

Learn by doing: less theory, more results

Spring MVC

Second Edition

Unleash the power of the latest Spring MVC 4.x to develop a complete application

Beginner's Guide

Amuthan Ganeshan

[PACKT] open source PUBLISHING
community experience distilled

Spring MVC Beginner's Guide

Second Edition

Unleash the power of the latest Spring MVC 4.x to
develop a complete application

Amuthan Ganeshan



BIRMINGHAM - MUMBAI

Spring MVC Beginner's Guide

Second Edition

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2014

Second edition: July 2016

Production reference: 1220716

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-063-6

www.packtpub.com

Credits

Author **Copy Editor**

Amuthan Ganeshan Ameesha Smith-Green

Reviewer **Project Coordinator**

Rafał Borowiec Ulhas Kambali

Commissioning Editor **Proofreader**

Nadeem Bagban Safis Editing

Acquisition Editor **Indexer**

Vinay Argekar Rekha Nair

Content Development Editor **Production Coordinator**

Prashanth G Rao Melwyn Dsa

Technical Editor **Cover Work**

Murtaza Tinwala Melwyn Dsa

About the Author

Amuthan Ganeshan is a software engineer with more than nine years of experience specializing in building distributed applications. He currently works as a senior software engineer at Uptake. He is a big data enthusiast and loves sharing knowledge about software development and practices through his blog at www.codeculture.guru. He can be contacted at amuthan@codeculture.guru.

I would like to gratefully and sincerely thank Mr. Vincent Kok for his guidance, understanding, patience, and, most importantly, his friendship during my first job at Educator Inc. His mentorship has helped me to become a well-rounded professional. He encouraged me to not only grow as a developer, but also as an independent thinker.

I want to take a moment and express my gratitude to the entire team at Packt Publishing especially Murtaza Tinwala, Anish Dhurat, and Vinay Argekar, for their patience and cooperation. When I signed up for this book, I really had no idea how things would turn out. I couldn't have pulled this off without their guidance.

I would like to express my gratitude to all my friends and family for providing me with unending encouragement and support. I owe every challenge and accomplishment to all my lovely colleagues who taught me a lot over the years.

A special thanks to Divya and Arun for their encouragement, friendship, and support. They were a strong shoulder to lean on in the most difficult times during the writing of this book.

Finally, and most importantly, I would like to thank my wife Manju, who believes in me more than I do myself. Her support, encouragement, quiet patience, and unwavering love were undeniably the bedrock upon which my life has been built.

About the Reviewer

Rafał Borowiec is an IT specialist, specializing in software development, software testing and quality assurance, project management, and team leadership. He currently holds the position of a development manager at Goyello (goyello.com), where he is mainly responsible for building and managing teams of professional developers and testers.

He believes in agile project management and is a big fan of technology, especially technology that is Java-related (but not limited to this). Rafał likes sharing knowledge about software development and practices through his blog, blog.codeleak.pl, and Twitter (@kolorobot).

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Table of Contents

Preface	1
Chapter 1: Configuring a Spring Development Environment	9
Setting up Java	9
Time for action – installing JDK	10
Time for action – setting up environment variables	11
Configuring a build tool	12
Time for action – installing the Maven build tool	13
Installing a web server	15
Time for action – installing the Tomcat web server	15
Configuring a development environment	16
Time for action – installing Spring Tool Suite	16
Time for action – configuring Maven on STS	17
Time for action – configuring Tomcat on STS	18
Creating our first Spring MVC project	20
Time for action – creating a Spring MVC project in STS	21
Time for action – adding Java version properties in pom.xml	22
What just happened?	24
Spring MVC dependencies	25
Time for action – adding Spring jars to the project	25
What just happened?	27
A jump-start to MVC	29
Time for action – adding a welcome page	29
What just happened?	30
The Dispatcher servlet	31
Time for action – configuring the Dispatcher servlet	32
What just happened?	34
Deploying our project	35
Time for action – running the project	35
Summary	37
Chapter 2: Spring MVC Architecture – Architecting Your Web Store	38
Dispatcher servlet	38
Time for action – examining request mapping	39
What just happened?	40
Pop quiz – request mapping	41
Understanding the Dispatcher servlet configuration	41

Time for action – examining the servlet mapping	42
What just happened?	43
Servlet mapping versus request mapping	43
Pop quiz – servlet mapping	44
Web application context	44
View resolvers	45
Time for action – understanding web application context	46
What just happened?	47
Understanding the web application context configuration	49
Pop quiz – web application context configuration	50
Model View Controller	51
Overview of the Spring MVC request flow	52
The web application architecture	53
The Domain layer	53
Time for action – creating a domain object	54
What just happened?	58
The Persistence layer	59
Time for action – creating a repository object	60
What just happened?	65
The Service layer	70
Time for action – creating a service object	71
What just happened?	73
Have a go hero – accessing the product domain object via a service	75
An overview of the web application architecture	76
Have a go hero – listing all our customers	76
Summary	78
Chapter 3: Control Your Store with Controllers	79
The role of a Controller in Spring MVC	79
Defining a Controller	80
Time for action – adding class-level request mapping	81
What just happened?	82
Default request mapping method	83
Pop quiz – class level request mapping	84
Handler mapping	85
Using URI template patterns	86
Time for action – showing products based on category	86
What just happened?	88
Pop quiz – request path variable	90
Using matrix variables	91
Time for action – showing products based on filters	92
What just happened?	94

Understanding request parameters	96
Time for action – adding a product detail page	97
What just happened?	99
Pop quiz – the request parameter	101
Time for action – implementing a master detail View	101
What just happened?	103
Have a go hero – adding multiple filters to list products	104
Summary	105
Chapter 4: Working with Spring Tag Libraries	106
The JavaServer Pages Standard Tag Library	106
Serving and processing forms	107
Time for action – serving and processing forms	108
What just happened?	112
Have a go hero – customer registration form	116
Customizing data binding	116
Time for action – whitelisting form fields for binding	117
What just happened?	119
Pop quiz – data binding	121
Externalizing text messages	121
Time for action – externalizing messages	122
What just happened?	123
Have a go hero – externalizing all the labels from all the pages	124
Summary	124
Chapter 5: Working with View Resolver	125
Resolving Views	125
RedirectView	127
Time for action – examining RedirectView	127
What just happened?	128
Pop quiz – RedirectView	129
Flash attribute	130
Serving static resources	131
Time for action – serving static resources	131
What just happened?	132
Pop quiz – static view	133
Time for action – adding images to the product detail page	134
What just happened?	136
Multipart requests in action	136
Time for action – adding images to a product	137
What just happened?	139
Have a go hero – uploading product user manuals to the server	142
Using ContentNegotiatingViewResolver	142

Time for action – configuring ContentNegotiatingViewResolver	143
What just happened?	145
Working with HandlerExceptionResolver	147
Time for action – adding a ResponseStatus exception	148
What just happened?	149
Time for action – adding an exception handler	150
What just happened?	153
Summary	154
Chapter 6: Internalize Your Store with Interceptor	155
Working with interceptors	156
Time for action – configuring an interceptor	156
What just happened?	158
Pop quiz – interceptors	160
LocaleChangeInterceptor – internationalization	161
Time for action – adding internationalization	162
What just happened?	165
Have a go hero – fully internationalize the product details page	167
Mapped interceptors	167
Time for action – mapped intercepting offer page requests	168
What just happened?	172
Summary	174
Chapter 7: Incorporating Spring Security	175
Using Spring Security	175
Time for action – authenticating users based on roles	176
What just happened?	182
Pop quiz – Spring Security	186
Have a go hero – play with Spring Security	186
Summary	186
Chapter 8: Validate Your Products with a Validator	187
Bean Validation	187
Time for action – adding Bean Validation support	188
What just happened?	191
Have a go hero – adding more validation in the Add new product page	195
Custom validation with JSR-303/Bean Validation	195
Time for action – adding Bean Validation support	195
What just happened?	198
Have a go hero – adding custom validation to a category	200
Spring validation	200
Time for action – adding Spring validation	201
What just happened?	203

Time for action – combining Spring validation and Bean Validation	204
What just happened?	207
Have a go hero – adding Spring validation to a product image	209
Summary	210
Chapter 9: Give REST to Your Application with Ajax	211
Introduction to REST	211
Time for action – implementing RESTful web services	212
What just happened?	227
Time for action – consuming REST web services	230
What just happened?	235
Handling web services in Ajax	236
Time for action – consuming REST web services via Ajax	236
What just happened?	242
Summary	245
Chapter 10: Float Your Application with Web Flow	246
Working with Spring Web Flow	246
Time for action – implementing the order processing service	247
What just happened?	260
Time for action – implementing the checkout flow	261
What just happened?	264
Understanding flow definitions	264
Understanding checkout flow	265
Pop quiz – web flow	270
Time for action – creating Views for every view state	271
What just happened?	284
Have a go hero – adding a decision state	286
Summary	287
Chapter 11: Template with Tiles	288
Enhancing reusability through Apache Tiles	288
Time for action – creating Views for every View state	289
What just happened?	295
Pop quiz – Apache Tiles	298
Summary	298
Chapter 12: Testing Your Application	299
Unit testing	300
Time for action – unit testing domain objects	300
What just happened?	302
Have a go hero – adding tests for Cart	304
Integration testing with the Spring Test context framework	304

Time for action – testing product validator	304
What just happened?	307
Time for action – testing product Controllers	309
What just happened?	311
Time for action – testing REST Controllers	312
What just happened?	314
Have a go hero – adding tests for the remaining REST methods	315
Summary	316
Thank you readers!	316
Chapter 13: Using the Gradle Build Tool	317
Installing Gradle	317
The Gradle build script for your project	318
Understanding the Gradle script	319
Chapter 14: Pop Quiz Answers	321
Chapter 2, Spring MVC Architecture – Architecting Your Web Store	321
Chapter 3, Control Your Store with Controllers	321
Chapter 4, Working with Spring Tag Libraries	323
Chapter 5, Working with View Resolver	323
Chapter 6, Internalize Your Store with Interceptor	323
Chapter 7, Incorporating Spring Security	324
Chapter 10, Float Your Application with Web Flow	325
Chapter 11, Template with Tiles	325
Index	326

Preface

This book has a very clear aim; to introduce you to the incredible simplicity and power of Spring MVC. I still remember first learning about the Spring framework back in 2009. The best way to test whether or not you really understand a concept is to try to teach it to someone else. In my case, I have taught Spring MVC to MVC; are you confused? I mean that back in 2009, I taught it to my wife Manju Viswambaran Chandrika (MVC). During that course, I was able to understand the kind of doubts that arise in a beginner's mind. I have gathered all my teaching knowledge and put it in this book in an elegant way so that it can be understood without confusion.

It has been more than two years since the first edition of this book presented readers with a beginner-friendly way of explaining the concepts of Spring MVC. The popular reception of the book and the rapid development of the Spring MVC framework naturally demands a new edition.

In the two years since the first edition of this book was published I have received all kinds of suggestions from readers how it could be improved. With the aid of all this information I have completely revised the book. The most obvious changes in this second edition are usage of the latest and greatest versions of Spring and other libraries.

The examples in this book are completely rewritten using Spring 4.3.0.RELEASE version with Java-based configuration. Also in this edition we incorporated the popular in-memory database (HSQL DB) as our backend data-store for the example project. Though this edition includes many changes, my main audience remains the beginners.

I hope you will find this second edition more useful for learning Spring MVC thoroughly from a beginner's perspective.

What this book covers

Chapter 1, *Configuring a Spring Development Environment*, will give you a quick overview of Spring MVC and guide you with detailed notes of step-by-step instructions to set up your development environment. After installing the required prerequisites, you will try out a quick example of how to develop an application with Spring MVC. Although the chapter doesn't explain all the code in detail, you'll pick up a few things intuitively.

Chapter 2, *Spring MVC Architecture – Architecting Your Web Store*, lays down the ground work for the sample application that we are going to build along the way, chapter by chapter. This chapter will introduce you to concepts such as Request mapping, the web application context, Spring MVC request flow, and the layered architecture of a typical web application. You will also learn about how to set up the in-memory database for our sample application.

Chapter 3, *Control Your Store with Controllers*, will take you into the concept of controller; you will learn in detail about defining a controller, how to use URI Template Patterns, Matrix variables and Request parameters.

Chapter 4, *Working with Spring Tag Libraries*, will show you how to use Spring and Spring-form tag libraries in web form handling. You will learn how to bind the domain objects with the views. You will also learn how to use message bundles to externalize label caption texts. At the end of this chapter you will see how to add a login form.

Chapter 5, *Working with View Resolver*, teaches you the inner mechanics of how `InternalResourceViewResolver` resolves a view and take you through how to use various view types such as redirect view and static view. You will also learn about Multipart resolver and content negotiation view resolver. Finally, you will learn how to use Exception handler resolvers.

Chapter 6, *Internalize Your Store with Interceptor*, presents the concept of the interceptor to you; you will learn how to leverage the interceptor to handle or transform requests and responses flexibly. This chapter will teach you how to make your webpage to support internalization with the help of `LocaleChangeInterceptor`. This chapter also introduces how to do audit logging in a log file using interceptor concept.

Chapter 7, *Incorporating Spring Security*, gives you an overview of how to incorporate Spring Security framework with Spring MVC. You will learn how to do simple authentication and authorization on a Spring MVC-based web application.

Chapter 8, *Validate Your Products with a Validator*, gives you an overview of validation concept. You will learn about bean validation, and you will learn how to perform custom validation along with the standard bean validation that bean validation. You will also learn about classic Spring validation and how to combine it with bean validation.

Chapter 9, *Give REST to Your Application with Ajax*, teaches you the basic principles of REST and Ajax, And you will learn how to develop application in RESTful services. The basic concept of HTTP verbs and how it is related to standard CRUD operations will be explained, and you will learn how to do fire Ajax requests and how to handle them from a web page.

Chapter 10, *Float Your Application with Web Flow*, will show you how to use Spring web flow to develop work flow-based web pages. You will learn more about states and transitions in web flow and how to define a flow definition.

Chapter 11, *Template with Tiles*, teaches you how to decompose a page using Apache tiles; you will learn more about `TileViewResolver` and how to define reusable Apache tile templates.

Chapter 12, *Testing Your Application*, introduces how to leverage the Spring testing capability to test your controllers. You will learn how to load the test context and how to mock the service and repository layers. This chapter also introduces you to the Spring MVC test module and how to use it.

Appendix A, *Using the Gradle Build Tool*, introduces you to using the Gradle build tool for our sample application. You will learn about the Gradle script that is required to build our project using Gradle build tool.

Appendix B, *Pop Quiz Answers*, will provide you with the answers to the *Pop quiz* sections in the book.

What you need for this book

To run the examples in the book the following softwares will be required:

1. Java SE Development Kit
2. Maven
3. Apache Tomcat
4. Spring Tool Suite

Who this book is for

The book is for Java developers who want to exploit Spring MVC and its features to build web applications. Some familiarity with basic servlet programming concepts would be a plus, but is not a prerequisite.

Sections

In this book, you will find several headings that appear frequently (Time for action, What just happened?, Pop quiz, and Have a go hero).

To give clear instructions on how to complete a procedure or task, we use these sections as follows:

Time for action

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation to ensure they make sense, so they are followed by these sections.

What just happened?

This section explains the working of the tasks or instructions that you have just completed. You will also find some other learning aids in the book.

Pop quiz

These are short multiple-choice questions intended to help you test your own understanding.

Have a go hero

These are practical challenges that give you ideas to experiment with what you have learned.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Enter the installed JDK directory path as the variable value; in our case, this would be C:\Program Files\Java\jdk1.8.0_91."

A block of code is set as follows:

```
<%@ taglib prefix="c"
    uri="http://java.sun.com/jsp/jstl/core"%>
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible"
          content="IE=edge">
    <meta name="viewport" content="width=device-width,
          initial-scale=1">
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.2.2.RELEASE</version>
</dependency>
```

Any command-line input or output is written as follows:

```
Java (TM) SE Runtime Environment (build 1.8.0_91-b15)
Java HotSpot (TM) 64-Bit Server VM (build 25.91-b15, mixed mode)
```

New **terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the **Java Platform (JDK) 8u91/8u92** download link"

Warnings or important notes appear in a box like this.



Tips and tricks appear like this.



Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Spring-MVC-Beginners-Guide-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/SpringMVCBeginnersGuideSecondEdition_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Configuring a Spring Development Environment

In this chapter, we are going take a look at how we can create a basic Spring MVC application. In order to develop a Spring MVC application, we need some prerequisite software and tools. First, we are going to see how to install all the prerequisites that are required to set up our development environment so that we can start developing the application.

The setup and installation steps given here are for Windows 10 operating systems, but don't worry, as the steps may change only slightly for other operating systems. You can always refer to the respective tools and software vendor's websites to install them in other operating system. In this chapter, we will learn to set up Java and configure the Maven build tool, install the Tomcat web server, install and configure the Spring Tool Suite, and create and run our first Spring MVC project.

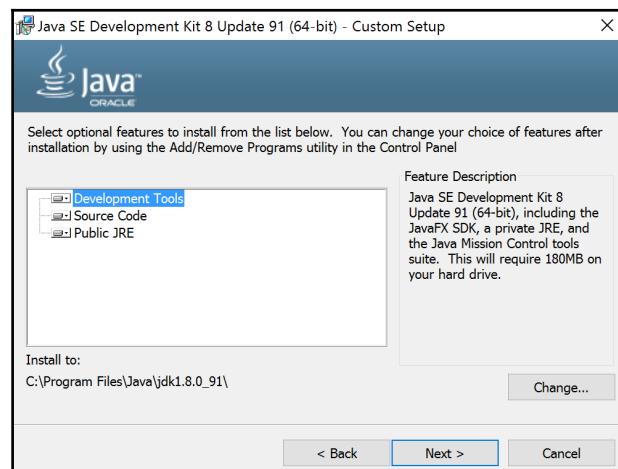
Setting up Java

Obviously, the first thing that we need to do is to install Java. The more technical name for Java is **Java Development Kit (JDK)**. JDK includes a Java compiler (javac), a Java virtual machine, and a variety of other tools to compile and run Java programs.

Time for action – installing JDK

We are going to use Java 8, which is the latest and greatest version of Java, but Java 6 or any higher version is also sufficient to complete this chapter, but I strongly recommend you use Java 8 since in later chapters of this book we may use some of the Java 8 features such as, streams and lambda expressions. Let's take a look at how we can install JDK on a Windows operating system:

1. Go to the Java SE download page on the Oracle website at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.
2. Click on the **Java Platform (JDK) 8u91/8u92** download link; this will take you to the license agreement page. Accept the license agreement by selecting the radio button option.
3. Now, click on the listed download link that corresponds to your Windows operating system architecture; for instance, if your operating system is of type 32 bit, click on the download link that corresponds to **Windows x86**. If your operating system is of type 64 bit, click on the download link that corresponds to **Windows x64**.
4. Now it will start downloading the installer. Once the download is finished, go to the downloaded directory and double-click on the installer. This will open up a wizard window. Just click through the next buttons in the wizard, leaving the default options alone, and click on the **Close** button at the end of the wizard:



JDK installation wizard



Additionally, a separate wizard also prompts you to install **Java Runtime Environment (JRE)**. Go through that wizard as well to install JRE in your system.

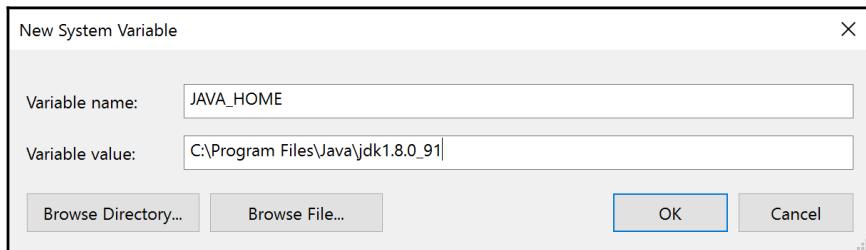
5. Now you can see the installed JDK directory in the default location; in our case, the default location is C:\Program Files\Java\jdk1.8.0_60.

Time for action – setting up environment variables

After installing JDK, we still need to perform some more configurations to use Java conveniently from any directory on our computer. By setting up the environment variables for Java in the Windows operating system, we can make the Java compiler and tools accessible from anywhere in the file system:

1. Navigate to **Start Menu | Settings | System | About | System info | Advanced system settings**.
2. A **System Properties** window will appear; in this window, select the **Advanced** tab and click on the **Environment Variables** button to open the **Environment Variables** window.
3. Now, click on the **New** button in the **System variables** panel and enter **JAVA_HOME** as the variable name and enter the installed JDK directory path as the variable value; in our case, this would be C:\Program Files\Java\jdk1.8.0_91. If you do not have proper rights for the operating system, you will not be able to edit **System variables**; in that case, you can create the **JAVA_HOME** variable under the **User variables** panel.
4. Now, in the same **System variables** panel, double-click on the path variable entry; an **Edit System Variable** window will appear.
5. Edit **Variable value of Path** by clicking the new button and enter the following text %JAVA_HOME%\bin as the value.

If you are using a Windows operating system prior to version 10, edit the path variable carefully; you should only append the text at the end of an existing path value. Don't delete or disturb the existing values; make sure you haven't missed the ; (semi-colon) delimited mark as the first letter in the text that you append:



6. Now click on the **OK** button.

Now we have installed JDK in our computer. To verify whether our installation has been carried out correctly, open a new command window, type `java -version`, and press *Enter*; you will see the installed version of Java compiler on the screen:

```
C:\Users\Amuthan>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b15, mixed mode)
```

Configuring a build tool

Building a java software project typically includes some activities as follows:

- Compiling all the source code
- Packaging the compiled code into a JAR or WAR archive file
- Deploying the packaged archives files on a server

Manually performing all these tasks is time-consuming and is prone to errors. Therefore, we take the help of a build tool. A build tool is a tool that automates everything related to building a software project, from compiling to deploying.

Time for action – installing the Maven build tool

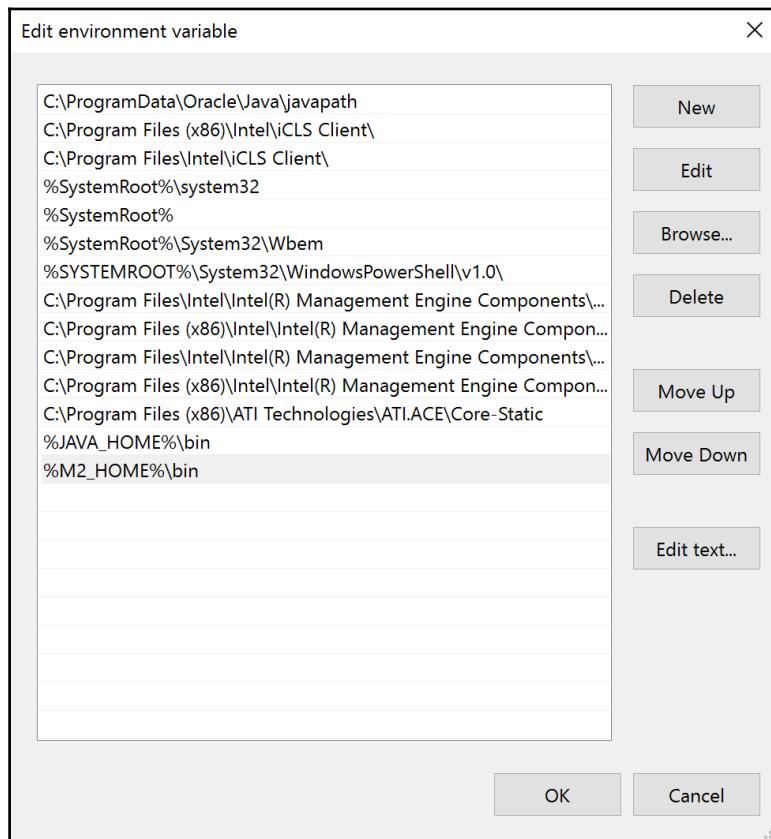
There are other build tools that are available for building Java projects such as Gradle and Ant. We are going to use Maven as our build tool. Let's take a look at how we can install Maven:

1. Go to Maven's download page at <http://maven.apache.org/download.cgi>.
2. Click on the **apache-maven-3.3.9-bin.zip** download link and start the download.



At the time of writing this book, the latest Maven version is 3.3.9; you can literally use any version of Maven after 3.0 to complete this book.

3. Once the download is finished, go to the downloaded directory and extract the **.zip** file into a convenient directory of your choice.
4. Now we need to create one more environment variable called **M2_HOME** in a way that is similar to the way in which we created **JAVA_HOME**. Enter the extracted Maven zip directory's path as the value for the **M2_HOME** environment variable.
5. Finally, append the **M2_HOME** variable to the **PATH** environment variable as well. Double-click on the path variable and click on the **New** button to enter **%M2_HOME%\bin** as the value.



Setting the M2 environment variable

6. Now we have installed the Maven build tool in our computer. To verify whether our installation has been carried out correctly, we need to follow steps that are similar to the Java installation verification. Open a new command window, type `mvn -version` and press *Enter*; you will see the following details of the Maven version:

```
C:\Users\Amuthan>mvn -version
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5;
2015-11-10T10:41:47-06:00)
Maven home: C:\Program Files\apache-maven-3.3.9
Java version: 1.8.0_91, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_91\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
```

Installing a web server

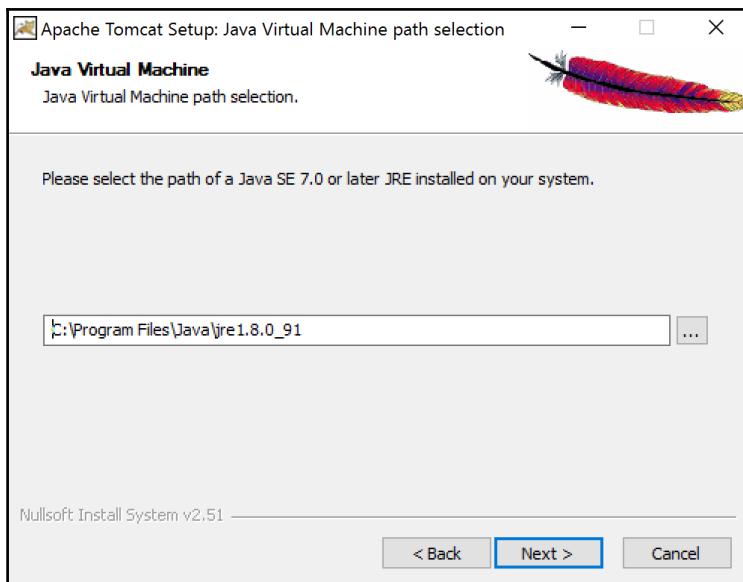
So far, we have seen how to install JDK and Maven. Using these tools, we can compile the Java source code into the .class files and package these .class files into the .jar or .war archives. However, how do we run our packaged archives? For this, we take the help of a web server, which will host our packaged archives as a running application.

Time for action – installing the Tomcat web server

Apache Tomcat is a popular Java web server and servlet container. We are going to use Apache Tomcat Version 8.0, which is the latest, but you can even use version 7.0. Let's take a look at how we can install the Tomcat web server:

1. Go to the Apache Tomcat home page at <http://tomcat.apache.org/>.
2. Click on the **Tomcat 8**. download link; it will take you to the download page.
3. Click on the **32-bit/64-bit Windows Service Installer (pgp, md5, sha1)** link; it will start downloading the installer.
4. Once the download is finished, go to the downloaded directory and double-click on the installer; this will open up a wizard window.
5. Just click through the **Next** buttons in the wizard, leaving the default options alone, and click on the **Finish** button at the end of the wizard. Note that before clicking on the **Finish** button, just ensure that you have unchecked **Run Apache Tomcat** checkbox.

Installing Apache Tomcat with the default option works successfully only if you have installed Java in the default location. Otherwise, you have to correctly provide the JRE path according to the location of your Java installation during the installation of Tomcat, as shown in the following screenshot:



The Java runtime selection for the Tomcat installation

Configuring a development environment

We installed Java and Maven to compile and package our Java source code and installed Tomcat to deploy and run our application. So now we have to start write Spring MVC code so that we can compile, package, and run the code. We can use any simple text editor on our computer to write our code, but that won't help us much with features like finding syntax errors as we type, auto-suggesting important key words, syntax highlighting, easy navigation, and so on.

An **integrated development environment (IDE)** can help us with these features to develop the code faster and error free. We are going to use **Spring Tool Suite (STS)** as our IDE.

Time for action – installing Spring Tool Suite

STS is the best Eclipse-powered development environment to build Spring applications. Let's take a look at how we can install STS:

1. Go to the STS download page at <http://spring.io/tools/sts/all>.
2. Click on the STS zip link to download the zip file that corresponds to your Windows operating system architecture type (32 bit or 62 bit); this will start the download of the zip file. The STS stable release version at the time of this writing is the STS 3.7.3.RELEASE based on Eclipse 4.6.
3. Once the download is finished, go to the downloaded directory and extract the .zip file into a convenient directory of your choice.
4. Open the extracted sts-bundle directory, you will find a directory called sts-3.7.3.RELEASE. Just open that directory and create a desktop shortcut for the STS.exe

We have almost installed all the tools and software required to develop a Spring MVC application; so now, we can create our Spring MVC project on STS. However, before jumping into creating a project, we need to perform the following two final configurations on our STS in order to use STS effectively:

1. Configuring Maven on STS
2. Configuring Tomcat on STS

The aforementioned settings are just a one-time configuration that you need to set up on your newly installed STS; you need not perform this configuration every time you open STS

Time for action – configuring Maven on STS

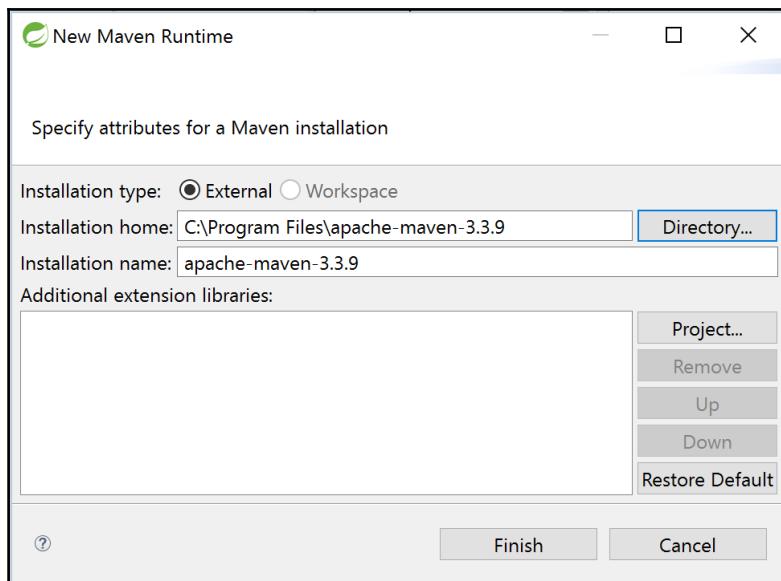
To build our projects, STS uses Maven build tool internally. But we have to tell STS where Maven has been installed so that it can use the Maven installation to build our projects. Let's take a look at how we can configure Maven on STS:

1. Open STS if it is not already open.



When you open STS for the very first time after installing, it will ask you to provide a workspace location. This is because when you create a project on STS, all your project files will be created under this location only. Provide a workspace directory path as you wish and click on the **OK** button.

2. Navigate to **Window | Preferences | Maven | Installations**.
3. On the right-hand side, you can see the **Add** button to locate Maven's installation.
4. Click on the **Add** button and choose our Maven's installed directory, and then click on the **Finish** button, as shown in the following screenshot:



Selecting Maven's location during the Maven configuration on STS

5. Now don't forget to select the newly added Maven installation as your default Maven installation by selecting the checkbox;
6. Click on the **OK** button in the **Preferences** window and close it.

Time for action – configuring Tomcat on STS

As mentioned previously, we can use the Tomcat web server to deploy our application, but we have to inform STS about the location of the Tomcat container so that we can easily deploy our project in to Tomcat. Let's configure Tomcat on STS:

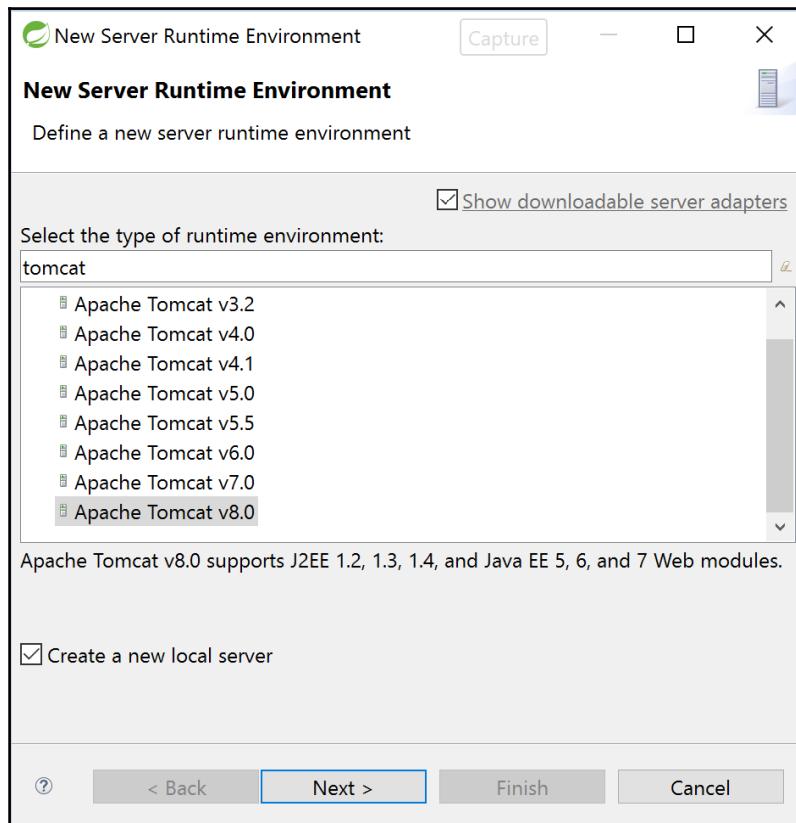
1. Open STS from the desktop icon, if it is not already open.
2. Go to the menu bar and navigate to **Window | Preferences | Server | Runtime Environments**.

3. You can see the available servers listed on the right panel. Now click on the **Add** button to add our Tomcat web server.



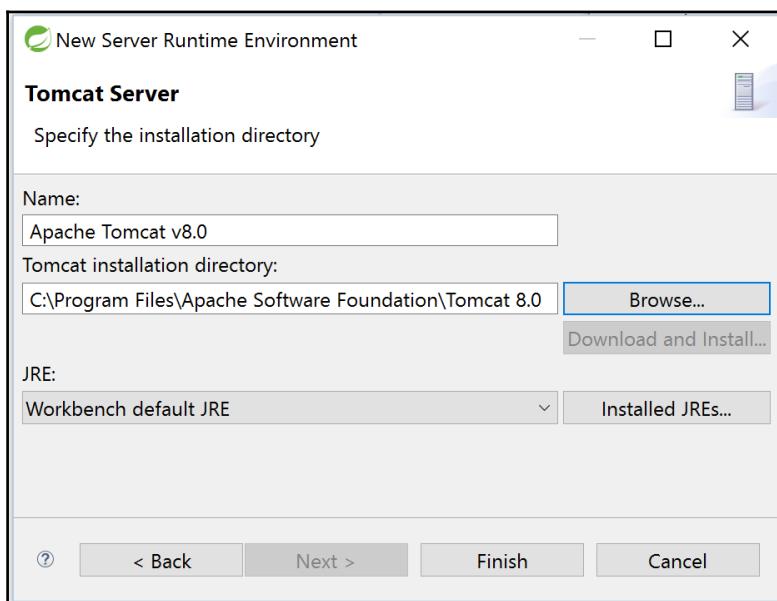
You may also see **Pivotal tc Server Developer Edition (Runtime) v3.1** listed under the available servers, which comes along with the STS installation. Although STS might come with an internal Pivotal tc Server, we chose to use the Tomcat web server as our server runtime environment because of its popularity.

4. A wizard window will appear; type `tomcat` in the **Select the type of runtime environment:** text box, and a list of available Tomcat versions will be shown. Just select **Tomcat v8.0** and select the **Create a new local server** checkbox. Finally, click on the **Next** button, as shown in the following screenshot:



Selecting the server type during the Tomcat configuration on STS

5. In the next window, click on the **Browse** button and locate Tomcat's installed directory, and then click on the **OK** button. You can find Tomcat's installed directory under C:\Program Files\Apache Software Foundation\Tomcat 8.0 if you have installed Tomcat in the default location. Then, click on the **Finish** button, as shown in the following screenshot:



Selecting the Tomcat location during the Tomcat configuration on STS

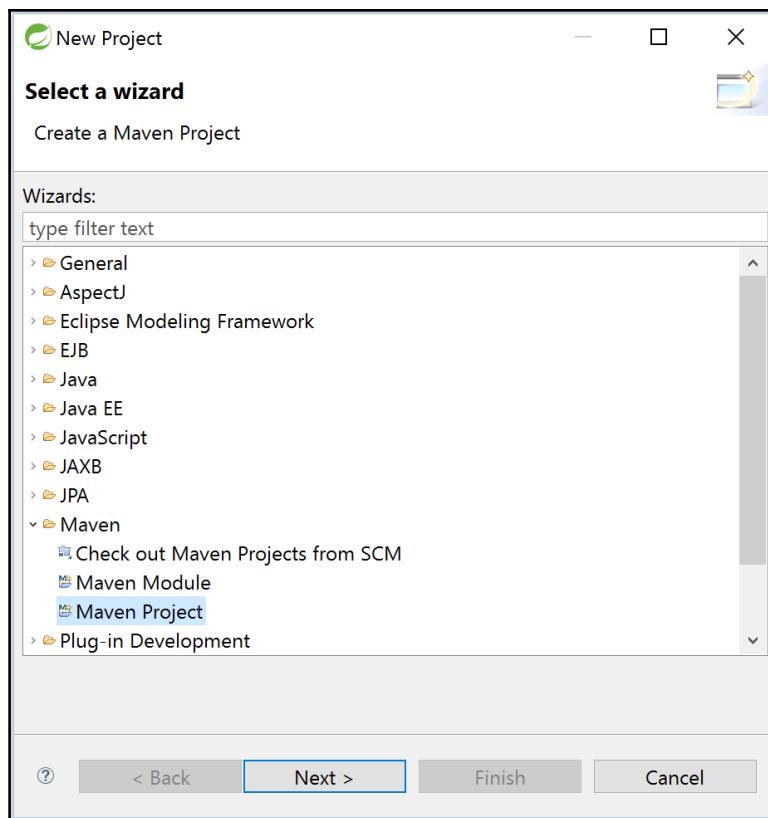
Creating our first Spring MVC project

So far, we have seen how we can install all the prerequisite tools and software. Now we are going to develop our first Spring MVC application using STS. STS provides an easy-to-use project template. Using these templates, we can quickly create our project directory structures without much problem.

Time for action – creating a Spring MVC project in STS

Let's create our first Spring MVC project in STS:

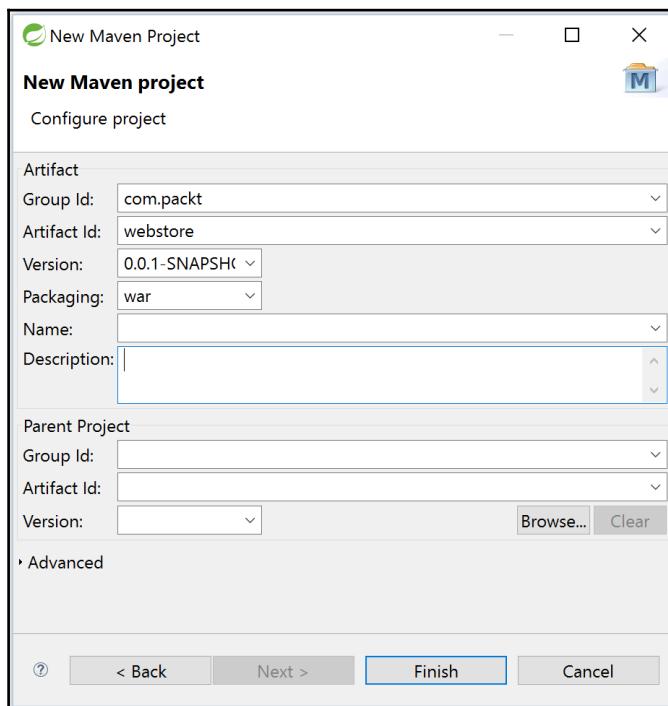
1. In STS, navigate to **File | New | Project**; a **New Project** wizard window will appear.
2. Select **Maven Project** from the list and click on the **Next** button, as shown in the following screenshot:



Maven project's template selection

3. Now, a **New Maven Project** dialog window will appear; just select the checkbox that has the **Create a simple project (skip archetype selection)** caption and click on the **Next** button.

4. The wizard will ask you to specify artifact-related information for your project; just enter **Group Id** as com.packt and **Artifact Id** as webstore. Then, select **Packaging** as war and click on the **Finish** button, as shown in the following screenshot:



Specifying artifact-related information during the project creation

Time for action – adding Java version properties in pom.xml

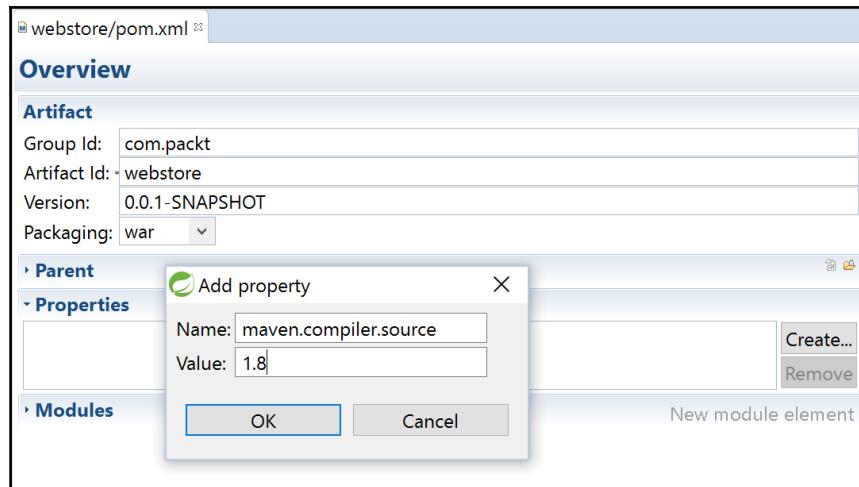
We have successfully created a basic project, but we need to perform one small configuration in our `pom.xml` file, that is, telling Maven to use Java Version 8 while compiling and building our project. How do we tell Maven to do this? Simply add two property entries in `pom.xml`. Let's do the following:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file. Select the **Overview** tab.



If you do not see these tabs, then right-click on `pom.xml`, select the **Open With...** option from the context menu, and choose **Maven POM editor**.

3. Expand the **Properties** accordingly and click on the **Create** button.
4. Now, an **Add property** window will appear; enter **Name** as `maven.compiler.source` and **Value** as `1.8`, as shown in the following screenshot:



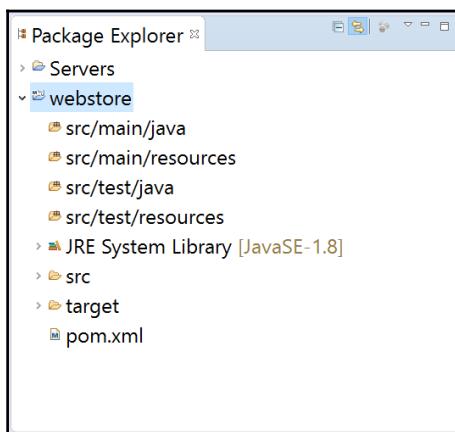
Adding the Java compiler version properties to POM

5. Similarly, create one more property with **Name** as `maven.compiler.target` and **Value** as `1.8`.
6. Finally, save `pom.xml`.

What just happened?

We just created the basic project structure. Any Java project follows a certain directory structure to organize its source code and resources. Instead of manually creating the whole directory hierarchy by ourselves, we just handed over that job to STS. By collecting some basic information about our project, such as **Group Id**, **Artifact Id**, and the **Packaging** style, from us, it is clear that STS is smart enough to create the whole project directory structure with the help of the Maven. Actually, what is happening behind the screen is that STS is internally using Maven to create the project structure.

We want our project to be deployable in any servlet container-based web server, such as Tomcat or Jetty, and that's why we selected the **Packaging** style as `war`. Finally, you will see the project structure in **Package Explorer**, as shown in the following screenshot:



The project structure of the application

If you encounter a maven error on your pom file saying **web.xml is missing and <failOnMissingWebXml> is set to true**, then it means it is expecting a `web.xml` file in your Maven project because it is a web application, as we have chosen packaging as `war`. However, nowadays in web applications `web.xml` file is optional. Add the following configuration in your `pom.xml` within `<project>` tag to fix the error:



```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
```



```
<version>2.6</version>
<configuration>
    <failOnMissingWebXml>false</failOnMissingWebXml>
</configuration>
</plugin>
</plugins>
</build>
```

Spring MVC dependencies

As we are going to use Spring MVC APIs heavily in our project, we need to add the Spring jars in our project to make use of it in our development. As mentioned previously, Maven will take care of managing dependency jars and packaging the project.

