spring-cloud-cloudfoundry/>

So, now we can start as many `pets-service` instances as we need, by executing the following code:

```
java -jar build/libs/ps-micro-1.1.0.RELEASE.jar pet PORT
```

In Figure 8-8, you can see that two `pets-service` instances were started and registered, the default one on port 4000, and a second one on port 4003.

DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
PETS-SERVICE	n/a (2)	(2)	UP (2) - 192.168.56.1:pets-service:4000 , 192.168.56.1:pets-service:4003
USERS-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.1:users-service:4001
WEB-SERVICE	n/a (1)	(1)	UP (1) - 192.168.56.1:web-service:4002

Figure 8-8. Two different `PETS-SERVICE` microservices registered with the Eureka Server

Microservices Communication

It has already been said that microservices processes communicate using agnostic protocols such as REST. The `users-service` and `pets-service` both expose RESTful interfaces over HTTP (although different protocols can be set up as well: JMS or AMQP, for example), and in the presented example, another service called `web-service` was introduced that uses them to access their data. To consume RESTful services, Spring provides the `RestTemplate` class, which can be used to send HTTP requests to a RESTful server and fetch data in a number of formats such as XML and JSON. For the simplicity of the application, we'll use the default format for the data, which is XML. The `web-service` microservice client uses a `RestTemplate` to connect and request data from the other two registered microservices while remaining agnostic as to their location and the exact URL, since Spring will automatically configure the locations for it.

The implementation of the `web-services` is a little different, since a web interface is configured for it. The Eureka server uses FreeMarker templates by default, so if a different implementation is desired, these have to be first ignored via configuration, by setting the `spring.application.freemarker.enabled` to `false`. The configuration file for the `web-service` is depicted in the following code snippet.

```
spring:
  application:
    name: web-service
  freemarker:
    enabled: false      #Do not use FreeMarker templates
    thymeleaf: #Thymeleaf templates will be used
    cache: false
    prefix: classpath:/web-server/templates/
```

```

error:
  path=/error

eureka:
  instance:
    leaseRenewalIntervalInSeconds: 5
  client:
    serviceUrl:
      defaultZone: http://localhost:3000/eureka/

# HTTP Server
server:
  port: 4002 # HTTP (Tomcat) port

```

The central class of this microservice is the `AllWebController`, which is the one that uses the `AllWebService` that requests information from the `pets-service` and the `users-service` microservices, and wraps it up in objects that can be rendered in a Thymeleaf view. The code for the `AllWebController` is nothing special, just a typical Spring controller class using a service instance to access data.

```

@Controller
public class AllWebController {

    private static Logger logger = LoggerFactory.getLogger(AllWebController.class);

    @Autowired
    private AllWebService allWebService;

    public AllWebController(AllWebService allWebService) {
        this.allWebService = allWebService;
    }

    @RequestMapping("/all/{ownerId}")
    public String byOwner(Model model,
        @PathVariable("ownerId") Long ownerId) {
        UserSkeleton owner = allWebService.findUserById(ownerId);
        if (owner != null) {
            owner.setPets(allWebService.findByOwnerId(ownerId));
        }
        model.addAttribute("owner", owner);
        return "all";
    }

    @RequestMapping("/pets/{type}")
    public String byOwner(Model model,
        @PathVariable("type") String type) {
        List<PetSkeleton> pets = allWebService.findByType(type);
        model.addAttribute("pets", pets);
        model.addAttribute("type", type);
        return "pets";
    }
}

```

The implementation for this service is depicted in the following code snippet.

```

package com.ps.web;

import com.ps.ex.UserNotFoundException;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.stereotype.Service;
import org.springframework.web.client.HttpClientErrorException;
import org.springframework.web.client.RestTemplate;
...

@Service
public class AllWebService {

    private static Logger logger = LoggerFactory.getLogger(AllWebService.class);

    @Autowired
    @LoadBalanced
    private RestTemplate restTemplate;

    private String petsServiceUrl;
    private String usersServiceUrl;

    public AllWebService(String usersServiceUrl, String petsServiceUrl) {
        this.usersServiceUrl = usersServiceUrl;
        this.petsServiceUrl = petsServiceUrl;
    }

    public UserSkeleton findUserById(Long id) {
        logger.info("findUserById(" + id + " ) called");
        User user = null;
        try {
            user = restTemplate.getForObject(usersServiceUrl
                + "/users/id/{id}", User.class, id);
        } catch (HttpClientErrorException e) {
            // no user
            throw new UserNotFoundException(id);
        }
        return new UserSkeleton(user.getId(), user.getUsername());
    }

    public List<PetSkeleton> findByOwnerId(Long ownerId) {
        logger.info("findByOwnerId(" + ownerId + " ) called");
        Pet pets = null;
        try {
            pets = restTemplate.getForObject(petsServiceUrl
                + "/pets/owner/{id}",
                com.ps.pet.Pet.class, ownerId);
        } catch (HttpClientErrorException e) {
            // no pets
        }
    }
}

```

```

        if(pets == null || pets.length ==0) {
            return null;
        }
        List<PetSkeleton> petsList = new ArrayList<>();
        for (Pet pet : pets) {
            petsList.add(new PetSkeleton(pet.getName(),
                pet.getAge(), pet.getPetType().toString()));
        }
        return petsList;
    }
}

```

The previous implementation is a normal Spring service class, annotated with the `@Service` stereotype annotation. It is a wrapper for a `RestTemplate` instance that accesses two beans that are references to the microservices `pets-service` and `users-service`. The classes `UserSkeleton` and `PetSkeleton` are stripped versions of the entity classes and were introduced to simplify the implementation a little, since the focus of the section is on the microservices' implementation and logic. The code for the two classes is nothing but the most basic implementation of a POJO and is path of the `com.ps.web` package from the `17-ps-micro-sample`.

The only novelty is the `@LoadBalanced` annotation, which marks the injected `RestTemplate` bean to be configured to use a `LoadBalancerClient` implementation.

As you probably recall from the previous chapter, the `RestTemplate` is thread-safe, so it can be used to access any number of services in different parts of an application. In this example, two microservices are accessed by the same `RestTemplate` bean.

The `petsServiceUrl` and `usersServiceUrl` should be resolved before being injected into the `AllWebService` bean. For this to happen, the auto-scanning on the main Spring Boot class must be disabled, and beans must be declared explicitly.

```

@SpringBootApplication
@EnableDiscoveryClient
@ComponentScan(useDefaultFilters = false) // Disable component scanner,
// needed because of the microservices we need access to
public class WebServer {

    private static Logger logger = LoggerFactory.getLogger(WebServer.class);

    public static final String USERS_SERVICE_URL = "http://USERS-SERVICE";

    public static final String PETS_SERVICE_URL = "http://PETS-SERVICE";

    public static void main(String args) throws IOException {
        // Tell server to look for web-server.properties or web-server.yml
        System.setProperty("spring.config.name", "web-server");
        ConfigurableApplicationContext ctx =
            SpringApplication.run(WebServer.class, args);
        assert (ctx != null);
        logger.info("Started ...");
        System.in.read();
        ctx.close();
    }
}

```

```

@Bean
RestTemplate restTemplate() {
    return new RestTemplate();
}

@Bean
public AllWebService allWebService() {
    return new AllWebService(USERS_SERVICE_URL, PETS_SERVICE_URL);
}
@Bean AllWebController allController() {
    return new AllWebController(allWebService());
}

@Bean
public HomeController homeController() {
    return new HomeController();
}
}

```

On starting this application, the service URLs are provided by the main program to the AllWebService.

■ ! In case you are confused as to why the term URL is used everywhere, although we are working with REST, here is the explanation. The two microservices URLs <http://USERS-SERVICE> and <http://PETS-SERVICE> are resolved to <http://localhost:4001> and <http://localhost:4000>, so they are in fact URLs (locations). They become URIs (resources) when concatenated with `/users/id/{id}` and `/pets/owner/{id}`.

The two microservices are identified by their application names, those that were set in their configuration files and that were used to register themselves with the discovery-server. The casing is not important, but it makes it obvious that PETS-NAME is a logical host that will be obtained via discovery and not an actual hostname. Also, previously, when the microservices details were depicted, you could see that the app name in the `<app>` element was depicted in uppercase as well. So this makes it quite obvious which service should be discovered and will be used further. In this example, the logical hostnames were hard-coded, but in production applications they are usually configured externally.

Since we are using Netflix software, the RestTemplate bean will be intercepted and auto-configured by Spring Cloud to use a custom HttpClient that uses Netflix Ribbon⁴ to do the microservice lookup.

The RibbonLoadBalancerClient implementation is the one that takes the logical service name (as registered with the discovery server) and converts it to the actual hostname of the chosen microservice. The method that does this is called `reconstructURI`, and the code is depicted in the following code snippet.

```

package org.springframework.cloud.netflix.ribbon;

import org.springframework.cloud.client.ServiceInstance;
import org.springframework.cloud.client.loadbalancer.LoadBalancerClient;
import org.springframework.cloud.client.loadbalancer.LoadBalancerRequest;
...