Time for action – adding Spring jars to the project

Let's take a look at how we can add the Spring-related jars via the Maven configuration:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab.



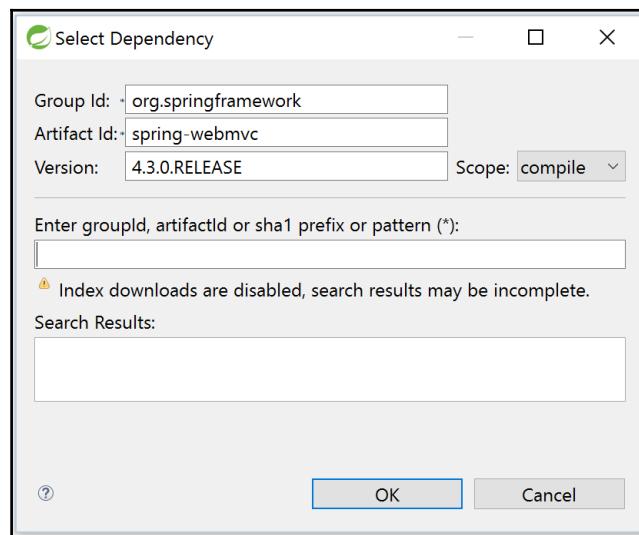
If you do not see these tabs, then right-click on `pom.xml`, select the **Open With...** option from the context menu, and choose **Maven POM editor**.

3. Click on the **Add** button in the **Dependencies** section.



Don't get confused with the **Add** button of the **Dependencies Management** section. You should choose the **Add** button from the left-hand side of the panel.

4. A **Select Dependency** window will appear; enter **Group Id** as `org.springframework`, **Artifact Id** as `spring-webmvc`, and **Version** as `4.3.0.RELEASE`. Select **Scope** as `compile` and then click on the **OK** button, as shown in the following screenshot:



Adding the spring-webmvc dependency

5. Similarly, add the dependency for **JavaServer Pages Standard Tag Library (JSTL)** by clicking on the same **Add** button; this time, enter **Group Id** as `javax.servlet`, **Artifact Id** as `jstl`, **Version** as `1.2`, and select **Scope** as `compile`.
6. Finally, add one more dependency for **servlet-api**; repeat the same step with **Group Id** as `javax.servlet`, **Artifact Id** as `javax.servlet-api`, and **Version** as `3.1.0`, but this time, select **Scope** as `provided` and then click on the **OK** button.
7. As a last step, don't forget to save the `pom.xml` file.

What just happened?

In the Maven world, `pom.xml` (Project Object Model) is the configuration file that defines the required dependencies. While building our project, Maven will read that file and try to download the specified jars from the Maven central binary repository. You need Internet access in order to download jars from Maven's central repository. Maven uses an addressing system to locate a jar in the central repository, which consists of **Group Id**, **Artifact Id**, and **Version**.

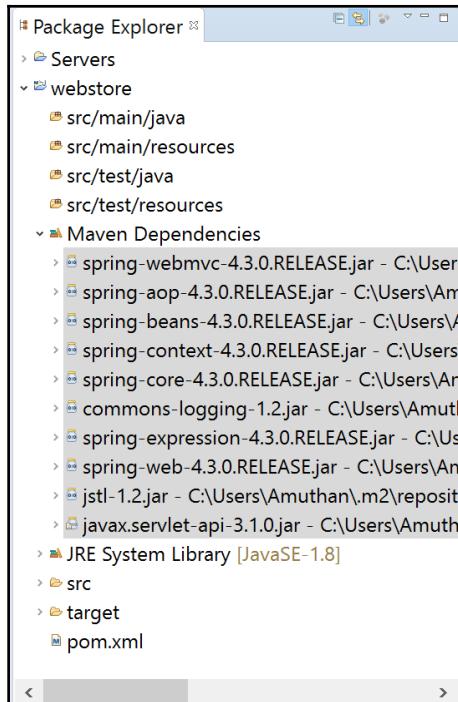
Every time we add a dependency, an entry will be made within the `<dependencies>` `</dependencies>` tags in the `pom.xml` file. For example, if you go to the **pom.xml** tab after finishing step 3, you will see an entry for `spring-webmvc` as follows with in `<dependencies>` `</dependencies>` tag:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.2.2.RELEASE</version>
</dependency>
```

We added the dependency for `spring-mvc` in step 3, and in step 4, we added the dependency for JSTL. JSTL is a collection of useful **JSP (JavaServer Pages)** tags that can be used to write JSP pages easily. Finally, we need a `servlet-api.jar` in order to use servlet-related code; this is what we added in step 5.

However, there is a little difference in the scope of the `servlet-api` dependency compared to the other two dependencies. We only need `servlet-api` while compiling our project. While packaging our project as `war`, we don't want to ship the `servlet-api.jar` as part of our project. This is because the Tomcat web server would provide the `servlet-api.jar` while deploying our project. This is why we selected the **Scope** as **provided** for `servlet-api`.

After finishing step 6, you will see all the dependent jars configured in your project, as shown in the following screenshot, under the **Maven Dependencies** library:



Showing Maven dependencies after configuring POM

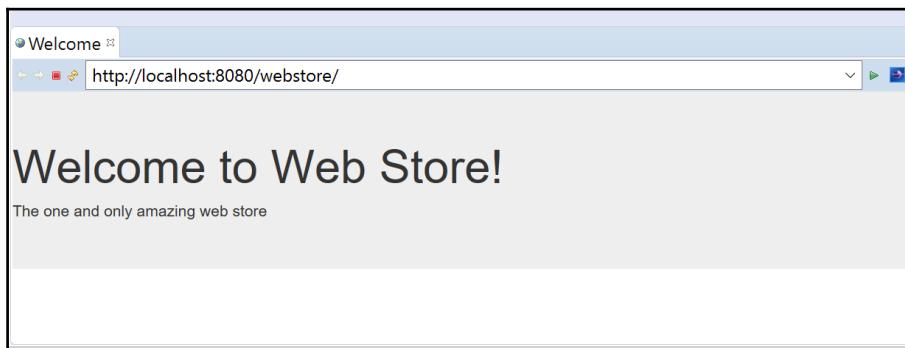
We added only three jars as our dependencies, but if you look in our Maven dependency library list, you will see more than three jar entries. Can you guess why? What if our dependent jars have a dependency on other jars and so on?

For example, our `spring-mvc.jar` is dependent on the `spring-core`, `spring-context`, and `spring-aop` jars, but we have not specified those jars in our `pom.xml` file; this is called **transitive dependencies** in the Maven world. In other words, we can say that our project is transitively dependent on these jars. Maven will automatically download all these transitive dependent jars; this is the beauty of Maven. It will take care of all the dependency management automatically; we need to inform Maven only about the first-level dependencies.

A jump-start to MVC

We have created our project and added all the required jars, so we are ready to code. We are going to incrementally build an online web store throughout this book, chapter by chapter. As a first step, let's create a home page in our project to welcome our customers.

Our aim is simple; when we enter the URL `http://localhost:8080/webstore/` on the browser, we would like to show a welcome page that is similar to the following screenshot:



Showing the welcome page of the web store

We are going to take a deep look at each concept in detail in the upcoming chapters. As of now, our aim is to have quick hands-on experience of developing a simple web page using Spring MVC. So don't worry if you are not able to understand some of the code here.

Time for action – adding a welcome page

To create a welcome page, we need to execute the following steps:

1. Create a `webapp/WEB-INF/jsp/` folder structure under the `src/main/` folder; create a JSP file called `welcome.jsp` under the `src/main/webapp/WEB-INF/jsp/` folder, and add the following code snippets into it and save it:

```
<%@ taglib prefix="c"  
uri="http://java.sun.com/jsp/jstl/core"%>  
  
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <meta charset="utf-8">
```

```
<meta http-equiv="X-UA-Compatible"
      content="IE=edge">
<meta name="viewport" content="width=device-width,
      initial-scale=1">
<title>Welcome</title>
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/
      bootstrap/3.3.5/css/bootstrap.min.css">
</head>
<body>
    <div class="jumbotron">
        <h1> ${greeting} </h1>
        <p> ${tagline} </p>
    </div>
</body>
</html>
```

2. Create a class called `HomeController` under the `com.packt.webstore.controller` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import
org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String welcome(Model model) {
        model.addAttribute("greeting", "Welcome to Web Store!");
        model.addAttribute("tagline", "The one and only amazing
        web store");

        return "welcome";
    }
}
```

What just happened?

In step 1, we just created a JSP file; the important thing we need to notice here is the `<h1>` tag and the `<p>` tag. Both the tags have some expression that is surrounded by curly braces and prefixed by the `$` symbol:

```
<h1> ${greeting} </h1>
<p> ${tagline} </p>
```

So, what is the meaning of \${greeting}? It means that greeting is a kind of variable; during rendering of this JSP page, the value stored in the greeting variable will be shown in the header 1 style, and similarly, the value stored in the tagline variable will be shown as a paragraph.

So now, the next question is where we will assign values to those variables arises. This is where the controller will be of help; within the welcome method of the HomeController class, take a look at the following lines of code:

```
model.addAttribute("greeting", "Welcome to Web Store!");
model.addAttribute("tagline", "The one and only amazing web store");
```

You can observe that the two variable names, greeting and tagline, are passed as a first parameter of the addAttribute method and the corresponding second parameter is the value for each variable. So what we are doing here is simply putting two strings, "Welcome to Web Store!" and "The one and only amazing web store", into the model with their corresponding keys as greeting and tagline. As of now, simply consider the fact that model is a kind of map data structure.



Folks with knowledge of servlet programming can consider the fact that model.addAttribute works exactly like request.setAttribute.

So, whatever value we put into the model can be retrieved from the view (JSP) using the corresponding key with the help of the \${ } placeholder expression. In our case, greeting and tagline are keys.

The Dispatcher servlet

We put values into the model, and we created the view that can read those values from the model. So, the Controller acts as an intermediate between the View and the Model; with this, we have finished all the coding part required to present the welcome page. So will we be able to run our project now? No. At this stage, if we run our project and enter the URL <http://localhost:8080/webstore/> on the browser, we will get an **HTTP Status 404** error. This is because we have not performed any servlet mapping yet.



So what is servlet mapping? Servlet mapping is a configuration of mapping a servlet to a URL or URL pattern. For example, if we map a pattern like `/status/*` to a servlet, all the HTTP request URLs starting with a text status such as `http://example.com/status/synopsis` or `http://example.com/status/complete?date=today` will be mapped to that particular servlet. In other words, all the HTTP requests that carry the URL pattern `/status/*` will be handed over to the corresponding mapped servlet class.

In a Spring MVC project, we must configure a servlet mapping to direct all the HTTP requests to a single front servlet. The front servlet mapping is a design pattern where all requests for a particular web application are directed to the same servlet. This pattern is sometimes called as **Front Controller Pattern**. By adapting the Front Controller design, we make front servlet have total control over the incoming HTTP request so that it can dispatch the HTTP request to the desired controller.

One such front servlet given by Spring MVC framework is the Dispatcher servlet (`org.springframework.web.servlet.DispatcherServlet`). We have not configured a Dispatcher servlet for our project yet; this is why we get the **HTTP Status 404** error if we run our project now.

Time for action – configuring the Dispatcher servlet

The Dispatcher servlet is what examines the incoming HTTP request and invokes the right corresponding controller method. In our case, the `welcome` method from the `HomeController` class needs to be invoked if we make an HTTP request by entering the `http://localhost:8080/webstore/` URL on the browser. So let's configure the Dispatcher servlet for our project:

1. Create a class called `WebApplicationContextConfig` under the `com.packt.webstore.config` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.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
.DefaultServletHandlerConfigurer;
import org.springframework.web.servlet.config.annotation
```

```
.EnableWebMvc;
import org.springframework.web.servlet.config.annotation
.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view
.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan("com.packt.webstore")
public class WebApplicationContextConfig extends
WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling
(DefaultServletHandlerConfigurer configurer) {
    configurer.enable();
}

@Bean
public InternalResourceViewResolver
getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new
InternalResourceViewResolver();
    resolver.setViewClass(JstlView.class);
    resolver.setPrefix("/WEB-INF/jsp/");
    resolver.setSuffix(".jsp");

    return resolver;
}
}
```

2. Create a class called `DispatcherServletInitializer` under the `com.packt.webstore.config` package in the source directory `src/main/java` and add the following code into it:

```
package com.packt.webstore.config;

import org.springframework.web.servlet.support
.AbstractAnnotationConfigDispatcherServletInitializer;

public class DispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
}
```

```
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {
            WebApplicationContextConfig.class
        };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

What just happened?

If you know about servlet programming, you might be quite familiar with the servlet configuration and `web.xml`, and in that case, you can consider

the `DispatcherServletInitializer` class as similar to `web.xml`. In step 2, we configured a servlet of type `DispatcherServlet` by extending the

`AbstractAnnotationConfigDispatcherServletInitializer` class, which is more or less similar to any other normal servlet configuration. The only difference is that we have not instantiated the `DispatcherServlet` class for that configuration. Instead, the servlet class (`org.springframework.web.servlet.DispatcherServlet`) is provided by the Spring MVC framework and we initialized it using the `AbstractAnnotationConfigDispatcherServletInitializer` class.

After this step, our configured `DispatcherServlet` will be ready to handle any requests that come to our application on runtime and will dispatch the request to the correct controller's method.

However, `DispatcherServlet` should know how to access the controller instances and view files that are located in our project, and only then can it properly dispatch the request to the correct controllers. So we have to give some hint to `DispatcherServlet` to locate the controller instances and view files.

This is what we configured within the `getServletConfigClasses` method of the `DispatcherServletInitializer` class. By overriding the `getServletConfigClasses` method, we are telling `DispatcherServlet` about our controller classes and view files. And in step 1, through the `WebApplicationContextConfig` class file, we configured the `InternalResourceViewResolver` and other default configurations.

Don't worry if you are not able to understand each and every configuration in the `WebApplicationContextConfig` and `DispatcherServletInitializer` classes; we will take a look deep into these configuration files in next chapter. As of now, just remember that this is a one-time configuration that is needed to run our project successfully.

Deploying our project

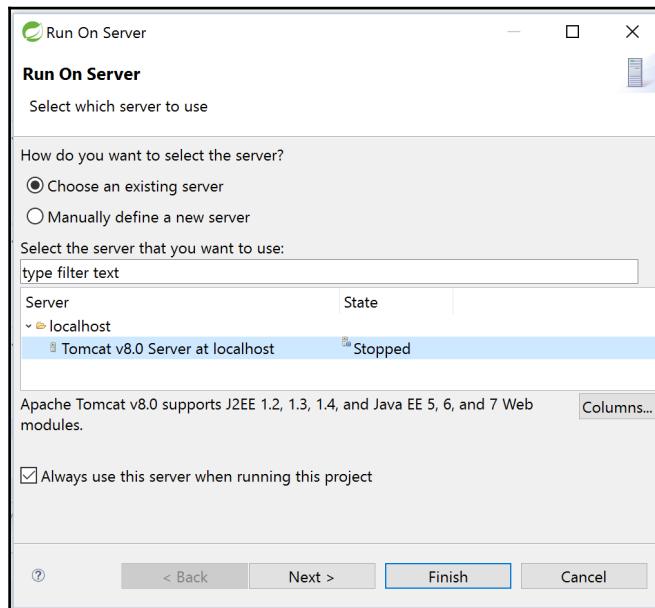
We successfully created the project in the last section, so you might be curious to know what would happen if we run our project now. As our project is a web project, we need a web server to run it.

Time for action – running the project

As we have already configured the Tomcat web server in our STS, let's use Tomcat to deploy and run our project:

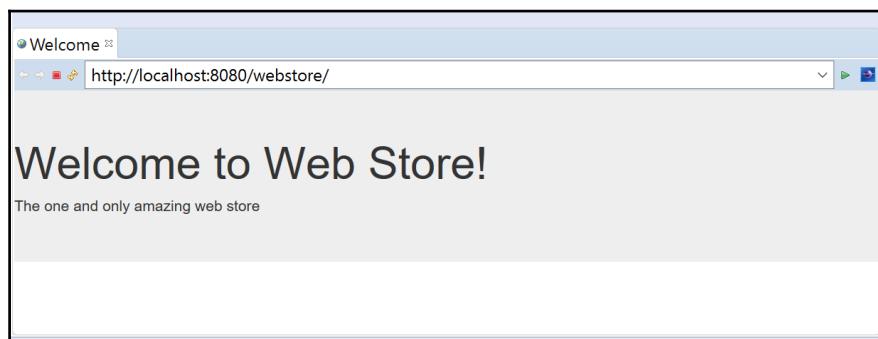
1. Right-click on your project from **Package Explorer** and navigate to **Run As | Run on Server**.
2. A server selection window will appear with all the available servers listed; just select the server that we have configured, **Tomcat v8.0**.

3. At the bottom of the window, you can see a checkbox with the caption that says **Always use this server when running this project**; select this checkbox and enter the **Finish** button, as shown in the following screenshot:



Configuring the default server for a Spring MVC project

4. Now you will see a web page that will show you a welcome message.



Showing the welcome page of the web store

Summary

In this chapter, you saw how to install all the prerequisites that are needed to get started and run your first Spring MVC application, for example, installing JDK, the Maven build tool, the Tomcat servlet container, and STS IDE.

You also learned how to perform various configurations in our STS IDE for Maven and Tomcat, created your first Spring MVC project, and added all Spring-related dependent jars through the Maven configuration.

We had a quick hands-on experience of developing a welcome page for our `webstore` application. During that course, you learned how to put values into a model and how to retrieve these values from the model.

Whatever we have seen so far is just a glimpse of Spring MVC, but there is much more to uncover, for example, how the model, view and controllers are connected to each other and how the request flow occurs. We are going to explore these topics in the next chapter, so see you there.

2

Spring MVC Architecture – Architecting Your Web Store

What we saw in the first chapter was nothing but a glimpse of Spring MVC. Our total focus was just to get a Spring MVC application running. Now it's time for us to deep dive into the Spring MVC architecture.

By the end of this chapter, you will have a clear understanding of:

- The Dispatcher servlet and request mapping
- Web application context and configuration
- Spring MVC request flow and Web MVC
- A typical Spring web application architecture

Dispatcher servlet

In the first chapter, we provided a little introduction to the Dispatcher servlet and you saw how to configure a Dispatcher servlet using the `DispatcherServletInitializer` class. You learned that every web request first comes to the Dispatcher servlet. The Dispatcher servlet is the thing that decides which controller method the web request should be dispatched to. In the previous chapter, we created a welcome page that will be shown whenever we enter the URL `http://localhost:8080/webstore/` in the browser.

Mapping a URL to the appropriate controller method is the primary duty of the Dispatcher servlet.

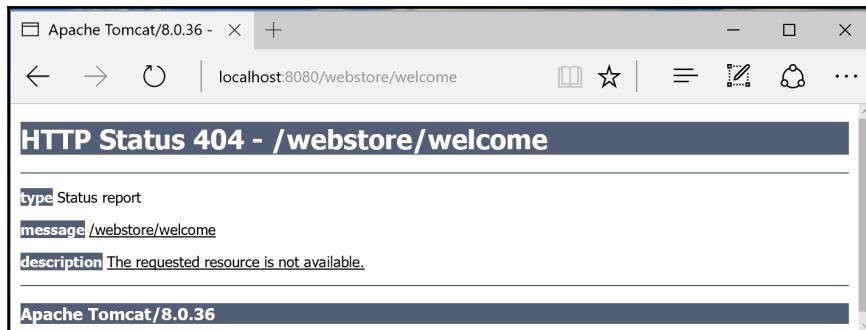
So the Dispatcher servlet reads the web request URL and finds the appropriate controller method that can serve that web request and invokes it. This process of mapping a web request onto a specific controller method is called request mapping. And the Dispatcher servlet is able to do this with the help of the `@RequestMapping` (`org.springframework.web.bind.annotation.RequestMapping`) annotation.

Time for action – examining request mapping

Let's observe what will happen when we change the value attribute of the `@RequestMapping` annotation.

1. Open your STS and run your `webstore` project; just right-click on your project and choose **Run As | Run on Server**. Now you will be able to see the same welcome message in the browser.
2. Now go to the address bar of the browser and enter the following URL
`http://localhost:8080/webstore/welcome`.
3. You will see the **HTTP Status 404** error page in the browser, and you will also see the following warning in the console:

```
WARNING: No mapping found for HTTP request with URI
[/webstore/welcome] in DispatcherServlet with name
'DefaultServlet'
```



Error showing no mapping found message

4. Now open your `HomeController` class and change the `@RequestMapping` annotation's `value` attribute to `/welcome` and save it. Basically, your new request mapping annotation will look like as follows: `@RequestMapping("/welcome")`.

5. Again, run your application and enter the same URL that you entered in step 2. Now you should be able to see the same welcome message again in the browser without any request mapping error.

What just happened?

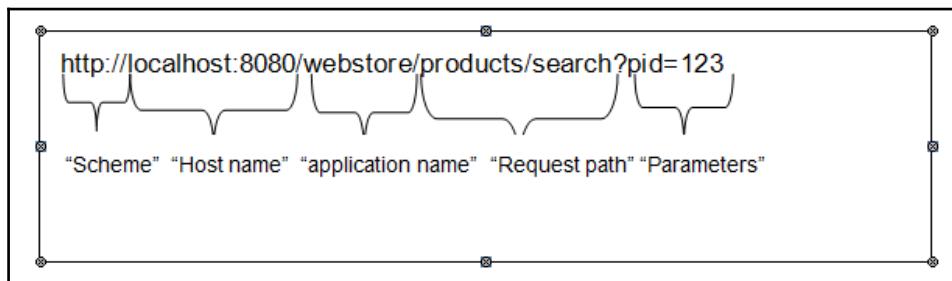
After starting our application, when we enter the URL

`http://localhost:8080/webstore/welcome` in the browser; the Dispatcher servlet (`org.springframework.web.servlet.DispatcherServlet`) immediately tried to find a matching controller's method for the request path `/welcome`.

In a Spring MVC application, a URL can be logically divided into five parts; see the figure that is present after this tip. The `@RequestMapping` annotation only matches against the URL request path; it will omit the scheme, host name, application name, and so on. Here the application name is just a context name where the application is deployed—it is totally under the control of how we configure the web server.



The `@RequestMapping` annotation has one more attribute called `method` to further narrow down the mapping based on HTTP request method types (GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE). If we do not specify the `method` attribute in the `@RequestMapping` annotation, the default `method` would be GET. You will see more about the `method` attribute of the `@RequestMapping` annotation in Chapter 4, *Working with Spring Tag Libraries* under *Time for action – serving and processing forms*.



The logical parts of a typical Spring MVC application URL

But since we don't have a corresponding request mapping for the given URL path `/welcome`, we are getting the **HTTP status 404** error in the browser and the following error log in the console:

```
WARNING: No mapping found for HTTP request with URI [/webstore/welcome] in DispatcherServlet with name 'DefaultServlet'
```

From the error log, we can clearly understand that there is no request mapping for the URL path `/webstore/welcome`. So we are trying to map this URL path to the existing controller's method; that's why in step 4 we only put the request path value `/welcome` in the `@RequestMapping` annotation as the `value` attribute. Now everything works perfectly fine.

Pop quiz – request mapping

Suppose I have a Spring MVC application for library management called *BookPedia* and I want to map a web request URL

`http://localhost:8080/BookPedia/category/fiction` to a controller's method—how would you form the `@RequestMapping` annotation?

1. `@RequestMapping("/fiction")`
2. `@RequestMapping("/category/fiction")`
3. `@RequestMapping("/BookPedia/category/fiction")`

What is the request path in the following URL `http://localhost:8080/webstore/?`

1. `webstore/`
2. `/`
3. `8080/webstore/`

Understanding the Dispatcher servlet configuration

Now we've got a basic idea of how request mapping works. I also mentioned that every web request first comes to the Dispatcher servlet, but the question is how does the Dispatcher servlet know it should handle every incoming request? The answer is we explicitly instructed it to do so through the `getServletMappings` method of the `DispatcherServletInitializer` class. Yes, when we return the string array containing

only the “/” character, it indicates the DispatcherServlet configuration as the default servlet of the application. So every incoming request will be handled by DispatcherServlet.

Time for action – examining the servlet mapping

Let's observe what will happen when we change the return value of the getServletMappings method.

1. Open DispatcherServletInitializer and change the return value of the getServletMappings method as `return new String[] { "/app/*"}`; basically your getServletMappings method should look like the following after your change:

```
@Override  
protected String[] getServletMappings() {  
    return new String[] { "/app/*" };  
}
```

2. Run your application by right-clicking on your project and choose **Run As | Run on Server**.
3. Go to the address bar of the browser and enter the following URL `http://localhost:8080/webstore/welcome`. You will see the **HTTP Status 404** error page in the browser.
4. Again go to the address bar of the browser and enter the following URL `http://localhost:8080/webstore/app/welcome` and you will be able to see the same welcome message in the browser.
5. Now revert the return value of the getServletMappings method to its original value. Basically, your getServletMappings method should look as follows after your change:

```
@Override  
protected String[] getServletMappings() {  
    return new String[] { "/" };  
}
```

What just happened?

As I have already mentioned, we can consider the `DispatcherServletInitializer` class as equivalent to `web.xml` as it extends from `the AbstractAnnotationConfigDispatcherServletInitializer`. If you look close enough, we are overriding three important methods in `DispatcherServletInitializer`, namely:

- `getRootConfigClasses`: This specifies the configuration classes for the root application context
- `getServletConfigClasses`: This specifies the configuration classes for the Dispatcher servlet application context
- `getServletMappings`: This specifies the servlet mappings for `DispatcherServlet`



Typically, the context loaded using the `getRootConfigClasses` method is the *root* context, which belongs to the whole application, while the one initialized using the `getServletConfigClasses` method is actually specific to that Dispatcher servlet. Technically, you can have multiple Dispatcher servlets in an application and so multiple such contexts, each specific for the respective Dispatcher servlet but with the same root context.

In step 1, when we changed the return value of the `getServletMappings` method, we instructed `DispatcherServlet` to handle only the web requests that start with a prefix text of `/app/`. That's why we entered the URL

`http://localhost:8080/webstore/welcome`. We saw the **HTTP Status 404** error page in the browser, since the URL request path didn't start with the `/app/` prefix. But when we tried the URL `http://localhost:8080/webstore/app/welcome`, we were able to see the same welcome message in the browser as the URL started with the `/app/` prefix.

Servlet mapping versus request mapping

The servlet mapping specifies which web container of the Java servlet should be invoked for a given URL. It maps the URL patterns to servlets. When there is a request from a client, the servlet container decides which servlet it should forward the request to based on the servlet mapping. In our case, we mapped all incoming requests to `DispatcherServlet`.

In contrast, request mapping guides the `DispatcherServlet` which controller method it needs to invoke as a response to the request based on the request path. In our case, we mapped the `/welcome` request path to the `welcome` method of the `HomeController` class.

Pop quiz – servlet mapping

Considering the following servlet mapping, identify the possible matching URLs:

```
@Override  
protected String[] getServletMappings() {  
    return new String[] { "*.do" };  
}
```

1. `http://localhost:8080/webstore/welcome`
2. `http://localhost:8080/webstore/do/welcome`
3. `http://localhost:8080/webstore/welcome.do`
4. `http://localhost:8080/webstore/welcome/do`

Considering the following servlet mapping, identify the possible matching URLs:

```
@Override  
protected String[] getServletMappings() {  
    return new String[] { "/" };  
}
```

1. `http://localhost:8080/webstore/welcome`
2. `http://localhost:8080/webstore/products`
3. `http://localhost:8080/webstore/products/computers`
4. All the above

Web application context

In a Spring-based application, our application objects will live within an object container. This container will create objects and associations between objects and manage their complete lifecycle. These container objects are called Spring managed beans (or simply beans) and the container is called application context in the Spring world.

Spring's container uses **dependency injection (DI)** to manage the beans that make up an application. An application context

(`org.springframework.context.ApplicationContext`) creates beans, associates beans together based on bean configuration, and dispenses beans upon request. A bean configuration can be defined via an XML file, annotation, or even via Java configuration classes. We are going to use annotation and Java configurations in our chapters.

A **web application context** is an extension of the application context, and is designed to work with the standard servlet context (`javax.servlet.ServletContext`). The web application context typically contains front-end related beans such as views and view resolvers, and so on. In the first chapter, we simply created a class called `WebApplicationContextConfig`, which is a bean configuration for our web application.

We learned that `WebApplicationContextConfig` is nothing but a Java-based bean configuration file for our web application context, where we can define the beans to be used in our application. Usually, we define beans using the `@Bean` annotation. In order to run a Spring MVC application successfully, Spring needs at least a bean that implements the `org.springframework.web.servlet.ViewResolver` interface. One such bean we defined in our web application context is `InternalResourceViewResolver`.

View resolvers

A view resolver helps the Dispatcher servlet to identify the views that have to be rendered as a response to a specific web request. Spring MVC provides various view resolver implementations to identify views and `InternalResourceViewResolver` is one such implementation:

```
@Bean
public InternalResourceViewResolver getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new
        InternalResourceViewResolver();
    resolver.setViewClass(JstlView.class);
    resolver.setPrefix("/WEB-INF/jsp/");
    resolver.setSuffix(".jsp");

    return resolver;
}
```

Through the above bean definition is in the web application context configuration (`WebApplicationContextConfig`), we are instructing Spring MVC to create a bean for the class `InternalResourceViewResolver` (`org.springframework.web.servlet.view.InternalResourceViewResolver`). We will see more about the view resolver in Chapter 5, *Working with View Resolver*.

Time for action – understanding web application context

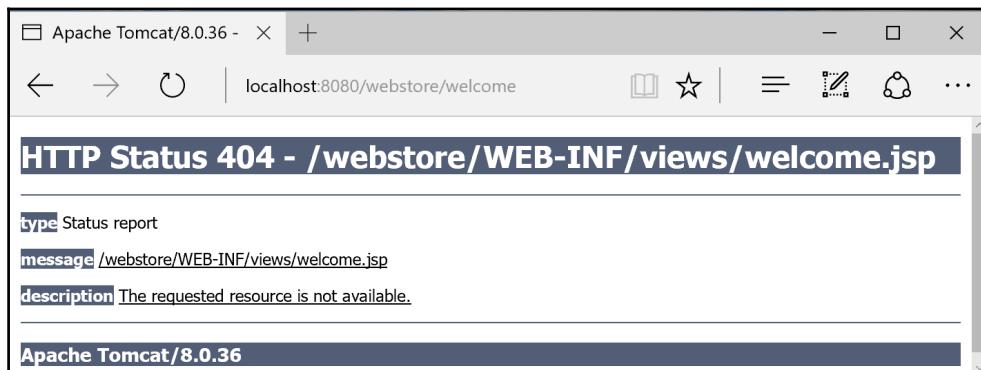
Okay we have seen enough introductions about web application context now, let's tweak the `InternalResourceViewResolver` bean from the web application context configuration (`WebApplicationContextConfig.java`) a little bit and observe the effect.

1. Open `WebApplicationContextConfig.java` and set the prefix as `/WEB-INF/views/` for the `InternalResourceViewResolver` bean. Basically your `InternalResourceViewResolver` bean definition should look as follows after your change:

```
@Bean
public InternalResourceViewResolver
getInternalResourceViewResolver() {
    InternalResourceViewResolver resolver = new
    InternalResourceViewResolver();
    resolver.setViewClass(JstlView.class);
    resolver.setPrefix("/WEB-INF/views/");
    resolver.setSuffix(".jsp");

    return resolver;
}
```

2. Run your application, then go to the address bar of the browser and enter the following URL: `http://localhost:8080/webstore/welcome`. You will see the **HTTP Status 404** error page in the browser:



The logical parts of a typical Spring MVC application URL

3. To fix this error, rename the folder `/webstore/src/main/webapp/WEB-INF/jsp` to `/webstore/src/main/webapp/WEB-INF/views`.
4. Now run your application and enter the URL
`http://localhost:8080/webstore/welcome`; you will see the welcome message again.

What just happened?

After changing the `prefix` property value of the `InternalResourceViewResolver` bean, when we entered the URL `http://localhost:8080/webstore/welcome` in the browser, we got a **HTTP status 404** error. The **HTTP status 404** error means that the server could not find the web page that you asked for. Okay if that is the case, which web page did we ask for?

As a matter of fact, we didn't ask for any web page to the server directly; instead the Dispatcher servlet asked for a particular web page to the server. And what we already learned is that the Dispatcher servlet will invoke a method in any of our controller beans that can serve this web request. In our case, that method is nothing but the `welcome` method of our `HomeController` class, because that is the only request mapping method that can match the request path of given URL

`http://localhost:8080/webstore/welcome` in its `@RequestMapping` annotation.

Now I want you to observe three things:

1. The `prefix` property value of the `InternalResourceViewResolver` bean definition in `WebApplicationContextConfig.java`:

```
/WEB-INF/views/
```

2. The return value of the `welcome` method from the `HomeController` class:

```
welcome
```

3. Finally, the `suffix` property value of the `InternalResourceViewResolver` bean:

```
.jsp
```

If you combine these three values together, you will get a web page request URL as `/WEB-INF/views/welcome.jsp`. Now notice the error message in the figure after step 2, which is showing a **HTTP status 404** error for the same web page URL `/WEB-INF/views/welcome.jsp` under the application name `/webstore/`.

So the conclusion is that `InternalResourceViewResolver` resolves the actual view's file path by prepending the configured `prefix` and appending the `suffix` value with the view name. The view name is the value usually returned by controller's method. So the controller's method doesn't return the path of the actual view file-it just returns the logical view name. It is the job of `InternalResourceViewResolver` to form the URL of the actual view file correctly.

Okay fine, but who is going to use this final formed URL? The answer is the Dispatcher servlet. Yes, after getting the final formed URL of a view file from the view resolver, the Dispatcher servlet will try to get the view file from the server. During that time, if the formed URL is found to be wrong, then you will get a **HTTP status 404** error.

Usually after invoking the controller's method, the Dispatcher servlet will wait to get the logical view name from the controller's method. Once the Dispatcher servlet gets the logical view name, it will give that to the view resolver (`InternalResourceViewResolver`) to get the URL path of the actual view file. Once the view resolver returns the URL path to the Dispatcher servlet, the rendered view file is served to the client browser as a web page by the Dispatcher servlet.

Okay fine, but why did we get the error in step 2? Since we changed the `prefix` property of `InternalResourceViewResolver` in step 2, the URL path value returned from `InternalResourceViewResolver` would become `/WEB-INF/views/welcome.jsp` in step 3, which is an invalid path value (there is no directory called `views` under `WEB-INF`). That's why we renamed the directory `jsp` to `views` in step 3 to align it with the path generated by `InternalResourceViewResolver`, so everything works fine again.

Understanding the web application context configuration

The web application context configuration file (`WebApplicationContextConfig.java`) is nothing but a simple Java-based Spring bean configuration class. Spring will create beans (objects) for every bean definition mentioned in this class during the boot up of our application. If you open this web application context configuration file, you will find the following annotations on top of the class definition:

- `@Configuration`: This indicates that a class declares one or more `@Bean` methods
- `@EnableWebMvc`: Adding this annotation to an `@Configuration` class imports some special Spring MVC configuration
- `@ComponentScan`: This specifies the base packages to scan for annotated components (beans)

The first annotation `@Configuration` indicates that this class declares one or more `@Bean` methods. If you remember, in the last section, I explained how we created a bean definition for `InternalResourceViewResolver`.

The second annotation is `@EnableWebMvc`. With this annotation, we are telling Spring MVC to configure the `DefaultAnnotationHandlerMapping`, `AnnotationMethodHandlerAdapter` and `ExceptionHandlerExceptionResolver` beans. These beans are required for Spring MVC to dispatch requests to the controllers.

Actually, `@EnableWebMvc` does many things behind the screen. It also enables support for various convenient annotations such as `@NumberFormat`, `@DateTimeFormat` to format the form bean's fields during form binding, and similarly the `@Valid` annotation to validate the controller method's parameters. It even supports Java objects being converted to/from XML or JSON via the `@RequestBody` and `@ResponseBody` annotation in the `@RequestMapping` or `@ExceptionHandler` methods during form binding. We will see the usage of these

annotations in later chapters. As for now, just remember that the `@EnableWebMvc` annotation is needed to enable annotations such as `@Controller` and `@RequestMapping` and so on.

Now the third annotation `@ComponentScan`-what is the purpose of this annotation? You need a little bit of background information to understand the purpose of the `@ComponentScan` annotation. The `@Controller` annotation indicates that a particular class serves the role of a controller. You have already learned that the Dispatcher servlet searches such annotated classes for mapped methods (`@RequestMapping` annotated methods) to serve a web request. In order to make the controller available for searching, we must create a bean for that controller in our web application context.

We can create beans for controllers explicitly via the bean configuration (using the `@Bean` annotation; you can see how we created a bean for the `InternalResourceViewResolver` class using the `@Bean` annotation for reference), or we can hand over that task to Spring via an auto-detection mechanism. To enable auto-detection of the `@Controller` annotated classes, we must add component scanning to our configuration using the `@ComponentScan` annotation. Now you understand the purpose of the `@ComponentScan` annotation.

Spring will create beans (objects) for every `@Controller` class at runtime. The Dispatcher servlet will search for the correct request mapping method in every `@Controller` bean based on the `@RequestMapping` annotation to serve a web request. The `base-package` property of a `@ComponentScan` annotation indicates under which package Spring should search for controller classes to create beans:

```
@ComponentScan("com.packt.webstore")
```

This line instructs Spring to search for controller classes in the `com.packt.webstore` package and its sub-packages.



The `@ComponentScan` annotation not only recognizes controller classes, it will also recognize other stereotypes such as services and repositories classes as well. We will explore services and repositories later.

Pop quiz – web application context configuration

In order to identify a class as a controller by Spring, what needs to be done?

1. That particular class should have an `@Controller` annotation.
2. The `@EnableWebMvc` annotation and `@ComponentScan` annotation should be specified in the web application context configuration file.

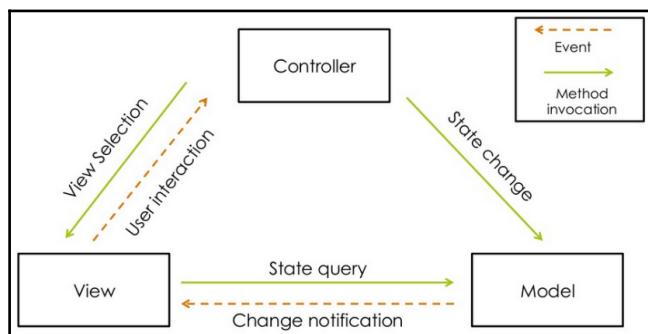
3. That particular class should be put up in a package or in a sub-package that has been specified as a value in the @ComponentScan annotation.
4. All of the above.

Model View Controller

So far, we have looked at lots of concepts such as the Dispatcher servlet, request mapping, controllers, and the view resolver, but it would be good to see the overall picture of the Spring MVC request flow so that we can understand each component's responsibilities. But before that, you need a basic understanding of the **Model View Controller (MVC)** concept. Every enterprise level application's Presentation layer can be logically divided into three major parts:

- The part that manages the data (**Model**)
- The part that creates the user interface and screens (**View**)
- The part that handles interactions between the user, the user interface, and the data (**Controller**)

The following diagram should help you to understand the event flow and command flow within an MVC pattern.



The classic MVC pattern

Whenever a user interacts with the view by clicking on a link or button, or something similar, the view issues an event notification to the controller and the controller issues a command notification to the model to update the data. Similarly, whenever the data in the model is updated or changed, a change notification event is issued to the view by the model, and in response the view issues a state query command to the model to get the latest

data from the model. Here, the model and view can directly interact. This pattern is called the classic MVC. But what Spring MVC employs is something called the Web MVC pattern because of the limitation in the HTTP protocol.

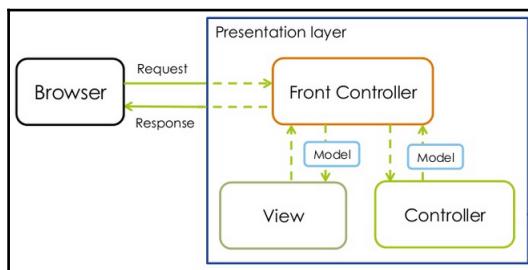


Web applications rely on the HTTP protocol, which is a stateless pull protocol. This means no request implies no reply every time we need to make a request to the application to know the state of the application. The MVC design pattern requires a push protocol for the views to be notified by the model. So the Web MVC controller takes more responsibility for state changing, state querying and change notification.

In Web MVC, every interaction between the model and view are done via controllers only. So the controller acts as a bridge between the model and the view. There is no direct interaction between the model and the view as in the classic MVC.

Overview of the Spring MVC request flow

The main entry point for a web request in a Spring MVC application is via the Dispatcher servlet. The Dispatcher servlet acts as a front controller and dispatches the requests to the other controller. The front controller's main duty is to find the appropriate right controller to hand over the request for further processing. The following diagram shows an overview of the request flow in a Spring MVC application:



The Spring MVC request flow

Okay, let's review the Spring MVC request flow in short:

1. When we enter a URL in the browser, the request comes to the Dispatcher servlet. The Dispatcher servlet acts as a centralized entry point to the web application.
2. The Dispatcher servlet determines a suitable controller that is capable of handling the request and dispatching that request to the controller.

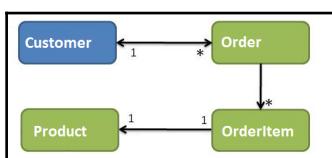
3. The controller method updates the objects in the model and returns the logical view name and updated model to the Dispatcher servlet.
4. The Dispatcher servlet consults with view resolver to determine which view to render and passes the model's data to that view.
5. View furnishes the dynamic values in the web page using the model's data and renders the final web page and returns that web page to the Dispatcher servlet.
6. At the end, the Dispatcher servlet returns the final rendered page as a response to the browser.

The web application architecture

Now you understand the overall request flow and the responsibility of each component in a typical Spring MVC application, but that is not enough for you to build an online web store application. We also need to know the best practices to develop an enterprise-level web application. One of the best practices in a typical web application is to organize source codes into layers, which will improve reusability and loose coupling. A typical web application would normally have four layers, namely presentation, domain, services, and persistence. So far, what we have seen like the Dispatcher servlet, controllers, view resolvers, and similar, are considered to be part of the Presentation layer's components. Now you need to understand the remaining layers and components one by one.

The Domain layer

So let's start with the Domain layer. The Domain layer typically consists of a domain model. So what is a domain model? A domain model is a representation of the data storage types required by the business logic. It describes the various domain objects (entities), their attributes, roles, and relationships, plus the constraints that govern the problem domain. You can look at the following order processing domain model diagram to get a quick idea about the domain model.



A sample domain model

Each block in the previous diagram represents a business entity and the lines represent the associations between the entities. Based on this domain model diagram, you should understand that in an order processing domain, a `Customer` can have many `Order` and each `order` can have many `OrderItem` and each `OrderItem` represents a single `Product`.

During actual coding and development this domain model, will be converted into corresponding domain objects and associations by a developer. A domain object is a logical container of purely domain information. Since we are going to build an online web store application, in our domain the primary domain object might be a product. So let's start with the domain object to represent a product.

Time for action – creating a domain object

So far in our web store, we have displayed only a welcome message. It is time for us to show our first product on our web page. Let's do this by creating a domain object to represent the product information.