```

⁴Ribbon is yet another Netflix OSS. It provides client-side software load-balancing algorithms. More about it here: <https://github.com/Netflix/ribbon/wiki>.

```

public class RibbonLoadBalancerClient implements LoadBalancerClient {
    public URI reconstructURI(ServiceInstance instance, URI original) {
        Assert.notNull(instance, "instance can not be null");
        String serviceId = instance.getServiceId();
        RibbonLoadBalancerContext context = this.clientFactory
            .getLoadBalancerContext(serviceId);
        Server server = new Server(instance.getHost(), instance.getPort());
        IClientConfig clientConfig = clientFactory.getClientConfig(serviceId);
        ServerIntrospector serverIntrospector = serverIntrospector(serviceId);
        URI uri = RibbonUtils.updateToHttpsIfNeeded(original, clientConfig,
            serverIntrospector, server);
        return context.reconstructURIWithServer(server, uri);
    }
    @Override
    public ServiceInstance choose(String serviceId) {
        Server server = getServer(serviceId);
        if (server == null) {
            return null;
        }
        return new RibbonServer(serviceId, server, isSecure(server, serviceId),
            serverIntrospector(serviceId).getMetadata(server));
    }
}
...
}

```

Ribbon can be used to select from multiple instances of the same service, the proper one to use, meaning the one to respond most rapidly, and this is usually done in `ClientHttpRequestFactory` by first calling the `choose(...)` method, also presented in the code snippet above.

More Novelties

Writing a microservices-based distributed application requires advanced knowledge of Spring Boot and Spring Cloud. The `17-ps-micro-sample` is detached from the `pet-sitter` project, because of its special setup and all the novelty software that is used. The Gradle setup is different, and many new annotations have been introduced, which should be explained before you are asked to solve the TODO tasks for this project. So here goes:

- The databases for the two microservices are separate, and even if JPA is used, a one-to-many relationship cannot be established between the `P_USER` and the `P_PET` in the classical way, using `@OneToMany` and `@ManyToOne` annotations, so the fact that a pet belongs to a certain user is implemented using a long field containing the owner ID. The `AllWebService` implementation starts with a user ID and checks its existence on the user database by requesting that information from the `users-service` microservice. If the user is found, it then uses the user ID as a search criterion for the `pets-service` microservice.
- `@EntityScan` configures the base packages used by auto-configuration when scanning for entity classes. This is a Spring Boot annotation that “under the hood” configures a `LocalContainerEntityManagerFactoryBean` and sets the `packagesToScan` property to the value of the annotation default attribute.

- The versions of the libraries used to build the application are externalized into the `gradle.properties` file. This is yet another way in which Gradle configuration can be done.
- Thymeleaf was used for the web interface, because it is supported by Eureka and little configuration is needed to get it working.

After all the application instances are in place, you can start playing with them. Try to access:

- `http://localhost:4000/pets/`: displays all pet information in XML format
- `http://localhost:4000/pets/1122664455`: displays information about pet with RFID=1122664455 in XML format
- `http://localhost:4000/pets/owner/1`: displays all pets for owner with id=1 in XML format
- `http://localhost:4001/users/`: displays all user information in XML format
- `http://localhost:4002/`: displays the index page of the web-service microservice with clickable URLs to test the access to `pets-service` and `users-service` microservices; the index page is depicted in Figure 8-9



Pet Sitter Microservices Demo - Web service that accesses both microservices

Overview

- This is a simple web application that accesses two microservices to extract data
- All the information is fetched using a RESTful interfaces to microservices.
- [Test: Owner with pets](#)
- [Test: Owner without pets](#)
- [Test: Owner does not exist](#)

Copyright © 2016 by Iuliana Cosmina and Apress. All rights reserved.

Figure 8-9. Index page for the web-service

Practice Section

The `17-ps-micro-sample` contains a method in the `PetController` class that is not fully implemented. This method should extract the pets that have a common type.

```
@RequestMapping("/type/{type}")
public List<Pet> byPetType(@PathVariable("type") String type) {
// TODO 62
return null;
}
```

Complete the method body so that when the `http://localhost:4000/pets/type/DOG` request is sent to the microservice, all dog information in the system will be returned.

Task TODO 63, located in the `AllWebService` class, requires you to complete the implementation of this method so that when `http://localhost:4002/pets/DOG` is accessed, a page with all dogs in the system will be displayed.

Summary

After reading this chapter you should have a proper understanding of how Spring can be used to develop distributed applications based on the microservices architecture. Below, you can find a simple list of topics that you should keep handy when reviewing what you have learned.

- Describe the microservices architecture.
- Spring Cloud is built on Spring Boot.
- What are typical Spring cloud annotations used in developing microservices applications?
- Netflix has developed and outsourced numerous projects designed to support development and running of distributed applications.
- How is Service Discovery set up?
- How is a microservice RESTful interface accessed?
- What is Eureka?
- What is Ribbon?

Quick Quiz

Question 1: Which of the following affirmations describe the microservices architecture?

- A. In a microservices architecture, services should have small granularity, and the communications protocols should be lightweight.
- B. In a microservices architecture, the interprocess communication can be done only using RESTful interfaces.
- C. In a microservices architecture, the design is unified, and components are interconnected and interdependent.

Question 2: Which of the following are advantages of the microservices architecture?

- A. Microservices-based applications are highly scalable.
- B. There is improvement in fault isolation.
- C. Transaction management is not needed.
- D. Deploying microservices-based application is painless.

Question 3: Which of the following is the core component of a microservices-based distributed application?