1. Create a class called `Product` under the `com.packt.webstore.domain` package in the `src/main/java` source folder and add the following code into it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.math.BigDecimal;

public class Product implements Serializable {
    private static final long serialVersionUID =
3678107792576131001L;

    private String productId;
    private String name;
    private BigDecimal unitPrice;
    private String description;
    private String manufacturer;
    private String category;
    private long unitsInStock;
    private long unitsInOrder;
    private boolean discontinued;
    private String condition;

    public Product() {
        super();
    }

    public Product(String productId, String name, BigDecimal
```

```
        unitPrice) {
            this.productId = productId;
            this.name = name;
            this.unitPrice = unitPrice;
        }

// add setters and getters for all the fields here

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Product other = (Product) obj;
    if (productId == null) {
        if (other.productId != null)
            return false;
    } else if (!productId.equals(other.productId))
        return false;
    return true;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
        + ((productId == null) ? 0 :
            productId.hashCode());
    return result;
}
}
```

2. Add setters and getters for all the fields as well as for the previous class. I have omitted it to make the code compact, but it is really needed, so please do add setters and getters for all the fields except serialVersionUID field.
3. Now create one more controller class called `ProductController` under the `com.packt.webstore.controller` package in the `src/main/java` source folder. And add the following code into it:

```
package com.packt.webstore.controller;

import java.math.BigDecimal;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import com.packt.webstore.domain.Product;

@Controller
public class ProductController {

    @RequestMapping("/products")
    public String list(Model model) {
        Product iphone = new Product("P1234", "iPhone 6s", new
        BigDecimal(500));
        iphone.setDescription("Apple iPhone 6s smartphone
with 4.00-inch 640x1136 display and 8-megapixel rear
camera");
        iphone.setCategory("Smartphone");
        iphone.setManufacturer("Apple");
        iphone.setUnitsInStock(1000);
        model.addAttribute("product", iphone);
        return "products";
    }
}
```

4. Finally, add one more JSP view file called `products.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>

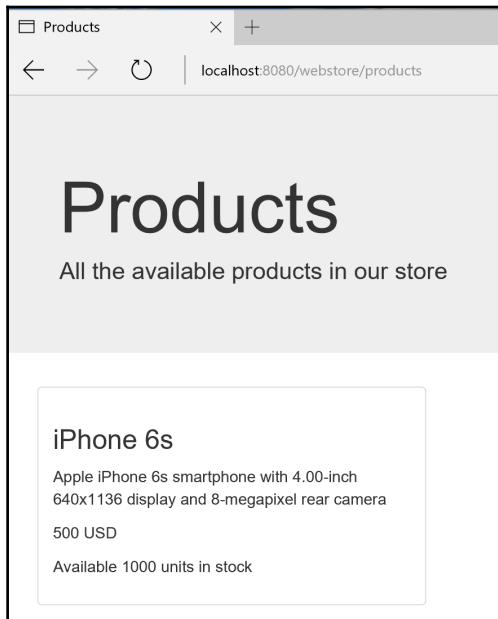
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Products</h1>
                <p>All the available products in our store</p>
            </div>
        </div>
    </section>

    <section class="container">
```

```
<div class="row">
    <div class="col-sm-6 col-md-3" style="padding-bottom: 15px">
        <div class="thumbnail">
            <div class="caption">
                <h3>${product.name}</h3>
                <p>${product.description}</p>
                <p>${product.unitPrice} USD</p>
                <p>Available ${product.unitsInStock} units in stock</p>
            </div>
        </div>
    </div>
</section>
</body>
</html>
```

5. Now run your application and enter the URL

<http://localhost:8080/webstore/products> You should be able to see a web page showing product information as shown in the following figure:



Products page showing product information

What just happened?

Our aim is to show the details of a product in our web page. In order to do that, first we need a domain object to hold the details of a product. That's what we did in step 1; we just created a class called `Product` (`Product.java`) to store information about the product such as the name, description, price, and more.

As you have already learned in the *Overview of Spring MVC request flow* section, to show any dynamic data in a web page, prior to doing so we need to put that data in a model, then only the view can read that data from the model and will render it in the web page. So to put product information in a model, we just created one more controller called `ProductController` (`ProductController.java`) in step 3.

In `ProductController`, we just have a single method called `list` whose responsibility it is to create a product domain object to hold the information about Apple's iPhone 5s and add that object to the model. And finally, we return the view name as `products`. That's what we were doing in the following lines of the `list` method of `ProductController`:

```
model.addAttribute("product", iphone);
return "products";
```

Since we configured `InternalResourceViewResolver` as our view resolver in the web application context configuration, during the process of resolving the view file for the given view name (in our case the view name is `products`), the view resolver will try to look for a file called `products.jsp` under `/WEB-INF/views/`. That's why we created `products.jsp` in step 4. If you skipped step 4, you will get a **HTTP status 404** error while running the project.

For a better visual experience, `products.jsp` contains lots of `<div>` tags with Bootstrap CSS styles applied (Bootstrap is an open source CSS framework). So don't think that `products.jsp` is very complex; as a matter of fact it is very simple-you don't need to bother about the `<div>` tags, as those are present just to get an appealing look. You only need to observe the following four tags carefully in `products.jsp` to understand the data retrieval from the model:

```
<h3>${product.name}</h3>
<p>${product.description}</p>
<p>${product.unitPrice} USD</p>
<p>Available ${product.unitsInStock} units in stock</p>
```

Look carefully at the expression `${product.unitPrice}`. The `product` text in the expression is nothing but the name of the key. We used this key to store the `iphone` domain object in the model; (remember this line `model.addAttribute("product", iphone)` from `ProductController`) and the `unitPrice` text is nothing but one of the fields from the `Product` domain class (`Product.java`). Similarly we are showing some important fields of the `product` domain class in the `products.jsp` file.



When I say that `price` is the field name, I am actually making an assumption here that you have followed the standard Java bean naming conventions for the getters and setters of your domain class.

When Spring evaluates the expression `${product.unitPrice}`, it is actually trying to call the getter method of the field to get the value, so it would expect the `getUnitPrice()` method to be in the `Product.java` file.

After finishing step 4, if we run our application and enter the URL `http://localhost:8080/webstore/products`, we are able to see a web page showing product information as shown in the screenshot after step 5.

So we created a domain class to hold information about a product, created a single product object in the controller and added it to the model, and finally showed that product's information in the view.

The Persistence layer

Since we had a single product, we just instantiated it in the controller itself and successfully showed the product information on our web page. But a typical web store would contain thousands of products, so all the product information for them would usually be stored in a database. This means we need to make our `ProductController` smart enough to load all the product information from the database into the model. But if we write all the data retrieval logic to retrieve the product information from the database in the `ProductController` itself, our `ProductController` will blow up into a big chunk of files. And logically speaking, data retrieval is not the duty of the controller because the controller is a Presentation layer component. And moreover, we want to organize the data retrieval code into a separate layer, so that we can reuse that logic as much as possible from the other controllers and layers.

So how do we retrieve data from a database in a Spring MVC way? Here comes the concept of the Persistence layer. A Persistence layer usually contains repository objects to access domain objects. A repository object sends queries to the data source for the data, then it maps the data from the data source to a domain object, and finally it persists the changes in

the domain object to the data source. So typically, a repository object is responsible for CRUD (Create, Read, Update, and Delete) operations on domain objects. And the `@Repository` (`org.springframework.stereotype.Repository`) annotation is an annotation that marks the specific class as a repository. The `@Repository` annotation also indicates that `SQLExceptions` thrown from the repository object's methods should be translated into Spring's specific `org.springframework.dao.DataAccessExceptions`. Let's create a repository layer for our application.

Time for action – creating a repository object

Let's create a repository class to access our `Product` domain objects.

1. Open `pom.xml` to add a dependency to **spring-jdbc**. In **Group Id** enter `org.springframework`, in **Artifact Id** enter `spring-jdbc`, and in **Version** enter `4.3.0.RELEASE`. Select **Scope** as `compile` and then click on the **OK** button.
2. Similarly, add the dependency for **HyperSQL DB** by clicking on the same **Add** button. This time, enter `org.hsqldb` for **Group Id**, `hsqldb` for **Artifact Id**, `2.3.2` for **Version**, select **Scope** as `compile`, and save `pom.xml`.
3. Create a class called `RootApplicationContextConfig` under the `com.packt.webstore.config` package in the `src/main/java` source folder and add the following code to it:

```
package com.packt.webstore.config;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@ComponentScan("com.packt.webstore")
public class RootApplicationContextConfig {
```

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new
    EmbeddedDatabaseBuilder();
    EmbeddedDatabase db = builder
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("db/sql/create-table.sql")
        .addScript("db/sql/insert-data.sql")
        .build();
    return db;
}
@Bean
public NamedParameterJdbcTemplate getJdbcTemplate() {
    return new NamedParameterJdbcTemplate(dataSource());
}
}
```

4. Create a folder structure called `db/sql/` under the `src/main/resources` source folder and create a file called `create-table.sql` in it. Add the following SQL script to it:

```
DROP TABLE PRODUCTS IF EXISTS;

CREATE TABLE PRODUCTS (
    ID VARCHAR(25) PRIMARY KEY,
    NAME VARCHAR(50),
    DESCRIPTION VARCHAR(250),
    UNIT_PRICE DECIMAL,
    MANUFACTURER VARCHAR(50),
    CATEGORY VARCHAR(50),
    CONDITION VARCHAR(50),
    UNITS_IN_STOCK BIGINT,
    UNITS_IN_ORDER BIGINT,
    DISCONTINUED BOOLEAN
);
```

5. Similarly create one more SQL script file called `insert-data.sql` under the `db/sql` folder and add the following script to it:

```
INSERT INTO PRODUCTS VALUES ('P1234', 'iPhone 6s', 'Apple
iPhone 6s smartphone with 4.00-inch 640x1136 display and 8-
megapixel rear
camera', '500', 'Apple', 'Smartphone', 'New', 450, 0, false);

INSERT INTO PRODUCTS VALUES ('P1235', 'Dell Inspiron',
'Dell Inspiron 14-inch Laptop (Black) with 3rd Generation
Core processors', Intel
```

```
    700, 'Dell', 'Laptop', 'New', 1000, 0, false);

    INSERT INTO PRODUCTS VALUES ('P1236', 'Nexus 7', 'Google
Nexus 7 is the lightest 7 inch tablet With a quad-core Qualcomm
Snapdragon™ S4 Pro processor',
    300, 'Google', 'Tablet', 'New', 1000, 0, false);
```

6. Open DispatcherServletInitializer and change the getRootConfigClasses method's return value to return new Class[] { RootApplicationContextConfig.class }; Basically, your getRootConfigClasses method should look as follows after your change:

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { RootApplicationContextConfig.class
};
```

7. Create an interface called ProductRepository under the com.packt.webstore.domain.repository package in the src/main/java source folder. And add a single method declaration in the interface as follows:

```
package com.packt.webstore.domain.repository;

import java.util.List;

import com.packt.webstore.domain.Product;

public interface ProductRepository {

    List <Product> getAllProducts();
}
```

8. Create a class called InMemoryProductRepository under the com.packt.webstore.domain.repository.impl package in the src/main/java source folder and add the following code to it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

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

```
.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam
.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.repository
.ProductRepository;

@Repository
public class InMemoryProductRepository implements
ProductRepository{
    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    @Override
    public List<Product> getAllProducts() {
        Map<String, Object> params = new HashMap<String,
Object>();
        List<Product> result = jdbcTemplate.query("SELECT *"
        FROM products", params, new ProductMapper());
        return result;
    }

    private static final class ProductMapper implements
RowMapper<Product> {
        public Product mapRow(ResultSet rs, int rowNum)
throws SQLException {
            Product product = new Product();
            product.setProductId(rs.getString("ID"));
            product.setName(rs.getString("NAME"));
            product.setDescription(rs.getString("DESCRIPTION"));
            product.setUnitPrice(rs.getBigDecimal("UNIT_PRICE"));
            product.setManufacturer(rs.getString("MANUFACTURER"));
            product.setCategory(rs.getString("CATEGORY"));
            product.setCondition(rs.getString("CONDITION"));
            product.setUnitsInStock(rs.getLong("UNITS_IN_STOCK"));
            product.setUnitsInOrder(rs.getLong("UNITS_IN_ORDER"));
            product.setDiscontinued(rs.getBoolean("DISCONTINUED"));
            return product;
        }
    }
}
```

9. Open `ProductController` from the `com.packt.webstore.controller` package in the `src/main/java` source folder. Add a private reference to `ProductRepository` with the `@Autowired` (`org.springframework.beans.factory.annotation.Autowired`) annotation as follows:

```
@Autowired  
private ProductRepository productRepository;
```

10. Now alter the body of the `list` method as follows in `ProductController`:

```
@RequestMapping("/products")  
public String list(Model model) {  
    model.addAttribute("products",  
    productRepository.getAllProducts());  
    return "products";  
}
```

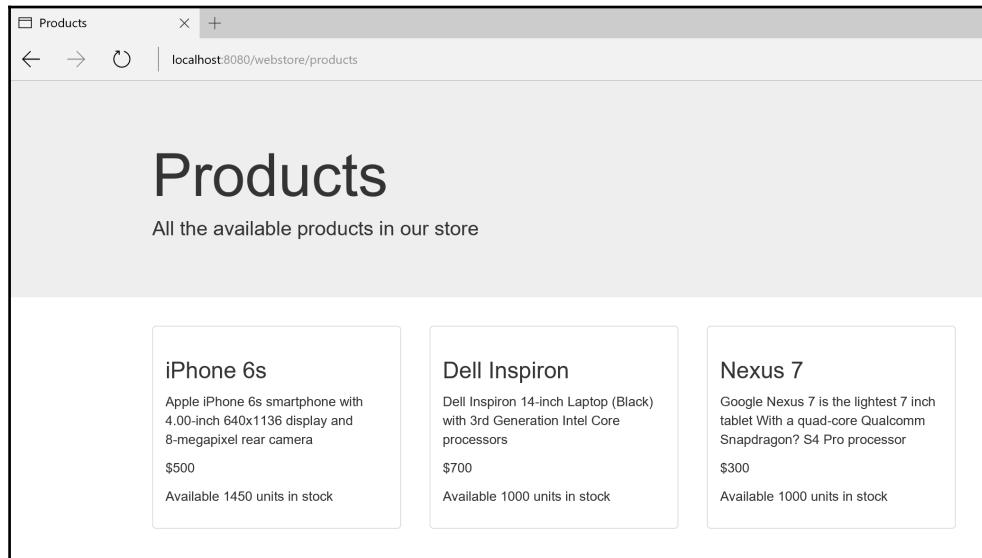
11. Finally, open your `products.jsp` view file from `src/main/webapp/WEB-INF/views/` and remove all the existing code and replace it with the following code snippet:

```
<%@ taglib prefix="c"  
uri="http://java.sun.com/jsp/jstl/core"%>  
  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html;  
charset=ISO-8859-1">  
<link rel="stylesheet"  
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css  
/bootstrap.min.css">  
<title>Products</title>  
</head>  
<body>  
    <section>  
        <div class="jumbotron">  
            <div class="container">  
                <h1>Products</h1>  
                <p>All the available products in our store</p>  
            </div>  
        </div>  
    </section>  
  
    <section class="container">  
        <div class="row">
```

```
<c:forEach items="${products}" var="product">
<div class="col-sm-6 col-md-3">
    <div class="thumbnail">
        <div class="caption">
            <h3>${product.name}</h3>
            <p>${product.description}</p>
            <p>${product.unitPrice}</p>
        <p>Available ${product.unitsInStock} units in stock</p>
    </div>
    </div>
</c:forEach>
</div>
</section>
</body>
</html>
```

12. Now run your application and enter the URL

<http://localhost:8080/webstore/products>. You will see a web page showing product information as shown in the following screenshot:



Products page showing all the products info from the in-memory repository

What just happened?

The most important step in the previous section is step 8, where we created the `InMemoryProductRepository` class. Since we don't want to write all the data retrieval logic inside the `ProductController` itself, we delegated that task to another class called `InMemoryProductRepository`. The `InMemoryProductRepository` class has a single method called `getAllProducts()`, which will return a list of product domain objects.

As the name implies, `InMemoryProductRepository` is trying to communicate with an in-memory database to retrieve all the information relating to the products. We decided to use one of the popular in-memory database implementations called **HyperSQL DB**; that's why we added the dependency for that jar in step 2. And in order to connect and query the HyperSQL database, we decided to use the `spring-jdbc` API. This means we need the `spring-jdbc` jar as well in our project, so we added that dependency in step 1.

Having the required dependency in place, the next logical step is to connect to the database. In order to connect to the database, we need a data source bean in our application context. So in step 3, we created one more bean configuration file called `RootApplicationContextConfig` and added two bean definitions in it. Let's see what they are one by one.

The first bean definition we defined in `RootApplicationContextConfig` is to create a bean for the `javax.sql.DataSource` class:

```
@Bean
public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    EmbeddedDatabase db = builder
        .setType(EmbeddedDatabaseType.HSQL)
        .addScript("db/sql/create-table.sql")
        .addScript("db/sql/insert-data.sql")
        .build();
    return db;
}
```

In this bean definition, we employed `EmbeddedDatabaseBuilder` to build an in-memory database (HyperSQL) with a specified script file to create the initial tables and data to insert. If you watch closely enough, you can see that we are passing two script files to `EmbeddedDatabaseBuilder`:

- `create-table.sql`: This file contains a SQL script to create a product table
- `insert-data.sql`: This file contains a SQL script to insert some initial product records into the product table

So the next step is to create the specified script file in the respective directory so that `EmbeddedDatabaseBuilder` can use that script file in order to initialize the in-memory database. That's what we did in steps 4 and 5.

Okay, using the data source bean, we created and initialized the in-memory database, but in order to communicate with the database we need one more bean called `NamedParameterJdbcTemplate`; that is the second bean we defined in `RootApplicationContextConfig`. If you watch the bean definition for `NamedParameterJdbcTemplate` closely enough, you can see that we have passed the `dataSource()` bean as a constructor parameter to the `NamedParameterJdbcTemplate` bean:

```
@Bean  
public NamedParameterJdbcTemplate getJdbcTemplate() {  
    return new NamedParameterJdbcTemplate(dataSource());  
}
```

Okay, so far so good. We created a bean configuration file (`RootApplicationContextConfig`) and defined two beans in it to initialize and communicate with the in-memory database. But for Spring MVC to actually pick up this bean configuration file and create beans (objects), for those bean definitions we need to hand over this file to Spring MVC during the initialization of our application. That's what we did in step 6 through the `DispatcherServletInitializer` class:

```
@Override  
protected Class<?>[] getRootConfigClasses() {  
    return new Class[] { RootApplicationContextConfig.class };  
}
```

As I have already mentioned, we did all the previously specified steps in order to use the `NamedParameterJdbcTemplate` bean in our `InMemoryProductRepository` class to communicate with the in-memory database. You may then be wondering what we did in step 7. In step 1, we are just creating an interface called `ProductRepository`, which defines the expected behavior of a product repository. As of now, the only expected behavior of a `ProductRepository` is to return a list of product domain objects (`getAllProducts`), and our `InMemoryProductRepository` is just an implementation of that interface.

Why do we have an interface and an implementation for the product repository? Remember we are actually creating a Persistence layer for our application. Who is going to use our Persistence layer repository object? Possibly a controller object (in our case `ProductController`) from the Controller layer, so it is not best practice to connect two layers (Controller and Persistence) with a direct reference. Instead we can have an interface

reference in the controller so that in future if we want to, we can easily switch to different implementations of the repository without making any code changes in the controller class.

That's the reason in step 9 that we had the `ProductRepository` reference in our `ProductController` not the `InMemoryProductRepository` reference. Notice the following lines in `ProductController`:

```
@Autowired  
private ProductRepository productRepository;
```

Okay, but why is the `@Autowired` annotation here? If you observe the `ProductController` class carefully, you may wonder why we didn't instantiate any object for the `productRepository` reference. Nowhere could we see a single line saying something like `productRepository = new InMemoryProductRepository()`.

So how come executing the line `productRepository.getAllProducts()` works fine without any `NullPointerException` in the `list` method of the `ProductController` class?

```
model.addAttribute("products", productRepository.getAllProducts() );
```

Who is assigning the `InMemoryProductRepository` object to the `productRepository` reference? The answer is that the Spring framework is the one assigning the `InMemoryProductRepository` object to the `productRepository` reference.

Remember you learned that Spring would create and manage beans (objects) for every `@controller` class? Similarly, Spring would create and manage beans for every `@Repository` class as well. As soon as Spring sees the annotation `@Autowired` on top of the `ProductRepository` reference, it will assign an object of `InMemoryProductRepository` to that reference, since Spring already created and holds the `InMemoryProductRepository` object in its object container (web application context).

Remember we configured the component scan through the following annotation in the web application context configuration file:

```
@ComponentScan("com.packt.webstore")
```

And you learned earlier that if we configure our web application context with `@ComponentScan` annotation, it not only detects controllers (`@controller`), it also detects other stereotypes such as repositories (`@Repository`) and services (`@Service`) as well.

Since we added the `@Repository` annotation on top of the `InMemoryProductRepository` class, Spring knows that if any reference of the type `productRepository` has an `@Autowired` annotation on top of it, then it should assign the implementation object `InMemoryProductRepository` to that reference. This process of managing dependencies between classes is called **dependency injection** or **wiring** in the Spring world. So to mark any class as a repository object, we need to annotate that class with the `@Repository` (`org.springframework.stereotype.Repository`) annotation.

Okay, you understand how the Persistence layer works, but after the repository object returns a list of products, how do we show it in the web page? If you remember how we added our first product to the model, it is very similar to that instead of a single object. This time we are adding a list of objects to the model through the following line in the `list` method of `ProductController`:

```
model.addAttribute("products", productRepository.getAllProducts() );
```

In the previous code, `productRepository.getAllProducts()` just returns a list of product domain objects (`List<Product>`) and we directly add that list to the model.

And in the corresponding view file (`products.jsp`), using the `<c:forEach>` tag, we loop through the list and show each product's information inside a styled `<div>` tag:

```
<c:forEach items="${products}" var="product">
<div class="col-sm-6 col-md-3" style="padding-bottom: 15px">
<div class="thumbnail">
    <div class="caption">
        <h3>${product.name}</h3>
        <p>${product.description}</p>
        <p>${product.unitPrice} USD</p>
    <p> Available ${product.unitsInStock} units in stock </p>
    </div>
</div>
</c:forEach>
```

Again, remember the `products` text in the `${products}` expression is nothing but the key that we used while adding the product list to the model from the `ProductController` class.

The for each loop is a special **JavaServer Pages Standard Tag Library (JSTL)** looping tag that will run through the list of products and assign each product to a variable called `product` (`var="product"`) on each iteration. From the `product` variable, we are fetching information such as the name, description, and price of the product and showing it within `<h3>` and `<p>` tags. That's how we are finally able to see the list of products in the products web page.



The JSTL is a collection of useful JSP tags that encapsulates the core functionality common to many JSP applications.

The Service layer

So far so good, we have created a Presentation layer that contains a controller, a dispatcher servlet, view resolvers, and more. And then we created the Domain layer, which contains a single domain class `Product`. Finally, we created the Persistence layer, which contains a repository interface and an implementation to access our `Product` domain objects from an in-memory database.

But we are still missing one more concept called the Service layer. Why do we need the Service layer? We have seen a Persistence layer dealing with all data access (CRUD) related logic, a Presentation layer dealing with all web requests and view-related activities, and a Domain layer containing classes to hold information that is retrieved from database records/the Persistence layer. But where can we put the business operations code?

The Service layer exposes business operations that could be composed of multiple CRUD operations. Those CRUD operations are usually performed by the repository objects. For example, you could have a business operation that would process a customer order, and in order to perform such a business operation, you would need to perform the following operations in order:

1. First, ensure that all the products in the requested order are available in your store.
2. Second, ensure there are sufficient quantities of those products in your store.
3. Finally, update the product inventory by reducing the available count for each product ordered.

Service objects are good candidates to put such business operation logic, where it requires multiple CRUD operations to be carried out by the repository layer for a single service call. The service operations could also represent the boundaries of SQL transactions meaning that all the elementary CRUD operations performed inside the business operations should be inside a transaction and either all of them should succeed, or they should roll back in the case of an error.

Time for action – creating a service object

Let's create a service object that will perform the simple business operation of updating stock. Our aim is dead simple: whenever we enter the URL

`http://localhost:8080/webstore/update/stock/`, our web store should go through the inventory of products and add 1,000 units to the existing stock if the number in stock is less than 500:

1. Open the `ProductRepository` interface from the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder and add one more method declaration in it as follows:

```
void updateStock(String productId, long noOfUnits);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previous declared method as follows:

```
@Override  
public void updateStock(String productId, long noOfUnits) {  
    String SQL = "UPDATE PRODUCTS SET UNITS_IN_STOCK =  
    :unitsInStock WHERE ID = :id";  
    Map<String, Object> params = new HashMap<>();  
    params.put("unitsInStock", noOfUnits);  
    params.put("id", productId);  
    jdbcTemplate.update(SQL, params);  
}
```

3. Create an interface called `ProductService` under the `com.packt.webstore.service` package in the `src/main/java` source folder. And add a method declaration in it as follows:

```
void updateAllStock();
```

4. Create a class called `ProductServiceImpl` under the `com.packt.webstore.service.impl` package in the `src/main/java` source folder. And add the following code to it:

```
package com.packt.webstore.service.impl;

import java.util.List;

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

import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.repository
    .ProductRepository;
import com.packt.webstore.service.ProductService;

@Service
public class ProductServiceImpl implements ProductService{

    @Autowired
    private ProductRepository productRepository;
    @Override
    public void updateAllStock() {
        List<Product> allProducts =
            productRepository.getAllProducts();
        for(Product product : allProducts) {
            if(product.getUnitsInStock()<500)
                productRepository.updateStock
                    (product.getProductId(),
                     product.getUnitsInStock()+1000);
        }
    }
}
```

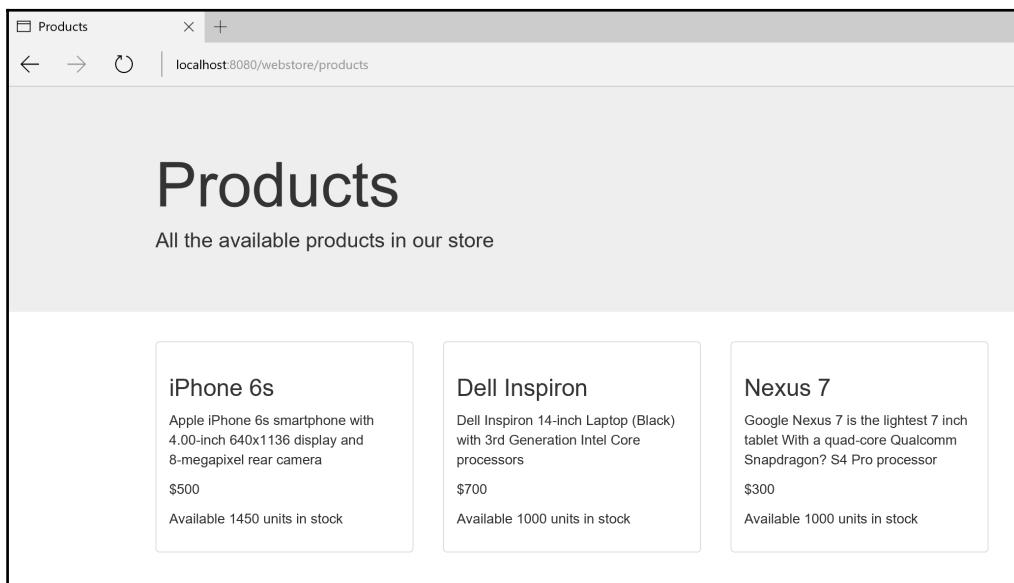
5. Open `ProductController` from the `com.packt.webstore.controller` package in the `src/main/java` source folder and add a private reference to `ProductService` with the `@Autowired` annotation as follows:

```
@Autowired
private ProductService productService;
```

6. Now add one more method definition as follows in `ProductController`:

```
@RequestMapping("/update/stock")
public String updateStock(Model model) {
    productService.updateAllStock();
    return "redirect:/products";
}
```

7. Run your application and enter the URL `http://localhost:8080/webstore/products`. You will be able to see a web page showing all the products. Notice the available units in stock for iPhone 6s will now show as **Available 450 units in stock**. All other products will show 1,000 as **Available 450 units in stock**.
8. Now enter the URL `http://localhost:8080/webstore/update/stock`, you will be able to see the same web page showing all the products. But this time, you can see that the available units in stock for iPhone 6s have been updated, and will show as **Available 1450 units in stock**.



Products page showing the product after stock has been updated via a service call

What just happened?

Okay, before going through the steps I just want to remind you of two facts regarding repository objects—that all the data access (CRUD) operations in a domain object should be carried out through repository objects only. Fact number two is that service objects rely on repository objects to carry out all data access related operations. That's why before creating the actual service interface/implementation, we created a repository interface/implementation method (`updateStock`) in steps 1 and 2.

The `updateStock` method from the `InMemoryProductRepository` class just updates a single product domain object's `unitsInStock` property for the given product. We need this method when we write logic for our service object method (`updateAllStock`) in the `OrderServiceImpl` class.

Now we come to steps 3 and 4 where we created the actual service definition and implementation. In step 3, we created an interface called `ProductService` to define all the expected responsibilities of an order service. As of now, we defined only one responsibility within that interface, which updates all the stock via the `updateAllStock` method. In step 4, we implemented the `updateAllStock` method within the `OrderServiceImpl` class, where we retrieved all the products and went through them one by one in a for loop to check whether the `unitsInStock` is less than 500. If so, we add 1,000 more units only to that product.

In the previous exercise, within the `ProductController` class, we connected to the repository through the `ProductRepository` interface reference to maximize loose coupling. Similarly, now we have connected the Service layer and repository layer through the `ProductRepository` interface reference as follows in the `ProductServiceImpl` class:

```
@Autowired  
private ProductRepository productRepository;
```

As you already learned, Spring assigns the `InMemoryProductRepository` object to `productRepository` reference in the previously mentioned code because the `productRepository` reference has an `@Autowired` annotation and we know that Spring creates and manages all the `@Service` and `@Repository` objects. Remember that `OrderServiceImpl` has an `@Service` annotation on top of it.

To ensure transactional behavior, Spring provides an `@Transactional` (`org.springframework.transaction.annotation.Transactional`) annotation. We must annotate service methods with an `@Transactional` annotation to define transaction attributes, and we need to make some more configurations in our application context to ensure the transactional behavior takes effect.



Since our book is about Spring MVC and the Presentation layer, I omitted the `@Transactional` annotation from our Service layer objects. To find out more about transactional management in Spring, check out <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/transaction.html>.

Okay, we have created the Service layer, and now it is ready to be consumed from the Presentation layer. It is time for us to connect our Service layer with the controller. In step 5, we created one more controller method called `updateStock` in `ProductController` to call the service object method:

```
@RequestMapping("/update/stock")
public String updateStock(Model model) {
    productService.updateAllStock();
    return "redirect:/products";
}
```

The `updateStock` method from `ProductController` class uses our `productService` reference to update all the stock.

You can also see that we mapped the `/update/stock` URL path to the `updateStock` method using the `@RequestMapping` annotation. So finally, when we are trying to hit the URL `http://localhost:8080/webstore/update/stock`, we are able to see the available units in stock being updated by 1,000 more units for the product P1234.

Have a go hero – accessing the product domain object via a service

In our `ProductController` class, we only have the `ProductRepository` reference to access the Product domain object within the `list` method. But accessing `ProductRepository` directly from the `ProductController` is not the best practice, as it is always good to access the Persistence layer repository via a service object.

Why don't you create a Service layer method to mediate between `ProductController` and `ProductRepository`? Here are some of things you can try out:

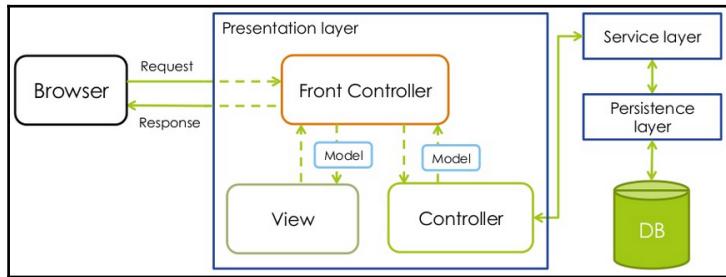
- Create a method declaration as `List <Products> getAllProducts()` within the `ProductService` interface
- Create an implementation method for the previous method declaration in `ProductServiceImpl`
- Use the `ProductRepository` reference within the `getAllProducts` method of `ProductServiceImpl` to get all the products from `ProductRepository`
- Remove the `ProductRepository` reference in the `ProductController` class and accordingly change the `list` method in `ProductController`

After finishing this, you will be able to see the same product listings under the URL `http://localhost:8080/webshop/products/` without any problems.

An overview of the web application architecture

So far you have seen how to organize our code into layers so that we can avoid tight coupling between codes and improve reusability and separation of concerns. We just created one domain class, one repository class, and one service class to demonstrate a purpose, but a typical real-world MVC application may contain many domain, repository, and service classes. Each layer is usually connected through interfaces, and the controller always accesses domain objects from the repository via a service interface only.

So every typical enterprise-level Spring MVC application will logically have four layers, namely Presentation, Domain, Persistence, and Services. The Domain layer is sometimes called the model layer. The following block diagram will help you conceptualize this idea:



Layers of a Spring MVC application

So you have learned how to create a Service layer object and a repository layer object, but what you saw in the Service layer and repository layer was just a glimpse. Spring has extensive support for database and transaction handling, which is a vast topic and deserves its own book. So in the upcoming chapters, we will concentrate more on the Presentation layer, which contains more Spring MVC related concepts, rather than the database and transaction-related concepts.

Have a go hero – listing all our customers

It's great that we have listed all our products in our web application under the URL <http://localhost:8080/webstore/products>, but in order to become a successful web store, maintaining only the product information is not enough-we need to maintain information about the customers as well, so that we can attract them by giving them special discounts based on their purchase history.

So why don't you maintain customer information in your application too? Here are some improvements you can make to your application to maintain customer information as well as product information:

- Add one more `Customer` domain class in the same package where `Product` exists:
 - Add fields such as `customerId`, `name`, `address`, and `noOfOrdersMade` to the `Customer` class
- Create a Persistence layer to return all the customers
 - Create an interface called `CustomerRepository` with a method declaration such as `List <Customers> getAllCustomers()`
 - Create an `InMemoryCustomerRepository` implementation for `CustomerRepository` and instantiate some dummy customer in the constructor of `InMemoryCustomerRepository` like we did in `InMemoryProductRepository`
- Create a Service layer to get all the customers from the repository
 - Create an interface called `CustomerService` with a method declaration such as `List <Customers> getAllCustomers()`
 - Create an implementation `CustomerServiceImpl` for `CustomerService`
- Create one more controller called `CustomerController`
 - Add a request mapping method to map the URL
`http://localhost:8080/webstore/customers`
- Create a view file called `customers.jsp`

After finishing this exercise, you will be able to see all your customers under the URL `http://localhost:8080/webstore/customers`. It is very similar to the way we listed all our products under the URL `http://localhost:8080/webstore/products`.

Summary

At the start of this chapter, you learned about the duties of the Dispatcher servlet and how it maps requests using the `@RequestMapping` annotation. Next you saw what the web application context is and how to configure a web application context for our web application. After that, you had a brief introduction to view resolvers and how `InternalResourceViewResolver` resolves the view file for a given logical view name.

You also learned the concept of MVC and saw the overall request flow of a Spring MVC application. Then you learned about the web application architecture. In the web application architecture section, you saw how to create and organize code under the various layers of a Spring MVC application such as the Domain layer, the Persistence layer, and the Service layer. During its course, we showed you how to retrieve product domain objects from the repository and present them on a web page using the controller. You also learned where a service object fits in. Finally, you got an overview of the web application architecture.

We hope you now have a good overall understanding about Spring MVC and the various components involved in developing a Spring MVC application. In the next chapter, you will learn more about controllers and related concepts in depth. So see you in the next chapter!

3

Control Your Store with Controllers

In Chapter 2, Spring MVC Architecture – Architecting Your Web Store you saw the overall architecture of a Spring MVC application. We didn't go into any of the concepts in depth; our aim was for you to understand the overall flow.

In Spring MVC, the concept of Controllers has an important role, so we are going to look at Controllers in-depth in this chapter. This chapter will cover concepts such as:

- Defining a Controller
- URI template patterns
- Matrix variables
- Request parameters

The role of a Controller in Spring MVC

In Spring MVC, the Controller's methods are the final destination point that a web request can reach. After being invoked, the Controller's method starts to process the web request by interacting with the Service layer to complete whatever work needs to be done. Usually, the Service layer executes business operations on domain objects and calls the Persistence layer to update the domain objects. After the processing is completed by the Service layer object, the Controller is responsible for updating and building up the `model` object and chooses a View for the user to see next as a response.

Remember that Spring MVC always keeps the Controllers unaware of any View technology used. That's why the Controller returns only the logical View name, and later DispatcherServlet consults with ViewResolver to find out the exact View to render. According to the Controller, the Model is a collection of arbitrary Java objects and the View is identified by a logical View name. Rendering the correct View for the given logical View name is the responsibility of ViewResolver.

In all our previous exercises, Controllers are used to return the logical View name and the Model is updated via the `model` parameter available in the controller method. There is another, seldom used way of updating the `model` parameter and returning the View name from the `controller` parameter with the help of the `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object. Look at the following code snippets as an example:

```
@RequestMapping("/all")
public ModelAndView allProducts() {
    ModelAndView modelAndView = new ModelAndView();

    modelAndView.addObject("products", productService.getAllProducts());

    modelAndView.setViewName("products");

    return modelAndView;
}
```

This code snippet just shows you how to encapsulate the Model and View using the `modelAndView` object.

Defining a Controller

Controllers are the Presentation layer components that are responsible for responding to the user's actions. These actions could be entering a particular URL in the browser, clicking on a link, submitting a form on a web page, or something similar. Any regular Java classes can be transformed into a Controller by simply annotating them with the `@Controller` (`org.springframework.stereotype.Controller`) annotation.

And as you have already learned, the `@Controller` annotation supports Spring's component scanning mechanism in auto-detecting/registering the bean definition in the web application's context. To enable this auto-registering capability, we must add the `@ComponentScan` (`org.springframework.context.annotation.ComponentScan`) annotation in the web application context configuration file. You saw how to do this in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* under the section

Understanding the web application context configuration.

A Controller class is made up of request-mapped methods, also in short called handler methods, and handler methods are annotated with the `@RequestMapping` (`org.springframework.web.bind.annotation.RequestMapping`) annotation. The `@RequestMapping` annotation is used to map the request path of a URL to a particular handler method. In Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* you got a short introduction to the `@RequestMapping` annotation and learned how to apply the `@RequestMapping` annotation on the handler method level, but in Spring MVC you can even specify the `@RequestMapping` annotation on the Controller class level. In that case, Spring MVC will consider the Controller class level `@RequestMapping` annotation's value before mapping the remaining URL request path to the handler methods. This feature is called relative request mapping.



The terms **request mapped method**, **mapped method**, **handler method**, and **controller method** all have the same meaning; these terms are used to specify a controller method with an `@RequestMapping` annotation. These terms are used interchangeably in this book.

Time for action – adding class-level request mapping

Let's add an `@RequestMapping` annotation on the class level of our `ProductController` to demonstrate the relative request mapping feature on the handler methods. But before that, we just want to ensure that you have already replaced the `ProductRepository` reference with the `ProductService` reference in the `ProductController` class as part of the previous chapter's *Have a go hero – accessing the product domain object via a service* section. Because contacting the Persistence layer directly from Presentation layer is not best practice, all access to the Persistence layer should go through the Service layer. Those who completed the exercise can directly start from step 5; others should continue from step 1.

1. Open the `ProductService` interface from the `com.packt.webstore.service` package in the `src/main/java` source folder, and add the following method declarations to it:

```
List<Product> getAllProducts();
```

2. Open the corresponding `ProductServiceImpl` implementation class from the `com.packt.webstore.service.impl` package in the `src/main/java` source folder, and add the following method implementation to it:

```
@Override
```

```
public List<Product> getAllProducts() {  
    return productRepository.getAllProducts();  
}
```

3. Open `ProductController` and remove the existing `ProductRepository` reference. Basically delete the following two lines from `ProductController`:

```
@Autowired  
private ProductRepository productRepository;
```

4. Now alter the body of the `list` method as follows in the `ProductController` class. Note this time we used the `productService` reference to get all the products:

```
@RequestMapping("/products")  
public String list(Model model) {  
    model.addAttribute("products",  
        productService.getAllProducts());  
  
    return "products";  
}
```

5. In the `ProductController` class, add the following annotation on top of the class:

```
@RequestMapping("market")
```

6. Again run our application and enter our regular URL to list the products (`http://localhost:8080/webstore/products/`). You will see the **HTTP Status 404** error page in the browser.
7. But now you can access the same products page under `http://localhost:8080/webstore/market/products/`.

What just happened?

What we demonstrated here is a simple concept called relative request mapping. In step 5, we just added an extra `@RequestMapping` annotation at the class level with a value attribute defined as `market`. Basically, your class signature will look as follows after your change:

```
@Controller  
@RequestMapping("market")  
public class ProductController {
```

In all our previous examples, we annotated `@RequestMapping` annotations only on the Controller's method level, but Spring MVC also allows us to specify request mapping on the Controller's class level. In that case, Spring MVC maps a specific URL request path on the method level that is relative to the class level of the `@RequestMapping` URL value. So if we have defined any class level request mapping, Spring MVC will consider that class level request path before mapping the remaining request path to the handler methods.

That's why when we accessed our regular products page in step 6, we got the **HTTP Status 404** error. Because we added an extra request mapping on the Controller class level, our list method is now being mapped to the URL

`http://localhost:8080/webstore/market/products`.

Now have we changed our Controller's request mapping in `ProductController`, we also need to change the return value of the `updateStock` method as follows to avoid side effects because of our change:

```
return "redirect:/market/products";
```

We will consider more about the `redirect :` prefix in the next chapter. For now, just remember this change is needed to avoid side-effects since we changed the Controller request mapping.

Default request mapping method

Every Controller class can have one default request mapping method. What is the default request mapping method? If we simply don't specify any request path value in the `@RequestMapping` annotation of a Controller's method, that method is designated as the default request mapping method for that class. So whenever a request URL just ends up with the Controller's class level request path value without any further relative path down the line, then Spring MVC will invoke this method as a response to that request.



If you specify more than one default mapping method inside a Controller, you will get `IllegalStateException` with the message **Ambiguous mapping found**. So a Controller can have only one default request mapping method at most.

Why not change the `welcome` method to the default request mapping method for our `HomeController` class?

1. In the `HomeController` class, add the following annotation on top of the class:

```
@RequestMapping("/")
```

2. And from the `welcome` method's `@RequestMapping` annotation, remove the `value` attribute completely, so now the `welcome` method will have a plain `@RequestMapping` annotation without any attributes, as follows:

```
@RequestMapping  
public String welcome(Model model) {
```

3. Now you can access the same welcome page under `http://localhost:8080/webstore/`.

Pop quiz – class level request mapping

If you imagine a web application called *Library* with the following request mapping on a Controller class level and in the method level, which is the appropriate request URL to map the request to the `productDetails` method?

```
@RequestMapping("/books")  
public class BookController {  
    ...  
  
    @RequestMapping(value = "/list")  
    public String books(Model model) {  
        ...  
  
        1. http://localhost:8080/library/books/list  
        2. http://localhost:8080/library/list  
        3. http://localhost:8080/library/list/books  
        4. http://localhost:8080/library/
```

Similarly, suppose we have another handler method called `bookDetails` under `BookController` as follows, what URL will map to that method?

```
@RequestMapping  
public String details(Model model) {  
    ...  
  
    1. http://localhost:8080/library/books/details  
    2. http://localhost:8080/library/books  
    3. http://localhost:8080/library/details  
    4. http://localhost:8080/library/
```

Handler mapping

You have learned that `DispatcherServlet` is the thing that dispatches the request to the handler methods based on the request mapping, but in order to interpret the mappings defined in a request mapping, `DispatcherServlet` needs a `HandlerMapping` (`org.springframework.web.servlet.HandlerMapping`) implementation.

`DispatcherServlet` consults with one or more `HandlerMapping` implementations to know which handler can handle the request. So `HandlerMapping` determines which Controller to call.

`HandlerMapping` interface provides the abstraction for mapping requests to handlers. The `HandlerMapping` implementations are able to inspect the request and come up with an appropriate Controller. Spring MVC provides many `HandlerMapping` implementations, and the one we are using to detect and interpret mappings from the `@RequestMapping` annotation is the `RequestMappingHandlerMapping` (`org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping`) class. To start using

`RequestMappingHandlerMapping`, we have to add the `@EnableWebMvc` annotation in our web application context configuration file, so that Spring MVC can create and register a bean for `RequestMappingHandlerMapping` in our web application context. If you remember, we already configured the `@EnableWebMvc` annotation in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store* under the *Understanding web application context configuration* section.

Using URI template patterns

In the previous chapters, you saw how to map a particular URL to a Controller's method; for example, if we entered the URL

`http://localhost:8080/webstore/market/products`, we would map that request to the `list` method of `ProductController` and list all the product information in the web page.

What if we want to list only a subset of products based on category, for instance, if we want to display only the products that fall under the category of laptops? If the URL entered is `http://localhost:8080/webstore/market/products/Laptop`, and similarly if the URL is `http://localhost:8080/webstore/market/products/Tablet`, we will like to show only tablets on the web page.

One way to do this is to have a separate request mapping method in the Controller for every unique category. But it won't scale if we have hundreds of categories; in that case we have to write hundreds of request mapping methods in the Controller. So how do we do that in an elegant way?

The Spring MVC URI template patterns feature is the answer. If you look at the following URLs carefully, the only part of the URL that changes is the category type (`Laptop` and `Tablet`). Other than that, everything is the same:

- `http://localhost:8080/webstore/products/Laptop`
- `http://localhost:8080/webstore/products/Tablet`

So we can define a common URI template for these URLs, which might look like `http://localhost:8080/webstore/products/{category}`. Spring MVC can leverage this fact and make the template portion (`{category}`) of the URL a variable, which is called a path variable in the Spring MVC world.

Time for action – showing products based on category

Let's add a category-wise View to our products page using the path variable.

1. Open the `ProductRepository` interface and add one more method declaration to `getProductsByCategory`:

```
List<Product> getProductsByCategory(String category);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previously declared method as follows:

```
@Override  
public List<Product> getProductsByCategory(String category) {  
    String SQL = "SELECT * FROM PRODUCTS WHERE CATEGORY =  
:category";  
    Map<String, Object> params = new HashMap<String, Object>();  
    params.put("category", category);  
  
    return jdbcTemplate.query(SQL, params, new  
    ProductMapper());  
}
```

3. Similarly open the `ProductService` interface and add one more method declaration to `getProductsByCategory`:

```
List<Product> getProductsByCategory(String category);
```

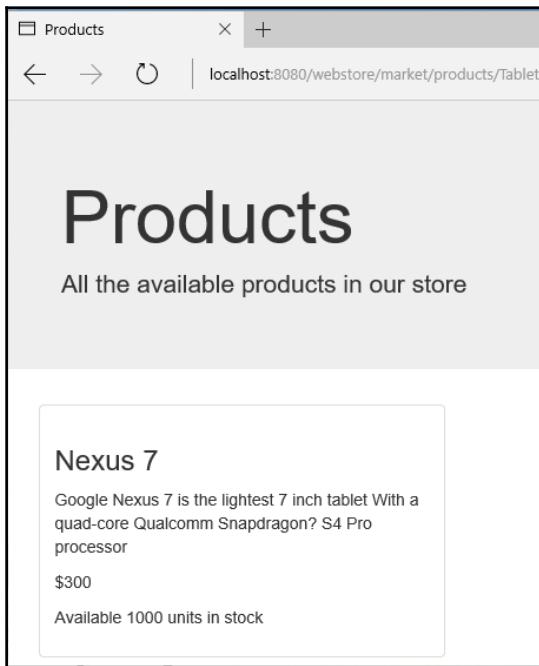
4. Open the `ProductServiceImpl` service implementation class and add an implementation as follows:

```
public List<Product> getProductsByCategory(String category) {  
    return productRepository.getProductsByCategory(category);  
}
```

5. Now open our `ProductController` class and add one more request mapping method as follows:

```
@RequestMapping("/products/{category}")  
public String getProductsByCategory(Model model,  
@PathVariable("category") String productCategory) {  
    model.addAttribute("products",  
    productService.getProductsByCategory(productCategory));  
    return "products";  
}
```

6. Now run our application and enter the following URL:
`http://localhost:8080/webstore/market/products/Tablet`. You should see the following screen:



Screen showing products by category with the help of path variables

What just happened?