- A. a service implementation
- B. a database
- C. a registry and discovery server

Question 4: What can be said about Spring Cloud?

- A. It is built on Spring Boot.
- B. It is an umbrella project that provides tools for developers to quickly build some of the common patterns in distributed systems.
- C. It is a proprietary Pivotal service for building distributed applications available only by paid subscription.

Question 5: Which of the following annotations is used to declare an instance of a Eureka server?

- A. `@EnableNetflixEureka`
- B. `@EurekaAutoConfiguration`
- C. `@EnableEurekaServer`

Question 6: Which of the following annotations is used for service registration and discovery with a discovery server?

- A. `@EnableDiscoveryClient`
- B. `@EnableEurekaClient`
- C. `@EnableSeviceRegistration`

Index

■ A

- Access Control List (ACL), 324
- Apache ActiveMQ, 367–370
- Apache Tomcat, 294, 298–299, 358
- Aspect Oriented Programming (AOP)
 - advice implementation
 - after advice, 176–177
 - around advice, 177–178
 - before advice, 172
 - IllegalArgumentExceptio method, 173
 - returning, 173–174
 - target method, 173
 - UML sequence diagram, 172
 - aspects, 153
 - business/base code, 153
 - cross-cutting concerns, 153
 - problem solving
 - code tangling and scattering, 157
 - conceptual schema, 156
 - database implementation, 154
 - repository components, 156
 - transactions, 156
 - UML call diagram, 155
 - spring
 - advice implementation, 172–177
 - application, 164
 - AspectJ, 157
 - aspect support configuration, 165
 - @EnableAspectJAutoProxy method, 161
 - findById method, 160
 - framework, 157
 - IDEA debugger view, 164
 - injected bean, 163
 - JdbcTemplateUserRepo
 - method, 159–160, 179
 - JdkDynamicAopProxy method, 163
 - libraries, 158
 - limitation, 178
 - pointcut definition, 165–171
 - proxy nature via UML sequence diagram, 180–181

- proxyTargetClass parameter, 161
- terminology, 158–159
- testFindById method, 162
- testProxyBubuDeps method, 180
- updateDependencies method, 180
- UserRepoMonitor method, 162
- userTemplateRepo proxy bean, 163

■ B

- Bean LifeCycle and configuration
 - @Bean and @Component methods, 112
 - bean definition, 112
 - configuration class, 113
 - connections, 17
 - default scope, 111
 - dependency injection, types of, 11, 117
 - development style
 - AbstractRepo interface, 18
 - contextual menu, 23
 - IntelliJ IDEA console, 23
 - JUnit test failure, 23
 - operations, 19
 - production and test environment, 20
 - repository interfaces hierarchy, 17–18
 - sequence of, 22
 - service interfaces and implementations, 19–20
 - SimpleOperationsService class
 - definition, 21
 - stub implementations, 18
 - test environment, 21
 - UserRepo interface, 21
 - factory post processor, 113
 - IoC (*see* Inversion of Control (IoC))
 - meaning, 17, 111
 - quizBean bean definition, 112
 - Spring applications, 110
 - Spring IoC container, 114
 - stereotype annotations, 112
- Bean LifeCycle and configuration. *See* Spring configuration

■ **C**

- Constructor injection
 - bean definition, 31
 - code and configuration, 31
 - ComplexBeanImpl class, 31
 - ComplexBean types, 34
 - configuration element, 33
 - handle constructors, 32
 - index attribute, 32
 - value attribute, 32

■ **D**

- DAO layer, 272
- Data access
 - HibernateORM (*see* Object relational mapping (ORM))
 - JDBC
 - database, 187
 - implementation, 187
 - source code, 187–189
 - NoSQL, 185
 - software application architecture, 185–186
 - Spring (*see* Spring data access)
 - transactional environment
 - abstract schema, 210
 - hibernate configuration, 210
 - JPA configuration, 211
 - JTA Spring environment, 212
 - local JDBC configuration, 210
 - properties, 209
 - transaction manager, 210
- Data access exceptions. *See* Spring data access
- Data Definition Language (DDL), 206
- Data Manipulation Language (DML), 206
- DataSourceTransactionManager, 213
- Dependency injection
 - bean definition, 40
 - CollectionHolder, 41–42
 - collection properties, 42
 - conversion definition, 39
 - injected elements, 43
 - logger.info statement, 43–44
 - MultipleTypesBean type, 40–41
 - primitive types, 38
 - PropertyEditor implementations, 38
 - reference wrapper types, 38
 - set element, 43
 - SimpleBean elements, 43
 - util namespace, 44
 - XML configuration, 38
- Dependencyinjection. *See* Inversion of Control (IoC)
- Declarative transactions, 217

■ **E**

- EntityManager JPA methods, 251
- Exception handling, 402–403

■ **F**

- findById() method, 156

■ **G**

- Gradle 2.x project, 8–10

■ **H**

- Hibernate configuration, 210
- HibernateTransactionManager, 213
- HTTP message converters, 404–405

■ **I**

- Inversion of Control (IoC)
 - application context, 26–27
 - application development process, 24
 - configuration file, 28
 - datasource.properties file, 28
 - repository, 25–26
 - UserService implementation, 25
 - XML Spring configuration file, 26

■ **J, K, L**

- Java configuration and annotations
 - bean naming
 - @AliasFor, 97
 - @Bean and @Component method, 96
 - JavaConfigurations, 94–95
 - loginTimeout method, 97
 - meta-annotation, 98
 - source code, 98
 - beans
 - creation, 107
 - definitions loading, 107
 - dependencies, 107
 - lifecycle and scopes, 106
 - constructor injection, 102
 - core annotation, 87–88
 - autowiring and initialization, 88
 - JSR 250 annotations, 89
 - stereotypes, 88
 - definition, 85
 - dependency injection
 - advantages, 110
 - @Bean annotation, 108
 - dataSource method, 105

- initMethod attribute, 108
 - @Lazy, 109
 - transaction, 109
 - @Value annotation, 106
 - XML, 107
- field, constructor and setter injection, 98–101
- multiple sources
 - AnnotationConfigApplicationContext
 - method, 93
 - @ContextConfiguration method, 91
 - @Import annotation, 93
 - @ImportResource annotation, 91–92
 - jdbcRequestRepo method, 93–94
 - PropertySource annotation, 89, 90
- prefixes and corresponding paths, 105
- setter injection, 103
- source code, 86
- Spring evolution, 86
- Spring security section, 104
- test class, 87
- Java Database Connectivity (JDBC)
 - database, 187
 - implementation, 187
 - source code, 187–189
- Java Data Objects (JDO), 247
- Java Management Extensions (JMX)
 - architecture
 - layers, 418, 421–422
 - MBeans, 422–423
 - plain JMX, 423
 - spring JMX, 419, 424–427, 429–431
 - definition, 350, 421
 - JDK versions 5.0 and 6, 421
 - MBeans, 421
- Java Messaging Service (JMS)
 - Apache ActiveMQ, 367–370
 - API programming model, 363
 - client application, 362
 - components, 362
 - connections and sessions, 363–364
 - description, 350
 - destinations, 365–367
 - JmsTemplate
 - ActiveMQ web application, 376–377
 - advantages, 370–371
 - connection factories and destinations, 371–372
 - connection factory, queues and message converter, 375–376
 - convertAndSend methods, 374
 - JmsCommonConfig.java, 375
 - MessageConverter, 371
 - producer/consumer, 375
 - pubSubNoLocal property, 372
 - userQueue, 374
 - UserReceiver class, 372–373
 - UserSender class, 373
 - messages, 364–365
 - SpringBoot (*see* Spring Boot JMS application)
- Java Open Transaction Manager (JOTM), 234
- Java Persistence API (JPA), 235
 - Apache OpenJPA, 248
 - components, 247
 - data nucleus, 248
 - definition, 247
 - EclipseLink, 248
 - entity manager factory, 247
 - hibernate, 248
 - JTA and JNDI, 255–256
 - MongoDB and Spring project
 - basePackages attribute, 263
 - column family stores, 260
 - db.user.find() function, 264–265
 - document databases, 260
 - graph databases, 260
 - key-values stores, 260
 - NoSQL databases, 260
 - operating system, 261
 - source code, 261–262
 - user data manipulation, 263–264
 - UserRepo interface, 262
 - persistence context, 247
 - persistence unit, 247
 - provider, 248
 - queries, 252–253, 255
- Spring configuration
 - conceptual UML sequence diagram, 251–252
 - debug mode, 250
 - EntityManager mapped, 249
 - EntityManager JPA methods, 251
 - Entity Manager operations, 249
 - SessionFactory bean, 248
- Spring project
 - definition, 256
 - instant repositories, 258
 - MongoDB, 260
 - NoRepositoryBean annotation, 258
 - repository hierarchy, 256–258
 - source code, 259
- Java Persistence Query Language (JPQL), 251
- Java Remote Method Protocol (JRMP), 352–353
- Java Web application architecture, 271
- JdbcTemplate
 - abstraction framework, 190
 - configuration file, 190
 - embedded database, 192
 - JdbcTemplateUserRepo code, 194–195
 - location files, 196–197
 - NamedParameterJdbcTemplate

JdbcTemplate (*cont.*)

- DDL database, 206
 - DELETE, 206
 - DML, 206
 - findById method, 204
 - INSERT, 205
 - relationship, 203
 - SELECT queries, 205
 - source code, 204, 206
 - UPDATE, 205
 - project structure, 197–198
 - queries
 - callback approach, 200
 - HTMLUserRowCallbackHandler, 201
 - ORM, 200
 - queryForObject, 198
 - ResultSet, 198–199
 - source code, 203
 - testing and auditing, 199
 - UserWithPetsExtractor, 202
 - sequece steps, 196
 - source code, 190–191
 - TestJdbcTemplateUserRepo code, 195–196
 - XML source code, 193–194
- JMX. *See* Java Management Extensions (JMX)
- JpaTransactionManager, 213
- JtaTransactionManager, 213