Step 5 is the most important in the whole sequence, because all the steps prior to step 5 are a prerequisite for it. What we are doing in step 5 is nothing but one normal way of adding a list of product objects to the model.

```
model.addAttribute("products",
    productService.getProductsByCategory(productCategory));
```

One thing you need to notice here is the `getProductsByCategory` method from `productService`; we need this method to get the list of products for the given category. And `productService` as such cannot give the list of products for the given category; it will ask the repository.

That's why in step 4 we used the `productRepository` reference to get the list of products by category in the `ProductServiceImpl` class. Notice the following line from `ProductServiceImpl`:

```
return productRepository.getProductsByCategory(category);
```

Another important thing you need to notice in the step 5 code snippet is the `@RequestMapping` annotation's request path value:

```
@RequestMapping("/products/{category}")
```

By enclosing a portion of a request path within curly braces, we are indicating to Spring MVC that it is a URI template variable. According to the Spring MVC documentation, a URI template is a URI-like string containing one or more variable names. When you substitute values for these variables, the template becomes a URI.

For example, the URI template

`http://localhost:8080/webstore/market/products/{category}` contains the `category` variable. Assigning the value `laptop` to the variable yields `http://localhost:8080/webstore/market/products/Laptop`. In Spring MVC, we can use the `@PathVariable` (`org.springframework.web.bind.annotation.PathVariable`) annotation to read a URI template variable.

Since we have the `@RequestMapping("/market")` annotation on the `ProductController` level, the actual request path for the `getProductsByCategory` method will be `/market/products/{category}`. So at runtime, if we provide a web request URL such as `http://localhost:8080/webstore/market/products/Laptop`, then the `category` path variable will have the value `laptop`. Similarly for the web request `http://localhost:8080/webstore/market/products/Tablet`, the `category` path variable will have the value `tablet`.

Now how do we retrieve the value stored in the URI template path variable `category`? As we already mentioned, the `@PathVariable` annotation will help us to read that variable. All we need to do is simply annotate the `getProductsByCategory` method's parameter with the `@PathVariable` annotation as follows:

```
public String getProductsByCategory(@PathVariable("category") String productCategory, Model model) {
```

Spring MVC will read whatever value is present in the `category` URI template variable and assign it to the `productCategory` method parameter. So we have the category value in a variable; we just pass it to `productService` to get the list of products in that category. Once we get that list of products, we simply add it to the Model and return the same View name that we used to list all the products.

The value attribute in the `@PathVariable` annotation should be the same as the variable name in the path expression of the `@RequestMapping` annotation. For example, if the path expression is `"/products/{identity}"`, then to retrieve the path variable `identity` you have to form the `@PathVariable` annotation as `@PathVariable("identity")`.

If the `@PathVariable` annotation is specified without any `value` attribute, it will try to retrieve a path variable with the name of the variable it has been annotated with.



For example, if you specify simply `@PathVariable String productId`, then Spring will assume that it should look for a URI template variable `{productId}` in the URL. A request mapping method can have any number of `@PathVariable` annotations.

Finally in step 6, when we enter the URL

`http://localhost:8080/webstore/market/products/Tablet`, we see information about Google's Nexus 7, which is a tablet. Similarly, if you enter the URL `http://localhost:8080/webstore/products/Laptop`, you will able to see Dell's Inspiron laptop's information.

Pop quiz – request path variable

If we have a web application called `webstore` with the following request mapping on the Controller class level and in the method level, which is the appropriate request URL?

```
@RequestMapping("/items")
public class ProductController {
    ...
    @RequestMapping(value = "/type/{type}", method = RequestMethod.GET)
    public String productDetails(@PathVariable("type") String productType,
        Model model) {
```

1. `http://localhost:8080/webstore/items/electronics`
2. `http://localhost:8080/webstore/items/type/electronics`

3. `http://localhost:8080/webstore/items/productType/electronics`
4. `http://localhost:8080/webstore/type/electronics`

For the following request mapping annotation, which are the correct methods' signatures to retrieve the path variables?

```
@RequestMapping(value="/manufacturer/{  
    manufacturerId}/product/{productId}")
```

1. `public String productByManufacturer(@PathVariable String manufacturerId, @PathVariable String productId, Model model)`
2. `public String productByManufacturer (@PathVariable String manufacturer, @PathVariable String product, Model model)`
3. `public String productByManufacturer
 (@PathVariable("manufacturer") String manufacturerId,
 @PathVariable("product") String productId, Model model)`
4. `public String productByManufacturer
 (@PathVariable("manufacturerId") String manufacturer,
 @PathVariable("productId") String product, Model model)`

Using matrix variables

In the previous section, you saw the URI template facility to bind variables in the URL request path. But there is one more way to bind variables in the request URL in a name-value pair style, referred to as matrix variables within Spring MVC. Look at the following URL:

`http://localhost:8080/webstore/market/products/filter/price;low=500;high=1000`

In this URL, the actual request path is just up to

`http://localhost:8080/webstore/market/products/filter/price`, and after that we have something like `low=500;high=1000`. Here, `low` and `high` are just matrix variables. But what makes Matrix variables so special is the ability to assign multiple values for a single variable; that is, we can assign a list of values to a URI variable. Look at the following URL:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop`

In this URL, we have two variables, namely `brand` and `category`. Both have multiple values: brands (Google, Dell) and categories (Tablet, Laptop). How can we read these variables from the URL during request mapping? Here comes the special binding annotation `@MatrixVariable`

(`org.springframework.web.bind.annotation.MatrixVariable`). One cool thing about the `@MatrixVariable` annotation is that it allows us to collect the matrix variables in the map of a collection (`Map<String, List<String>>`), which will be more helpful when we are dealing with complex web requests.

Time for action – showing products based on filters

Consider a situation where we want to filter the product list based on brands and categories. For example, you want to list all the products that fall under the category Laptop and Tablets and from the manufacturer Google and Dell. With the help of Matrix variables, we can form a URL something like the following to bind the brands and categories variables' values into the URL:

```
http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop
```

Let's see how to map this URL to a handler method with the help of the `@MatrixVariable` annotation:

1. Open the `ProductRepository` interface and add one more method declaration to `getProductsByFilter`:

```
List<Product> getProductsByFilter(Map<String, List<String>>
filterParams);
```

2. Open the `InMemoryProductRepository` implementation class and add the following method implementation for `getProductsByFilter`:

```
@Override
public List<Product> getProductsByFilter(Map<String,
List<String>> filterParams) {
    String SQL = "SELECT * FROM PRODUCTS WHERE CATEGORY IN (
:categories ) AND MANUFACTURER IN ( :brands )";

    return jdbcTemplate.query(SQL, filterParams, new
ProductMapper());
}
```

3. Open the `ProductService` interface and add one more method declaration to `getProductsByFilter`:

```
List<Product> getProductsByFilter(Map<String, List<String>>  
filterParams);
```

4. Open the `ProductServiceImpl` service implementation class and add the following method implementation for `getProductsByFilter`:

```
public List<Product> getProductsByFilter(Map<String,  
List<String>> filterParams) {  
    return productService.getProductsByFilter(filterParams);  
}
```

5. Open `ProductController` and add one more request mapping method as follows:

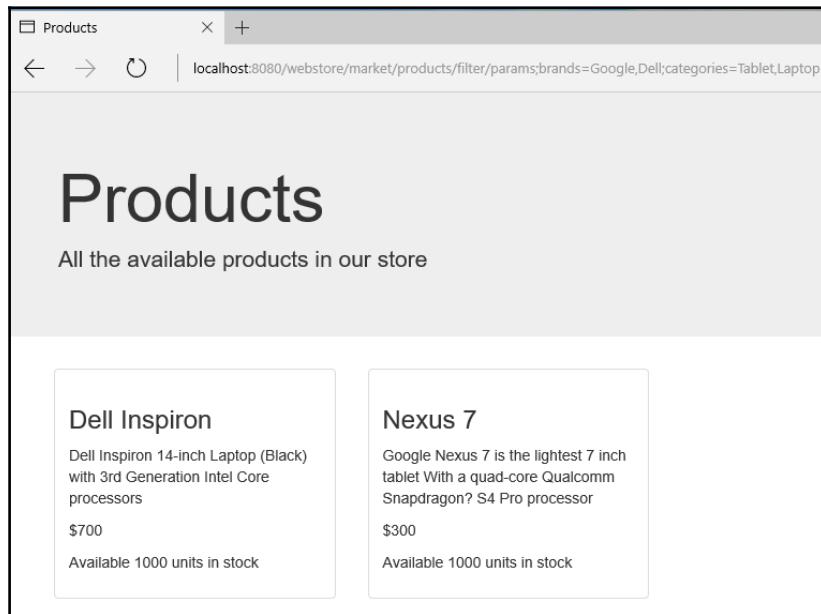
```
@RequestMapping("/products/filter/{params}")  
public String  
getProductsByFilter(@MatrixVariable(pathVar="params")  
Map<String, List<String>> filterParams, Model model) {  
    model.addAttribute("products",  
        productService.getProductsByFilter(filterParams));  
    return "products";  
}
```

6. Open our web application context configuration file (`WebApplicationContextConfig.java`) and enable matrix variable support by overriding the `configurePathMatch` method as follows:

```
@Override  
public void configurePathMatch(PathMatchConfigurer  
configurer) {  
    UrlPathHelper urlPathHelper = new UrlPathHelper();  
    urlPathHelper.setRemoveSemicolonContent(false);  
  
    configurer.setUrlPathHelper(urlPathHelper);  
}
```

7. Now run our application and enter the following URL:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop`. You will see the products list as shown in the following screen:



Using matrix variables to show the product list filtered by criteria

What just happened?

Our aim was to retrieve the matrix variable values from the URL and do something useful; in our case the URL we were trying to map is `http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop`. Here we want to extract the matrix variables brands and categories. The brands and categories variables have values: (Google, Dell) and (Tablet, Laptop) respectively. In the previously specified URL, the request path is just up to `http://localhost:8080/webstore/market/products/filter/params` only. That's why in step 5 we annotated our `getProductsByFilter` request mapping method as follows:

```
@RequestMapping("/products/filter/{params}")
```

But you may be wondering why we have a URI template (`/{params}`) in the `@RequestMapping` annotation as a mapping to a path variable. This is because, if our request URL contains a matrix variable, then we have to form the `@RequestMapping` annotation with a URI template to identify the matrix variable segments. That's why we defined `params` as a URI template in the request mapping `(@RequestMapping("/products/filter/{params}"))` annotation.



A URL can have multiple matrix variables, and each matrix variable must be separated with a ";" (semicolon). To assign multiple values to a single variable, each value must be "," (comma) separated or we can repeat the variable name. See the following URL, which is a variable repeated version of the same URL that we used in our example:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google;brands=Dell;categories=Tablet;categories=Laptop`

Note that we repeated the variable `brands` and `categories` twice in the URL.

Okay, we mapped the web request to the `getProductsByFilter` method, but how do we retrieve the value from the matrix variables? The answer is the `@MatrixVariable` annotation.

`@MatrixVariable` is very similar to the `@PathVariable` annotation; if you look at the `getProductsByFilter` method signature in step 5, we annotated the method's parameter `filterParams` with the `@MatrixVariable` annotation as follows:

```
public String getProductsByFilter(@MatrixVariable(pathVar="params")  
Map<String, List<String>> filterParams, Model model)
```

So Spring MVC will read all the matrix variables found in the URL after the `/{params}` URI template and put them into the method parameter `filterParams` map. The `filterParams` map will have each matrix variable name as the key and the corresponding list will contain multiple values assigned for the matrix variable. The `pathVar` attribute from `@MatrixVariable` is used to identify the matrix variable segment in the URL; that's why it has the value `params`, which is nothing but the URI template value we used in our request mapping URL.

A URL can have multiple matrix variable segments. See the following URL:

`http://localhost:8080/webstore/market/products/filter/params;brands=Google,Dell;categories=Tablet,Laptop/specification;dimension=10,20,15;color=red,green,blue`

It contains two matrix variable segments each identified by the prefix `params` and `specification` respectively. So, in order to capture each matrix variable segment into maps, we have to form the controller method signature as follows:

```
@RequestMapping("/products/filter/{params}/{specification}")
public String filter(@MatrixVariable(pathVar="params")
Map<String, List<String>> criteriaFilter, @MatrixVariable(pathVar="
specification") Map<String, List<String>> specFilter, Model model)
```

Okay, we got the value of the matrix variables' values into the method parameter `filterParams`, but what we have done with that `filterParams` map? We simply passed it as parameters to the service method to retrieve the products based on the criteria:

```
productService.getProductsByFilter(filterParams)
```

Again, the service passes that map to the repository to get the list of products based on the criteria. Once we get the list, as usual, we simply add that list to the Model, and return the same logical View name that was used to list the products.

To enable the use of matrix variables in Spring MVC, we must set the `RemoveSemicolonContent` property of `UrlPathHelper` to false; we did that in step 6. Finally, we are able to see products based on the specified criteria in step 7 on our product listing page.

Understanding request parameters

Matrix variables and path variables are a great way to bind variables in the URL request path. However, there is one more way to bind variables in the HTTP request, not only as a part of the URL but also in the body of the HTTP web request; these are the so-called HTTP parameters. You might have heard about GET or POST parameters; GET parameters have been used for years as a standard way to bind variables in URLs, and POST is used to bind variables in the body of an HTTP request. You will learn about POST parameters in the next chapter during form submission.

Okay, now let's see how to read GET request parameters in the Spring MVC style. To demonstrate the usage of a request parameter, let's add a product details page to our application.

Time for action – adding a product detail page

So far in our product listing page, we have only shown product information such as the product's name, description, price, and available units in stock. But we haven't shown information such as the manufacturer, category, product ID, and more. Let's add a product detail page to show them:

1. Open the `ProductRepository` interface from the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder and add one more method declaration on it as follows:

```
Product getProductById(String productID);
```

2. Open the `InMemoryProductRepository` implementation class and add an implementation for the previously declared method as follows:

```
@Override  
public Product getProductById(String productID) {  
    String SQL = "SELECT * FROM PRODUCTS WHERE ID = :id";  
    Map<String, Object> params = new HashMap<String, Object>();  
    params.put("id", productID);  
  
    return jdbcTemplate.queryForObject(SQL, params, new  
ProductMapper());  
}
```

3. Open the `ProductService` interface and add one more method declaration to it as follows:

```
Product getProductById(String productID);
```

4. Open the `ProductServiceImpl` service implementation class and add the following method implementation for `getProductById`:

```
@Override  
public Product getProductById(String productID) {  
    return productRepository.getProductById(productID);  
}
```

5. Open our `ProductController` class and add one more request mapping method as follows:

```
@RequestMapping("/product")  
public String getProductById(@RequestParam("id") String  
productId, Model model) {  
    model.addAttribute("product",
```

```
productService.getProductById(productId));
    return "product";
}
```

6. And finally add one more JSP View file called `product.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>

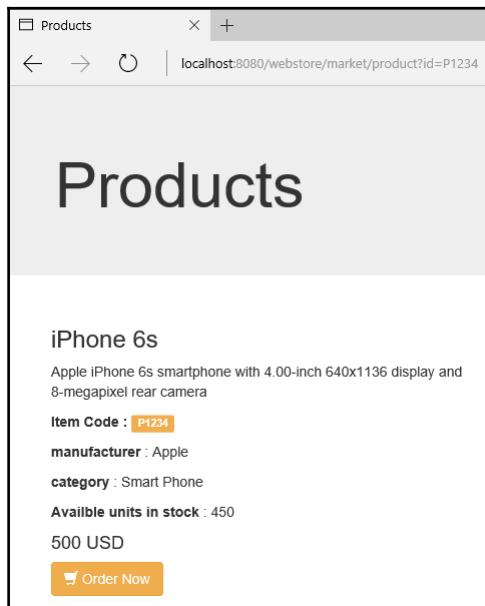
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"

href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css/
bootstrap.min.css">
<title>Products</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Products</h1>
            </div>
        </div>
    </section>
    <section class="container">
        <div class="row">
            <div class="col-md-5">
                <h3>${product.name}</h3>
                <p>${product.description}</p>
                <p>
                    <strong>Item Code : </strong><span
class="label label-warning">${product.productId}
                </span>
                </p>
                <p>
                    <strong>manufacturer : ${product.manufacturer}
                </strong>
                </p>
                <p>
                    <strong>category : ${product.category}
                </strong>
                </p>
                <p>
```

```
<strong>Available units in stock </strong> :  
 ${product.unitsInStock}  
</p>  
<h4>${product.unitPrice} USD</h4>  
<p>  
 <a href="#" class="btn btn-warning btn-large">  
 <span class="glyphicon-shopping-cart glyphicon">  
</span> Order Now  
 </a>  
</p>  
</div>  
</div>  
</section>  
</body>  
</html>
```

7. Now run our application and enter the following URL:

`http://localhost:8080/webstore/market/product?id=P1234`. You will be able to see the product detail page as shown in the following screenshot:



Using request parameters to show the product detail page

What just happened?

In steps 1 and 2, we just created a repository method declaration/implementation to get products for the given product ID (`getProductById`). Similarly in steps 3 and 4, we created a corresponding Service layer method declaration and implementation to access the `getProductById` method. What we did in step 5 is very similar to what we did in the `getProductsByCategory` method of `ProductController`. We just added a product object to the model that is returned by the service object:

```
model.addAttribute("product", productService.getProductById(productId));
```

But here, the important question is, who is providing the value for the `productId` parameter? The answer is simple, as you guessed; since we annotated the parameter `productId` with `@RequestParam("id")` annotation

(`org.springframework.web.bind.annotation.RequestParam`), Spring MVC will try to read a GET request parameter with the name `id` from the URL and will assign that to the `getProductById` method's parameter `productId`.

The `@RequestParam` annotation also follows the same convention as other binding annotations; that is, if the name of the GET request parameter and the name of the variable it is annotating are the same, then there is no need to specify the value attribute in the `@RequestParam` annotation.

And finally in step 6, we added one more View file called `product.jsp`, because we want a detailed view of the product where we can show all the information about the product. Nothing fancy in this `product.jsp` file; as usual we are getting the value from the `model` and showing it within HTML tags using the usual JSTL expression notation `${ }`:

```
<h3>${product.name}</h3>
<p>${product.description}</p>
...
...
```

Okay, you saw how to retrieve a GET request parameter from an URL, but how do you pass more than one GET request parameter in the URL? The answer is simple: the standard HTTP protocol defines a way for it; we simply need to delimit each parameter value pair with an & symbol; for example, if you want to pass `category` and `price` as GET request parameters in a URL, you have to form the URL as follows:

```
http://localhost:8080/webstore/product?category=laptop&price=700
```

Similarly, to map this URL in a request mapping method, your request mapping method should have at least two parameters with the `@RequestParam` annotation annotated:

```
public String getProducts(@RequestParam String category, @RequestParam String price) {
```

Pop quiz – the request parameter

For the following request mapping method signature, which is the appropriate request URL?

```
@RequestMapping(value = "/products", method = RequestMethod.GET)
public String productDetails(@RequestParam String rate, Model model)
```

1. <http://localhost:8080/webstore/products/rate=400>
2. <http://localhost:8080/webstore/products?rate=400>
3. <http://localhost:8080/webstore/products?rate/400>
4. <http://localhost:8080/webstore/products/rate=400>

Time for action – implementing a master detail View

A master detail View is nothing but a display of very important information in a master page; once we select an item in the master View, a detailed page of the selected item will be shown in the detailed View page. Let's build a master detail View for our product listing page, so that, when we click on any product, we can see the detailed View of that product.

We have already implemented the product listing page (<http://localhost:8080/webstore/market/products>) and product detail page (<http://localhost:8080/webstore/market/product?id=P1234>), so the only thing we need to do is connect those two Views to make it a master detail View. Let's see how to do that:

1. Open `products.jsp`. You can find `products.jsp` under `src/main/webapp/WEB-INF/views/`. Add the following spring tag lib reference on top of the file:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

2. Add the following lines after the Available units in stock paragraph tag in products.jsp:

```
<p>
<a href="
```

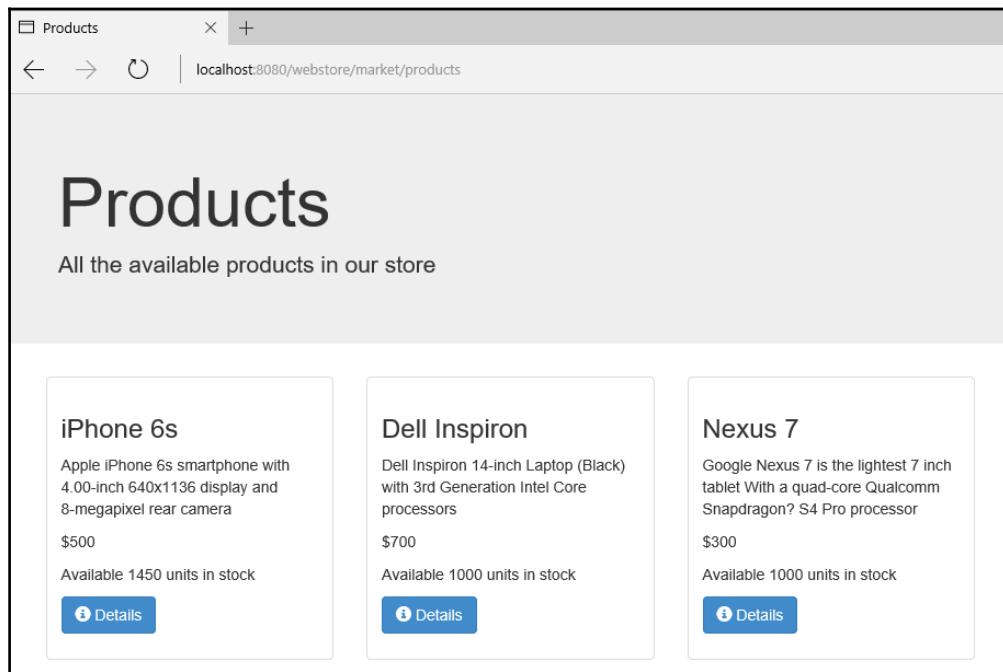
3. Now open product.jsp. You can find product.jsp under src/main/webapp/WEB-INF/views/. Add the following spring tag lib reference on top of the file:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

4. And, add the following lines just before the Order Now link in product.jsp:

```
<a href="
```

5. Now run our application and enter the following URL
`http://localhost:8080/webstore/market/products`. You will be able to see the product list page, and every product will have a **Details** button as shown in the following screenshot:



Master View of product listings

6. Now click on the any product's **Details** button and you will be able to see the detailed view with the back button link to the product listing page.

What just happened?

What we did is simple; in step 2 we just added a hyperlink using the following tag in `products.jsp`:

```
<a href=" <spring:url value="/market/product?id=${product.productId}" /> "
htmlEscape="true" />" class="btn btn-primary">
    <span class="glyphicon-info-sign glyphicon"/></span> Details
</a>
```

Notice the `href` attribute of a `<a>` tag, which has a `<spring:url>` tag as its value:

```
<spring:url value="/market/product?id=${product.productId}" />
```

This `<spring:url>` tag is used to construct a valid Spring URL. We need this `<spring:url>` to be used in step 2; that's why we added reference to the Spring tag library in step 1. Observe the value attribute of the `<spring:url>` tag, and you can see that, for the `id` URL parameter, we assigned the expression `${product.productID}`. So during the rendering of this link, Spring MVC will assign the corresponding product ID in that expression.

For example, while rendering the link for the first product, Spring MVC will assign the value `P1234` for the product ID. So the final URL value with `<spring:url>` in it will become `/market/product?id=P1234`, which is nothing but the request mapping path for the product's details page. So when you click this link, you will land on the details page for that particular product.

Similarly, we need a link back to the product listing page from product detail page; that's why we added another link in the `product.jsp` tag as follows in step 4:

```
<a href="<spring:url value="/market/products" />" class="btn btn-default">  
    <span class="glyphicon-hand-left glyphicon"></span> back  
</a>
```

Note the `` tag is just to style the button with an icon, so you do not need to pay attention to it too much; the only interesting thing for us is the `href` attribute of the `<a>` tag, which has the `<spring:url>` tag with the value attribute `/market/products` in it.

Have a go hero – adding multiple filters to list products

It is good that you learned various techniques to bind parameters with URLs such as using path variables, matrix variables, and GET parameters. We saw how to get the products of a particular category using path variables, how to get products within a particular brand and category using the matrix variable, and finally how to get a particular product by the product ID using a request parameter.

Now imagine you want to apply multiple criteria to view a desired product; for example, what if you want to view a product that falls under the `Tablet` category, and within the price range of \$200 to \$400, and from the manufacturer `Google`?

To retrieve a product that can satisfy all this criteria, we can form a URL something like this:

`http://localhost:8080/webstore/products/Tablet/price;low=200;high=400?brand="Google"`

Why don't you write a corresponding controller method to serve this request URL? Here are some hints to accomplish this requirement:

- Create one more request mapping method called `filterProducts` in the `productController` class to map the following URL:

```
http://localhost:8080/webstore/products/Tablet/price;low=200  
;high=400?brand="Google"
```

Remember this URL contains matrix variables `low` and `high` to represent the price range, a GET parameter called `brand` to identify the manufacturer, and finally a URI template path variable called `Tablet` to represent the category.

- You can use the same View file `products.jsp` to list the filtered products.

Good luck!

Summary

In this chapter, you learned about the role of a Controller in Spring MVC first. Next you learned how to define a Controller and saw the usage of the `@Controller` annotation. After that, you learned the concept of relative request mapping, where you saw how to define request mapping on the Controller level and understood how Spring relatively maps a web request to the Controller's request mapping method. You also learned how the Dispatcher servlet uses handler mapping to find out the exact handler methods. You saw various parameter binding techniques such as URI template patterns, matrix variables, and HTTP GET request parameters to bind parameter with URL. Finally, you saw how to implement a master detail View.

In the next chapter, we are going explore various Spring tags that are available in the Spring tag library. You will also learn more about form processing and how to bind form data with the HTTP POST parameter. Get ready for the next chapter...

4

Working with Spring Tag Libraries

You learned that one of the biggest advantages of using Spring MVC is its ability to separate View technologies from the rest of the MVC framework. Spring MVC supports various View technologies such as JSP/JSTL, Thymeleaf, Tiles, FreeMarker, Velocity, and more. In previous chapters, you saw some basic examples of how to use InternalResourceViewResolver to implement JSP/JSTL views. Spring MVC provides first-class support for JSP/JSTL views with the help of Spring tag libraries. In this chapter, you are going to learn more about the various tags that are available as part of Spring tag libraries.

After finishing this chapter, you will have a good idea about the following topics:

- JavaServer Pages Standard Tag Library (JSTL)
- Serving and processing web forms
- Form-binding and whitelisting
- Spring tag libraries

The JavaServer Pages Standard Tag Library

JavaServer Pages (JSP) is a technology that lets you embed Java code inside HTML pages. This code can be inserted by means of `<% %>` blocks or by means of JSTL tags. To insert Java code into JSP, JSTL tags are generally preferred, since tags adapt better to their own tag representation of HTML, making JSP pages look more readable.



JSP even lets you define your own tags; you must write the code that actually implements the logic of your own tags in Java.

JSTL is just a standard tag library provided by Oracle. We can add a reference to the JSTL tag library in our JSP pages as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Similarly, Spring MVC also provides its own tag library to develop Spring JSP views easily and effectively. These tags provide a lot of useful common functionality such as form binding, evaluating errors, and outputting messages, and more when we work with Spring MVC.

In order to use these, Spring MVC has provided tags in our JSP pages. We must add a reference to that tag library in our JSP pages as follows:

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

These `taglib` directives declare that our JSP page uses a set of custom tags related to Spring and identify the location of the library. They also provide a means to identify custom tags in our JSP page. In the `taglib` directive, the `uri` attribute value resolves to a location that the servlet container understands and the `prefix` attribute announces which bits of markup are custom actions.

Serving and processing forms

In the previous chapters, you learned how to retrieve data from an in-memory database using the Controller, but you didn't learn how to store the data in an in-memory database from the View. In Spring MVC, the process of putting an HTML form element's values into model data is called form binding.

In all the previous chapter's examples, you saw that the data transfer took place from the Model to the View via the Controller. The following line is a typical example of how we put data into the Model from the Controller:

```
model.addAttribute(greeting, "Welcome")
```

Similarly, the next line shows how we retrieve that data in the View using a JSTL expression:

```
<p> ${greeting} </p>
```

But what if we want to put data into the Model from the View? How do we retrieve that data in the Controller? For example, consider a scenario where an admin of our store wants to add new product information to our store by filling out and submitting an HTML form. How can we collect the values filled out in the HTML form elements and process them in the Controller? This is where Spring tag library tags help us to bind the HTML tag element's values to a form backing bean in the Model. Later, the Controller can retrieve the form backing bean from the Model using the `@ModelAttribute` (`org.springframework.web.bind.annotation.ModelAttribute`) annotation.



The form backing bean (sometimes called the form bean) is used to store form data. We can even use our domain objects as form beans; this works well when there's a close match between the fields in the form and the properties in our domain object. Another approach is creating separate classes for form beans, which is sometimes called **Data Transfer Objects (DTO)**.

Time for action – serving and processing forms

The Spring tag library provides some special `<form>` and `<input>` tags, which are more or less similar to HTML form and input tags, but have some special attributes to bind form elements' data with the form backed bean. Let's create a Spring web form in our application to add new products to our product list:

1. Open our `ProductRepository` interface and add one more method declaration to it as follows:

```
void addProduct(Product product);
```

2. Add an implementation for this method in the `InMemoryProductRepository` class as follows:

```
@Override  
public void addProduct(Product product) {  
    String SQL = "INSERT INTO PRODUCTS (ID, "  
        + "NAME,"  
        + "DESCRIPTION,"  
        + "UNIT_PRICE,"  
        + "MANUFACTURER,"
```

```
+ "CATEGORY,"  
+ "CONDITION,"  
+ "UNITS_IN_STOCK,"  
+ "UNITS_IN_ORDER,"  
+ "DISCONTINUED)"  
+ "VALUES (:id, :name, :desc, :price,  
:manufacturer, :category, :condition, :inStock,  
:inOrder, :discontinued)";  
Map<String, Object> params = new HashMap<>();  
params.put("id", product.getProductId());  
params.put("name", product.getName());  
params.put("desc", product.getDescription());  
params.put("price", product.getUnitPrice());  
params.put("manufacturer", product.getManufacturer());  
params.put("category", product.getCategory());  
params.put("condition", product.getCondition());  
params.put("inStock", product.getUnitsInStock());  
params.put("inOrder", product.getUnitsInOrder());  
params.put("discontinued", product.isDiscontinued());  
  
jdbcTemplate.update(SQL, params);  
}
```

3. Open our `ProductService` interface and add one more method declaration to it as follows:

```
void addProduct(Product product);
```

4. And add an implementation for this method in the `ProductServiceImpl` class as follows:

```
@Override  
public void addProduct(Product product) {  
    productRepository.addProduct(product);  
}
```

5. Open our `ProductController` class and add two more request mapping methods as follows:

```
@RequestMapping(value = "/products/add", method =  
RequestMethod.GET)  
public String getAddNewProductForm(Model model) {  
    Product newProduct = new Product();  
    model.addAttribute("newProduct", newProduct);  
    return "addProduct";  
}  
@RequestMapping(value = "/products/add", method =
```

```
RequestMethod.POST)
public String
processAddNewProductForm(@ModelAttribute("newProduct")
Product newProduct) {
    productService.addProduct(newProduct);
    return "redirect:/market/products";
}
```

6. Finally, add one more JSP View file called `addProduct.jsp` under the `src/main/webapp/WEB-INF/views/` directory and add the following tag reference declaration as the very first line in it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
```

7. Now add the following code snippet under the tag declaration line and save `addProduct.jsp`. Note that I skipped some `<form:input>` binding tags for some of the fields of the product domain object, but I strongly encourage you to add binding tags for the skipped fields while you are trying out this exercise:

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css/
bootstrap.min.css">
<title>Products</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1>Products</h1>
<p>Add products</p>
</div>
</div>
</section>
<section class="container">
<form:form method="POST" modelAttribute="newProduct"
class="form-horizontal">
<fieldset>
<legend>Add new product</legend>

<div class="form-group">
```

```
<label class="control-label col-lg-2 col-lg-2"
      for="productId">Product Id</label>
<div class="col-lg-10">
    <form:input id="productId" path="productId"
    type="text" class="form:input-large"/>
</div>
</div>

<!-- Similarly bind &lt;form:input&gt; tag for
     name,unitPrice,manufacturer,category,unitsInStock
and unitsInOrder fields--&gt;

&lt;div class="form-group"&gt;
    &lt;label class="control-label col-lg-2"
          for="description"&gt;Description&lt;/label&gt;
    &lt;div class="col-lg-10"&gt;
        &lt;form:textarea id="description"
                      path="description" rows = "2"/&gt;
    &lt;/div&gt;
&lt;/div&gt;

&lt;div class="form-group"&gt;
    &lt;label class="control-label col-lg-2"
          for="discontinued"&gt;Discontinued&lt;/label&gt;
    &lt;div class="col-lg-10"&gt;
        &lt;form:checkbox id="discontinued"
                      path="discontinued"/&gt;
    &lt;/div&gt;
&lt;/div&gt;
&lt;div class="form-group"&gt;
    &lt;label class="control-label col-lg-2"
          for="condition"&gt;Condition&lt;/label&gt;
    &lt;div class="col-lg-10"&gt;
        &lt;form:radiobutton path="condition"
                          value="New" /&gt;New
        &lt;form:radiobutton path="condition"
                          value="Old" /&gt;Old
        &lt;form:radiobutton path="condition"
                          value="Refurbished" /&gt;Refurbished
    &lt;/div&gt;
&lt;/div&gt;
&lt;div class="form-group"&gt;
    &lt;div class="col-lg-offset-2 col-lg-10"&gt;
        &lt;input type="submit" id="btnAdd" class="btn
btn-primary" value ="Add"/&gt;
    &lt;/div&gt;
&lt;/div&gt;
&lt;/fieldset&gt;</pre>
```

```
</form:form>
</section>
</body>
</html>
```

8. Now run our application and enter the URL:

<http://localhost:8080/webstore/market/products/add>. You will be able to see a web page showing a web form to add product information as shown in the following screenshot:

The screenshot shows a web-based form titled "Add new product". The form contains the following fields:

- Product Id: An input field.
- Name: An input field.
- Unit Price: An input field.
- Manufacturer: An input field.
- Category: An input field.
- Unit in Stock: An input field with the value "0".
- Unit in Order: An input field with the value "0".
- Description: A text area.
- Discontinued: A checkbox.
- Condition: Radio buttons for "New", "Old", and "Refurbished".
- Add: A blue button.

Add a products web form

9. Now enter all the information related to the new product that you want to add and click on the **Add** button. You will see the new product added in the product listing page under the URL

<http://localhost:8080/webstore/market/products>.

What just happened?

In the whole sequence, steps 5 and 6 are very important steps and need to be observed carefully. Everything mentioned prior to step 5 was very familiar to you I guess. Anyhow, I will give you a brief note on what we did in steps 1 to 4.

In step 1, we just created an `addProduct` method declaration in our `ProductRepository` interface to add new products. And in step 2, we just implemented the `addProduct` method in our `InMemoryProductRepository` class. Steps 3 and 4 are just a Service layer extension for `ProductRepository`. In step 3, we declared a similar `addProduct` method in our `ProductService` and implemented it in step 4 to add products to the repository via the `productRepository` reference.

Okay, coming back to the important step; all we did in step 5 was add two request mapping methods, namely `getAddNewProductForm` and `processAddNewProductForm`:

```
@RequestMapping(value = "/products/add", method = RequestMethod.GET)
public String getAddNewProductForm(Model model) {
    Product newProduct = new Product();
    model.addAttribute("newProduct", newProduct);
    return "addProduct";
}
@RequestMapping(value = "/products/add", method = RequestMethod.POST)
public String processAddNewProductForm(@ModelAttribute("newProduct")
Product productToBeAdded) {
    productService.addProduct(productToBeAdded);
    return "redirect:/market/products";
}
```

If you observe those methods carefully, you will notice a peculiar thing: both the methods have the same URL mapping value in their `@RequestMapping` annotations (`value = "/products/add"`). So if we enter the URL

`http://localhost:8080/webstore/market/products/add` in the browser, which method will Spring MVC map that request to?

The answer lies in the second attribute of the `@RequestMapping` annotation (`method = RequestMethod.GET` and `method = RequestMethod.POST`). Yes if you look again, even though both methods have the same URL mapping, they differ in the request method.

So what is happening behind the scenes is that, when we enter the URL `http://localhost:8080/webstore/market/products/add` in the browser, it is considered as a GET request, so Spring MVC will map that request to the `getAddNewProductForm` method. Within that method, we simply attach a new empty `Product` domain object with the `model`, under the attribute name `newProduct`. So in the `addproduct.jsp` View, we can access that `newProduct` Model object:

```
Product newProduct = new Product();
model.addAttribute("newProduct", newProduct);
```

Before jumping into the `processAddNewProductForm` method, let's review the `addproduct.jsp` View file in some detail, so that you understand the form processing flow without confusion. In `addproduct.jsp`, we just added a `<form:form>` tag from Spring's tag library:

```
<form:form modelAttribute="newProduct" class="form-horizontal">
```

Since this special `<form:form>` tag is coming from a Spring tag library, we need to add a reference to that tag library in our JSP file; that's why we added the following line at the top of the `addProducts.jsp` file in step 6:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

In the Spring `<form:form>` tag, one of the important attributes is `modelAttribute`. In our case, we assigned the value `newProduct` as the value of `modelAttribute` in the `<form:form>` tag. If you remember correctly, you can see that this value of the `modelAttribute`, and the attribute name we used to store the `newProduct` object in the Model from our `getAddNewProductForm` method, are the same. So the `newProduct` object that we attached to the `model` from the Controller method (`getAddNewProductForm`) is now bound to the form. This object is called the form backing bean in Spring MVC.

Okay, now you should look at every `<form:input>` tag inside the `<form:form>` tag. You can observe a common attribute in every tag. That attribute is path:

```
<form:input id="productId" path="productId" type="text" class="form:input-large"/>
```

The `path` attribute just indicates the field name that is relative to the form backing bean. So the value that is entered in this input box at runtime will be bound to the corresponding field of the form bean.

Okay, now it's time to come back and review our `processAddNewProductForm` method. When will this method be invoked? This method will be invoked once we press the **submit** button on our form. Yes, since every form submission is considered a POST request, this time the browser will send a POST request to the same URL
`http://localhost:8080/webstore/products/add`.

So this time the `processAddNewProductForm` method will get invoked since it is a POST request. Inside the `processAddNewProductForm` method, we are simply calling the `addProduct` service method to add the new product to the repository:

```
productService.addProduct (productToBeAdded) ;
```

But the interesting question here is, how come the `productToBeAdded` object is populated with the data that we entered in the form? The answer lies in the `@ModelAttribute` (`org.springframework.web.bind.annotation.ModelAttribute`) annotation. Notice the method signature of the `processAddNewProductForm` method:

```
public String processAddNewProductForm(@ModelAttribute("newProduct")
                                         Product productToBeAdded)
```

Here if you look at the value attribute of the `@ModelAttribute` annotation, you can observe a pattern. Yes, the `@ModelAttribute` annotation's value and the value of the `modelAttribute` from the `<form:form>` tag are the same. So Spring MVC knows that it should assign the form bounded `newProduct` object to the `processAddNewProductForm` method's `productToBeAdded` parameter.

The `@ModelAttribute` annotation is not only used to retrieve an object from the Model, but if we want we can even use the `@ModelAttribute` annotation to add objects to the Model. For instance, we can even rewrite our `getAddNewProductForm` method to something like the following, using the `@ModelAttribute` annotation:

```
@RequestMapping(value = "/products/add", method = RequestMethod.GET)
public String getAddNewProductForm(@ModelAttribute("newProduct") Product
newProduct) {
    return "addProduct";
}
```

You can see that we haven't created a new empty `Product` domain object and attached it to the model. All we did was add a parameter of the type `Product` and annotated it with the `@ModelAttribute` annotation, so Spring MVC will know that it should create an object of `Product` and attach it to the model under the name `newProduct`.

One more thing that needs to be observed in the `processAddNewProductForm` method is the logical View name it is returning: `redirect:/market/products`. So what we are trying to tell Spring MVC by returning the string `redirect:/market/products`? To get the answer, observe the logical View name string carefully; if we split this string with the ":" (colon) symbol, we will get two parts. The first part is the prefix `redirect` and the second part is something that looks like a request path: `/market/products`. So, instead of returning a View name, we are simply instructing Spring to issue a redirect request to the request path `/market/products`, which is the request path for the `list` method of our `ProductController`. So, after submitting the form, we list the products using the `list` method of `ProductController`.



As a matter of fact, when we return any request path with the `redirect:` prefix from a request mapping method, Spring will use a special View object called `RedirectView`

(`org.springframework.web.servlet.view.RedirectView`) to issue the redirect command behind the scenes. We will see more about `RedirectView` in the next chapter.

Instead of landing on a web page after the successful submission of a web form, we are spawning a new request to the request path `/market/products` with the help of `RedirectView`. This pattern is called redirect-after-post; it is a commonly used pattern with web-based forms. We are using this pattern to avoid double submission of the same form.



Sometimes after submitting the form, if we press the browser's refresh button or back button, there are chances to resubmit the same form. This behavior is called double submission.

Have a go hero – customer registration form

It is great that we created a web form to add new products to our web application under the URL `http://localhost:8080/webstore/market/products/add`. Why don't you create a customer registration form in our application to register a new customer in our application? Try to create a customer registration form under the URL `http://localhost:8080/webstore/customers/add`.

Customizing data binding

In the last section, you saw how to bind data submitted by an HTML form to a form backing bean. In order to do the binding, Spring MVC internally uses a special binding object called `WebDataBinder` (`org.springframework.web.bind.WebDataBinder`).

`WebDataBinder` extracts the data out of the `HttpServletRequest` object and converts it to a proper data format, loads it into a form backing bean, and validates it. To customize the behavior of data binding, we can initialize and configure the `WebDataBinder` object in our Controller. The `@InitBinder` (`org.springframework.web.bind.annotation.InitBinder`) annotation helps us to do that. The `@InitBinder` annotation designates a method to initialize `WebDataBinder`.

Let's look at a practical use for customizing `WebDataBinder`. Since we are using the actual domain object itself as a form backing bean, during the form submission there is a chance of security vulnerabilities. Because Spring automatically binds HTTP parameters to form bean properties, an attacker could bind a suitably-named HTTP parameter with form properties that weren't intended for binding. To address this problem, we can explicitly tell Spring which fields are allowed for form binding. Technically speaking, the process of explicitly telling which fields are allowed for binding is called whitelisting binding in Spring MVC; we can do whitelisting binding using `WebDataBinder`.

Time for action – whitelisting form fields for binding

In the previous exercise, while adding a new product we bound every field of the `Product` domain in the form, but it is meaningless to specify `unitsInOrder` and `discontinued` values during the addition of a new product because nobody can make an order before adding the product to the store; similarly discontinued products need not be added in our product list. So we should not allow these fields to be bound with the form bean while adding a new product to our store. However we should only allow all the other fields of the `Product` domain object to be bound. Let's see how to do this with the following steps:

1. Open our `ProductController` class and add a method as follows:

```
@InitBinder  
public void initialiseBinder(WebDataBinder binder) {  
    binder.setAllowedFields("productId",  
        "name",  
        "unitPrice",  
        "description",  
        "manufacturer",  
        "category",  
        "unitsInStock",  
        "condition");  
}
```

2. Add an extra parameter of the type `BindingResult` (`org.springframework.validation.BindingResult`) to the `processAddNewProductForm` method as follows:

```
public String  
processAddNewProductForm(@ModelAttribute("newProduct")  
Product productToBeAdded, BindingResult result)
```

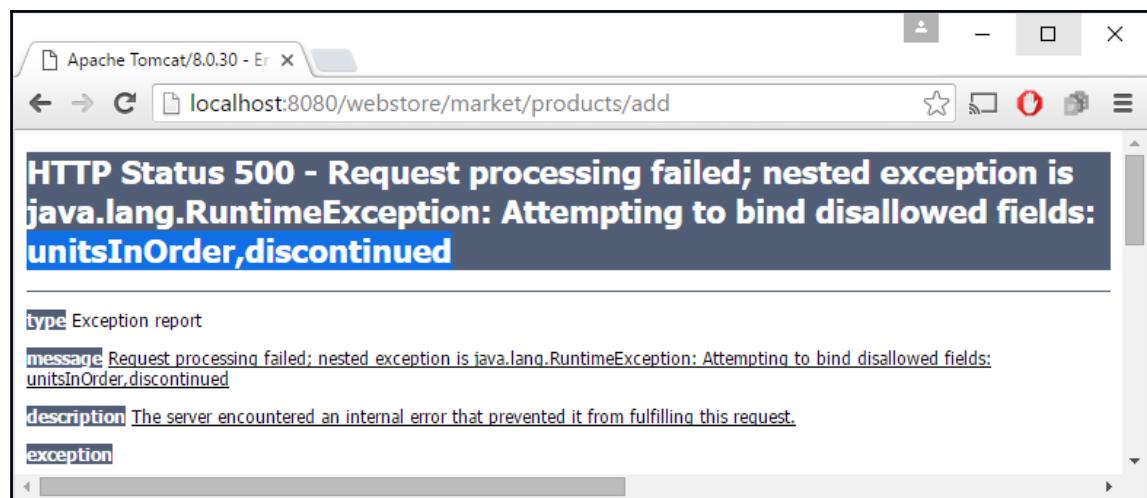
3. In the same `processAddNewProductForm` method, add the following condition just before calling the `productService.addProduct(newProduct)` line:

```
String[] suppressedFields = result.getSuppressedFields();
if (suppressedFields.length > 0) {
    throw new RuntimeException("Attempting to bind
        disallowed fields: " +
    StringUtils.arrayToCommaDelimitedString(suppressedFields));
}
```

4. Now run our application and enter the URL

`http://localhost:8080/webstore/market/products/add`. You will be able to see a web page showing a web form to add new product information. Fill out all the fields, particularly `Units in order` and `discontinued`.

5. Now press the **Add** button and you will see a **HTTP Status 500** error on the web page as shown in the following image:



6. Now open `addProduct.jsp` from `src/main/webapp/WEB-INF/views/` in your project and remove the input tags that are related to the `Units in order` and `discontinued` fields. Basically, you need to remove the following block of code:

```
<div class="form-group">
    <label class="control-label col-lg-2"
        for="unitsInOrder">Units In
```

```
        Order</label>
    <div class="col-lg-10">
        <form:input id="unitsInOrder" path="unitsInOrder"
            type="text" class="form:input-large"/>
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="discontinued">Discontinued</label>
    <div class="col-lg-10">
        <form:checkbox id="discontinued" path="discontinued"/>
    </div>
</div>
```

7. Now run our application again and enter the URL <http://localhost:8080/webstore/market/products/add>. You will be able to see a web page showing a web form to add a new product, but this time without the **Units in order** and **Discontinued** fields.
8. Now enter all information related to the new product and click on the **Add** button. You will see the new product added in the product listing page under the URL <http://localhost:8080/webstore/market/products>.

What just happened?

Our intention was to put some restrictions on binding HTTP parameters with the form baking bean. As we already discussed, the automatic binding feature of Spring could lead to a potential security vulnerability if we used a domain object itself as form bean. So we have to explicitly tell Spring MVC which fields are allowed. That's what we are doing in step 1.

The `@InitBinder` annotation designates a Controller method as a hook method to do some custom configuration regarding data binding on `WebDataBinder`. And `WebDataBinder` is the thing that is doing the data binding at runtime, so we need to specify which fields are allowed to bind to `WebDataBinder`. If you observe our `initialiseBinder` method from `ProductController`, it has a parameter called `binder`, which is of the type `WebDataBinder`. We are simply calling the `setAllowedFields` method on the `binder` object and passing the field names that are allowed for binding. Spring MVC will call this method to initialize `WebDataBinder` before doing the binding since it has the `@InitBinder` annotation.



WebDataBinder also has a method called `setDisallowedFields` to strictly specify which fields are disallowed for binding. If you use this method, Spring MVC allows any HTTP request parameters to be bound except those field names specified in the `setDisallowedFields` method. This is called blacklisting binding.

Okay, we configured which fields are allowed for binding, but we need to verify whether any fields other than those allowed are bound with the form baking bean. That's what we are doing in steps 2 and 3.

We changed `processAddNewProductForm` by adding one extra parameter called `result`, which is of the type `BindingResult`. Spring MVC will fill this object with the result of the binding. If any attempt is made to bind any fields other than the allowed fields, the `BindingResult` object will have a `getSuppressedFields` count greater than zero. That's why we were checking the suppressed field count and throwing a `RuntimeException` exception:

```
if (suppressedFields.length > 0) {  
    throw new RuntimeException("Attempting to bind disallowed fields: " +  
        StringUtils.arrayToCommaDelimitedString(suppressedFields));  
}
```



Here the static class `StringUtils` comes from
`org.springframework.util.StringUtils`.

We want to ensure that our binding configuration is working; that's why we run our application without changing the View file `addProduct.jsp` in step 4. And as expected, we got the **HTTP Status 500** error saying **Attempting to bind disallowed fields** when we submitted the **Add products** form with the `unitsInOrder` and `discontinued` fields filled out. Now we know our binder configuration is working, we could change our View file so as not to bind the disallowed fields. That's what we were doing in step 6: just removing the input field elements that are related to the disallowed fields from the `addProduct.jsp` file.

After that, our added new products page works just fine, as expected. If any outside attackers try to tamper with the POST request and attach a HTTP parameter with the same field name as the form baking bean, they will get a `RuntimeException`.

The whitelisting is just an example of how we can customize the binding with the help of WebDataBinder. But by using WebDataBinder, we can perform many more types of binding customization as well. For example, WebDataBinder internally uses many PropertyEditor (java.beans.PropertyEditor) implementations to convert the HTTP request parameters to the target field of the form backing bean. We can even register custom PropertyEditor objects with WebDataBinder to convert more complex data types. For instance, look at the following code snippet that shows how to register the custom PropertyEditor to convert a Date class:

```
@InitBinder  
public void initialiseBinder (WebDataBinder binder) {  
    DateFormat dateFormat = new SimpleDateFormat("MMM d, YYYY");  
    CustomDateEditor orderDateEditor = new CustomDateEditor(dateFormat,  
    true);  
    binder.registerCustomEditor(Date.class, orderDateEditor);  
}
```

There are many advanced configurations we can make with WebDataBinder in terms of data binding, but for a beginner level we don't need that level of detail.

Pop quiz – data binding

Consider the following data binding customization and identify the possible matching field bindings:

```
@InitBinder  
public void initialiseBinder(WebDataBinder binder) {  
    binder.setAllowedFields("unit*");  
}
```

1. NoOfUnit
2. unitPrice
3. priceUnit
4. united

Externalizing text messages

So far, in all our View files we hardcoded text values for all the labels. As an example, take our addProduct.jsp file—for the productId input tag, we have a label tag with the hardcoded text value as Product id:

```
<label class="control-label col-lg-2 col-lg-2" for="productId">Product  
Id</label>
```

Externalizing these texts from a View file into a properties file will help us to have a single centralized control for all label messages. Moreover, it will help us to make our web pages ready for internationalization. We will talk more about internationalization in Chapter 6, *Internalize Your Store with Interceptor*, but, in order to perform internalization, we need to externalize the label messages first. So now you are going to see how to externalize locale-sensitive text messages from a web page to a property file.

Time for action – externalizing messages

Let's externalize the labels texts in our `addProduct.jsp`:

1. Open our `addProduct.jsp` file and add the following tag lib reference at the top:

```
<%@ taglib prefix="spring"  
uri="http://www.springframework.org/tags" %>
```

2. Change the `productId`'s `<label>` tag value `Product Id` to `<spring:message code="addProduct.form.productId.label"/>`. After changing your `productId`'s `<label>` tag value, it should look as follows:

```
<label class="control-label col-lg-2 col-lg-2"  
for="productId"> <spring:message  
code="addProduct.form.productId.label"/> </label>
```

3. Create a file called `messages.properties` under `/src/main/resources` in your project and add the following line to it:

```
addProduct.form.productId.label = New Product ID
```

4. Now open our web application context configuration file `WebApplicationContextConfig.java` and add the following bean definition to it:

```
@Bean  
public MessageSource messageSource() {  
    ResourceBundleMessageSource resource = new  
    ResourceBundleMessageSource();  
    resource.setBasename("messages");  
    return resource;  
}
```

5. Now run our application again and enter the URL `http://localhost:8080/webstore/market/products/add`. You will be able to see the added product page with the product ID label showing as **New Product ID**.

What just happened?

Spring MVC has a special a tag called `<spring:message>` to externalize texts from JSP files. In order to use this tag, we need to add a reference to a Spring tag library; that's what we did in step 1. We just added a reference to the Spring tag library in our `addProduct.jsp` file:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
```

In step 2, we just used that tag to externalize the label text of the product ID input tag:

```
<label class="control-label col-lg-2 col-lg-2" for="productId">  
<spring:message code="addProduct.form.productId.label"/> </label>
```

Here, an important thing you need to remember is the `code` attribute of the `<spring:message>` tag; we have assigned the value `addProduct.form.productId.label` as the code for this `<spring:message>` tag. This `code` attribute is a kind of key; at runtime Spring will try to read the corresponding value for the given key (`code`) from a message source property file.

We said that Spring will read the message's value from a message source property file, so we need to create that property file. That's what we did in step 3. We just created a property file with the name `messages.properties` under the resource directory. Inside that file, we just assigned the label text value to the message tag `code`:

```
addProduct.form.productId.label = New Product ID
```

Remember, for demonstration purposes I just externalized a single label, but a typical web application will have externalized messages for almost all tags; in that case `messages.properties` file will have many code-value pair entries.

Okay, we created a message source property file and added the `<spring:message>` tag in our JSP file, but to connect these two we need to create one more Spring bean in our web application context for the

`org.springframework.context.support.ResourceBundleMessageSource` class with the name `messageSource`—we did that in step 4:

```
@Bean  
public MessageSource messageSource() {  
    ResourceBundleMessageSource resource = new  
    ResourceBundleMessageSource();  
    resource.setBasename("messages");  
    return resource;  
}
```

One important property you need to notice here is the `basename` property; we assigned the value `messages` for that property. If you remember, this is the name of the property file that we created in step 3.

That is all we did to enable the externalizing of messages in a JSP file. Now if we run the application and open up the **Add products** page, you can see that the product ID label will have the same text as we assigned to the `addProdcut.form.productID.label` code in the `messages.properties` file.

Have a go hero – externalizing all the labels from all the pages

I just showed you how to externalize the message for a single label; you can now do that for every single label available in all the pages.

Summary

At the start of this chapter, you saw how to serve and process forms, and you learned how to bind form data with a form backing bean. You also learned how to read a bean in the Controller. After that, we went a little deeper into form bean binding and configured the binder in our Controller to whitelist some of the POST parameters from being bound to the form bean. Finally, you saw how to use one more Spring special tag `<spring:message>` to externalize the messages in a JSP file.

In the next chapter, you will find out more about Views and view resolvers.

5

Working with View Resolver

In the last chapter, you saw how to use some of the Spring tags that could only be used in JSP and JSTL Views, but Spring has excellent support for other View technologies as well. Spring MVC maintains a high level of decoupling between the View and the Controller; the Controller knows nothing about the View except the View name. It is the responsibility of the view resolver to map the correct View for the given View name.

In this chapter, we will have a deeper look into Views and view resolvers. After finishing this chapter, you will have a clear idea about:

- Views and resolving Views
- Static Views
- A multipart view resolver
- Content negotiation
- Handler Exception Resolver

Resolving Views

As I already mentioned, Spring MVC does not make any assumptions about specific View technologies. According to Spring MVC, a View is identifiable as an implementation of the `org.springframework.web.servlet.View` interface:

```
public interface View {  
  
    String getContentType();  
  
    void render(Map<String, ?> model, HttpServletRequest request,  
    HttpServletResponse response) throws Exception;  
}
```

The render method from the Spring MVC View interface defines, as the main responsibility for a view object, that it should render proper content as a response (`javax.servlet.http.HttpServletResponse`) based on the given Model and request (`javax.servlet.http.HttpServletRequest`).

Because of the simplicity of the Spring MVC View interface, if we want we can write our own View implementation. But Spring MVC provides many convenient View implementations that are ready to use by simply configuring them in our web application context configuration file.

One such View is `InternalResourceView` (`org.springframework.web.servlet.view.InternalResourceView`) for rendering a response as a JSP page. Similarly, there are other View implementations such as `RedirectView`, `TilesView`, `FreeMarkerView`, `VelocityView`, and more available for specific View technologies. Spring MVC does not encourage you to couple the view object with the Controller as it will lead the controller method to tightly couple with one specific View technology. But if you want to do so, you can do something like the following code snippet:

```
@RequestMapping("/home")
public ModelAndView greeting(Map<String, Object> model) {
    model.put("greeting", "Welcome to Web Store!");
    model.put("tagline", "The one and only amazing web store");
    View view = new InternalResourceView("/WEB-INF/views/welcome.jsp");
    return new ModelAndView(view, model);
}
```

In this code handler method, we didn't return any logical View name; rather we instantiated `InternalResourceView` out of `welcome.jsp` directly and composed it into the `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object. This example is not encouraged since it tightly coupled the `greeting` handler method with `InternalResourceView`. Instead, what we can do is return a logical View name and configure an appropriate view resolver of our choice in our web app context to create a view object.

Spring comes with quite a few view resolvers to resolve various type of Views. You already saw how to configure `InternalResourceViewResolver` as our view resolver to resolve JSP Views in Chapter 2, *Spring MVC Architecture – Architecting Your Web Store*, and you also saw how `InternalResourceViewResolver` resolves a particular logical View name into a View. Anyhow, I will repeat it briefly here.

`InternalResourceViewResolver` will resolve the actual View file path by prepending the configured `prefix` value and appending the `suffix` value with the logical View name;

the logical View name is the value usually returned by the Controller's method. So the Controller's method didn't return any actual View, it just returns the View name. It is the role of `InternalResourceViewResolver` to form the correct URL path for the actual `InternalResourceView`.

RedirectView

In a web application, URL redirection or forwarding is the technique of moving visitors to a different web page than the one they requested. Most of the time, this technique is used after submitting a web form to avoid resubmission of the same form due to pressing the browser's back button or refresh. Spring MVC has a special View object that handles redirection and forwarding. To use a `RedirectView`

(`org.springframework.web.servlet.view.RedirectView`) with our Controller, we simply need to return the target URL string with the redirection prefix from the Controller. There are two redirection prefixes available in Spring MVC:

- `redirect`
- `forward`

Time for action – examining RedirectView

Though both redirection and forwarding are used to present a different web page than the one requested, there is a small difference between them. Let's try to understand them by examining them:

1. Open our `HomeController` class and add one more request mapping method as follows:

```
@RequestMapping("/welcome/greeting")
public String greeting() {
    return "welcome";
}
```

2. Now alter the return statement of the existing `welcome` request mapping method as follows and save it:

```
return "forward:/welcome/greeting";
```

3. Now run our application and enter the URL
`http://localhost:8080/webstore/`. You will able to see the welcome message on the web page.

4. Now again alter the return statement of the existing `welcome` request mapping method as follows and save it:

```
return "redirect:/welcome/greeting";
```

5. Now run our application and enter the URL

`http://localhost:8080/webstore/`. You will see a blank page without any welcome message.

6. Revert the return value of the `welcome` method to the original value:

```
return "welcome";
```

What just happened?

What we demonstrated here is how to invoke `RedirectView` from the Controller's method. In step 1, we simply created a request mapping method called `greeting` for the `welcome/greeting` request path. This method simply returns a logical View name as `welcome`.

Since we returned the logical View name as `welcome`, the `welcome.jsp` file will be rendered by `InternalResourceView` at runtime. The `InternalResourceView` file expects two model attributes, `greeting` and `tagline`, while rendering `welcome.jsp`. And in step 2, we altered the return statement of the existing request mapping method to return a redirect URL as follows:

```
@RequestMapping("/")
public String welcome(Model model) {
    model.addAttribute("greeting", "Welcome to Web Store!");
    model.addAttribute("tagline", "The one and only amazing web store");
    return "forward:/welcome/greeting";
}
```

So what we did in step 2 was more important; instead of returning a logical View name, we simply returned the request path value of the `greeting` handler method with the `forward` keyword prefixed.

The moment Spring MVC sees this, it can understand that it is not a regular logical View name, so it won't search for any View file under the `src/main/webapp/WEB-INF/views/` directory; rather it will consider this request for forwarding to another request mapping method based on the request path attached after the `forward:` keyword.

One important thing to remember here is that the forwarded request is still the active original request, so whatever value we put in the model at the start of the request would still be available; that's why we did not add any value to the Model inside the greeting method. We simply returned the View name as welcome and the welcome.jsp file with the assumption that there will be model attributes, greeting and tagline, available in the model. So when we finally run our application as mentioned in step 3, even though we issued the request to the URL `http://localhost:8080/webstore/`, RedirectView will forward our request to `http://localhost:8080/webstore/welcome/greeting` and we will be able to see the welcome message on the web page.

Again in step 4, we simply changed the return statement of the `processAddNewProductFormwelcome` method to the `redirect:` prefix. This time Spring will consider this request as a new request, so whatever value we put in the model (inside the `welcome` method) at the start of the original request will be gone. This is why you saw an empty welcome page in step 6, since the `welcome.jsp` page can't read the `greeting` and `tagline` model attributes from the model.

So, based on this exercise, we understand that `RedirectView` will come into the picture if we return a redirection URL with the appropriate prefix from the Controller method. `RedirectView` will keep the original request or spawn a new request based on redirection or forwarding.

Pop quiz – RedirectView

Consider the following customer Controller:

```
@Controller("/customers")
public class CustomerController {
    @RequestMapping("/list")
    public String list(Model model) {
        return "customers";
    }

    @RequestMapping("/process")
    public String process(Model model) {
        // return
    }
}
```

If I want to redirect the `list` method from `process`, how should I form the return statement with the `process` method?

1. `return "redirect:list"`
2. `return "redirect:/list"`
3. `return "redirect:customers/list"`
4. `return "redirect:/customers/list"`

Flash attribute

In a normal Spring MVC application, every form submitted POSTs the form data to the server; a normal Spring Controller retrieves the data from those forms from the request and processes it further. Once the operation is successful, the user is forwarded to another page showing a message that the operation was a success.

Traditionally, if we handle this scenario via the POST/Forward/GET pattern, then it may sometimes cause multiple form submission issues. The user might press *F5* and the same form will be submitted again. To resolve this issue, the POST/Redirect/GET pattern is used in many web applications. Once the user's form is submitted successfully, we redirect the request to another success page instead of forwarding it. This makes the browser perform a new GET request and load the GET page. Thus if the user even presses *F5* multiple times, the GET request gets loaded instead of submitting the form again and again.

While the POST/Redirect/GET pattern seems to perfectly solve the problem of multiple form submissions, it adds one more problem of retrieving request parameters and attributes from the initial POST request. Usually, when we perform an HTTP request redirection, the data stored in the original request is lost, making it impossible for the next GET request to access it after redirection. Flash attributes can help in such cases. Flash attributes provide a way for us to store information that is intended to be used in another request. Flash attributes are saved temporarily in a session to be available for an immediate request after redirection.

In order to use Flash attributes in your Spring MVC application, just add the `RedirectAttributes redirectAttributes` parameter to your Spring Controller's method as follows:

```
@RequestMapping  
public String welcome(Model model, RedirectAttributes redirectAttributes) {  
    model.addAttribute("greeting", "Welcome to Web Store!");  
    model.addAttribute("tagline", "The one and only amazing web store");
```

```
redirectAttributes.addFlashAttribute("greeting", "Welcome to Web Store!");
redirectAttributes.addFlashAttribute("tagline", "The one and only amazing
web store");
    return "redirect:/welcome/greeting";
}
```

In this example, we stored two attributes, namely `greeting` and `tagline`, in Flash attributes; thus when the redirection happens to the `/welcome/greeting` (the `greeting` controller method in `HomeController`) request mapping method, it will have access to those variable values that are already in the model object.

Serving static resources

So far, we have seen that every request goes through the Controller and returns a corresponding View file for the request, and most of the time these View files contain dynamic content. By dynamic content, I mean that, during the request processing, the model values are dynamically populated in the View file. For example, if the View file is of the type JSP, then we populate the model values in the JSP file using the JSTL notation `${}.`

But what if we have some static content that we want to serve to the client? For example, consider an image that is static content; we don't want to go through Controllers in order to serve (fetch) an image, as there is nothing to process or update in terms of values in the model—we simply need to return the requested image.

Let's say we have a directory (`/resources/images/`) that contains some product images and we want to serve those images upon request. For example, if the requested URL is `http://localhost:8080/webstore/img/P1234.png`, then we will like to serve the image with the name `P1234.png`. Similarly, if the requested URL is `http://localhost:8080/webstore/img/P1236.png`, then an image with the name `P1236.png` needs to be served.

Time for action – serving static resources

Let's see how to serve static images with Spring MVC:

1. Put some images under the directory `src/main/webapp/resources/images/`; I put three product images in: `P1234.png`, `P1235.png`, and `P1236.png`.

2. Override the `addResourceHandlers` method from our `WebApplicationContextConfig.java` web application context configuration file as follows:

```
@Override  
public void addResourceHandlers(ResourceHandlerRegistry  
registry) {  
    registry.addResourceHandler("/img/**")  
        .addResourceLocations("/resources/images/");  
}
```

3. Now run our application and enter the URL `http://localhost:8080/webstore/img/P1234.png` (change the image name in the URL based on the images you put in the directory in step 1).
4. You will be able to view the image you requested in the browser.

What just happened?

What just happened was simple. In step 1, we put some image files under the `src/main/webapp/resources/images/` directory. And in step 2, we just overrode the `addResourceHandlers` method from our web application context configuration file, `WebApplicationContextConfig.java`, to tell Spring where those image files are located in our project, so that Spring can serve those files upon request:

```
@Override  
public void addResourceHandlers(ResourceHandlerRegistry registry) {  
    registry.addResourceHandler("/img/**")  
        .addResourceLocations("/resources/images/");  
}
```

The `addResourceLocations` method from `ResourceHandlerRegistry` defines the base directory location of the static resources that you want to serve. In our case, we want to serve all the images that are available under the `src/main/webapp/resources/images/` directory; you may wonder then why we have only given `/resources/images` as the location value instead of `src/main/webapp/resources/images/`. This is because, during our application build and deployment time, Spring MVC will copy everything available under the `src/main/webapp/` directory to the root directory of our web application. So during resource look up, Spring MVC will start looking up from the root directory.

The other method—`addResourceHandler`—just indicated the request path that needs to be mapped to this resource directory. In our case, we assigned `/img/**` as the mapping value. So if any web request comes with the request path `/img`, then it will be mapped to the `resources/images` directory, and the `/**` symbol indicates to recursively look for any resource files underneath the base resource directory.

That is why, if you noticed in step 3, we formed the URL as follows:

`http://localhost:8080/webstore/img/P1234.png`. So while serving this web request, Spring MVC will consider `/img/P1234.png` as the request path, so it will try to map `/img` to the resource base directory `resources/images`. From that directory, it will try to look for the remaining path of the URL, which is `/P1234.png`. Since we have the `images` directory under the `resources` directory, Spring can easily locate the image file from the `images` directory.

So in our application, if any request comes with the request path prefix `/img` in its URL, then Spring will look into the `location` directory that is configured in `ResourceHandlerRegistry` and will return the requested file to the browser. Remember Spring allows you to not only host images, but also any type of static files such as PDFs, Word documents, Excel sheets, and so in this fashion.

It is good that we are able to serve product images without adding any extra request mapping methods in our Controller.

Pop quiz – static view

Consider the following resource configuration:

```
@Override  
public void addResourceHandlers(ResourceHandlerRegistry registry) {  
    registry.addResourceHandler("/resources/**")  
        .addResourceLocations("/pdf/");  
}
```

Under the `pdf` directory, if I have a sub-directory such as `product/manuals/`, which contains a PDF file called `manual-P1234.pdf`, how can I form the request path to access that PDF file?

1. `/pdf/product/manuals/manual-P1234.pdf`
2. `/resources/product/manuals/manual-P1234.pdf`
3. `/product/manuals/manual-P1234.pdf`
4. `/resource/pdf/product/manuals/manual-P1234.pdf`

Time for action – adding images to the product detail page

Let's extend this technique to show the product images in our product listing page and in the product detail page. Perform the following steps:

1. Open `products.jsp`, which you can find under the `/src/main/webapp/WEB-INF/views/` directory in your project, and add the following `` tag after the `<div class="thumbnail">` tag:

```
` tag:

```
<div class="col-md-5">
 ` tag has an expression to fetch the product ID. After getting the product ID, we simply concatenate it to the existing value to form a valid request path, as follows:

```
"/img/${product.productId}.png
```

For example, if the product ID is `P1234`, then we will get an image request URL as `/img/P1234.png`, which is nothing but one of the image file names that we already put in the `/resources/images` directory. So Spring can easily return that image file, which we showed using the `` tag in steps 1 and 2.

Multipart requests in action

In the preceding exercise, you saw how to incorporate a static View to show product images on the product details page. We simply put some images in a directory on the server and did some configuration, and Spring MVC was able to pick up those files while rendering the product details page. What if we automated this process? I mean, instead of putting those images in the directory, what if we were able to upload the images to the image directory?

How can we do this? Here comes the multipart request. A multipart request is a type of HTTP request to send files and data to the server. Spring MVC provides good support for multipart requests. Let's say we want to upload some files to the server, then we have to form a multipart request to accomplish that.

Time for action – adding images to a product

Let's add an image upload facility in our add_products page:

1. Add a bean definition in our web application context configuration file (`WebApplicationContextConfig.java`) for `CommonsMultipartResolver` as follows:

```
@Bean  
public CommonsMultipartResolver multipartResolver() {  
    CommonsMultipartResolver resolver=new  
    CommonsMultipartResolver();  
    resolver.setDefaultEncoding("utf-8");  
    return resolver;  
}
```

2. Open `pom.xml`, which you can find under the project root directory itself.
3. You will be able to see some tabs under `pom.xml`; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
4. A **Select Dependency** window will appear; in **Group Id** enter `commons-fileupload`, in **Artifact Id** enter `commons-fileupload`, in **Version** enter `1.2.2`, select **Scope** as `compile`, then click on the **OK** button.
5. Similarly, add one more dependency: `org.apache.commons` as **Group Id**, `commons-io` as **Artifact Id**, `1.3.2` as **Version**, and **Scope** as `compile`, then click on the **OK** button and save `pom.xml`.
6. Open our product's domain class (`product.java`) and add a reference to `org.springframework.web.multipart.MultipartFile` with corresponding setters and getters as follows (don't forget to add getters and setters for this field):

```
private MultipartFile productImage;
```

7. Open `addProduct.jsp`, which you can find under `the/src/main/webapp/WEB-INF/views/` directory in your project, and add the following set of tags after the `<form:input id="condition">` tag group:

```
<div class="form-group">  
    <label class="control-label col-lg-2" for="productImage">  
        <spring:message code="addProduct.form.productImage.label"/>  
</label>  
    <div class="col-lg-10">  
        <form:input id="productImage" path="productImage"
```

```
    type="file" class="form:input-large" />
    </div>
</div>
```

8. Add an entry in our message bundle source (`messages.properties`) for the product's image label, as follows:

```
addProduct.form.productImage.label = Product Image file
```

9. Now set the `enctype` attribute to `multipart/form-data` in the `form` tag as follows and save `addProduct.jsp`:

```
<form:form modelAttribute="newProduct" class="form-
horizontal" enctype="multipart/form-data">
```

10. Open our `ProductController.java` and modify the `processAddNewProductForm` method's signature by adding an extra method parameter of the type `HttpServletRequest` (`javax.servlet.http.HttpServletRequest`); so basically your `processAddNewProductForm` method signature should look like the following code snippet:

```
public String processAddNewProductForm(
@ModelAttribute("newProduct") Product newProduct,
BindingResult result, HttpServletRequest request) {
```

11. Add the following code snippet inside the `processAddNewProductForm` method just before `productService.addProduct(newProduct)`:

```
MultipartFile productImage = newProduct.getProductImage();
String rootDirectory =
request.getSession().getServletContext().getRealPath("/");
if (productImage!=null && !productImage.isEmpty()) {
    try {
        productImage.transferTo(new
File(rootDirectory+"resources\\images"+
newProduct.getProductId() + ".png"));
    } catch (Exception e) {
        throw new RuntimeException("Product Image saving
failed", e);
    }
}
```

12. Within the `initialiseBinder` method, add a `productImage` field to the whitelisting set as follows:

```
binder.setAllowedFields("productId",
    "name",
    "unitPrice",
    "description",
    "manufacturer",
    "category",
    "unitsInStock",
    "condition",
    "productImage");
```

13. Now run our application and enter the URL

<http://localhost:8080/webstore/market/products/add>. You will be able to see our add products page with an extra input field so you can choose which file to upload. Just fill out all the information as usual and, importantly, pick an image file of your choice for the newly-added image file; click on the Add button. You will be able to see that the image has been added to the **Products** page and to the product details page.

The screenshot shows a web application interface for adding a new product. The main form has fields for Product ID (P1237), Name (MDR Head Phone), Unit Price (250), Manufacturer (Sony), Category (Headphone), Units in stock (1000), Description (With Built-in Noise cancellation), Condition (New), and a Product Image file input field. An 'Add' button is at the bottom. A file upload dialog is overlaid, showing a file list with three items: P1235 (a smartphone), P1236 (a pair of headphones), and P1237 (another pair of headphones). The P1237 file is selected, highlighted with a blue border. The dialog includes standard file navigation controls and an 'Open' button.

The add products page with the image selection option

What just happened?

Spring's CommonsMultipartResolver (`org.springframework.web.multipart.commons.CommonsMultipartResolver`) class is the thing that determines whether the given request contains multipart content and parses the given HTTP request into multipart files and parameters. That's the reason we created a bean for that class within our web application context in step 1. And, through the `setMaxUploadSize` property, we set a maximum of 10,240,000 bytes as the allowed file size to be uploaded:

```
@Bean
public CommonsMultipartResolver multipartResolver() {
    CommonsMultipartResolver resolver=new CommonsMultipartResolver();
    resolver.setDefaultEncoding("utf-8");
    resolver.setMaxUploadSize(10240000);
    return resolver;
}
```

From steps 2 to 5, we added some of the `org.apache.commons` libraries as our Maven dependencies. This is because Spring uses those libraries internally to support the file uploading feature.

Since the image that we were uploading belongs to a product, it is better to keep that image as part of the product information; that's why in step 6 we added a reference to the `MultipartFile` in our domain class (`Product.java`) and added corresponding setters and getters. This `MultipartFile` reference holds the actual product image file that we are uploading.

We want to incorporate the image uploading facility in our `add_products` page; that's why, in the `addProduct.jsp` View file, we added a file input tag to choose the desired image:

```
<div class="form-group">
<label class="control-label col-lg-2" for="productImage"> <spring:message
code="addProduct.form.productImage.label"/>
</label>
<div class="col-lg-10">
<form:input id="productImage" path="productImage" type="file"
class="form:input-large" />
</div>
</div>
```

In the preceding set of tags, the important one is the `<form:input>` tag, which has the `type` attribute as `file` so that it can make the **Choose File** button display the file chooser window. As usual, we want this `form` field to be bound with the domain object field; that's

the reason we gave the path attribute as `productImage`. If you remember, this path name is just the same `MultipartFile` reference name that we added in step 6.

As usual, we want to externalize the label message for this file input tag as well; that's why we added `<spring:message>`, and in step 8 we added the corresponding message entry in the message source file (`messages.properties`).

Since our add product form is now capable of sending an image file as well as part of the request, we need to encode the request as a multipart request. This is why in step 9 we added the `enctype` attribute to the `<form:form>` tag and set its value as `multipart/form-data`. The `enctype` attribute indicates how the form data should be encoded when submitting it to the server.

We wanted to save the image file in the server under the `resources/images` directory, as this directory structure will be available directly under the root directory of our web application at runtime. So, in order to get the root directory of our web application, we need `HttpServletRequest`. See the following code snippet:

```
String rootDirectory =  
    request.getSession().getServletContext().getRealPath("/");
```

That's the reason we added an extra method parameter called `request` of the type `HttpServletRequest` to our `processAddNewProductForm` method in step 10. Remember, Spring will fill this `request` parameter with the actual HTTP request.

In step 11, we simply read the image file from the domain object and wrote it into a new file with the product ID as the name:

```
MultipartFile productImage = newProduct.getProductImage();  
String rootDirectory =  
    request.getSession().getServletContext().getRealPath("/");  
if (!productImage.isEmpty()) {  
    try {  
        productImage.transferTo(new  
File(rootDirectory+"resources\\images"+newProduct.getId() +  
.png));  
    } catch (Exception e) {  
        throw new RuntimeException("Product Image saving failed", e);  
    }  
}
```

Remember, we purposely saved the images with the product ID name because we have already designed our products (`products.jsp`) page and details (`product.jsp`) page accordingly to show the right image based on the product ID.

And as a final step, we added the newly introduced `productImage` file to the whitelisting set in the binder configuration within the `initialiseBinder` method.

Now if you run your application and enter

`http://localhost:8080/webstore/market/products/add`, you will be able to see your add products page with an extra input field to choose the file to upload.

Have a go hero – uploading product user manuals to the server

It's nice that we were able to upload the product image to the server while adding a new product. Why don't you extend this facility to upload a PDF file to the server? For example, consider that every product has a user manual and you want to upload these user manuals while adding a product.

Here are some of the things you can do to upload PDF files:

- Create a directory with the name `pdf` under the `src/main/webapp/resources/` directory in your project
- Add one more `MultipartFile` reference in your product domain class (`Product.java`) to hold the PDF file and change `Product.java` accordingly
- Extend `addProduct.jsp`
- Extend `ProductController.java` accordingly; don't forget to add the newly added field to the whitelist

So finally, if the newly added product ID is `P1237`, you will be able to access the PDF under `http://localhost:8080/webstore/pdf/P1237.pdf`.

Good luck!

Using ContentNegotiatingViewResolver

Content negotiation is a mechanism that makes it possible to serve different representations of the same resource. For example, so far we have shown our product detail page in a JSP representation. What if we want to represent the same content in an XML format. Similarly, what if we want the same content in a JSON format? Here comes Spring MVC's `ContentNegotiatingViewResolver` (`org.springframework.web.servlet.view.ContentNegotiatingViewResolver`) to help us.

The XML and JSON formats are popular data interchange formats that are heavily used in web service communications. Using `ContentNegotiatingViewResolver`, we can incorporate many Views such as `MappingJacksonJsonView` (for JSON) and `MarshallingView` (for XML) to represent the same product information in a XML or JSON format.

Time for action – configuring ContentNegotiatingViewResolver

`ContentNegotiatingViewResolver` does not resolve Views itself, but rather delegates to other view resolvers based on the request. Now, let's add a content negotiation capability to our application:

1. Open `pom.xml`, which you can find under the project root directory.
2. You should be able to see some tabs under `pom.xml`; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; in **Group Id** enter `org.springframework`, in **Artifact Id** enter `spring-oxm`, in **Version** enter `4.3.0.RELEASE`, select **Scope** as `compile`, then click on the **OK** button.
4. Add another dependency: `org.codehaus.jackson` as **Group Id**, `jackson-mapper-asl` as **Artifact Id**, `1.9.10` as **Version**, and select **Scope** as `compile`. Then click on the **OK** button.
5. Similarly, add one more dependency: `com.fasterxml.jackson.core` as **Group Id**, `jackson-databind` as **Artifact Id**, `2.8.0` as **Version**, and select **Scope** as `compile`. Then click on the **OK** button and save `pom.xml`.
6. Now add the bean configuration in our web application context configuration (`WebApplicationContextConfig.java`) for the JSON View as follows:

```
@Bean  
public MappingJackson2JsonView jsonView() {  
    MappingJackson2JsonView jsonView = new  
    MappingJackson2JsonView();  
    jsonView.setPrettyPrint(true);  
    return jsonView;  
}
```

7. Add another bean configuration for the XML View as follows:

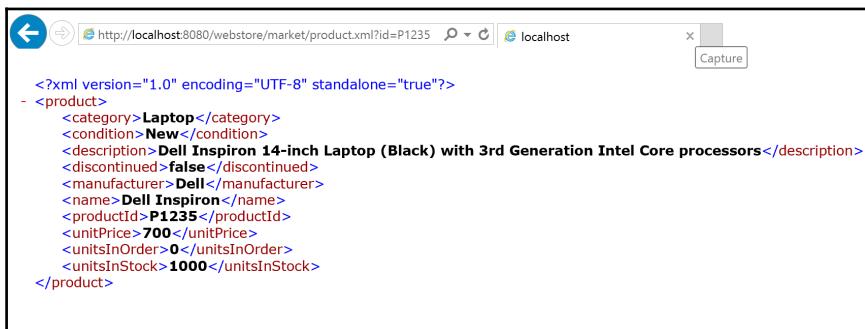
```
@Bean  
public MarshallingView xmlView() {  
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
```

```
        marshaller.setClassesToBeBound(Product.class);
        MarshallingView xmlView = new MarshallingView(marshaller);
        return xmlView;
    }
```

8. Finally, add the bean configuration for ContentNegotiatingViewResolver in our WebApplicationContextConfig web application context configuration file as follows:

```
@Bean
public ViewResolver contentNegotiatingViewResolver(
    ContentNegotiationManager manager) {
    ContentNegotiatingViewResolver resolver = new
ContentNegotiatingViewResolver();
    resolver.setContentNegotiationManager(manager);
    ArrayList<View> views = new ArrayList<>();
    views.add(jsonView());
    views.add(xmlView());
    resolver.setDefaultViews(views);
    return resolver;
}
```

9. Open our product domain class (Product.java) and add the @XmlRootElement annotation at the top of the class.
10. Similarly, add the @XmlTransient annotation at the top of the getProductImage() method and add another annotation @JsonIgnore on top of the productImage field.
11. Now run our application and enter the URL
`http://localhost:8080/webstore/market/product?id=P1235`. You will now be able to view the details page of the product with the ID P1234.
12. Now change the URL to the .xml extension:
`http://localhost:8080/webstore/market/product.xml?id=P1235`. You should be able to see the same content in the XML format as shown in the following screenshot:



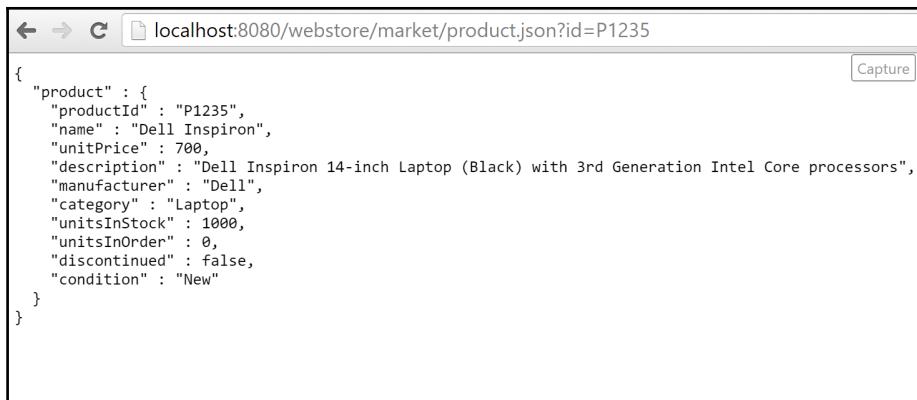
A screenshot of a web browser window. The address bar shows the URL `http://localhost:8080/webstore/market/product.xml?id=P1235`. The page content displays an XML document with the following structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <product>
  <category>Laptop</category>
  <condition>New</condition>
  <description>Dell Inspiron 14-inch Laptop (Black) with 3rd Generation Intel Core processors</description>
  <discontinued>false</discontinued>
  <manufacturer>Dell</manufacturer>
  <name>Dell Inspiron</name>
  <productId>P1235</productId>
  <unitPrice>700</unitPrice>
  <unitsInOrder>0</unitsInOrder>
  <unitsInStock>1000</unitsInStock>
</product>
```

The product details page showing product information in the XML format

13. Similarly, this time change the URL to the .json extension:

`http://localhost:8080/webstore/market/product.json?id=P1235`. You will be able to see the JSON representation of that content as shown in the following screenshot:



A screenshot of a web browser window. The address bar shows the URL `http://localhost:8080/webstore/market/product.json?id=P1235`. The page content displays a JSON object with the following structure:

```
{
  "product" : {
    "productId" : "P1235",
    "name" : "Dell Inspiron",
    "unitPrice" : 700,
    "description" : "Dell Inspiron 14-inch Laptop (Black) with 3rd Generation Intel Core processors",
    "manufacturer" : "Dell",
    "category" : "Laptop",
    "unitsInStock" : 1000,
    "unitsInOrder" : 0,
    "discontinued" : false,
    "condition" : "New"
  }
}
```

The product details page showing the product information in the JSON format

What just happened?

Since we want an XML representation of our model data to convert our model objects into XML, we need Spring's object/XML mapping support—that's why we added the dependency for `spring-oxm.jar` through steps 1 to 3. The `spring-oxm` notation will help us convert an XML document to and from a Java object.

Similarly, to convert model objects into JSON, Spring MVC will use `jackson-mapper-asl.jar` and `jackson-databind.jar`; thus we need those jars in our project as well. In steps 4 and 5, we just added the dependency configuration for those jars.

If you remember in our servlet context (`servlet-context.xml`), we already defined `InternalResourceViewResolver` as our view resolver to resolve JSP-based Views, but this time we want a view resolver to resolve XML and JSON Views. That's why in step 8 we configured `ContentNegotiatingViewResolver` (`org.springframework.web.servlet.view.ContentNegotiatingViewResolver`) in our servlet context.

As I already mentioned, `ContentNegotiatingViewResolver` does not resolve Views itself but rather it delegates to other Views based on the request, so we need to introduce other Views to `ContentNegotiatingViewResolver`. How we do that is through the `setDefaultViews` method in `ContentNegotiatingViewResolver`:

```
@Bean
public ViewResolver contentNegotiatingViewResolver(
    ContentNegotiationManager manager) {
    ContentNegotiatingViewResolver resolver = new
    ContentNegotiatingViewResolver();
    resolver.setContentNegotiationManager(manager);
    ArrayList<View> views = new ArrayList<>();
    views.add(jsonView());
    views.add(xmlView());
    resolver.setDefaultViews(views);
    return resolver;
}
```

To configure bean references for `jsonView` and `xmlView` inside `ContentNegotiatingViewResolver`, we need a bean definition for those references. That is the reason we defined those beans in steps 6 and 7.

The `xmlView` bean configuration especially has one important property to be set called `classesToBeBound`; this lists the domain objects that require XML conversion during the request processing. Since our product domain object requires XML conversion, we added `com.packt.webstore.domain.Product` to the list `classesToBeBound`:

```
@Bean
public MarshallingView xmlView() {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setClassesToBeBound(Product.class);
    MarshallingView xmlView = new MarshallingView(marshaller);
    return xmlView;
}
```

In order to convert it to XML, we need to give `MarshallingView` one more hint to identify the root XML element in the `Product` domain object. This is why in step 9 we annotated our class with the `@XmlRootElement` (`javax.xml.bind.annotation.XmlRootElement`) annotation.

In step 10, we added the `@XmlTransient` (`javax.xml.bind.annotation.XmlTransient`) annotation on top of the `getImage()` method and added another annotation—`@JsonIgnore` (`org.codehaus.jackson.annotate.JsonIgnore`)—on top of the `productImage` field. This is because we don't want to represent the product image as part of the XML View or the JSON View since both formats are purely a text-based representation, and it is not possible to represent images in text.

In step 10, we simply accessed our product details page in the regular way by firing the web request (`http://localhost:8080/webstore/products/product?id=P1234`) from the browser. We could see the normal JSP View, as expected.

In step 11, we just changed the URL slightly by adding an `.xml` extension to the request path:
`http://localhost:8080/webstore/market/products/product.xml?id=P1235`. This time we were able to see the same product information in the XML format.

Similarly for the JSON View, we changed the extension for the `.json` path to
`http://localhost:8080/webstore/market/products/product.json?id=P1235` and we were able to see the JSON representation of the same product information.

Working with HandlerExceptionResolver

Spring MVC provides several approaches to exception handling. In Spring, one of the main exception handling constructs is the `HandlerExceptionResolver` (`org.springframework.web.servlet.HandlerExceptionResolver`) interface. Any objects that implement this interface can resolve exceptions thrown during Controller mapping or execution. `HandlerExceptionResolver` implementers are typically registered as beans in the web application context.

Spring MVC creates two such `HandlerExceptionResolver` implementations by default to facilitate exception handling:

- `ResponseStatusExceptionResolver` is created to support the `@ResponseStatus` annotation

- `ExceptionHandlerExceptionResolver` is created to support the `@ExceptionHandler` annotation

Time for action – adding a `ResponseStatus` exception

We will look at them one by one. First, the `@ResponseStatus` (`org.springframework.web.bind.annotation.ResponseStatus`) annotation; in Chapter 3, *Control Your Store with Controllers* we created a request mapping method to show products by category under the URI template:

`http://localhost:8080/webstore/market/products/{category}`. If no products were found under the given category, we showed an empty web page, which is not correct semantically as we should show a HTTP status error to indicate that no products exist under the given category. Let's see how to do that with the help of the `@ResponseStatus` annotation:

1. Create a class called `NoProductsFoundUnderCategoryException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder. Now add the following code to it:

```
package com.packt.webstore.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.
    ResponseStatus;

@ResponseStatus(value=HttpStatus.NOT_FOUND, reason="No
products found under this category")
public class NoProductsFoundUnderCategoryException extends
RuntimeException{

    private static final long serialVersionUID =
3935230281455340039L;
}
```

2. Now open our `ProductController` class and modify the `getProductsByCategory` method as follows:

```
@RequestMapping("/products/{category}")
public String getProductsByCategory(Model model,
@PathVariable("category") String category) {
    List<Product> products =
productService.getProductsByCategory(category);

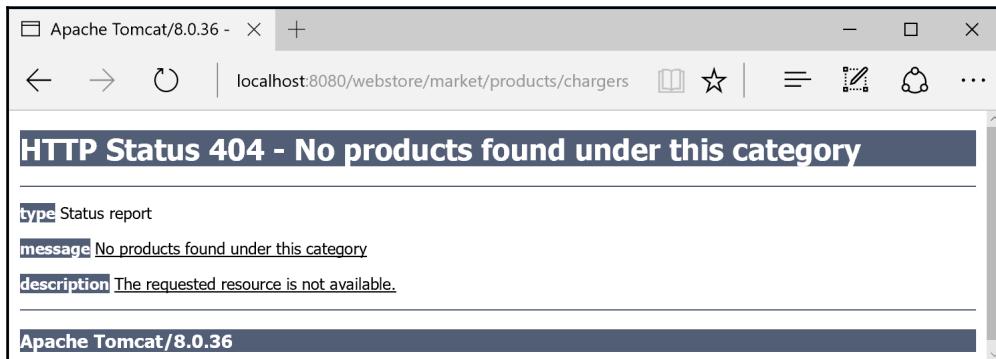
    if (products == null || products.isEmpty()) {
```

```
        throw new NoProductsFoundUnderCategoryException();
    }

    model.addAttribute("products", products);
    return "products";
}
```

3. Now run our application and enter the URL

`http://localhost:8080/webstore/market/products/chargers`. You will see a HTTP status error saying **No products found under this category**, as shown here:



Product category page showing the HTTP Status 404 for No products found under this category

What just happened?

In step 1, we created a runtime exception called `NoProductsFoundUnderCategoryException` to indicate that no products were found under the given category. One of the important constructs in the `NoProductsFoundUnderCategoryException` class is the `@ResponseStatus` annotation, which instructs the Spring MVC to return a specific HTTP status if this exception has been thrown from a request mapping method.

We can configure which HTTP status needs to be returned via the `value` attribute of the `@ResponseStatus` annotation; in our case we configured `HttpStatus.NOT_FOUND` (`org.springframework.http.HttpStatus`), which displays the familiar HTTP 404 response. The second attribute, `reason`, denotes the reason for the HTTP response error.

In step 2, we just modified the `getProductsByCategory` method in the `ProductController` class to check whether the product list for the given category is empty. If so, we simply throw the exception we created in step 1, which returns a **HTTP Status 404** error to the client saying **No products found under this category**.

So finally in step 3, we fired the web request

`http://localhost:8080/webstore/market/products/chargers`, which would try to look for products under the category `chargers` but, since we didn't have any products under the `chargers` category, we got the **HTTP Status 404** error.

It's good that we can show the HTTP status error for products not found under a given category, but sometimes you may wish to have an error page where you want to show your error message in a more detailed manner.

For example, run our application and enter

`http://localhost:8080/webstore/market/product?id=P1234`. You will be able to see a detailed View of the iPhone 6s—now change the product ID in the URL to an invalid one such as `http://localhost:8080/webstore/market/product?id=P1000`, and you will see an error page.

Time for action – adding an exception handler

We should show a nice error message saying No products found with the given product ID, so let's do that with the help of `@ExceptionHandler`:

1. Create a class called `ProductNotFoundException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder. Add the following code to it:

```
package com.packt.webstore.exception;

public class ProductNotFoundException extends
RuntimeException{

    private static final long serialVersionUID =
-694354952032299587L;
    private String productId;

    public ProductNotFoundException(String productId) {
        this.productId = productId;
    }

    public String getProductId() {
```

```
        return productId;
    }

}
```

2. Now open our `InMemoryProductRepository` class and modify the `getProductById` method as follows:

```
@Override
public Product getProductById(String productID) {
    String SQL = "SELECT * FROM PRODUCTS WHERE ID = :id";
    Map<String, Object> params = new HashMap<>();
    params.put("id", productID);
    try {
        return jdbcTemplate.queryForObject(SQL, params, new
ProductMapper());
    } catch (DataAccessException e) {
        throw new ProductNotFoundException(productID);
    }
}
```

3. Add an exception handler method using the `@ExceptionHandler` (`org.springframework.web.bind.annotation.ExceptionHandler`) annotation, as follows, in the `ProductController` class:

```
@ExceptionHandler(ProductNotFoundException.class)
public ModelAndView handleError(HttpServletRequest req,
ProductNotFoundException exception) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("invalidProductId",
exception.getProductId());
    mav.addObject("exception", exception);
    mav.addObject("url",
req.getRequestURL()+"?"+req.getQueryString());
    mav.setViewName("productNotFound");
    return mav;
}
```

4. Finally, add one more JSP View file called `productNotFound.jsp` under the `src/main/webapp/WEB-INF/views/` directory, add the following code snippets to it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
```

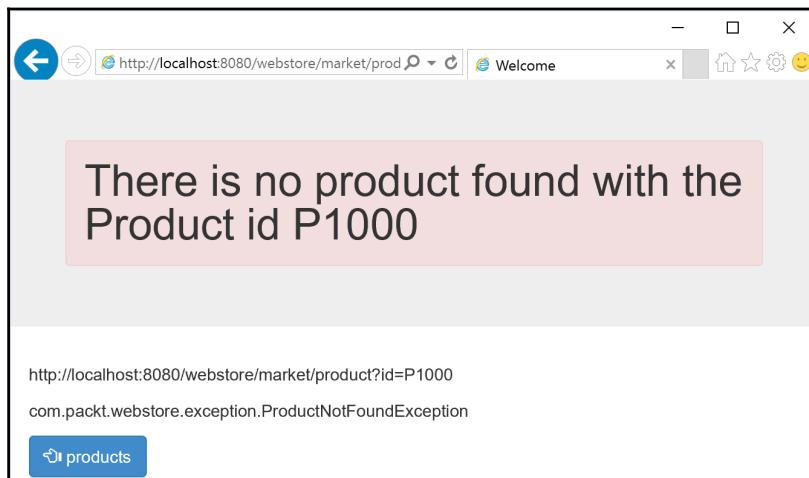
```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Welcome</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1 class="alert alert-danger"> There is no
product found with the Product id
${invalidProductId}</h1>
</div>
</div>
</section>

<section>
<div class="container">
<p>${url}</p>
<p>${exception}</p>
</div>

<div class="container">
<p>
<a href="
```

5. Now run our application and enter

<http://localhost:8080/webstore/market/product?id=P1000>. You will see an error page showing **There is no product found with the Product id P1000** as follows:



Product detail page showing a custom error page for the product ID P1000

What just happened?

We decided to show a custom-made error page instead of showing the raw exception in the case of a product not being found for a given product ID. So, in order to achieve that, in step 1 we just created a runtime exception called `ProductNotFoundException` to be thrown when the product is not found for the given product ID.

In step 2, we just modified the `getProductById` method of the `InMemoryProductRepository` class to check whether any products were found for the given product ID. If not, we simply throw the exception (`ProductNotFoundException`) we created in step 1.

In step 3, we added our exception handler method to handle `ProductNotFoundException` with the help of the `@ExceptionHandler` annotation. Within the `handleError` method, we just created a `ModelAndView` (`org.springframework.web.servlet.ModelAndView`) object and stored the requested invalid product ID, exception, and the requested URL, and returned it with the View name `productNotFound`:

```
@ExceptionHandler(ProductNotFoundException.class)
public ModelAndView handleError(HttpServletRequest req,
ProductNotFoundException exception) {
    ModelAndView mav = new ModelAndView();
    mav.addObject("invalidProductId", exception.getProductId());
```

```
mav.addObject("exception", exception);
mav.addObject("url", req.getRequestURL()+"?"+req.getQueryString());
mav.setViewName("productNotFound");
return mav;
}
```

Since we returned the `ModelAndView` object with the View name `productNotFound`, we must have a View file with the name `productNotFound`. That's why we created this View file (`productNotFound.jsp`) in step 4. `productNotFound.jsp` just contains a CSS-styled `<h1>` tag to show the error message and a link button to the product listing page.

So, whenever we request to show a product with an invalid ID such as `http://localhost:8080/webstore/product?id=P1000`, the `ProductController` class will throw the `ProductNotFoundException`, which will be handled by the `handleError` method and will show the custom error page (`productNotFound.jsp`).

Summary

In this chapter, you learned how `InternalResourceViewResolver` resolves Views, and you saw how to activate `RedirectView` from a Controller's method. You learned the important difference between `redirect` and `forward` and also found out about Flash attributes. After that, you saw how to host static resources files without going through Controller configuration. You discovered how to attach a static image file to the product details page. You learned how to upload files to the server. You saw how to configure `ContentNegotiatingViewResolver` to give alternate XML and JSON Views for product domain objects in our application. Finally, you made use of `HandlerExceptionResolver` to resolve exceptions.

In the next chapter, you will learn how to intercept regular web requests with the help of an interceptor. See you in the next chapter.

6

Internalize Your Store with Interceptor

In all our previous chapters, we have only seen how to map a request to the Controller's method; once the request reaches the Controller method, we execute some logic and return a logical View name, which can be used by the view resolver to resolve Views, but what if we want to execute some logic before actual request processing happens? Similarly, what if we want to execute some other instruction before dispatching the response?

Spring MVC interceptor intercepts the actual request and response. Interceptors are a special web programming technique, where we can execute a certain piece of logic before or after a web request is processed. In this chapter, we are going to learn more about interceptors. After finishing this chapter, you will know:

- How to configure an interceptor
- How to add internalization support
- Data auditing using an interceptor
- Conditional redirecting using an interceptor

Working with interceptors

As I have already mentioned, interceptors are used to intercept actual web requests before or after processing them. We can relate the concept of interceptors in Spring MVC to the filter concept in Servlet programming. In Spring MVC, interceptors are special classes that must implement the `org.springframework.web.servlet.HandlerInterceptor` interface. The `HandlerInterceptor` interface defines three important methods, as follows:

- `preHandle`: This method will get called just before the web request reaches the Controller for execution
- `postHandle`: This method will get called just after the Controller method execution
- `afterCompletion`: This method will get called after the completion of the entire web request cycle

Once we create our own interceptor, by implementing the `HandlerInterceptor` interface, we need to configure it in our web application context in order for it to take effect.

Time for action – configuring an interceptor

Every web request takes a certain amount of time to get processed by the server. In order to find out how much time it takes to process a web request, we need to calculate the time difference between the starting time and ending time of a web request process. We can achieve that by using the interceptor concept. Let's see how to configure our own interceptor in our project to log the execution time of every web request:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project itself.
2. You will see some tabs at the bottom of the `pom.xml` file; select the **Dependencies** tab and click on the **Add** button in the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `log4j`, **Artifact Id** as `log4j`, and **Version** as `1.2.17`, select **Scope** as `compile`, click the **OK** button, and save `pom.xml`.
4. Create a class named `ProcessingTimeLogInterceptor` under the `com.packt.webstore.interceptor` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.interceptor;  
  
import javax.servlet.http.HttpServletRequest;
```

```
import javax.servlet.http.HttpServletResponse;
import org.apache.log4j.Logger;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class ProcessingTimeLogInterceptor implements
HandlerInterceptor {
    private static final Logger LOGGER =
Logger.getLogger(ProcessingTimeLogInterceptor.class);
    public boolean preHandle(HttpServletRequest request,
HttpServletRequest response, Object handler) {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);

        return true;
    }
    public void postHandle(HttpServletRequest request,
HttpServletRequest response, Object handler, ModelAndView
modelAndView) {
        String queryString = request.getQueryString() == null ?
"" : "?" + request.getQueryString();
        String path = request.getRequestURL() + queryString;

        long startTime = (Long)
request.getAttribute("startTime");
        long endTime = System.currentTimeMillis();
        LOGGER.info(String.format("%s millisecond taken to
process the request %s.", (endTime - startTime), path));
    }
    public void afterCompletion(HttpServletRequest request,
HttpServletRequest response, Object handler, Exception
exceptionIfAny){
        // NO operation.
    }
}
```

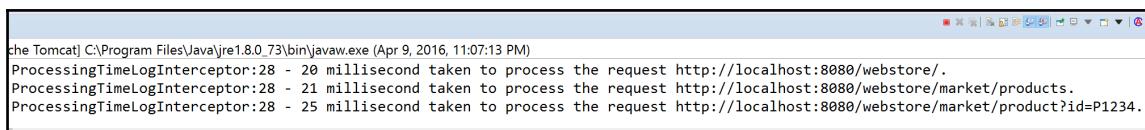
5. Now open your web application context configuration file `WebApplicationContextConfig.java`, add the following method to it, and save the file:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
    ProcessingTimeLogInterceptor());
}
```

6. Create a property file named `log4j.properties` under the `src/main/resources` directory, and add the following content. Then, save the file:

```
# Root logger option
log4j.rootLogger=INFO, file, stdout
# Direct log messages to a log file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File= C:\\webstore\\webstore-
performance.log
log4j.appender.file.MaxFileSize=1MB
log4j.appender.file.MaxBackupIndex=1
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %-5p %c{1}:%L - %m%n
```

7. Now run our application, enter the `http://localhost:8080/webstore/market/products` URL, and navigate to some pages; you should able to see the logging in the console as follows:



The screenshot shows a Java console window with the following log output:

```
che Tomcat| C:\Program Files\Java\jre1.8.0_73\bin\javaw.exe (Apr 9, 2016, 11:07:13 PM)
ProcessingTimeLogInterceptor:28 - 20 millisecond taken to process the request http://localhost:8080/webstore/.
ProcessingTimeLogInterceptor:28 - 21 millisecond taken to process the request http://localhost:8080/webstore/market/products.
ProcessingTimeLogInterceptor:28 - 25 millisecond taken to process the request http://localhost:8080/webstore/market/product?id=P1234.
```

Showing ProcessingTimeLogInterceptor logging messages in the console

8. Open the file `C:\\webstore\\webstore-performance.log`; you can see the same log message in the logging file as well.

What just happened?

Our intention was to record the execution time of every request coming to our web application, so we decided to record the execution times in a log file. In order to use a logger, we needed the `log4j` library, so we added the `log4j` library as a Maven dependency in step 3.

In step 4, we defined an interceptor class named `ProcessingTimeLogInterceptor` by implementing the `HandlerInterceptor` interface. As we already saw, there are three methods that need to be implemented. We will see each method one by one. The first method is `preHandle()`, which is called before the execution of the Controller method:

```
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
    long startTime = System.currentTimeMillis();
    request.setAttribute("startTime", startTime);
    return true;
}
```

In the previously shown `preHandle` method, we just set the current time value in the `request` object for later retrieval. Whenever a request comes to our web application, it first comes through this `preHandle` method and sets the current time in the `request` object before reaching the Controller. We are returning `true` from this method because we want the execution chain to proceed with the next interceptor or the Controller itself. Otherwise, `DispatcherServlet` assumes that this interceptor has already dealt with the response itself. So if we return `false` from the `preHandle` method, the request won't proceed to the Controller or the next interceptor.

The second method is `postHandle`, which will be called after the Controller method's execution:

```
public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {
    String queryString = request.getQueryString() == null ? "" : "?" +
        request.getQueryString();
    String path = request.getRequestURL() + queryString;

    long startTime = (Long) request.getAttribute("startTime");
    long endTime = System.currentTimeMillis();
    LOGGER.info(String.format("%s millisecond taken to process the request %s.", (endTime - startTime), path));
}
```

In the preceding method, we are simply logging the difference between the current time, which is considered to be the request processing finish time, and the start time, which we got from the `request` object. Our final method is `afterCompletion`, which is called after the View is rendered. We don't want to put any logic in this method, that's why I left the method empty.



If you don't want to implement all the methods from the `HandlerInterceptor` interface in your interceptor class, you could consider extending your interceptor from `org.springframework.web.servlet.handler.HandlerInterceptorAdapter`, which is a convenient class provided by Spring MVC as a default implementation of all the methods from the `HandlerInterceptor` interface.

After creating our `ProcessingTimeLogInterceptor`, we need to register our interceptor with Spring MVC, which is what we have done in step 5 through the `addInterceptors` overridden method of `WebMvcConfigurerAdapter`:

```
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(new ProcessingTimeLogInterceptor());  
}
```

In step 6, we have added a `log4j.properties` file in order to specify some logger-related configurations. You can see we have configured the log file location in `log4j.properties` as follows:

```
log4j.appenders.file.File= C:\\webstore\\webstore-performance.log
```

Finally, in step 7 we ran our application in order to record some performance logging, and we were able to see that the logger is just working fine via the console. You can open the log file to view the performance logs.

So we understood how to configure an interceptor and saw `ProcessingTimeLogInterceptor` in action. In the next exercise, we will see how to use some Spring-provided interceptors.

Pop quiz – interceptors

Consider the following interceptor:

```
public class SecurityInterceptor extends HandlerInterceptorAdapter{  
  
    @Override  
    public void afterCompletion(HttpServletRequest request,  
        HttpServletResponse response, Object handler, Exception ex) throws  
        Exception {  
        // just some code related to after completion  
    }  
}
```

Is this `SecurityInterceptor` class a valid interceptor?

1. It is not valid because it does not implement the `HandlerInterceptor` interface.
2. It is valid because it extends the `HandlerInterceptorAdapter` class.

Within the interceptor methods, what is the order of execution?

1. `preHandle`, `afterCompletion`, `postHandle`.
2. `preHandle`, `postHandle`, `afterCompletion`.

LocaleChangeInterceptor – internationalization

In the previous sections, we saw how to create an interceptor (`ProcessingTimeLogInterceptor`) and configure it in our web application context. Spring provides some pre-built interceptors that we can configure in our application context as and when needed. One such pre-built interceptor is `LocaleChangeInterceptor`, which allows us to change the current locale on every request and configures `LocaleResolver` to support internationalization.

Internationalization means adapting computer software to different languages and regional differences. For example, if you are developing a web application for a Dutch-based company, they may expect all the web page text to be displayed in the Dutch language, use the Euro for currency calculations, expect a space as a thousand separator when displaying numbers, and use a "," (comma) as a decimal point. On the other hand, when the same Dutch company wants to open a market in America, they expect the same web application to be adapted for American locales; for example, the web pages should be displayed in English, dollars should be used for currency calculations, numbers should be formatted with "," (comma) as the thousand separator, a "." (dot) acts as a decimal point, and so on.

The technique for designing a web application that can automatically adapt to different regions and countries without needing to be re-engineered is called internationalization, sometimes shortened to i18n (i-eighteen letters-n).

In Spring MVC, we can achieve internationalization through `LocaleChangeInterceptor` (`org.springframework.web.servlet.i18n.LocaleChangeInterceptor`). The `LocaleChangeInterceptor` allows us to change the current locale for every web request via a configurable request parameter.

In Chapter 4, *Working with Spring Tag Libraries*, we have seen how to externalize text messages in the add products page; now we are going to add internationalization support for the same add products page (`addProducts.jsp`), because in Spring MVC, prior to internationalizing a label, we must externalize that label first. Since we already externalized all the label messages in the add products page (`addProducts.jsp`), we can proceed to internationalize the add products page.

Time for action – adding internationalization

Technically we can add support for as many languages as we want with internationalization, but for demonstration purposes I am going to show you how to make our add products page with Dutch language support:

1. Create a file called `messages_nl.properties` under `/src/main/resources` in your project, add the following lines in it, and save the file:

```
addProduct.form.productId.label = Nieuw product ID  
addProduct.form.name.label = Naam  
addProduct.form.unitPrice.label = prijs per eenheid  
addProduct.form.manufacturer.label = fabrikant  
addProduct.form.category.label = categorie  
addProduct.form.unitsInStock.label = Aantal op voorraad  
addProduct.form.description.label = Beschrijving  
addProduct.form.condition.label = Product Staat  
addProduct.form.productImage.label = product afbeelding
```

2. Open our `addProduct.jsp` and add the following set of tags right after the `<body>` tag:

```
<section>
    <div class="pull-right" style="padding-right:50px">
        <a href="?language=en" >English</a>|<a href=?language=nl" >Dutch</a>
    </div>
</section>
```

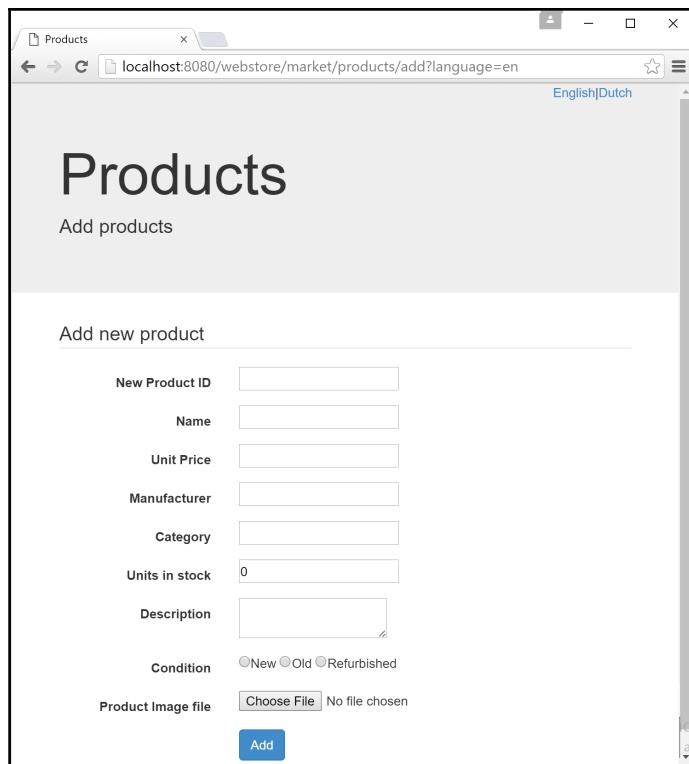
3. Now open our web application context configuration file, `WebApplicationContextConfig.java`, and add one more bean definition for the locale resolver as follows:

```
@Bean
public LocaleResolver localeResolver(){
    SessionLocaleResolver resolver = new
    SessionLocaleResolver();
    resolver.setDefaultLocale(new Locale("en"));
    return resolver;
}
```

4. Now change our `addInterceptors` method to configure one more interceptor in our `InterceptorRegistry` as follows:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
        ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
LocaleChangeInterceptor();
localeChangeInterceptor.setParamName("language");
    registry.addInterceptor(localeChangeInterceptor);
}
```

5. Now run our application and enter the `http://localhost:8080/webstore/market/products/add` URL; you will be able to see our regular add products page with two extra links in the top-right corner to choose the language:



The add products page showing internationalization support to choose languages

6. Now click on the **Dutch** link; you will see the product ID label transformed into the Dutch caption **Nieuw product ID**.
7. Since our configured `LocaleChangeInterceptor` will add a request parameter called `language` to the web request, we need to add this `language` request parameter to our whitelisting set in our `ProductController`. Open our `ProductController` and, within the `initialiseBinder` method, add the `language` request parameter to the whitelisting set as follows:

```
binder.setAllowedFields("productId",
```

```
"name",
"unitPrice",
"description",
"manufacturer",
"category",
"unitsInStock",
"condition",
"productImage",
"language");
```

What just happened?

In step 1, we just created a property file called `messages_nl.properties`. This file acts as a Dutch-based message source for all our externalized label messages in the `addProducts.jsp` file. In order to display the externalized label messages, we used the `<spring:message>` tag in our `addProducts.jsp` file.

But by default, the `<spring:message>` tag will read messages from the `messages.properties` file only; we need to make a provision for our end user to switch to the Dutch locale when they view the webpage, so that the label messages can come from the `messages_nl.properties` file. We provided such a provision through a locale-choosing link in `addProducts.jsp`, as mentioned in step 2:

```
<a href="?language=en" >English</a> | <a href="?language=nl" >Dutch</a>
```

In step 2, we created two links, one each to choose English or Dutch as the preferred locale. When the user clicks on one of these links, it will add a request parameter called `language` to the URL with the corresponding locale value. For example, when we click on the **English** link on the add products page at runtime, it will change the request URL to `http://localhost:8080/webstore/products/add?language=en`; similarly if you click on the **Dutch** link, it will change the request URL to `http://localhost:8080/webstore/products/add?language=nl`.

In step 3, we created a `SessionLocaleResolver` bean in our web application context as follows:

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver resolver = new SessionLocaleResolver();
    resolver.setDefaultLocale(new Locale("en"));
    return resolver;
}
```

`SessionLocaleResolver` is the one that sets the locale attribute in the user's session. One important property of `SessionLocaleResolver` is `defaultLocale`. We assigned `en` as the value for the default locale, which indicates that by default our page should use English as its default locale.

In step 4, we created a `LocaleChangeInterceptor` bean and configured it in the existing interceptor list:

```
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(new ProcessingTimeLogInterceptor());  
    LocaleChangeInterceptor localeChangeInterceptor = new  
    LocaleChangeInterceptor();  
    localeChangeInterceptor.setParamName("language");  
    registry.addInterceptor(localeChangeInterceptor);  
}
```

We assigned the string `language` as the value for the `paramName` property in `LocaleChangeInterceptor`. There is a reason for this because, if you notice, in step 2 when we created the locale choosing link in add products page (`addProduct.jsp`), we used the same parameter name as the request parameter within the `<a>` tag:

```
<a href="?language=en" >English</a>|<a href="?language=nl" >Dutch</a>
```

This way we can give a hint to `LocaleChangeInterceptor` to choose the correct user-preferred locale. So whatever parameter name you plan to use in your URL, use the same name as the value for the `paramName` property in `LocaleChangeInterceptor`. One more thing: whatever value you have given to the `language` request parameter in the link should match the translation message source file suffix. For example, in our case we have created a Dutch translation message source file, `messages_nl.properties`; here the suffix is `nl` and `messages.properties` without any suffix is considered the default for the `en` suffix. That's why in step 2 we have given `nl` and `en` as the values for the `language` parameters for Dutch and English, respectively:

```
<a href="?language=en" >English</a>|<a href="?language=nl" >Dutch</a>
```

So finally, when we run our application and enter the `http://localhost:8080/webstore/market/products/add` URL, you will see our regular add products page with two extra links in the top-right corner to choose the language.

Clicking on **Dutch** changes the request URL

to `http://localhost:8080/webstore/market/products/add?language=nl`, which brings up the `LocaleChangeInterceptor` and reads the Dutch-based label messages from `messages_nl.properties`.

Note that, if we do not give a language parameter in our URL, Spring will use the default message source file (`messages.properties`) for translation; if we give a language parameter, Spring will use that parameter value as the suffix to identify the correct language message source file (`messages_nl.properties`).

Have a go hero – fully internationalize the product details page

As I already mentioned just for demonstration purposes, I have internationalized a single web page (`addProducts.jsp`). I encourage you to internationalize the product detail web page (`product.jsp`) in our project. You can use the Google translate service (<https://translate.google.com/>) to find out the Dutch translation for labels. Along with this, try to add language support for one more language of your choice.

Mapped interceptors

Interceptors are a great way to add logic that needs to be executed during every request to our web application. The cool thing about interceptors is that, not only can we have a logic that executes before and/or after the request, we can even bypass or redirect the original web request itself. So far, we have seen a couple of examples of interceptors, such as performance logging and internationalization, but one problem with those examples is that there is no proper way to stop them from being run for specific requests.

For example, we might have a specific web request that does not need to run that additional logic in our interceptor. How can we tell Spring MVC to selectively include or exclude interceptors? Mapped interceptors to the rescue; using mapped interceptors we can selectively include or exclude interceptors, based on a mapped URL. Mapped interceptors use Ant-based path pattern matching to determine whether a web request path matches the given URL path pattern.

Consider a situation where you want to show the special-offer products page only to those users who have a valid promo code, but others who are trying to access the special-offer products page with an invalid promo code should be redirected to an error page.

Time for action – mapped intercepting offer page requests

Let's see how we can achieve this piece of functionality with the help of a mapped interceptor:

1. Create a class named `PromoCodeInterceptor` under the `com.packt.webstore.interceptor` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.interceptor;

import java.io.IOException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.handler
.HandlerInterceptorAdapter;

public class PromoCodeInterceptor extends
HandlerInterceptorAdapter {

    private String promoCode;
    private String errorRedirect;
    private String offerRedirect;

    public boolean preHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler) throws
    IOException{
        String givenPromoCode = request.getParameter("promo");

        if (promoCode.equals(givenPromoCode)) {
            response.sendRedirect(request.getContextPath() + "/"
            + offerRedirect);
        } else {
            response.sendRedirect(errorRedirect);
        }

        return false;
    }

    public void setPromoCode(String promoCode) {
        this.promoCode = promoCode;
    }

    public void setErrorRedirect(String errorRedirect) {
```

```
        this.errorRedirect = errorRedirect;
    }

    public void setOfferRedirect(String offerRedirect) {
        this.offerRedirect = offerRedirect;
    }
}
```

2. Create a bean definition for `PromoCodeInterceptor` in our web application context (`WebApplicationContextConfig.java`) as follows:

```
@Bean
public HandlerInterceptor promoCodeInterceptor() {
    PromoCodeInterceptor promoCodeInterceptor = new
    PromoCodeInterceptor();
    promoCodeInterceptor.setPromoCode("OFF3R");
    promoCodeInterceptor.setOfferRedirect("market/products");
    promoCodeInterceptor.setErrorRedirect("invalidPromoCode");
    return promoCodeInterceptor;
}
```

3. Now configure the `PromoCodeInterceptor` interceptor bean in our `InterceptorRegistry` as follows:

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(new
        ProcessingTimeLogInterceptor());
    LocaleChangeInterceptor localeChangeInterceptor = new
    LocaleChangeInterceptor();
    localeChangeInterceptor.setParamName("language");
    registry.addInterceptor(localeChangeInterceptor);
    registry.addInterceptor(promoCodeInterceptor())
        .addPathPatterns("/**/market/products/specialOffer");
}
```

4. Open our `ProductController` class and add one more request mapping method as follows:

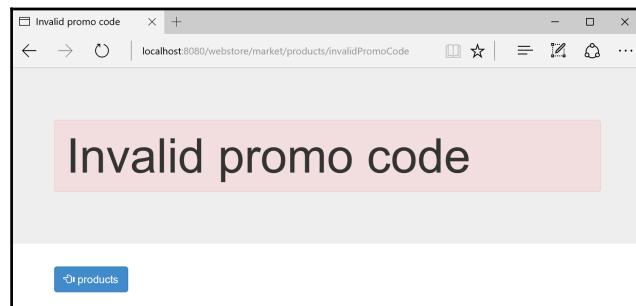
```
@RequestMapping("/products/invalidPromoCode")
public String invalidPromoCode() {
    return "invalidPromoCode";
}
```

5. Finally add one more JSP View file called `invalidPromoCode.jsp` under the directory `src/main/webapp/WEB-INF/views/`, add the following code snippets to it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags" %>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet" href="//netdna.bootstrapcdn.com/
bootstrap/3.0.0/css/bootstrap.min.css">
<title>Invalid promo code</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger"> Invalid promo
                code</h1>
            </div>
        </div>
    </section>

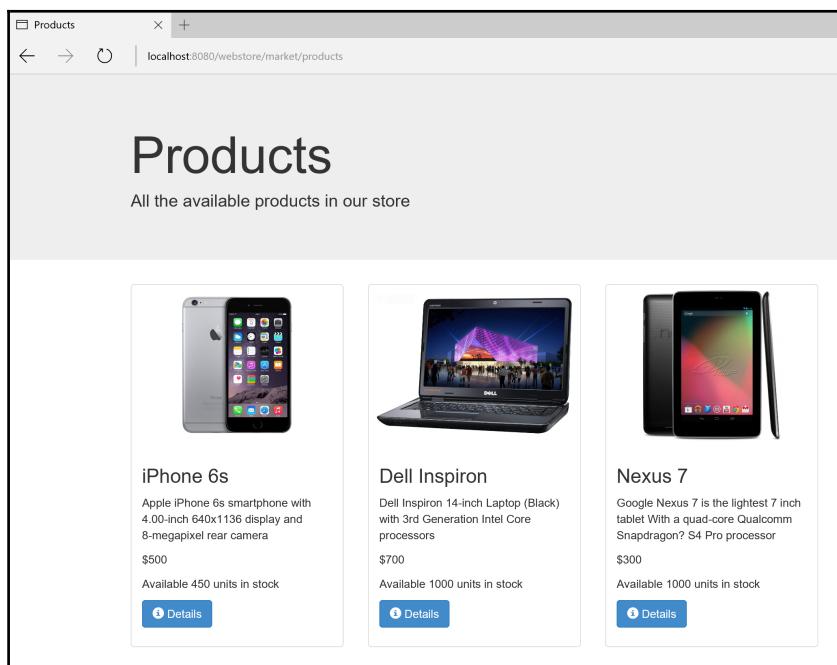
    <section>
        <div class="container">
            <p>
                <a href=<spring:url value="/market/products" />">
                    <span class="glyphicon-hand-left glyphicon">
                        </span> products
                </a>
            </p>
        </div>
    </section>
</body>
</html>
```

6. Now run our application and enter the `http://localhost:8080/webstore/market/products/specialOffer?promo=offer` URL; you should be able to see an error message page as follows:



Showing invalid promo code exception

7. Now enter the
`http://localhost:8080/webstore/market/products/specialOffer?promo=OFF3R` URL; you will land on a special-offer products page.



Showing special-offer page

What just happened?

The `PromoCodeInterceptor` class that we created in step 1 is very similar to `ProcessingTimeLogInterceptor`; the only difference is that we have extended `HandlerInterceptorAdapter` and only overridden the `preHandle` method:

```
public boolean preHandle(HttpServletRequest request, HttpServletResponse
    response, Object handler) throws IOException{
    String givenPromoCode = request.getParameter("promo");

    if (promoCode.equals(givenPromoCode)) {
        response.sendRedirect(request.getContextPath() + "/" +
offerRedirect);
    } else {
        response.sendRedirect(errorRedirect);
    }

    return false;
}
```

In the `preHandle` method, we are simply checking whether the request contains the correct promo code as the HTTP parameter. If so, we redirect the request to the configured special-offer page; otherwise, we redirect it to the configured error page.

The `PromoCodeInterceptor` class has three properties, `promoCode`, `errorRedirect`, and `offerRedirect`. The `promoCode` property is used to configure the valid promo code; in our case, we have assigned `OFF3R` as the valid promo code in step 2, so whoever is accessing the special-offer page should provide `OFF3R` as the promo code in their HTTP parameter in order to access the special-offer page.

The next two attributes, `errorRedirect` and `offerRedirect`, are used for redirection. `errorRedirect` indicates the redirect URL mapping in the case of an invalid promo code and `offerRedirect` indicates the redirect URL mapping for a successful promo code redirection.



I have not created a special-offer products page; for demonstration purposes I have reused the same regular products page as a special-offer products page, which is why we have assigned `market/products` as the value for `offerRedirect`; thus, in the case of a valid promo code we are simply redirecting to the regular products page. But if we create a special-offer products page, we can assign that page URL as the value for `offerRedirect`.

Okay, we have created the `PromoCodeInterceptor` bean, but we have to configure this interceptor with our Spring MVC runtime, which is what we have done in step 3 by adding the `PromoCodeInterceptor` bean to our `InterceptorRegistry` within the `addInterceptors` method:

```
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(new ProcessingTimeLogInterceptor());  
    LocaleChangeInterceptor localeChangeInterceptor = new  
    LocaleChangeInterceptor();  
    localeChangeInterceptor.setParamName("language");  
    registry.addInterceptor(localeChangeInterceptor);  
    registry.addInterceptor(promoCodeInterceptor())  
    .addPathPatterns("/**/market/products/specialOffer");  
}
```

If you notice, while adding `promoCodeInterceptor` to our `InterceptorRegistry`, we can specify URL patterns using the `addPathPatterns` method. This way we can specify the URL patterns to which the registered interceptor should apply. So our `promoCodeInterceptor` will get executed only for a request that ends with `market/specialOffer`.



While adding an interceptor, we can also specify the URL patterns that the registered interceptor should not apply to via the `excludePathPatterns` method.

In step 4, we have added one more request mapping method called `invalidPromoCode` to show an error page in the case of an invalid promo code. In step 5, we have added the corresponding error View file, called `invalidPromoCode.jsp`.

So in step 6, we purposely entered the

`http://localhost:8080/webstore/market/products/specialOffer?promo=offer` URL into our running application to demonstrate `PromoCodeInterceptor` in action; we saw the error page because the promo code we gave in the URL is `offer` (`?promo=offer`), which is wrong. In step 7, we gave the correct promo code in the `http://localhost:8080/webstore/market/products/specialOffer?promo=OFF3R` URL, so we should be able to see the configured special-offer products page.

Summary

In this chapter, we learned about the concept of interceptors and we learned how to configure interceptors in Spring MVC. We have seen how to do performance logging using interceptors. We also learned how to use Spring's `LocaleChangeInterceptor` to support internationalization. Later, we have seen how to do conditional redirecting using a mapped interceptor.

In the next chapter, I will introduce you to incorporating Spring Security into our web application. You will learn how to do authentication and authorization using the Spring Security framework.

7

Incorporating Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework for enterprise Java applications. The Spring Security framework is mainly used to ensure web application security such as authentication and authorization on application-level operations.

For web-layer security, Spring Security heavily leverages the existing Servlet filter architecture; it does not depend on any particular web technology. Spring Security mainly concerns `HttpRequest` and `HttpResponse` objects; it doesn't care about the source of the request and the response target. A request may originate from a web browser, web service, HTTP client, or JavaScript-based Ajax request. The only critical requirement for Spring Security is that it must be an `HttpRequest`, so that it can apply standard Servlet filters.

Spring Security provides various pre-built Servlet filters as part of the framework; the only requirement for us is to configure the appropriate filters in our web application in order to intercept the request and perform security checks. Spring Security is a vast topic, so we are not going to see all the capabilities of Spring Security; instead we are only going to see how to add basic authentication to our web pages.

Using Spring Security

In Chapter 4, *Working with Spring Tag Libraries*, we saw how to serve and process web forms; in that exercise we created a web page to add products. Anyone with access to the add products page could add new products to our web store. But in a typical web store, only administrators can add products. So how can we prevent other users from accessing the add products page? Spring Security comes to the rescue.

Time for action – authenticating users based on roles

We are going to restrict access to all our web pages using Spring Security. Only an authorised user or administrator with a valid username and password can access our web pages from a browser:

1. Open pom.xml; you can find pom.xml under the project root folder itself.
2. You should see some tabs at the bottom of pom.xml; select the **Dependencies** tab and click the **add** button in the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as org.springframework.security, **Artifact Id** as spring-security-config, **Version** as 4.1.1.RELEASE, select **Scope** as **compile**, and click the **OK** button.
4. Similarly, add one more dependency **Group Id** as org.springframework.security, **Artifact Id** as spring-security-web, **Version** as 4.1.1.RELEASE, select **Scope** as **compile**, and click the **OK** button. And most importantly, save pom.xml.
5. Now create one more controller class called `LoginController` under the com.packt.webstore.controller package in the src/main/java source folder. Add the following code to it:

```
package com.packt.webstore.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class LoginController {

    @RequestMapping(value = "/login", method =
    RequestMethod.GET)
    public String login() {
        return "login";
    }
}
```

6. Add one more JSP view file called `login.jsp` under the src/main/webapp/WEB-INF/views/ directory, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
```

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Products</title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1>Welcome to Web Store!</h1>
<p>The one and only amazing web store</p>
</div>
</div>
</section>
<div class="container">
<div class="row">
<div class="col-md-4 col-md-offset-4">
<div class="panel panel-default">
<div class="panel-heading">
<h3 class="panel-title">Please sign in</h3>
</div>
<div class="panel-body">
<c:url var="loginUrl" value="/login" />
<form action="${loginUrl}" method="post"
class="form-horizontal">

<c:if test="${param.error != null}">
<div class="alert alert-danger">
<p>Invalid username and password.</p>
</div>
</c:if>

<c:if test="${param.logout != null}">
<div class="alert alert-success">
<p>You have been logged out
successfully.</p>
</div>
</c:if>

<c:if test="${param.accessDenied !=
```

```
        null}">
            <div class="alert alert-danger">
                <p>Access Denied: You are not
                    authorised! </p>
            </div>
        </c:if>

        <div class="input-group input-sm">
            <label class="input-group-addon"
                for="username"><i
                    class="fa fa-user"></i></label>
            <input type="text" class="form-control"
                id="userId" name="userId"
                placeholder="Enter Username"
                required>
        </div>
        <div class="input-group input-sm">
            <label class="input-group-addon"
                for="password"><i
                    class="fa fa-lock"></i></label>
            <input type="password"
                class="form-control" id="password"
                name="password" placeholder="Enter
                Password" required>
        </div>

        <div class="form-actions">
            <input type="submit"
                class="btn btn-block btn-primary
                btn-default" value="Log in">
        </div>
    </div>
</div>
</body>
```

7. Now create one more configuration file called `SecurityConfig.java` under the `com.packt.webstore.config` package in the `src/main/java` source folder, add the following content into it, and save it:

```
package com.packt.webstore.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
configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {

    auth.inMemoryAuthentication().withUser("john").password("pa55word")
    .roles("USER");

    auth.inMemoryAuthentication().withUser("admin").password("root123")
    .roles("USER", "ADMIN");
}

@Override
protected void configure(HttpSecurity httpSecurity)
throws Exception {

    httpSecurity.formLogin().loginPage("/login")
        .usernameParameter("userId")
        .passwordParameter("password");

    httpSecurity.formLogin().defaultSuccessUrl
    ("/market/products/add")
        .failureUrl("/login?error");

    httpSecurity.logout().logoutSuccessUrl("/login?
    logout");

    httpSecurity.exceptionHandling().accessDeniedPage
    ("/login?accessDenied");

    httpSecurity.authorizeRequests()
        .antMatchers("/").permitAll()
        .antMatchers("//**/add").access("hasRole('ADMIN')");
}
```

```
    .antMatchers("/**/market/**").access  
    ("hasRole('USER')");  
  
    httpSecurity.csrf().disable();  
}  
}
```

8. Create one more initializer class called `SecurityWebApplicationInitializer` under the `com.packt.webstore.config` package in the `src/main/java` source folder, add the following content into it, and save it:

```
package com.packt.webstore.config;  
  
import org.springframework.security.web.context  
.AbstractSecurityWebApplicationInitializer;  
  
public class SecurityWebApplicationInitializer extends  
AbstractSecurityWebApplicationInitializer {  
  
}
```

9. Now open your `addProduct.jsp` file and add the following code after the `English|Dutch` anchor tag:

```
<a href="
```

10. Now run your application and enter the URL,
`http://localhost:8080/webstore/`; you will see the welcome screen.
11. Now try to access the products page by entering the URL,
`http://localhost:8080/webstore/market/products`; a login page will be shown. Enter an arbitrary username and password; you will see an **Invalid username and password** error:

A screenshot of a web-based login form titled "Please sign in". The form includes two input fields: "Enter Username" and "Enter Password", both with placeholder text. Below the inputs is a large blue "Log in" button. Above the "Log in" button, a red error message box displays the text "Invalid username and password.".

Login page showing error messages for invalid credentials

12. Now enter the username `john` and the password `pa55word`, and press the **Log in** button; you should be able to see the regular products page.
13. Now try to access the add products page by entering the URL, `http://localhost:8080/webstore/market/products/add`; again the login page will be shown with the error message **Access Denied: You are not authorised!**

A screenshot of a web-based login form titled "Please sign in". The form includes two input fields: "Enter Username" and "Enter Password", both with placeholder text. Below the inputs is a large blue "Log in" button. Above the "Log in" button, a red error message box displays the text "Access Denied: You are not authorised!".

Login page showing error messages for unauthorised users

14. Now enter the username `admin` and the password `root123`, and press the **Log in** button; you should be able to see the regular add products page with a **logout** button in the top-right corner.

What just happened?

As usual, in order to use Spring Security in our project, we need some Spring Security-related JARs; from steps 1 to 4 we just added those JARs as Maven dependencies.

In step 5, we created one more controller called `LoginController` to handle all login-related web requests. It simply contains a single request mapping method to handle login-login failure and log out requests. Since the request mapping method returns a view named `login`, we need to create a view file called `login.jsp`, which is what we did in step 6.

`login.jsp` contains many tags with the `bootstrap` style class applied to enhance the look and feel of the login form; we don't need to concentrate on those tags. But some important tags are used to understand the flow; the first one is the `<c:if>` tag:

```
<c:if test="${param.error != null}">
    <div class="alert alert-danger">
        <p>Invalid username and password.</p>
    </div>
</c:if>
```

`<c:if>` is a special JSTL tag to check a condition; it is more like an `if...else` condition that we use in our programming language. Using this `<c:if>` tag we are simply checking whether the page request parameter contains a variable called `error`; if the request parameter contains a variable called `error` we simply show an error message, **Invalid username and password**, within the `<p>` tag using the `<spring:message>` tag.

Similarly, we are also checking whether the request parameter contains variables called `logout` and `accessDenied`; if so we show the corresponding message, also within the `<p>` tag:

```
<c:if test="${param.logout != null}">
    <div class="alert alert-success">
        <p>You have been logged out successfully.</p>
    </div>
</c:if>

<c:if test="${param.accessDenied != null}">
    <div class="alert alert-danger">
        <p>Access Denied: You are not authorised! </p>
    </div>
</c:if>
```

```
</div>
</c:if>
```

So now we facilitated all possible login-related messages being shown in the login page. The other important tag in `login.jsp` is the `form` tag, which represents the login form. Notice the action attribute of the `form` tag:

```
<c:url var="loginUrl" value="/login" />
<form action="${loginUrl}" method="post" class="form-horizontal">
```

We are simply posting our login form values, such as username and password, to the Spring Security authentication handler URL, which is stored in the variable called `${loginUrl}`. Here the special JSTL tag `<c:url>` is used to encode the URL.

Okay, now we have created a controller (`LoginController`) to dispatch the login page. But we need to tell Spring to present this login page to users if they try to access a page without logging in. How can we enforce that? This is where

the `WebSecurityConfigurerAdapter` class comes in; by extending

`WebSecurityConfigurerAdapter`, we can configure the `HttpSecurity` object for various security-related settings in our web application. So in step 7, we are simply creating a class called `SecurityConfig` to configure the security-related aspects of our web application.

One of the important methods in the `SecurityConfig` class is `configureGlobalSecurity`; under this method we are simply configuring `AuthenticationManagerBuilder` to create two users, `john` and `admin`, with a specified password and roles:

```
@Autowired
public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {
    auth.inMemoryAuthentication().withUser("john")
        .password("pa55word")
        .roles("USER");

    auth.inMemoryAuthentication().withUser("admin")
        .password("root123")
        .roles("USER", "ADMIN");
}
```

The next important method is `configure`; within this method we are doing some authentication-and authorization-related configuration and we will see these one by one. The first configuration tells Spring MVC that it should redirect the users to the login page if authentication is required; here the `loginpage` attribute denotes to which URL it should forward the request to get the login form.

Remember this request path should be the same as the request mapping of the `login()` method of `LoginController`. We are also setting the user name parameter and password parameter name in this configuration:

```
httpSecurity.formLogin().loginPage("/login")
    .usernameParameter("userId")
    .passwordParameter("password");
```

With this configuration, while posting the username and password to the Spring Security authentication handler through the login page, Spring expects those values to be bound under the variable name `userId` and `password` respectively; that's why, if you notice, the input tags for username and password carry the name attributes `userId` and `password` in step 6:

```
<input type="text" class="form-control" id="userId" name="userId"
placeholder="Enter Username" required>
<input type="password" class="form-control" id="password" name="password"
placeholder="Enter Password" required>
```

Similarly, Spring handles the log out operation under the `/logout` URL; that's why in step 9 we formed the logout link on the add products page, as follows:

```
<a href=<c:url value="/logout" />>Logout</a>
```

Next, we are just configuring the default success URL, which denotes the default landing page after a successful login; similarly the authentication failure URL indicates to which URL the request needs to be forwarded in the case of login failure:

```
httpSecurity.formLogin().defaultSuccessUrl("/market/products/add")
    .failureUrl("/login?error");
```

Notice we are setting the request parameter to `error` in the failure URL; thus when the login page is rendered, it will show the error message **Invalid username and password** in the case of login failure. Similarly, we can also configure the logout success URL, which denotes where the request needs to be forwarded to after a logout:

```
httpSecurity.logout().logoutSuccessUrl("/login?logout");
```

You can also see that we are setting the request parameter as `logout` for the logout success URL to match the condition checking that we performed in `login.jsp`:

```
<c:if test="${param.logout != null}">
    <div class="alert alert-success">
        <p>You have been logged out successfully.</p>
    </div>
</c:if>
```

And similarly, we also configured the redirection URL for the access denied page in the case of authorization failure as follows:

```
httpSecurity.exceptionHandling().accessDeniedPage("/login?accessDenied");
```

The request parameter we are setting here for the access denied page URL should match the parameter name that we are checking in the `login.jsp` page:

```
<c:if test="#{param.accessDenied != null}">
    <div class="alert alert-danger">
        <p>Access Denied: You are not authorised! </p>
    </div>
</c:if>
```

So now we have configured almost all redirection URLs that specify to which page the user should get redirected in the case of login success, login failure, logout success, and access denial. The next configuration is the more important as it defines which user should get access to which page:

```
httpSecurity.authorizeRequests()
    .antMatchers("/").permitAll()
    .antMatchers("/**/add").access("hasRole('ADMIN')")
    .antMatchers("/**/market/**").access("hasRole('USER')");
```

The preceding configuration defines three important authorization rules for our web application, in terms of Ant pattern matchers. The first one allows the request URLs that end with `/`, even if the request doesn't carry any roles. The next rule allows all request URLs that end with `/add`, if the request has the role `ADMIN`. The third rule allows all request URLs that have the path `/market/` if they have the role `USER`.

Okay we defined all security-related configurations in the security context file, but Spring should know about this configuration file and will have to read this configuration file before booting the application. Only then can it create and manage those security-related configurations. How can we instruct Spring to pick up this file? The answer is `AbstractSecurityWebApplicationInitializer`; by extending the `AbstractSecurityWebApplicationInitializer` class, we can instruct Spring MVC to pick up our `SecurityConfig` class during bootup. So in step 8, that's what we are doing.

After finishing all the steps, if you run your application and enter the `http://localhost:8080/webstore/` URL, you are able to see the welcome screen. Now try to access the products page by entering the `http://localhost:8080/webstore/market/products` URL; a login page will be shown. Enter an arbitrary username and password; you will see an **Invalid username and password** error.

Now enter the username `john` and the password `pa55word`, and press the **Log in** button; you should be able to see the regular products page. Now try to access the add products page by entering the `http://localhost:8080/webstore/market/products/add` URL; again a login page will be shown with an error message, **Access Denied: You are not authorised!** Now enter the username `admin` and the password `root123`, and press the **Log in** button; you should be able to see the regular add products page with a **logout** button in the top-right corner.

Pop quiz – Spring Security

Which URL is the Spring Security default authentication handler listening on for the username and password?

1. /login
2. /login?userName&password
3. /handler/login

What is the default logout handler URL for Spring Security?

1. /logout
2. /login?logout
3. /login?logout=true

Have a go hero – play with Spring Security

Just for demonstration purposes, I added the **logout** link only on the add products page; why don't you add the **logout** link to every page? Also, try adding new users and assigning new roles to them.

Summary

Web application security is a complex multidimensional problem that needs to be addressed from multiple perspectives such as coding, designing, operations, systems, networking, and policy. What we have seen in this chapter is just a glimpse into Spring Security and how it can be used for authentication and authorization. In the next chapter we will further explore validators.

8

Validate Your Products with a Validator

One of the most commonly expected behaviors of any web application is that it should validate user data. Every time a user submits data into our web application, it needs to be validated. This is to prevent security attacks, wrong data, or simple user errors. We don't have control over what the user may type while submitting data into our web application. For example, they may type some text instead of a date, they may forget to fill a mandatory field, or suppose we used 12-character lengths for a field in the database and the user entered 15-character length data, then the data cannot be saved in the database. Similarly, there are lots of ways a user can feed incorrect data into our web application. If we accept those values as valid, then it will create errors and bugs when we process such inputs. This chapter will explain the basics of setting up validation with Spring MVC.

After finishing this chapter, you will have a clear idea about the following:

- JSR-303 Bean Validation
- Custom validation
- Spring validation

Bean Validation

Java Bean Validation (JSR-303) is a Java specification that allows us to express validation constraints on objects via annotations. It allows the APIs to validate and report violations. **Hibernate Validator** is the reference implementation of the Bean Validation specification. We are going to use Hibernate Validator for validation.

You can see the available Bean Validation annotations at the following site:

<https://docs.oracle.com/javaee/7/tutorial/bean-validation001.htm>.

Time for action – adding Bean Validation support

In this section, we will see how to validate a form submission in a Spring MVC application. In our project, we have the **Add new product** form already. Now let's add some validation to that form:

1. Open `pom.xml`, which you will find under the root directory of the actual project.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `org.hibernate`, **Artifact Id** as `hibernate-validator`, **Version** as `5.2.4.Final`, select **Scope** as `compile`, click the **OK** button, and save `pom.xml`.
4. Open our `Product` domain class and add the `@Pattern` (`javax.validation.constraints.Pattern`) annotation at the top of the `productId` field, as follows:

```
@Pattern(regexp="P[1-9]+", message=""  
{Pattern.Product.productId.validation})  
private String productId;
```

5. Similarly, add the `@Size`, `@Min`, `@Digits`, and `@NotNull` (`javax.validation.constraints.*`) annotations on the top of the `name` and `unitPrice` fields respectively, as follows:

```
@Size(min=4, max=50, message=""  
{Size.Product.name.validation})  
private String name;  
@Min(value=0, message=""  
{Min.Product.unitPrice.validation})@Digits(integer=8,  
fraction=2, message=""  
{Digits.Product.unitPrice.validation})@NotNull(message=" "  
{NotNull.Product.unitPrice.validation})  
private BigDecimal unitPrice;
```

6. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entries into it:

```
Pattern.Product.productId.validation = Invalid product ID.  
It should start with character P followed by number.
```

Size.Product.name.validation = Invalid product name. It should be minimum 4 characters to maximum 50 characters long.

Min.Product.unitPrice.validation = Unit price is Invalid. It cannot have negative values.

Digits.Product.unitPrice.validation = Unit price is Invalid. It can have maximum of 2 digit fraction and 8 digit integer.

NotNull.Product.unitPrice.validation = Unit price is Invalid. It cannot be empty.

Min.Product.unitPrice.validation = Unit price is Invalid. It cannot be negative value.

7. Open our ProductController class and change the processAddNewProductForm request mapping method by adding a @Valid (`javax.validation.Valid`) annotation in front of the newProduct parameter. After finishing that, your processAddNewProductForm method signature should look as follows:

```
public String  
processAddNewProductForm(@ModelAttribute("newProduct")  
@Valid Product newProduct, BindingResult result,  
HttpServletRequest request)
```

8. Now, within the body of the processAddNewProductForm method, add the following condition as the first statement:

```
if(result.hasErrors()) {  
    return "addProduct";  
}
```

9. Open our addProduct.jsp from src/main/webapp/WEB-INF/views/ in your project and add the <form:errors> tag for the productId, name, and unitPrice input elements. For example, the product ID input tag will have the <form:errors> tag beside it, as follows:

```
<form:input id="productId" path="productId" type="text"  
class="form:input-large"/>  
<form:errors path="productId" cssClass="text-danger"/>
```

Remember, the path attribute value should be the same as the corresponding input tag.

10. Now, add one global <form:errors> tag within the <form:form> tag, as follows:

```
<form:errors path="*" cssClass="alert alert-danger" element="div"/>
```

11. Add bean configuration for LocalValidatorFactoryBean in our web application context configuration file WebApplicationContextConfig.java, as follows:

```
@Bean(name = "validator")
public LocalValidatorFactoryBean validator() {
    LocalValidatorFactoryBean bean = new
    LocalValidatorFactoryBean();
    bean.setValidationMessageSource(messageSource());
    return bean;
}
```

12. Finally, override the getValidator method in WebApplicationContextConfig to configure our validator bean as the default validator, as follows:

```
@Override
public Validator getValidator(){
    return validator();
}
```



Note that here the return type
is org.springframework.validation.Validator

13. Now run our application and enter the URL

<http://localhost:8080/webstore/market/products/add>. We will see a webpage showing a web form to add product info. Without entering any values in the form, simply click on the **Add** button. You will see validation messages at the top of the form, as shown in the following screenshot:

Add new product

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.
Invalid product ID. It should start with character P followed by number.
Unit price is Invalid. It cannot be empty.

New Product ID	<input type="text"/>	Invalid product ID. It should start with character P followed by number.
Name	<input type="text"/>	Invalid product name. It should be minimum 4 characters to maximum 50 characters long.
Unit Price	<input type="text"/>	Unit price is Invalid. It cannot be empty.
Manufacturer	<input type="text"/>	
Category	<input type="text"/>	
Units in stock	<input type="text" value="0"/>	
Description	<input type="text"/>	
Condition	<input checked="" type="radio"/> New <input type="radio"/> Old <input type="radio"/> Refurbished	
Product Image file	<input type="button" value="Choose File"/> No file chosen	
<input type="button" value="Add"/>		

The Add new product web form showing validation message.

What just happened?

Since we decided to use the Bean Validation (JSR-303) specification, we needed an implementation of the Bean Validation specification, and we decided to use a Hibernate Validator implementation in our project; thus we need to add that JAR to our project as a dependency. That's what we did in steps 1 through 3.

From steps 4 and 5, we added some `javax.validation.constraints` annotations such as `@Pattern`, `@Size`, `@Min`, `@Digits`, and `@NotNull` on our domain class (`Product.java`) fields. Using these annotations, we can define validation constraints on fields. There are more validation constraint annotations available under the `javax.validation.constraints` package. Just for demonstration purposes, we have used a couple of annotations; you can check out the Bean Validation documentation for all available lists of constraints.

For example, take the `@Pattern` annotation on top of the `productId` field; it will check whether the given value for the field matches the regular expression that is specified in the `regexp` attribute of the `@Pattern` annotation. In our example, we just enforce that the value given for the `productId` field should start with the character `P` and should be followed by digits:

```
@Pattern(regexp="P[1-9]+", message=" {Pattern.Product.productId.validation}")
private String productId;
```

The `message` attribute of every validation annotation is just acting as a key to the actual message from the message source file (`messages.properties`). In our case, we specified `Pattern.Product.productId.validation` as the key, so we need to define the actual validation message in the message source file. That's why we added some message entries in step 6. If you noticed the corresponding value for the key `Pattern.Product.productId.validation` in the `messages.properties` file, you will notice the following value:

`Pattern.Product.productId.validation = Invalid product ID. It should start with character P followed by number.`



You can even add localized error messages in the corresponding message source file if you want. For example, if you want to show error messages in Dutch, simply add error message entries in the `messages_nl.properties` file as well. During validation, this message source will be automatically picked up by Spring MVC based on the chosen locale.

We defined the validation constraints in our domain object, and also defined the validation error messages in our message source file. What else do we need to do? We need to tell our controller to validate the form submission request. We did that through steps 7 and 8 in the `processAddNewProductForm` method:

```
@RequestMapping(value = "/products/add", method = RequestMethod.POST)
public String processAddNewProductForm(@ModelAttribute("newProduct") @Valid
Product newProduct, BindingResult result, HttpServletRequest request) {
    if(result.hasErrors()) {
        return "addProduct";
    }

    String[] suppressedFields = result.getSuppressedFields();
    if (suppressedFields.length > 0) {
        throw new RuntimeException("Attempting to bind disallowed fields: " +
StringUtils.arrayToCommaDelimitedString(suppressedFields));
    }
}
```

```
MultipartFile productImage = newProduct.getProductImage();
String rootDirectory =
request.getSession().getServletContext().getRealPath("/");
if (productImage!=null && !productImage.isEmpty()) {
    try {
        productImage.transferTo(new
File(rootDirectory+"resources\\images"+ newProduct.getProductId() +
".png"));
    } catch (Exception e) {
        throw new RuntimeException("Product Image saving failed", e);
    }
}

productService.addProduct(newProduct);
return "redirect:/market/products";
}
```

We first annotated our method parameter `newProduct` with the `@Valid` (`javax.validation.Valid`) annotation. By doing so, Spring MVC will use the Bean Validation framework to validate the `newProduct` object. As you already know, the `newProduct` object is our form-backed bean. After validating the incoming form bean (`newProduct`), Spring MVC will store the results in the `result` object; this is again another method parameter of our `processAddNewProductForm` method.

In step 8 we simply checked whether the `result` object contains any errors. If so, we redirected to the same **Add new product** page; otherwise we proceeded to add `productToBeAdded` to our repository.

So far everything is fine. At first we defined the constraints on our domain object and defined the error messages in the message source file (`messages.properties`). Later we validated and checked the validation result in the controller's form processing method (`processAddNewProductForm`), but we haven't said how to show the error messages in the view file. Here comes Spring MVC's special `<form:errors>` tag to the rescue.

We added this tag for the `productId`, `name`, and `unitPrice` input elements in step 9. If any of the input fields failed during validation, the corresponding error message will be picked up by this `<form:errors>` tag:

```
<form:errors path="productId" cssClass="text-danger"/>
```

The `path` attribute is used to identify the field in the form bean to look for errors, and the `cssClass` attribute will be used to style the error message. I have used Bootstrap's style class `text-danger`, but you can use any valid CSS- style class that you prefer to apply on the error message.

Similarly, in step 10, we have added a global `<form:errors>` tag to show all error messages as a consolidated view at the top of the form:

```
<form:errors path="*" cssClass="alert alert-danger" element="div"/>
```

Here we have used the “`*`” symbol for the `path` attribute, which means we want to show all the errors and `element` attributes, just indicating which type of element Spring MVC should use to list all the errors.

So far, we have done all the coding-related stuff that is needed to enable validation, but we have to do one final configuration in our web application context to enable validation; that is we need to introduce the Bean Validation framework to our Spring MVC. In steps 11 and 12 we did just that. We have defined a bean for `LocalValidatorFactoryBean` (`org.springframework.validation.beanvalidation.LocalValidatorFactoryBean`). This `LocalValidatorFactoryBean` will initiate the Hibernate Validator during the booting of our application:

```
@Bean(name = "validator")
public LocalValidatorFactoryBean validator() {
    LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
    bean.setValidationMessageSource(messageSource());
    return bean;
}
```

The `setValidationMessageSource` method of `LocalValidatorFactoryBean` indicates which message source bean it should look to for error messages. Since, in Chapter 6, *Internalize Your Store with Interceptor*, we already configured message sources in our web application context, we just use that bean, which is why we assigned the value `messageSource()` as the value for the `setValidationMessageSource` method. You will see a bean definition under the method `messageSource()` already in our web application context.

And finally, we introduced our `validator` bean to Spring MVC by overriding the `getValidator` method:

```
@Override
public Validator getValidator() {
    return validator();
}
```

That is all we did to enable validation; now, if you run our application and bring the **Add new product** page using the <http://localhost:8080/webstore/market/products/add> URL, you can see the empty form ready to submit. If you submit that form without filling in any information, you

will see error messages in red.

Have a go hero – adding more validation in the Add new product page

I have just added validation for the first three fields in the `Product` domain class; you can extend the validation for the remaining fields. And try to add localisedlocalized error messages for the validation you are defining.

Here are some hints you can try out:

- Add validation to show a validation message in case the `category` field is empty
- Try to add validation to the `unitsInStock` field to validate that the minimum number of **Units in stock** allowed is zero

Custom validation with JSR-303/Bean Validation

In the previous sections, we learned how to use standard JSR-303 Bean Validation annotations to validate fields of our domain object. This works great for simple validations, but sometimes we need to validate some custom rules that aren't available in the standard annotations. For example, what if we need to validate a newly added product's ID, which should not be the same as any existing product ID? To accomplish this type of thing, we can use custom validation annotations.

Time for action – adding Bean Validation support

In this exercise we are going to learn how to create custom validation annotations and how to use them. Let's add a custom product ID validation to our **Add new product** page to validate duplicate product IDs:.

1. Create an annotation interface called `ProductId` (`ProductId.java`) under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code snippet:

```
package com.packt.webstore.validator;  
  
import static java.lang.annotation.ElementType
```

```
.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.validation.Constraint;
import javax.validation.Payload;

@Target( { METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = ProductIdValidator.class)
@Documented
public @interface ProductId {
    String message() default "
{com.packt.webstore.validator.ProductId.message}";

    Class<?>[] groups() default {};
    public abstract Class<? extends Payload>[] payload()
default {};
}
```

2. Now, create a class called `ProductIdValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
import org.springframework.beans.factory.annotation.Autowired;
import com.packt.webstore.domain.Product;
import com.packt.webstore.exception.ProductNotFoundException;
import com.packt.webstore.service.ProductService;

public class ProductIdValidator implements
ConstraintValidator<ProductId, String>{
    @Autowired
    private ProductService productService;

    public void initialize(ProductId constraintAnnotation) {
        // intentionally left blank; this is the place to
        initialize the constraint annotation for any sensible default
        values.
    }
}
```

```
public boolean isValid(String value,
ConstraintValidatorContext context) {
    Product product;
    try {
        product = productService.getProductById(value);
    } catch (ProductNotFoundException e) {
        return true;
    }
    if(product!= null) {
        return false;
    }
    return true;
}
```

3. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entry into it:

```
com.packt.webstore.validator.ProductId.message = A product
already exists with this product id.
```

4. Finally, open our `Product` (`Product.java`) domain class and annotate our `productId` field with our newly created `ProductId` annotation, as follows:

```
@Pattern(regexp="P[1-9]+", message="
{Pattern.Product.productId.validation}")
@ProductId
private String productId;
```

5. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. You should be able to see a webpage showing a web form to add product info. Enter all the values in the form, particularly filling in the product ID field with the value `P1234`, and simply click the **Add** button.

6. You will see validation messages at the top of the form, as shown in the following screenshot:

The screenshot shows a web form titled "Add new product". The form fields are as follows:

- New Product ID:** P1234 (Validation message: A product already exists with this product id.)
- Name:** Sony Head Phone
- Unit Price:** 250
- Manufacturer:** (empty)
- Category:** (empty)
- Units in stock:** 0
- Description:** (empty)
- Condition:** New Old Refurbished
- Product Image file:** Choose File (No file chosen)

At the bottom is a blue "Add" button.

The Add new product web form showing custom validation.

What just happened?

In step 1 we just created our custom validation annotation called `ProductId`. Every custom validation annotation we create needs to be annotated with the `@Constraint` (`javax.validation.Constraint`) annotation. The `@Constraint` annotation has an important property called `validatedBy`, which indicates the class that is performing the actual validation. In our case, we have given a value `ProductIdValidator.class` for the `validatedBy` property. So our `ProductId` validation annotation would expect a class called `ProductIdValidator`. That's why in step 2 we have created the class `ProductIdValidator` by implementing the interface `ConstraintValidator` (`javax.validation.ConstraintValidator`).

We annotated the `ProductIdValidator` class with the `@Component` (`org.springframework.stereotype.Component`) annotation; the `@Component` annotation is another stereotype annotation that is available in Spring. It is very similar to the `@Repository` or `@Service` annotations; during the booting of our application, Spring would create and maintain an object for the `ProductIdValidator` class. So `ProductIdValidator` will become a managed bean in our web application context, which is the reason we were able to autowire the `productService` bean in `ProductIdValidator`.

Next we autowired the `ProductService` object in the `ProductIdValidator` class. Why? Because, inside the `isValid` method of the `ProductIdValidator` class, we have used the `productService` to check whether any product with the given ID exists:

```
public boolean isValid(String value, ConstraintValidatorContext context) {  
    Product product;  
    try {  
        product = productService.getProductById(value);  
    } catch (ProductNotFoundException e) {  
        return true;  
    }  
    if (product != null) {  
        return false;  
    }  
    return true;  
}
```

If any product exists with the given product ID, we are invalidating the validation by returning `false`, otherwise we are passing the validation by returning `true`.

In step 3 we just added our default error message for our custom validation annotation in the message source file (`messages.properties`). If you observed carefully, the key (`com.packt.webstore.validator.ProductId.message`) we have used in our message source file is the same as the default key that we have defined in the `ProductId` (`ProductId.java`) validation annotation:

```
String message() default  
    "{com.packt.webstore.validator.ProductId.message}";
```

So, finally, in step 4 we have used our newly created `ProductId` validation annotation in our domain class (`Product.java`). So it will act similarly to any other JSR-303 Bean Validation annotation.

Thus, you were able to see the error message on the screen when you entered the existing product ID as the product ID for the newly added product.

Have a go hero – adding custom validation to a category

Create a custom validation annotation called `@Category`, which will allow only some of the predefined configured categories to be entered. Consider the following things while implementing your custom annotation:

- Create an annotation interface called `CategoryValidator` under the package `com.packt.webstore.validator`
- Create a corresponding `ConstraintValidator` called `CategoryValidator` under the package `com.packt.webstore.validator`
- Add a corresponding error message in the message source file
- Your `CategoryValidator` should maintain a list of allowed categories (`List<String> allowedCategories`) to check whether the given category exists under the list of allowed categories
- Don't forget to initialize the `allowedCategories` list in the constructor of the `CategoryValidator` class
- Annotate the `category` field of the `Product` domain class with the `@Category` annotation

After applying your custom validation annotation `@category` on the `category` field of the `Product` domain class, your **Add new product** page should reject products of categories that have been not configured in the `CategoryValidator`.

Spring validation

We have seen how to incorporate JSR-303 Bean Validation with Spring MVC. In addition to Bean Validation, Spring has its own classic mechanism to perform validation as well, which is called Spring vValidation. The JSR-303 Bean Validation is much more elegant, expressive, and, in general, simpler to use when compared to the classic Spring validation. But the classic Spring validation is very flexible and extensible. For example, consider a cross field validation where we want to compare two or more fields to see whether their values can be considered as valid in combination. In such a case we can use Spring validation.

In the last section, using JSR-303 Bean Validation we have validated some of the individual fields on our `Product` domain object; we haven't done any validation that combines one or more fields. We don't know whether the combination of different fields makes sense.

Time for action – adding Spring validation

For example, say we have a constraint that we should not allow more than 99 units of any product to be added if the unit price is greater than \$ 1,000 for that product. Let's see how to add such a validation using Spring validation in our project:

1. Create a class called `UnitsInStockValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;

import java.math.BigDecimal;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import com.packt.webstore.domain.Product;

@Component
public class UnitsInStockValidator implements Validator{

    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Product product = (Product) target;

        if(product.getUnitPrice() != null && new
        BigDecimal(1000).compareTo(product.getUnitPrice())<=0 &&
        product.getUnitsInStock()>99) {
            errors.rejectValue("unitsInStock",
                "com.packt.webstore.validator
                .UnitsInStockValidator.message");
        }
    }
}
```

2. Open our message source file `messages.properties` from `/src/main/resources` in your project and add the following entry into it:

```
com.packt.webstore.validator.UnitsInStockValidator.message =
You cannot add more than 99 units if the unit price is
greater than 1000.
```

3. Open the `ProductController` class and autowire a reference to the `UnitsInStockValidator` class as follows:

```
@Autowired  
private UnitsInStockValidator unitsInStockValidator;
```

4. Now, inside the `initialiseBinder` method in the `ProductController` class, add the following line:

```
binder.setValidator(unitsInStockValidator);
```

5. Now run our application and enter

`http://localhost:8080/webstore/market/products/add`, and you will be able to see a webpage showing a web form to add product info. Enter all the valid values in the form; particularly fill in the **Unit Price** field with the value 1500 and **Units in stock** field with the value 150. Now simply click on the **Add** button and you will see validation messages at the top of the form as shown in the following screenshot:

The screenshot shows a web browser window with a form titled "Add new product". The form fields are as follows:

Field	Value
New Product ID	P1237
Name	Sony Head Phone
Unit Price	1500
Manufacturer	Sony
Category	Head phones
Units in stock	150
Description	With built-in noise canceling technology
Condition	<input checked="" type="radio"/> New <input type="radio"/> Old <input type="radio"/> Refurbished
Product Image file	Choose File No file chosen

A red error message box at the top of the form contains the text: "You cannot add more than 99 units if the unit price is greater than 1000." A blue "Add" button is located at the bottom right of the form.

The Add new product web form showing cross field validation.

What just happened?

In classic Spring validation, the main validation construct is the Validator (`org.springframework.validation.Validator`) interface. The Spring Validator interface defines two methods for validation purposes, namely `supports` and `validate`. The `supports` method indicates whether the validator can validate a specific class. If so, the `validate` method can be called to validate an object of that class.

Every Spring-based validator we are creating should implement this interface. In step 1 we just did just that; we simply created a class called `UnitsInStockValidator`, which implements the Spring Validator interface.

Inside the `validate` method of the `UnitsInStockValidator` class, we simply check whether the given `Product` object has a unit price greater than 1,000 and the number of units in stock is more than 99; if so, we reject that value with the corresponding error key to show the error message from the message source file:

```
@Override
public void validate(Object target, Errors errors) {
    Product product = (Product) target;

    if(product.getUnitPrice() != null && new
        BigDecimal(1000).compareTo(product.getUnitPrice())<=0 &&
        product.getUnitsInStock()>99) {
        errors.rejectValue("unitsInStock",
            "com.packt.webstore.validator.UnitsInStockValidator.message");
    }
}
```

In step 2 we simply added the actual error message for the error key `com.packt.webstore.validator.UnitsInStockValidator.message` in the message source file (`messages.properties`).

We created the validator but, to kick in the validation, we need to associate that validator with the controller. That's what we did in steps 3 and 4. In step 3 we simply added and autowired the reference to `UnitsInStockValidator` in the `ProductController` class. And we associated the `unitsInStockValidator` with `WebDataBinder` in the `initialiseBinder` method:

```
@InitBinder
public void initialiseBinder(WebDataBinder binder) {

    binder.setValidator(unitsInStockValidator);

    binder.setAllowedFields("productId",
```

```
    "name",
    "unitPrice",
    "description",
    "manufacturer",
    "category",
    "unitsInStock",
    "condition",
    "productImage",
    "language");
}
```

That's it, we created and configured our Spring-based validator to do the validation. Now run our application and enter

<http://localhost:8080/webstore/market/products/add> to show the web form for adding product info. Fill all the values in the form, particularly fill the **Unit Price** field with the value **1000** and **Units in stock** field with the value **150**, and click on the **Add** button. You will see validation messages at the top of the form saying **You cannot add more than 99 units if the unit price is greater than 1000.**

It is good that we have added Spring-based validation into our application. But since we have configured our Spring-based validator (`unitsInStockValidator`) with `WebDataBinder`, the Bean Validation that we have configured earlier would not take effect. Spring MVC will simply ignore those JSR-303 Bean Validation annotations (`@Pattern`, `@Size`, `@Min`, `@Digits`, `@NotNull`, and more).

Time for action – combining Spring validation and Bean Validation

So we need to write those Bean Validations again in classic Spring-based validation. This is not a good idea but, thanks to the flexibility and extensibility of Spring validation, we can combine both Spring-based validation and Bean Validation together with a little extra code. Let's do that:

1. Create a class called `ProductValidator` under the package `com.packt.webstore.validator` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.validator;

import java.util.HashSet;
import java.util.Set;
import javax.validation.ConstraintViolation;
import org.springframework.beans.factory
```

```
.annotation.Autowired;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;
import com.packt.webstore.domain.Product;

public class ProductValidator implements Validator{

    @Autowired private javax.validation.
    .Validator beanValidator;

    private Set<Validator> springValidators;

    public ProductValidator() {
        springValidators = new HashSet<Validator>();
    }

    public void setSpringValidators(Set<Validator>
springValidators) {
        this.springValidators = springValidators;
    }

    public boolean supports(Class<?> clazz) {
        return Product.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        Set<ConstraintViolation<Object>> constraintViolations =
beanValidator.validate(target);

        for (ConstraintViolation<Object> constraintViolation :
constraintViolations) {
            String propertyPath =
constraintViolation.getPropertyPath().toString();
            String message = constraintViolation.getMessage();
            errors.rejectValue(propertyPath, "", message);
        }

        for(Validator validator: springValidators) {
            validator.validate(target, errors);
        }
    }
}
```

2. Now open our web application context configuration file `WebApplicationContextConfig.java` and add the following bean definition into it:

```
@Bean
public ProductValidator productValidator () {
    Set<Validator> springValidators = new HashSet<>();
    springValidators.add(new UnitsInStockValidator());
    ProductValidator productValidator = new
ProductValidator();
    productValidator.setSpringValidators(springValidators);
    return productValidator;
}
```

3. Open our `ProductController` class and replace the existing reference of the `UnitsInStockValidator` field with our newly created `ProductValidator` class, as follows:

```
@Autowired
private ProductValidator productValidator;
```

4. Now, inside the `initialiseBinder` method of the `ProductController` class, replace the `binder.setValidator(unitsInStockValidator)` statement with the following statement:

```
binder.setValidator(productValidator);
```

5. Now run our application and enter the URL `http://localhost:8080/webstore/market/products/add` to check whether all the validations are working fine. Just click the **Add** button without filling anything in on the form. You will notice Bean Validation taking place; similarly fill the **Unit Price** field with the value 1500 and the **Units in stock** field with the value 150 to see Spring validation, as shown in the following screenshot:

Add new product

Invalid product name. It should be minimum 4 characters to maximum 50 characters long.
A product already exists with this product id.
You cannot add more than 99 units if the unit price is greater than 1000.

New Product ID	P1234	A product already exists with this product id.
Name		Invalid product name. It should be minimum 4 characters to maximum 50 characters long.
Unit Price	1500	
Manufacturer	Sony	
Category	Head phones	
Units in stock	150	
Description	With built-in noise canceling technology	
Condition	<input checked="" type="radio"/> New <input type="radio"/> Old <input type="radio"/> Refurbished	
Product Image file	<input type="button" value="Choose File"/>	No file chosen
<input type="button" value="Add"/>		

The Add new product web form showing Bean Validation and Spring validation together

What just happened?

Well, our aim was to combine Bean Validations and our Spring-based validation (`unitsInStockValidator`) together, to create a common adapter validator called `ProductValidator`. If you notice closely, the `ProductValidator` class is nothing but an implementation of the regular Spring validator.

We have autowired our existing bean validator into the `ProductValidator` class through the following line:

```
@Autowired  
private javax.validation.Validator beanValidator;
```

Later, we used this `beanValidator` reference inside the `validate` method of the `ProductValidator` class to validate all Bean Validation annotations, as follows:

```
Set<ConstraintViolation<Object>> constraintViolations =  
beanValidator.validate(target);
```

```
for (ConstraintViolation<Object> constraintViolation :  
constraintViolations) {  
    String propertyPath = constraintViolation.getPropertyPath().toString();  
    String message = constraintViolation.getMessage();  
    errors.rejectValue(propertyPath, "", message);  
}
```

The `beanValidator.validate(target)` statement returned all the constraint violations. Then, using the `errors` object, we threw all the invalid constraints as error messages. So every Bean Validation annotation we specified in the `Product` domain class will get handled within a `for` loop.

Similarly, we have one more `for` loop to handle all Spring validations in the `validate` method of the `ProductValidator` class:

```
for(Validator validator: springValidators) {  
    validator.validate(target, errors);  
}
```

This `for` loop iterates through the set of Spring validators and validates them one by one, but if you notice, we haven't initiated the `springValidators` reference. Thus, you may wonder where we have initiated the `springValidators` set. You can find the answer in step 2; we have created a bean for the `ProductValidator` class in our web application context (`WebApplicationContextConfig.java`) and assigned the `springValidators` set:

```
@Bean  
public ProductValidator productValidator () {  
    Set<Validator> springValidators = new HashSet<>();  
    springValidators.add(new UnitsInStockValidator());  
  
    ProductValidator productValidator = new ProductValidator();  
    productValidator.setSpringValidators(springValidators);  
  
    return productValidator;  
}
```

So now we have created a common adapter validator that can adopt Bean Validation and Spring validation, and validates all Spring and Bean-based validations together. Now we have to replace the `UnitsInStockValidator` reference with the `ProductValidator` reference in our `ProductController` class to kick in our new `ProductValidator`, which we have done in steps 3 and 4. We simply replaced the `UnitsInStockValidator` with `ProductValidator` in the binder, as follows:

```
@InitBinder
```

```
public void initialiseBinder(WebDataBinder binder) {  
  
    binder.setValidator(productValidator);  
  
    binder.setAllowedFields("productId",  
        "name",  
        "unitPrice",  
        "description",  
        "manufacturer",  
        "category",  
        "unitsInStock",  
        "condition",  
        "productImage",  
        "language");  
  
}
```

So we have successfully configured our newly created `ProductValidator` with `ProductController`. To see it in action, just run our application and enter the URL `http://localhost:8080/webstore/market/products/add`. Then, enter some invalid values such as an existing product ID, or fill the **Unit Price** field with the value 1000 and the **Units in stock** field with the value 100. You will notice the Bean Validation error messages and Spring validation error messages on the screen.

Have a go hero – adding Spring validation to a product image

Create a Spring validation class called `ProductImageValidator`, which will validate the size of the product image. It should only allow images whose size is less than or equal to the predefined configured size. Consider the following things while implementing your `ProductImageValidator`:

- Create a validation class called `ProductImageValidator` under the package `com.packt.webstore.validator` by implementing the `org.springframework.validation.Validator` interface
- Add a corresponding error message in the message source file
- Your `ProductImageValidator` should maintain a long variable called `allowedSize` to check whether the given image size is less than or equal to it

- Create a bean for the `ProductImageValidator` class in the servlet context and add it under the `springValidators` set of the `productValidator` bean
- Remember, don't forget to set the `allowedSize` property in the `ProductImageValidator` bean

After applying your custom validation annotation `@category` on the `category` field of the `Product` domain class, your **Add new product** page should reject products of other categories that have been not configured in the `CategoryValidator`.

Summary

In this chapter, you learned the concept of validation and saw how to enable Bean Validation in Spring MVC for form processing. We also learned how to set up a custom validation using the extension capability of the Bean Validation framework. After that, we learned how to do cross field validation using Spring validation. Finally, we saw how to integrate Bean Validation and Spring validation together.

In our next chapter, we will see how to develop an application in RESTful services. We will be covering the basic concepts of HTTP verbs and will try to understand how they are related to standard CRUD operations. We will also cover how to fire an Ajax request and how to handle it.

9

Give REST to Your Application with Ajax

REST stands for REpresentational State Transfer; REST is an architectural style. Everything in REST is considered as a resource and every resource is identified by a URI. RESTful web services have been embraced by large service providers across the Web as an alternative to SOAP-based Web Services due to their simplicity.

After finishing this chapter, you will have a clear idea about:

- REST web services
- Ajax

Introduction to REST

As I already mentioned, in a REST-based application, everything including static resources, data, and operations is considered as a resource and identified by a URI. For example, consider a piece of functionality to add a new product to our store; we can represent that operation by a URI, something such as

`http://localhost:8080/webstore/products/add`, and we can pass the new product details in XML or JSON representation to that URL. So, in REST, URIs are used to connect clients and servers to exchange resources in the form of representations (HTML, XML, JSON, and so on). In order to exchange data, REST relies on basic HTTP protocol methods GET, POST, PUT, and DELETE.

Spring provides extensive support for developing REST-based web services. In our previous chapters, we have seen that, whenever a web request comes in, we returned a web page to serve that request; usually that web page contained some states (that is, dynamic data). However, in REST-based applications, we only return the states and it is up to the client to decide how to render or present the data to the end user.

Usually, REST-based web services return data in two formats, XML and JSON. We are going to develop some REST-based web services that can return data in the JSON format. After we get the JSON data, we are going to render that data as an HTML page using some JavaScript libraries in the browser.

In our `webstore` application, we have successfully listed some of the products, but the store cannot make a profit without enabling the end user to pick up some products in his/her shopping cart. So let's add a shopping cart facility to our store.