■ **M, N**

Microservices

- 17-ps-micro-sample
 - application, 448
 - architecture, 444
 - structure, 442–443
- advantages, 435–436
- classic application, 435
- communication
 - AllWebController, 452
 - AllWebService, 452–455
 - PETS-NAME, 455
 - protocols, 451
 - RestTemplate, 454
 - Ribbon, 456
 - RibbonLoadBalancerClient
 - implementation, 455–456
 - Spring Boot class, 454–455
 - UserSkeleton and PetSkeleton, 454
 - web-services, 451–452
- description, 435
- Eureka clients, 446
- functionality, 442
- monolithic application, 435
- Netflix GitHub page, 447
- per-service and user-service, 444

- PetController class, 457
- PetServer class, 447
- pet-sitter project, 456–457
- popularity, 435
- registration and discovery
 - server, 439–441, 448–451
- RESTful interface, 444
- spring
 - characteristics, 438
 - database implementation strategies, 437
 - monolith architecture, 437
 - Spring Boot, 438
 - Spring Cloud, 438–439
 - Spring Cloud Netflix, 439
 - technologies, 438
 - Spring Boot, 445, 448

Mock object

- EasyMock, 126
 - findByName method, 126–127
 - SimpleUserService, 125
 - TestObjectsBuilder, 126
- jMock
 - findAllByUser method, 128–129
 - generic types, 128
 - SimpleRequestService, 127
- libraries and frameworks, 124
- Mockito
 - findAllByUser method, 131
 - InjectMock methods, 133
 - SimpleReviewService method, 130
 - static methods, 133
 - verify and times methods, 132
- PowerMock, 133

Model-View-Controller (MVC)

- design paradigm, 273
- behavior, 272
- Java Configuration, 286–288
- software, 272

■ **O**

Object relational mapping (ORM)

- benefits, 245–246
- cache management, 246
- domain object, 245
- entries, 245
- exception mapping, 243–245
- framework, 235
- JPA (*see* Java Persistence API (JPA))
- one-to-many relationships, 245
- queries
 - AbstractEntity, 240
 - repository class, 243
 - dataSource bean, 237
 - hibernateProperties, 237

- Hibernate Query Language, 241
- HQL queries, 241
- Session instances, 243
- SessionFactory, 238, 240–242
- synchronize domain objects, 242
- session and hibernate configuration
 - entities, 238
 - @ManyToOne annotation, 239–240
 - @OneToMany annotation, 239
 - source code, 236
 - Spring application, 235–236

■ P, Q

- Password-salting (encryption method), 310
- Plain Old Java Objects (POJOs), 17
- Pet Sitter project
 - account type, 11
 - application layers, 13
 - entity class hierarchy, 14
 - modules, 11–12
 - UML diagram, 15
- Pointcut expression
 - advice target object, 170
 - definition, 165–166
 - identify methods, 166
 - JointPoint methods, 171
 - named pointcuts, 168–169
 - parameter methods, 166
 - repoUpdate and serviceUpdate
 - method, 169
 - return type, 166–167
 - update methods, 168–169
 - UserRepo method, 167
- Pseudoobject. *See* Mock object

■ R

- Remote Method Invocation (RMI), 351–352, 356–358
- Remoting
 - client and server applications, 350–351
 - configuration
 - bean of type, 355
 - client application, 356–357
 - Hessian protocol, 358, 361–362
 - HessianProxyFactoryBean, 358
 - HessianServiceExporter, 358
 - Http Invoker classes, 358, 360
 - Http Invoker proxy, 361
 - HttpInvokerProxyFactoryBean, 359
 - HttpInvokerServiceExporter, 359
 - HTTP Invoker test, 361
 - HTTP methods, 358
 - IntelliJ IDEA test execution, 357

- Java, 353–354, 356
 - properties, 354
 - proxy points, 360
 - RmiExporterBootstrap, 355
 - RmiServiceExporter and
 - RmiProxyFactoryBean, 352–353, 358
 - server application, 354–355
 - web application, 360
 - XML, 353
- Java marshaling, 352
- RMI model, 351–352
- service beans, 350
- and web services, 349
- Representational state transfer (REST)
 - advantages, 416–417
 - architecture, 396–397
 - description, 349
 - exception handling, 402–403
 - HTTP message converters, 404–405
 - HTTP specifications, 396
 - mechanisms, 395
 - message converters, 396
 - requests and responses, 397
- RestTemplate
 - controller method, 415
 - DELETE method, 414–415
 - exchange method, 408, 409
 - execute method, 408
 - GET method, 410
 - HTTP methods, 407
 - JMock and Mockito components, 415–416
 - message converters, 404, 409
 - POST method, 411–412
 - PUT method, 412–414
 - ResponseEntity object, 409
 - StandaloneRestControllerTest
 - class, 415
 - URI templates, 408
- SoapUI IntelliJ IDEA plugin, 397
- SoapUI Navigator interface, 395
- Spring Boot, 418–420
- Spring MVC, 405–407
- spring support
 - controller methods, 400
 - DELETE methods, 400
 - getLocationForUser method, 399
 - HTTP status codes, 399–401
 - JAX-RS 2.0, 397
 - @ResponseBody, 397
 - @RestController, 398
 - SoapUI Navigator interface, 397
 - Spring MVC, 401
 - URL, 398–399
- TODO tasks, 417–418
- web applications, 395

REST. *See* Representational state transfer (REST)

RMI. *See* Remote method invocation (RMI)

RootApplicationContext, 275

S

Setter injection

- advantage, 36
- bean definition, 34–35
- creation (bean), 37
- dependencies, 34
- setSimpleBean method, 35
- XML definition, 36

Simple Object Access Protocol (SOAP), 382, 384–386, 394–395

SOAP. *See* Simple Object Access Protocol (SOAP)

Spring Boot, 326, 438

- configuration, 327, 329–331, 333–338
 - logging, 341
 - using YAML, 338–340
- testing, 341–344

Spring Boot JMS application

- abstract schema, 378
- ActiveMQ web application, 377–378
- Application class, 379
- ConfirmationSender class, 379
- JmsTemplate object, 380
- MappingJackson2MessageConverter, 380–381
- mechanism, 382
- MessageConverter interface, 380
- message listener, 381
- message type, 381
- UserReceiver and ConfirmationReceiver
 - classes, 378–379

Spring Boot WS application, 387–390

Spring Cloud Netflix, 439

Spring configuration

- application context
 - application-configuration.xml file, 50
 - ApplicationContext implementation, 48
 - bean definitions, 49
 - configuration files, 52
 - directories, 51
 - Gradle view, 52
 - in-memory database, 50
 - prefixes and corresponding paths, 48
 - professional database, 50
 - resources directory, 51
 - test classes, 48
 - types, 49
 - unit testing, 50
 - XML files, 49

bad bean naming, 62

bean definition inheritance, 57

- configuration file, 57

overrides, 58

bean factories

- FactoryBean interface, 47
- getObject method, 46
- static method, 45

bean scopes

- AOP framework, 85
- configuration file, 84
- context classes, 83
- scope attribute, 83–84
- session scope, 84
- singleton design pattern, 84

context and bean lifecycle

- application lifecycle, 66–67
- initialization phase, 65
- phases, 64

expression language, 57

hood

- afterProperties method, 73
- afterPropertiesSet() method, 72
- bean creation steps, 67–68
- BeanFactoryPostProcessor method, 69
- close method, 80
- CommonAnnotationBeanPostProcessor, 76
- ComplexBean class, 74–75
- configuration file, 82–83
- context namespace, 73
- dependencies, 78
- destroy method, 80
- DisposableBean method, 81
- finalize method, 79
- initialization stage, 71
- initMethod method, 71, 77
- InstantiatingBean interface, 72
- IntelliJ IDEA, 78
- load bean definition, 68
- log files, 76
- @PostConstruct method, 73, 75
- PropertyPlaceholderConfigurer, 69
- stages, 70

inner beans, 58–59

Java (*see* Java configuration and annotations)

name definition

- application context, 61, 63
- bean definition, 61
- configuration file, 60, 62
- convention, 61
- CustomDateEditor method, 59
- detailed explanation, 62
- getBeanDefinitionNames()
 - method, 60
- getBeansOfType(...) method, 60
- getBean(...) method, 60
- override, 63

- namespaces
 - memory datasources, 54
 - read properties, 54
 - typed collections, 53
 - util namespace, 53
 - reading properties
 - advantage, 56
 - bean namespace, 55
 - context file, 55
 - util namespace, 56
 - schema declaration, 29
 - XML configuration
 - application context, 48–52
 - bean definition, 30
 - bean factories, 45–47
 - constructor injection, 31–34
 - dependency injection, 38–39, 41, 43–45
 - setter injection, 34–37
 - XML-based metadata, 29
 - Spring data access
 - exceptions
 - branches, 208
 - data sources, 207
 - hierarchy, 207
 - non-transient, 208
 - TODO view, 209
 - transient, 208
 - JdbcTemplate (*see* JdbcTemplate)
 - repository classes interact, 189
 - transaction management
 - atomic execution, 218–219
 - @BeforeTransaction, 226
 - conceptual UML sequence diagram, 213–214
 - configuration support, 214
 - console log, 220
 - declarative model clarification, 228–232
 - declare transactional methods, 215–217
 - dirty reads, 221
 - distributed transaction, 233–234
 - isolation attribute, 221
 - manager framework, 213
 - nested transaction, 218, 220
 - noRollbackFor attribute values, 222
 - phantom reads, 222
 - programatic transaction model, 232–233
 - propagation attribute, 218
 - readOnly attribute, 217
 - repeatable read, 221
 - @Rollback annotation, 225
 - rollbackFor attribute values, 222
 - serialization, 222
 - testing methods, 223–225
 - third-party implementation, 226–227
 - @Transactional annotation, 222–223
 - @TransactionConfiguration, 226
 - transactionManager attribute, 217
 - UML sequence diagram, 212–213
 - Spring project, 1
 - certification exam, 3–4
 - core components, 3
 - development environment
 - build tool, 8–10
 - IDE, 10
 - JVM, 8
 - Pet Sitter, 11–12, 14–15
 - flash drive, 4
 - frameworks, 1
 - infrastructure beans, 1
 - Java application, 2
 - libraries, 1
 - nutshell, 3
 - objectives, 3
 - structure
 - author information, 7
 - code downloads, 7
 - conventions, 7
 - list of, 5–6
 - modules, 6
 - use of, 5
 - web stack, 2
 - Spring Security, 298–300
 - chained filters, 303–304, 316
 - configuration, 301
 - Java Configuration, 313–317
 - method security, 321–325
 - security tag library, 317–320
 - XML configuration, 301–312
 - Spring Web, 271
 - App Configuration, 274–276
 - application, 291
 - controllers, 277–281
 - @MVC, 285–286
 - running application, 291–292
 - running with jetty, 292–293
 - running with Tomcat, 294–295, 297
 - steps to develop MVC
 - application, 276
 - XML, 281–285
 - Spring WebFlow, 271, 274
 - Spring Web MVC application, 344
- **T**
- Testing
 - integration test, 117, 146
 - Mockito methods
 - Gradle test task, 148–149
 - IntelliJ IDEA, 148
 - pet-sitter project build, 150–151
 - web report, 149–150

Testing (*cont.*)

- mock object
 - EasyMock, 125–127
 - jMock, 127–129
 - libraries and frameworks, 124
 - Mockito, 130–133
 - PowerMock, 133
- Spring application
 - buildPet method, 136
 - configuration classes, 139
 - @ContextConfiguration annotation, 139
 - integration test, 136–137
 - Java Configuration classes, 138
 - mapping interface, 141
 - profiles, 144–146
 - repo dependency, 138–139
 - rules, 142
 - simplePetService, 134–135, 138
 - source code, 140
 - spring-test, 134
 - stereotype annotations, 138
 - test class, 143–144
 - test context, 142
 - TestObjectsBuilder method, 136
- stubs
 - abstract class, 119–120
 - assert* methods, 124
 - delete methods, 123
 - JUnit components, 124
 - PetRepo methods, 120, 121
 - SimplePetService, 117–118
 - SimplePetServiceTest methods, 122–123
- test-driven development, 115, 116
- types of, 115
- unit testing, 116–117, 147
- Tomcat, 294–295, 297–299, 332, 334
- Transaction management
 - configuration support, 214–215
 - declare transactional methods, 215–217
 - @EnableTransactionManagement, 215
 - explicit exception, 216
 - transactionManager attribute, 217
 - console log, 220–221
 - declarative model
 - clarification, 228
 - IDEA execution, 229
 - repository transactional bean, 231–232
 - service method, 228, 230
 - @Transactional service class, 229
 - distributed transaction, 233–234
 - isolation attribute, 221
 - manager framework, 213
 - nested transaction, 218–220
 - noRollbackFor attribute values, 222

- operation, 214
- programatic transaction model, 232–233
- propagation attribute
 - MANDATORY, 218
 - NESTED, 218
 - NEVER, 218
 - NOT_SUPPORTED, 218
 - REQUIRED, 218
 - REQUIRES_NEW, 218
 - SUPPORTS, 218
- readOnly attribute, 217
- rollbackFor attribute values, 222
- testing methods, 223–225
 - @BeforeTransaction, 226
 - defaultRollback attribute, 226
 - repository methods, 225
 - @Rollback annotation, 225
 - third-party implementation, 226–227
 - @TransactionConfiguration, 226
 - @Transactional annotation, 222–223
- transactionManager attribute, 217
- UML sequence diagram, 212–213

■ **U, V**

- Uniform Resource Locator (URL), 278
- Unit testing, 116–117

■ **W**

- WebLogicJtaTransactionManager, 213
- WebMvcConfigurerAdapter class, 288
- Web services
 - description, 349, 382
 - designing and development, 383
 - Java Code with XJC, 386–387
 - and remoting, 349
 - and remotingfeatures, 382–383
 - and remotingXML, 382
 - SOAP messages, 382, 384–386
 - Spring Boot WS application, 387–390
 - testing, 392–395
 - WSDL, 391
- Web Services Description Language (WSDL), 391
- WSDL. *See* Web Services Description Language (WSDL)

■ **X**

- XML, Spring Web, 281–285

■ **Y, Z**

- YAML, 338–340