Time for action – implementing RESTful web services

We are going to add the shopping cart facility in two stages. Firstly, we will create a REST-style Controller to handle all shopping cart-related web service requests. Secondly, we will add some JavaScript code to render the JSON data returned by the REST web Service Controller. So first, let's implement some RESTful web services using Spring MVC Controllers, so that later we can add some JavaScript code to consume those web services:

1. Add the following schema definitions for `CART` and `CART_ITEM` tables in `create-table.sql`. You can find `create-table.sql` under the folder structure `src/main/resources/db/sql`:

```
CREATE TABLE CART (
    ID VARCHAR(50) PRIMARY KEY
);

CREATE TABLE CART_ITEM (
    ID VARCHAR(75),
    PRODUCT_ID VARCHAR(25) FOREIGN KEY REFERENCES
    PRODUCTS(ID),
    CART_ID varchar(50) FOREIGN KEY REFERENCES
    CART(ID),
    QUANTITY BIGINT,
    CONSTRAINT CART_ITEM_PK PRIMARY KEY (ID,CART_ID)
);
```

2. Add the following drop table command as the first line in `create-table.sql`:

```
DROP TABLE CART_ITEM IF EXISTS;
DROP TABLE CART IF EXISTS;
```

3. Create a domain class named `CartItem` under the package `com.packt.webstore.domain` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.math.BigDecimal;

public class CartItem implements Serializable{

    private static final long serialVersionUID = -
4546941350577482213L;

    private String id;
    private Product product;
    private int quantity;
    private BigDecimal totalPrice;

    public CartItem(String id) {
        super();
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public Product getProduct() {
        return product;
    }

    public void setProduct(Product product) {
        this.product = product;
        this.updateTotalPrice();
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
```

```
        this.quantity = quantity;
    }

    public BigDecimal getTotalPrice() {
        this.updateTotalPrice();
        return totalPrice;
    }

    public void updateTotalPrice() {
        totalPrice =
this.product.getUnitPrice().multiply(new
BigDecimal(this.quantity));
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 :
id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        CartItem other = (CartItem) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}
```

4. Similarly, add one more domain class named `Cart` in the same package and add the following code into it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.math.BigDecimal;
```

```
import java.util.List;
import java.util.function.Function;

public class Cart implements Serializable{

    private static final long serialVersionUID =
6554623865768217431L;

    private String id;
    private List<CartItem> cartItems;
    private BigDecimal grandTotal;

    public Cart(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public BigDecimal getGrandTotal() {
        updateGrandTotal();
        return grandTotal;
    }

    public void setGrandTotal(BigDecimal grandTotal) {
        this.grandTotal = grandTotal;
    }

    public List<CartItem> getCartItems() {
        return cartItems;
    }

    public void setCartItems(List<CartItem> cartItems) {
this.cartItems = cartItems;
    }

    public CartItem getItemByProductId(String
productId) {
        return cartItems.stream().filter(cartItem ->
cartItem.getProduct().getProductId()
.equals(productId))
            .findAny()
            .orElse(null);
    }

    public void updateGrandTotal() {
```

```
        Function<CartItem, BigDecimal> totalMapper =  
        cartItem -> cartItem.getTotalPrice();  
  
        BigDecimal grandTotal = cartItems.stream()  
            .map(totalMapper)  
            .reduce(BigDecimal.ZERO, BigDecimal::add);  
  
        this.setGrandTotal(grandTotal);  
    }  
  
    @Override  
    public int hashCode() {  
        final int prime = 31;  
        int result = 1;  
        result = prime * result + ((id == null) ? 0 :  
id.hashCode());  
        return result;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        Cart other = (Cart) obj;  
        if (id == null) {  
            if (other.id != null)  
                return false;  
        } else if (!id.equals(other.id))  
            return false;  
        return true;  
    }  
}
```

5. Create a data transfer object (**dto**) named `CartItemDto` under the package `com.packt.webstore.dto` in the source folder `src/main/java`, and add the following code into it:

```
package com.packt.webstore.dto;  
  
import java.io.Serializable;  
  
public class CartItemDto implements Serializable{  
  
    private static final long serialVersionUID = -
```

```
3551573319376880896L;

    private String id;
    private String productId;
    private int quantity;
    public String getId() {
        return id;
    }

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

    public String getProductId() {
        return productId;
    }
    public void setProductId(String productId) {
        this.productId = productId;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

6. Similarly, add one more dto object named `CartDto` in the same package and add the following code into it:

```
package com.packt.webstore.dto;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class CartDto implements Serializable{

    private static final long serialVersionUID = -
2017182726290898588L;

    private String id;
    private List<CartItemDto> cartItems;

    public CartDto() {}}
```

```
public CartDto(String id) {
    this.id = id;
    cartItems = new ArrayList<>();
}

public String getId() {
    return id;
}

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

public List<CartItemDto> getCartItems() {
    return cartItems;
}

public void setCartItems(List<CartItemDto>
cartItems) {
    this.cartItems = cartItems;
}

public void addCartItem(CartItemDto cartItemDto) {
    this.cartItems.add(cartItemDto);
}
}
```

7. Create a class named `CartItemMapper` under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.packt.webstore.domain.CartItem;
import com.packt.webstore.service.ProductService;

public class CartItemMapper implements
RowMapper<CartItem> {
    private ProductService productService;
    public CartItemMapper(ProductService
productService) {
        this.productService = productService;
    }
}
```

```
    }

    @Override
    public CartItem mapRow(ResultSet rs, int rowNum)
    throws SQLException {
        CartItem cartItem = new
CartItem(rs.getString("ID"));
        cartItem.setProduct(productService.getProductById
        (rs.getString("PRODUCT_ID")));
        cartItem.setQuantity(rs.getInt("QUANTITY"));

        return cartItem;
    }
}
```

8. Create a class named `CartMapper` under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.domain.repository.impl;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam
.NamedParameterJdbcTemplate;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain.CartItem;
import com.packt.webstore.service.ProductService;

public class CartMapper implements RowMapper<Cart> {
    private CartItemMapper cartItemMapper;
    private NamedParameterJdbcTemplate jdbcTemplate;
    public CartMapper(NamedParameterJdbcTemplate
jdbcTemplate, ProductService productService) {
        this.jdbcTemplate = jdbcTemplate;
        cartItemMapper = new
CartItemMapper(productService);
    }

    public Cart mapRow(ResultSet rs, int rowNum)
throws SQLException {
    String id = rs.getString("ID");
    Cart cart = new Cart(id);
```

```
        String SQL = String.format("SELECT * FROM
CART_ITEM WHERE CART_ID = '%s'", id);
        List<CartItem> cartItems =
jdbcTemplete.query(SQL, cartItemMapper);
        cart.setCartItems(cartItems);
        return cart;
    }
}
```

9. Create an interface named `CartRepository` under the package `com.packt.webstore.domain.repository` in the source folder `src/main/java` and add the following method declarations into it:

```
package com.packt.webstore.domain.repository;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.dto.CartDto;

public interface CartRepository {

    void create(CartDto cartDto);
    Cart read(String id);
    void update(String id, CartDto cartDto);
    void delete(String id);

    void addItem(String cartId, String productId);
    void removeItem(String cartId, String productId);
}
```

10. Create an implementation class named `InMemoryCartRepository` for the previous interface under the package `com.packt.webstore.domain.repository.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.domain.repository.impl;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.beans.factory
.annotation.Autowired;
import org.springframework.dao
.EmptyResultDataAccessException;
import org.springframework.jdbc.core.namedparam
.NamedParameterJdbcTemplate;
```

```
import org.springframework.stereotype.Repository;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain.CartItem;
import com.packt.webstore.domain.Product;
import com.packt.webstore.domain.
repository.CartRepository;
import com.packt.webstore.
dto.CartDto;
import com.packt.webstore.dto.CartItemDto;
import com.packt.webstore.service.ProductService;

@Repository
public class InMemoryCartRepository implements
CartRepository{
    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;
    @Autowired
    private ProductService productService;
    public void create(CartDto cartDto) {
        String INSERT_CART_SQL = "INSERT INTO CART(ID)
VALUES (:id)";

        Map<String, Object> cartParams = new
HashMap<String, Object>();
        cartParams.put("id", cartDto.getId());
        jdbcTemplate.update(INSERT_CART_SQL,
cartParams);

        cartDto.getCartItems().stream()
.forEach(cartItemDto ->{
            Product productById =
productService.getProductById
(cartItemDto.getProductId());
            String INSERT_CART_ITEM_SQL =
"INSERT INTO CART_ITEM(ID, PRODUCT_ID
,CART_ID, QUANTITY) "
                    + "VALUES (:id,
:product_id, :cart_id, :quantity)";

            Map<String, Object> cartItemsParams = new
HashMap<String, Object>();
            cartItemsParams.put("id",
cartItemDto.getId());
            cartItemsParams.put("product_id",
productById.getProductId());
            cartItemsParams.put("cart_id",
cartDto.getId());
        }
    }
}
```

```
        cartItemsParams.put("quantity",
cartItemDto.getQuantity());
        jdbcTemplete.update(INSERT_CART_ITEM_SQL,
cartItemsParams);
    });
}
public Cart read(String id) {
    String SQL = "SELECT * FROM CART WHERE ID = :id";
    Map<String, Object> params = new HashMap<String,
Object>();
    params.put("id", id);
    CartMapper cartMapper = new
CartMapper(jdbcTemplete, productService);
    try {
        return jdbcTemplete.queryForObject(SQL,
params, cartMapper);
    } catch (EmptyResultDataAccessException e) {
        return null;
    }
}
@Override
public void update(String id, CartDto cartDto) {
    List<CartItemDto> cartItems =
cartDto.getCartItems();
    for(CartItemDto cartItem :cartItems) {
        String SQL = "UPDATE CART_ITEM SET QUANTITY
= :quantity, PRODUCT_ID = :productId WHERE ID = :id
AND CART_ID = :cartId";
        Map<String, Object> params = new
HashMap<String, Object>();
        params.put("id", cartItem.getId());
        params.put("quantity",
cartItem.getQuantity());
        params.put("productId",
cartItem.getProductId());
        params.put("cartId", id);
        jdbcTemplete.update(SQL, params);
    }
}

@Override
public void delete(String id) {
    String SQL_DELETE_CART_ITEM = "DELETE FROM
CART_ITEM WHERE CART_ID = :id";
    String SQL_DELETE_CART = "DELETE FROM CART
WHERE ID = :id";
    Map<String, Object> params = new
HashMap<String, Object>();
```

```
        params.put("id", id);
        jdbcTempleate.update(SQL_DELETE_CART_ITEM,
params);
        jdbcTempleate.update(SQL_DELETE_CART, params);
    }
@Override
public void addItem(String cartId, String
productId) {
    String SQL=null;
    Cart cart = null;
    cart = read(cartId);
    if(cart ==null) {
        CartItemDto newCartItemDto = new
CartItemDto();
        newCartItemDto.setId(cartId+productId);
        newCartItemDto.setProductId(productId);
        newCartItemDto.setQuantity(1);
        CartDto newCartDto = new CartDto(cartId);
        newCartDto.addCartItem(newCartItemDto);
        create(newCartDto);
        return;
    }
    Map<String, Object> cartItemsParams = new
HashMap<String, Object>();

    if(cart.getItemByProductId(productId) == null) {
        SQL = "INSERT INTO CART_ITEM (ID,
PRODUCT_ID, CART_ID, QUANTITY) VALUES (:id,
:productId, :cartId, :quantity)";
        cartItemsParams.put("id", cartId+productId);
        cartItemsParams.put("quantity", 1);
    } else {
        SQL = "UPDATE CART_ITEM SET QUANTITY =
:quantity WHERE CART_ID = :cartId AND PRODUCT_ID =
:productId";
        CartItem existingItem =
        cart.getItemByProductId(productId);
        cartItemsParams.put("id",
existingItem.getId());
        cartItemsParams.put("quantity",
existingItem.getQuantity()+1);
    }
    cartItemsParams.put("productId", productId);
    cartItemsParams.put("cartId", cartId);
    jdbcTempleate.update(SQL, cartItemsParams);
}

@Override
```

```
    public void removeItem(String cartId, String
productId) {
    String SQL_DELETE_CART_ITEM = "DELETE FROM
CART_ITEM WHERE PRODUCT_ID = :productId AND CART_ID =
:id";
    Map<String, Object> params = new
HashMap<String, Object>();
    params.put("id", cartId);
    params.put("productId", productId);
    jdbcTemplete.update(SQL_DELETE_CART_ITEM,
params);
}
}
```

11. Create an interface named `CartService` under the package `com.packt.webstore.service` in the source folder `src/main/java` and add the following method declarations into it:

```
package com.packt.webstore.service;
import com.packt.webstore.domain.Cart;
import com.packt.webstore.dto.CartDto;

public interface CartService {
    void create(CartDto cartDto);
    Cart read(String cartId);
    void update(String cartId, CartDto cartDto);
    void delete(String id);

    void addItem(String cartId, String productId);

    void removeItem(String cartId, String productId);
}
```

12. Create an implementation class named `CartServiceImpl` for the preceding interface under the package `com.packt.webstore.service.impl` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.service.impl;

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

import com.packt.webstore.domain.Cart;
import com.packt.webstore.domain
.repository.CartRepository;
import com.packt.webstore.dto.CartDto;
```

```
import com.packt.webstore.service.CartService;

@Service
public class CartServiceImpl implements CartService{
    @Autowired
    private CartRepository cartRepository;

    public void create(CartDto cartDto) {
        cartRepository.create(cartDto);
    }

    @Override
    public Cart read(String id) {
        return cartRepository.read(id);
    }

    @Override
    public void update(String id, CartDto cartDto) {
        cartRepository.update(id, cartDto);
    }

    @Override
    public void delete(String id) {
        cartRepository.delete(id);
    }

    @Override
    public void addItem(String cartId, String
productId) {
        cartRepository.addItem(cartId, productId);
    }

    @Override
    public void removeItem(String cartId, String
productId) {
        cartRepository.removeItem(cartId, productId);
    }
}
```

13. Now create a class named `CartRestController` under the package `com.packt.webstore.controller` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import javax.servlet.http.HttpSession;

import org.springframework.beans.factory
```

```
.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind
.annotationPathVariable;
import org.springframework.web.bind
.annotation.RequestBody;
import org.springframework.web.bind
.annotation.RequestMapping;
import org.springframework.web.bind
.annotation.RequestMethod;
import org.springframework.web.bind
.annotation.ResponseStatus;
import org.springframework.web.bind
.annotation.RestController;

import com.packt.webstore.domain.Cart;
import com.packt.webstore.dto.CartDto;
import com.packt.webstore.service.CartService;

@RestController
@RequestMapping(value = "rest/cart")
public class CartRestController {

    @Autowired
    private CartService cartService;
    @RequestMapping(method = RequestMethod.POST)
    @ResponseStatus(value = HttpStatus.CREATED)
    public void create(@RequestBody CartDto cartDto) {
        cartService.create(cartDto);
    }

    @RequestMapping(value = "/{cartId}", method =
RequestMethod.GET)
    public Cart read(@PathVariable(value = "cartId")
String cartId) {
        return cartService.read(cartId);
    }

    @RequestMapping(value = "/{cartId}", method =
RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void update(@PathVariable(value = "cartId")
String cartId, @RequestBody CartDto cartDto) {
        cartDto.setId(cartId);
        cartService.update(cartId, cartDto);
    }

    @RequestMapping(value = "/{cartId}", method =
```

```
RequestMethod.DELETE)
    @ResponseStatus(value = HttpStatus.OK)
    public void delete(@PathVariable(value = "cartId")
String cartId) {
    cartService.delete(cartId);
}
@RequestMapping(value = "/add/{productId}", method
= RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void addItem(@PathVariable String
productId, HttpSession session) {
    cartService.addItem(session.getId(), productId);
}
@RequestMapping(value = "/remove/{productId}",
method = RequestMethod.PUT)
    @ResponseStatus(value = HttpStatus.OK)
    public void removeItem(@PathVariable String
productId, HttpSession session) {
    cartService.removeItem(session.getId(), productId);
}
}
```

14. Now run our webstore project from the STS.

What just happened?

Okay, in order to store cart-related information, first of all we need the database tables to store cart-related information, which is why we are creating `CART` and `CART_ITEM` tables in steps 1 and 2.

In steps 3 and 4, we have just created two domain classes called `CartItem` and `Cart` to hold the information about the shopping cart. The `CartItem` class just represents a single item in a shopping cart and it holds information such as the product, quantity, and the `totalPrice`. Similarly, the `Cart` represents the whole shopping cart itself; a `Cart` can have a collection of `cartItems` and `grandTotal`. Similarly, in steps 5 and 6 we have created two more data transfer objects called `CartItemDto` and `CartDto` to carry data between the REST client and our backend services.

In steps 7 and 8 we created a class called `CartItemMapper` and `CartMapper`, which we are going to use to map the database records to the `CartItem` and `Cart` domain object in the repository classes.

In steps 9 to 10, we just created a repository layer to manage `CartDto` objects. In the `CartRepository` interface, we have defined six methods to take care of CRUD operations (Create, Read, Update, and Delete) on the `CART_ITEM` table. The `InMemoryCartRepository` is just an implementation of the `CartRepository` interface.

Similarly, from steps 11 to 12, we have created the service layer for `Cart` objects. The `CartService` interface has the same methods of the `CartRepository` interface. The `CartServiceImpl` class just internally uses `InMemoryCartRepository` to carry out all the CRUD operations.

Step 13 is very crucial in the whole sequence because in that step we have created our REST-style Controller to handle all `Cart`-related REST web services. The `CartRestController` class mainly has six methods to handle web requests for CRUD operations on `Cart` objects, namely `create`, `read`, `update`, and `delete`, and two more methods, `addItem` and `removeItem`, to handle adding and removing a `CartItem` from the `Cart` object. We will explore the first four CRUD methods in greater details.

You will see, on the surface, the `CartRestController` class is just like any other normal Spring MVC Controller, because it has the same `@RequestMapping` annotations. What makes it special enough to become a REST-style Controller is the `@RestController` and `@RequestBody` annotations. See the following Controller method:

```
@RequestMapping(value = "/{cartId}", method = RequestMethod.GET)
public Cart read(@PathVariable(value = "cartId") String cartId) {
    return cartService.read(cartId);
}
```

Usually every Controller method is used to return a View name, so that the dispatcher servlet can find the appropriate View file and dispatch that View file to the client, but here we have returned the object (`Cart`) itself. Instead of putting the object into the Model, we have returned the object itself. Why? Because we want to return the state of the object in JSON format. Remember, REST-based web services should return data in JSON or XML format and the client can use the data however they want; they may render it to an HTML page, or they may send it to some external system as it is, as raw JSON/XML data.

Okay, let's come to the point. The `read` Controller method is just returning `Cart` Java objects; how come this Java object is being converted into JSON or XML format? This is where the `@RestController` annotation comes in. The `@RestController` annotation instructs Spring to convert all Java objects that are returned from the request-mapping methods into JSON/XML format and send a response in the body of the HTTP response.

Similarly, when you send an HTTP request to a Controller method with JSON/XML data in it, the `@RequestBody` annotation instructs Spring to convert it into the corresponding Java object. That's why the `create` method has a `cartDto` parameter annotated with a `@RequestBody` annotation.

If you closely observe the `@RequestMapping` annotation of all those six CRUD methods, you will end up with the following table:

URL	HTTP method	Description
<code>http://localhost:8080/webstore/rest/cart</code>	POST	Creates a new cart
<code>http://localhost:8080/webstore/rest/cart/1234</code>	GET	Retrieves cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/1234</code>	PUT	Updates cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/1234</code>	DELETE	Deletes cart with ID = 1234
<code>http://localhost:8080/webstore/rest/cart/add/P1234</code>	PUT	Adds the product with ID P1234 to the cart under session
<code>http://localhost:8080/webstore/rest/cart/remove/P1234</code>	PUT	Removes the product with ID P1234 to the cart under session

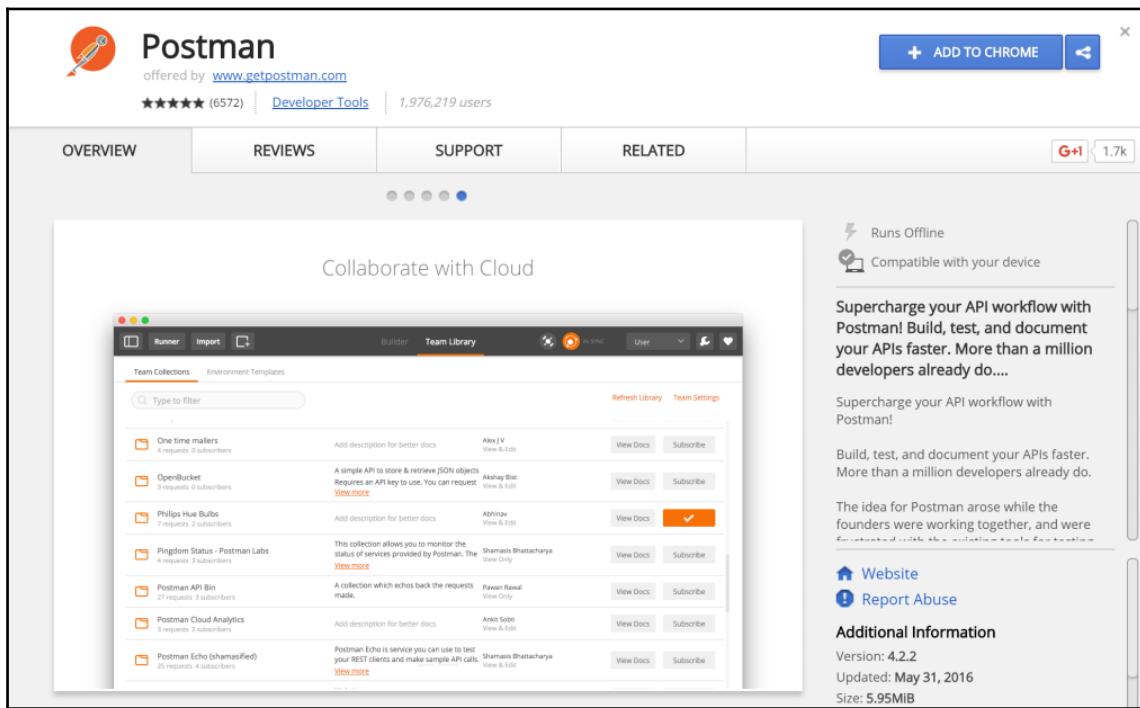
Though the request-mapping URL is more or less the same, based on the HTTP method, we are performing different operations. For example, if you send a GET request to the URL `http://localhost:8080/webstore/rest/cart/1234`, the `read` Controller method would get executed and the `Cart` object with ID 1234 will get returned in JSON format. Similarly, if you send a PUT or DELETE request to the same URL, the `update` or `delete` Controller method would get called correspondingly.

In addition to those four CRUD request-mapping methods, we have two more request-mapping methods that take care of adding and removing a `CartItem` from the `Cart` object. These methods are considered update methods, which is why both `theaddItem` and `removeItem` methods have PUT as a request method type in their `@RequestMapping` annotation. For instance, if you send a PUT request to the URL `http://localhost:8080/webstore/rest/cart/add/P1236`, a product with a product id P1236 will be added to the `Cart` object. Similarly, if you send a PUT request to the URL `http://localhost:8080/webstore/rest/cart/remove/P1236`, a product with P1236 will be removed from the `Cart` object.

Time for action – consuming REST web services

Okay, we have created our REST-style Controller, which can serve some REST-based web requests, but we have not seen our `CartRestController` in action. Using the standard browser we can only send GET or POST requests; in order to send a PUT or DELETE request we need a special tool. There are plenty of HTTP client tools available to send requests, let's use one such tool called **Postman** to test our `CartRestController`. Postman is a Google Chrome browser extension, so better install Google Chrome in your system before downloading the Postman HTTP client:

1. Go to the Postman download page at <http://www.getpostman.com/> from your Google Chrome browser and click **Chrome App**.
2. You will be taken to the Chrome webstore page; click the **+ ADD TO CHROME** button to install the Postman tool in your browser.



Postman – HTTP client app installing

3. Now a Google login page will appear to ask you to log in. Log in using your Google account.
4. A confirmation dialog will be shown on your screen asking your permission to add the Postman extension to your browser; click the **Add app** button.
5. Now open your Google Chrome browser and enter the URL `chrome://apps/`. A web page will be loaded with all the available apps for your Chrome browser. Just click on the Postman icon to launch the Postman app. Before launching Postman just ensure our `webstore` project is running.
6. Now, in the Postman app, enter the request URL as `http://localhost:8080/webstore/rest/cart`, the request method as **POST**, the **Body as raw** format, and the content type as **JSON(application/json)**.
7. Now enter the following JSON content in the content box and press the **Send** button. An HTTP respond status 201 will be created:

```
{  
  "id": "111",
```

```
"cartItems" : [
    {
        "id" :"1",
        "productId":"P1234",
        "quantity":1
    },
    {
        "id" :"2",
        "productId":"P1235",
        "quantity":2
    }
]
```

The screenshot shows the Postman application interface. The top navigation bar includes 'Runner', 'Import', 'Builder' (which is selected), and 'Team Library'. There are also sync and environment dropdowns. The URL field contains 'http://localhost:8080/web:'. The request method is set to 'POST' and the endpoint is 'http://localhost:8080/webstore/rest/cart'. The 'Body' tab is active, showing the JSON payload from the code block above. The 'raw' option is selected under the body type dropdown, which is set to 'JSON (application/json)'. The JSON content is displayed in a code editor-like view with line numbers. Below the body, there are tabs for 'Body', 'Cookies', 'Headers (9)', and 'Tests'. At the bottom right, the status is shown as 'Status: 201 Created' and 'Time: 228 ms'.

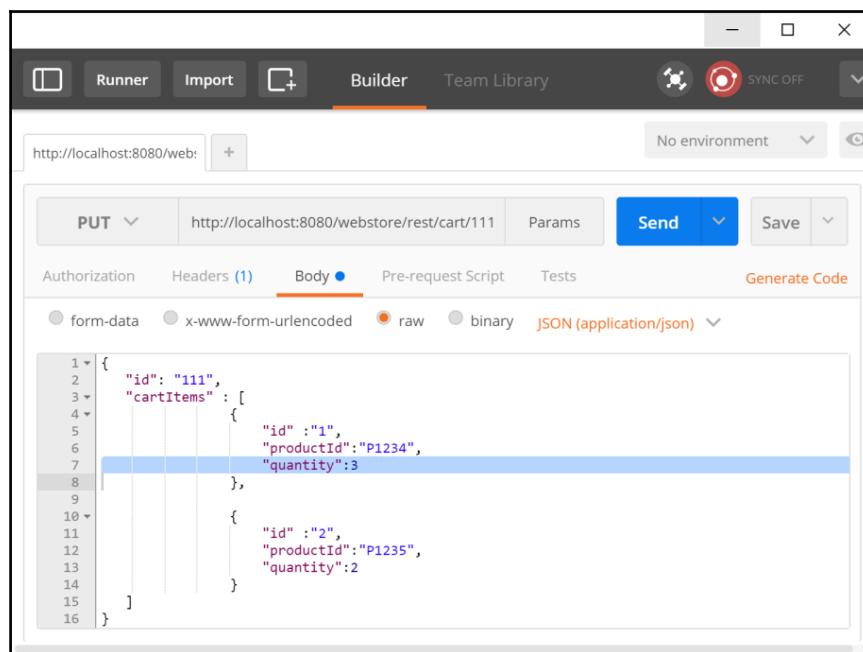
Postman – posting a web request

8. Now, similarly in the Postman app, enter the target URL as `http://localhost:8080/webstore/rest/cart/111` and the request method as **GET**, and press the **Send** button. You will get the following JSON as a response:

```
{  
    "id": "111",  
    "cartItems": [  
        {  
            "id": "1",  
            "product": {  
                "productId": "P1234",  
                "name": "iPhone 6s",  
                "unitPrice": 500,  
                "description": "Apple iPhone 6s smartphone  
with 4.00-inch 640x1136 display and 8-megapixel rear  
camera",  
                "manufacturer": "Apple",  
                "category": "Smart Phone",  
                "unitsInStock": 450,  
                "unitsInOrder": 0,  
                "discontinued": false,  
                "condition": "New"  
            },  
            "quantity": 1,  
            "totalPrice": 500  
        },  
        {  
            "id": "2",  
            "product": {  
                "productId": "P1235",  
                "name": "Dell Inspiron",  
                "unitPrice": 700,  
                "description": "Dell Inspiron 14-inch Laptop  
(Black) with 3rd Generation Intel Core processors",  
                "manufacturer": "Dell",  
                "category": "Laptop",  
                "unitsInStock": 1000,  
                "unitsInOrder": 0,  
                "discontinued": false,  
                "condition": "New"  
            },  
            "quantity": 2,  
            "totalPrice": 1400  
        }  
    "grandTotal": 1900
```

}

9. To update the cart in the Postman app, enter the target URL as `http://localhost:8080/webstore/rest/cart/111` and just change the JSON data. For instance, in the content box, just change the quantity to 3 for the P1234 cart item, choose the request method as **PUT** and the content type as **JSON(application/json)**, and send the request to the same URL by pressing the **Send** button:



Postman – posting a PUT web service request.

10. To verify whether your changes took place, just repeat step 8; you will see the `totalPrice` increased to 1500 for the cart item with the product ID P1234; `grandTotal` will increase accordingly as well.
11. Similarly, to delete the cart just enter the `http://localhost:8080/webstore/rest/cart/111` URL in the Postman app, enter the target URL and the request method as **DELETE**, and press the **Send** button. You will get the HTTP status **200 (OK)** as a response. To verify whether the cart got deleted, just repeat step 8; you will get an empty response.

What just happened?

At the start of the chapter, we discussed that most REST-based web services are designed to exchange data in JSON or XML format. This is because the JSON and XML formats are considered universal formats so that any system can easily understand, parse, and interpret that data. In order to test the REST-based web services that we have created, we need a tool that can send different (GET, PUT, POST, or DELETE) types of HTTP requests with JSON data in its request body. Postman is one such tool and is available as a Google Chrome extension.

From steps 1 to 4, we just installed the Postman app in our Google Chrome browser. In steps 6 and 7, we just sent our first REST-based web request to the target URL `http://localhost:8080/webstore/rest/cart` to create a new cart in our `webstore` application. We did this by sending a POST request with the cart information as JSON data to the target URL. If you notice the following JSON data, it represents a cart with a cart id 111, and it has two product (P1234 and P1235) cart items in it:

```
{
  "id": "111",
  "cartItems" : [
    {
      "id" :"1",
      "productId": "P1234",
      "quantity":1
    },
    {
      "id" :"2",
      "productId": "P1235",
      "quantity":2
    }
  ]
}
```

Since we have posted our first cart in our system, to verify whether that cart got stored in our system, we have sent another REST web request in step 8 to get the whole cart information in JSON format. Notice this time the request type is GET and the target URL is `http://localhost:8080/webstore/rest/cart/111`. Remember, we have learned that, in REST-based applications, every resource is identifiable by a URI. Here the URL `http://localhost:8080/webstore/rest/cart/111` represents a cart whose ID is 111. If you send a GET request to the previous URL you will get the cart information as JSON data.

Similarly, you can even update the whole cart by sending an updated JSON data as a PUT request to the same URL, which is what we have done in step 9. In a similar fashion we have sent a DELETE request to the same URL to delete the cart whose ID is 111.

Okay, we have tested or consumed our REST-based web services with the help of the Postman HTTP client tool, which is working quite well. But in a real-world application, most of the time these kinds of REST-based web service are consumed from the frontend with the help of a concept called Ajax. Using a JavaScript library, we can send an Ajax request to the backend. In the next section, we will see what Ajax requests are and how to consume REST-based web services using JavaScript/Ajax libraries.

Handling web services in Ajax

Ajax (Asynchronous JavaScript and XML) is a web development technique used on the client side to create asynchronous web applications. In a typical web application, every time a web request is fired as a response, we get a full web page loaded, but in an Ajax-based web application web pages are updated asynchronously by polling small data with the server behind the scenes. This means that, using Ajax, it is possible to update parts of a web page without reloading the entire web page. With Ajax, web applications can send data to, and retrieve data from, a server asynchronously. The asynchronous aspect of Ajax allows us to write code that can send some requests to a server and handles a server response without reloading the entire web page.

In an Ajax-based application, the XMLHttpRequest object is used to exchange data asynchronously with the server, whereas XML or JSON is often used as the format for transferring data. The “X” in AJAX stands for XML, but JSON is used instead of XML nowadays because of its simplicity, and it uses fewer characters to represent the same data compared to XML. So it can reduce the bandwidth requirements over the network to make the data transfer speed faster.

Time for action – consuming REST web services via Ajax

Okay, we have implemented some REST-based web services that can manage the shopping cart in our application, but we need a frontend that can facilitate the end user to manage a shopping cart visually. So let's consume those web services via Ajax in the frontend to manage the shopping cart. In order to consume REST web services via Ajax, perform the following steps:

1. Add a JavaScript file named `controllers.js` under the directory `/src/main/webapp/resources/js/`, add the following code snippets into it, and save it:

```
var cartApp = angular.module('cartApp', []);

cartApp.controller('cartCtrl', function($scope,
$http) {

    $scope.refreshCart = function(cartId) {
        $http.get('/webstore/rest/cart/' +
$scope.cartId)
            .success(function(data) {
                $scope.cart = data;
            });
    };

    $scope.clearCart = function() {
        $http.delete('/webstore/rest/cart/' +
$scope.cartId)
            .success(function(data) {
                $scope.refreshCart($scope.cartId);
            });
    };

    $scope.initCartId = function(cartId) {
        $scope.cartId = cartId;
        $scope.refreshCart($scope.cartId);
    };

    $scope.addToCart = function(productId) {
        $http.put('/webstore/rest/cart/add/' +
productId)
            .success(function(data) {
                alert("Product Successfully added to
the Cart!");
            });
    };
    $scope.removeFromCart = function(productId) {
        $http.put('/webstore/rest/cart/remove/' +
productId)
            .success(function(data) {
                $scope.refreshCart($scope.cartId);
            });
    };
});
```

2. Now create a class named `CartController` under the package `com.packt.webstore.controller` in the source folder `src/main/java` and add the following code into it:

```
package com.packt.webstore.controller;

import javax.servlet.http.HttpServletRequest;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping(value = "/cart")
public class CartController {

    @RequestMapping
    public String get(HttpServletRequest request) {
        return
    "redirect:/cart/" + request.getSession(true).getId();
    }

    @RequestMapping(value = "/{cartId}", method =
RequestMethod.GET)
    public String getCart(@PathVariable(value =
"cartId") String cartId, Model model) {
        model.addAttribute("cartId", cartId);
        return "cart";
    }
}
```

3. Add one more JSP View file named `cart.jsp` under the directory `src/main/webapp/WEB-INF/views/`, add the following code snippets into it, and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
```

```
<link rel="stylesheet"
      href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css/bootstrap.min.css">

<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.5.1/angular.min.js"></script>

<script src="/webstore/resources/js/controllers.js"></script>

<title>Cart</title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1>Cart</h1>
                <p>All the selected products in your
                    cart</p>
            </div>
        </div>
    </section>

    <section class="container" ng-app="cartApp">
        <div ng-controller="cartCtrl" ng-
            init="initCartId('${cartId}')">

            <div>
                <a class="btn btn-danger pull-left"
                   ng-click="clearCart()"> <span
                   class="glyphicon glyphicon-remove-
                   sign"></span> Clear Cart
                </a> <a href="#" class="btn btn-success
                   pull-right"> <span
                   class="glyphicon-shopping-cart
                   glyphicon"></span> Check out
                </a>
            </div>
            <table class="table table-hover">
                <tr>
                    <th>Product</th>
                    <th>Unit price</th>
```

```
<th>Qauntity</th>
<th>Price</th>
<th>Action</th>
</tr>
<tr ng-repeat="item in cart.cartItems">
    <td>{{item.product.productId}}-
    {{item.product.name}}</td>
    <td>{{item.product.unitPrice}}</td>
    <td>{{item.quantity}}</td>
    <td>{{item.totalPrice}}</td>
    <td><a href="#" class="label label-
danger" ng-
click="removeFromCart(item.product.productId)"> <span
class="glyphicon glyphicon-
remove" /></span> Remove
</a></td>
</tr>
<tr>
    <th></th>
    <th></th>
    <th>Grand Total</th>
    <th>{{cart.grandTotal}}</th>
    <th></th>
</tr>
</table>
<a href="
```

4. Open product.jsp from src/main/webapp/WEB-INF/views/ and add the following AngularJS-related script links in the head section as follows:

```
<script src="https://ajax.googleapis.com/ajax
/libs/angularjs/1.5.5/angular.min.js"></script>
```

5. Similarly, add more script links to our controller.js as follows:

```
<script src="/webstore/resources  
/js/controllers.js"></script>
```

6. Now add the ng-click AngularJS directive to the Order Now <a> tag as follows:

```
<a href="#" class="btn btn-warning btn-large" ng-  
click="addToCart('${product.productId}')">  
<span class="glyphicon-shopping-cart  
glyphicon"></span> Order Now  
</a>
```

7. Finally, add one more <a> tag beside the Order Now <a> tag, which will show the **View cart** button and save the product.jsp:

```
<a href="    <span class="glyphicon-hand-right  
glyphicon"></span> View Cart  
</a>
```

8. Now add the ng-app AngularJS directive to the Order Now <section> tag as follows:

```
<section class="container" ng-app="cartApp">
```

9. Now add the ng-controller AngularJS directive to the surrounding <p> tag of the Order Now link as follows:

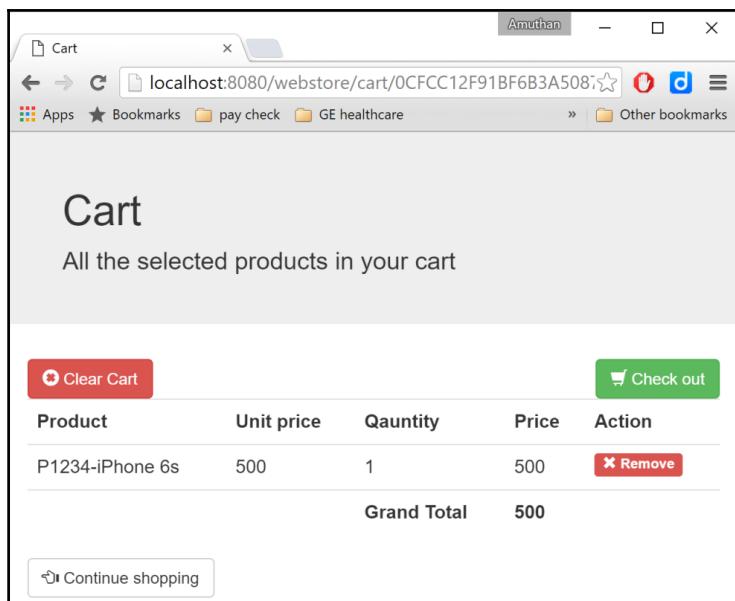
```
<p ng-controller="cartCtrl">
```

10. Now run our application and enter the URL

`http://localhost:8080/webstore/market/product?id=P1234`. You should be able to see the product detail page of a product whose ID is P1234.

11. Now click on the **Order Now** button; an alert message will display, saying **Product successfully added to the cart!!**.

12. Now click on the **View Cart** button; you will see a web page displaying a shopping cart page, as shown in the following screenshot:



Shopping cart page

What just happened?

There are plenty of JavaScript frameworks available to send an Ajax request to the server; we decided to use AngularJS (<https://angularjs.org/>) as our frontend JavaScript library to send Ajax requests. AngularJS is more or less like a frontend MVC framework, but also has the concepts of Model, View, Controller and more. The only difference is that it is designed to work in the frontend using JavaScript.

In step 1, we just created our AngularJS-based Controller called `controllers.js` in `/src/main/webapp/resources/js/`. Remember, we purposely put this file under the `resources` directory because from the client side we want to access this file as a static resource; we don't want to go through Spring MVC Controllers in order to get this file.

Okay, coming to the point, what have we written in `controllers.js`? We have written five frontend Controller methods, namely `refreshCart`, `clearCart`, `initCartId`, `addToCart`, and `removeFromCart`. These methods are used to communicate with the server using Ajax calls. For example, consider the following Controller method:

```
$scope.refreshCart = function(cartId) {
    $http.get('/webstore/rest/cart/' + $scope.cartId)
        .success(function(data) {
            $scope.cart = data;
        });
};
```

Within the `refreshCart` method, using the AngularJS `$http` object, we have sent an HTTP GET request to the URI template `/webstore/rest/cart/'+$scope.cartId`. Based on the value stored in the `$scope.cartId` variable, the actual request will be sent to the REST target URL. For instance, if the `$scope.cartId` contains a value of 111, then a GET request will be sent to the `http://localhost:8080/webstore/rest/cart/111` to get a `cart` object whose ID is 111 as JSON data. Once we get the `cart` object as JSON data, we store it in the frontend Angular model using the `$scope` object, as follows:

```
.success(function(data) {
    $scope.cart = data;
})
```

Similarly, all other AngularJS Controller methods fire some Ajax web requests to the server, and retrieve or update the `cart`. For example, the `addToCart` and `removeFromCart` methods are just adding a `cartItem` and removing a `cartItem` from the `cart` object.

Okay, we have just defined our AngularJS Controller methods, but we have to invoke this method in order to do something useful, which is what we have done in step 2. In step 2, we just defined our regular Spring MVC Controller named `CartController`, which has two request-mapping methods, namely `get` and `getCart`. Whenever a normal web request comes to the URL `http://localhost:8080/webstore/cart`, the `get` method will be invoked, and inside the `get` method we have retrieved the session ID and used it as a cart ID to invoke the `getCart` method. Here we maintained the session ID as the cart ID:

```
@RequestMapping
public String get(HttpServletRequest request) {
    return "redirect:/cart/" + request.getSession(true).getId();
}
```

And within the `getCart` method, we simply stored the `cartId` in the Spring MVC Model and returned a View name as `cart`. We did this kind of setup because we want our application to redirect the request to the correct cart based on the session ID whenever a request comes to the URL `http://localhost:8080/webstore/cart`. Okay, since we have returned a View name as `cart`, our dispatcher servlet would definitely look for a View file called `cart.jsp`. That is why we have created the `cart.jsp` in step 3.

`cart.jsp` just acts as a template for our shopping cart page. The `cart.jsp` page internally uses the AngularJS Controller's methods that we have created in step 1 to communicate with the server. The `ng-repeat` directive of AngularJS would repeat the HTML table rows dynamically based on the `cartItems` available in the `cart`:

```
<tr ng-repeat="item in cart.cartItems">
    <td>{{item.product.productId}}-{{item.product.name}}</td>
    <td>{{item.product.unitPrice}}</td>
    <td>{{item.quantity}}</td>
    <td>{{item.totalPrice}}</td>
    <td><a href="#" class="label label-danger" ng-
        click="removeFromCart(item.product.productId)">
        <span class="glyphicon glyphicon-remove" /></span> Remove
        </a>
    </td>
</tr>
```

And the `ng-click` directive from the remove `<a>` tag would call the `removeFromCart` Controller method. Similarly, to add a `cartItem` to the cart, in `product.jsp` we have added another `ng-click` directive as follows in step 6 to invoke the `addToCart` method:

```
<a href="#" class="btn btn-warning btn-large" ng-
    click="addToCart('${product.productId}')">
    <span class="glyphicon-shopping-cart glyphicon"></span> Order Now
</a>
```

So that's it, we have done everything to roll out our shopping cart in our application. After running our application, we can access our shopping cart under the URL `http://localhost:8080/webstore/cart` and can even add products to the cart from each product detail page as well.

Summary

In this chapter, we learned how to develop REST-based web services using Spring MVC and we also learned how to test those web services using the Postman HTTP client tool. We also covered the basic concepts of HTTP verbs and understood how they are related to standard CRUD operations. Finally, we learned how to use the AngularJS JavaScript framework to send Ajax requests to our server.

In the next chapter, we will see how to integrate the Spring Web Flow framework with Spring MVC.

10

Float Your Application with Web Flow

*When it comes to web application development, reusability and maintenance are two important factors that need to be considered. **Spring Web Flow (SWF)** is an independent framework that facilitates the development of highly configurable and maintainable flow-based web applications.*

In this chapter, we are going to see how to incorporate the Spring Web Flow framework within a Spring MVC application. Spring Web Flow facilitates the development of stateful web applications with controlled navigation flow. After finishing this chapter, you will have an idea about developing flow-based applications using Spring Web Flow.

Working with Spring Web Flow

Spring Web Flow allows us to develop flow-based web applications easily. A flow in a web application encapsulates a series of steps that guides the user through the execution of a business task, such as checking in to a hotel, applying for a job, checking out a shopping cart, and so on. Usually, a flow will have clear start and end points, include multiple HTTP requests/responses, and the user must go through a set of screens in a specific order to complete the flow.

In all our previous chapters, the responsibility for defining a page flow specifically lies with controllers, and we weaved the page flows into individual Controllers and Views; for instance, we usually mapped a web request to a controller, and the controller is the one who decides which logical View to return as a response.

This is simple to understand and sufficient for straightforward page flows, but when web applications get more and more complex in terms of user interaction flows, maintaining a large and complex page flow becomes a nightmare.

If you are going to develop such complex flow-based applications, then SWF is your trusty companion. SWF allows you to define and execute **user interface (UI)** flows within your web application. Without further ado, let's dive straight into Spring Web Flow by defining some page flows in our project.

It is nice that we have implemented a shopping cart in our previous chapter, but it is of no use if we do not provide a checkout facility to finish shopping and perform order processing. Let's do that in two phases. Firstly, we need to create the required backend services, domain objects, and repository implementation, in order to perform order processing (here strictly no web flow related stuff is involved, it's just a supportive backend service that can be used later by web flow definitions in order to complete the checkout process). Secondly, we need to define the actual Spring Web Flow definition, which can use our backend services in order to execute the flow definition. There, we will do the actual web flow configuration and definition.

Time for action – implementing the order processing service

We will start by implementing our order processing backend service first. We proceed as follows:

1. Add the following schema definitions for the `CART` and `CART_ITEM` tables in `create-table.sql`; you can find `create-table.sql` under the folder `src/main/resources/db/sql/`:

```
CREATE TABLE ADDRESS (
    ID INTEGER IDENTITY PRIMARY KEY,
    DOOR_NO VARCHAR(25),
    STREET_NAME VARCHAR(25),
    AREA_NAME VARCHAR(25),
    STATE VARCHAR(25),
    COUNTRY VARCHAR(25),
    ZIP VARCHAR(25),
);

CREATE TABLE CUSTOMER (
    ID INTEGER IDENTITY PRIMARY KEY,
    NAME VARCHAR(25),
    PHONE_NUMBER VARCHAR(25),
```

```
BILLING_ADDRESS_ID INTEGER FOREIGN KEY REFERENCES ADDRESS(ID),  
) ;  
  
CREATE TABLE SHIPPING_DETAIL (  
    ID INTEGER IDENTITY PRIMARY KEY,  
    NAME VARCHAR(25),  
    SHIPPING_DATE VARCHAR(25),  
    SHIPPING_ADDRESS_ID INTEGER FOREIGN KEY REFERENCES ADDRESS(ID),  
) ;  
  
CREATE TABLE ORDERS (  
    ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH  
        1000, INCREMENT BY 1) PRIMARY KEY,  
    CART_ID VARCHAR(50) FOREIGN KEY REFERENCES CART(ID),  
    CUSTOMER_ID INTEGER FOREIGN KEY REFERENCES CUSTOMER(ID),  
    SHIPPING_DETAIL_ID INTEGER FOREIGN KEY REFERENCES  
        SHIPPING_DETAIL(ID),  
) ;
```

2. Also add the following drop table command as the first line in `create-table.sql`:

```
DROP TABLE ORDERS IF EXISTS;  
DROP TABLE CUSTOMER IF EXISTS;  
DROP TABLE SHIPPING_DETAIL IF EXISTS;  
DROP TABLE ADDRESS IF EXISTS;
```

3. Create a class named `Address` under the `com.packt.webstore.domain` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.domain;  
  
import java.io.Serializable;  
  
public class Address implements Serializable{  
  
    private static final long serialVersionUID = -530086768384258062L;  
    private Long id;  
    private String doorNo;  
    private String streetName;  
    private String areaName;  
    private String state;  
    private String country;  
    private String zipCode;  
    public Long getId() {  
        return id;  
    }
```

```
public void setId(Long id) {
    this.id = id;
}
public String getDoorNo() {
    return doorNo;
}
public void setDoorNo(String doorNo) {
    this.doorNo = doorNo;
}
public String getStreetName() {
    return streetName;
}
public void setStreetName(String streetName) {
    this.streetName = streetName;
}
public String getAreaName() {
    return areaName;
}
public void setAreaName(String areaName) {
    this.areaName = areaName;
}
public String getState() {
    return state;
}
public void setState(String state) {
    this.state = state;
}
public String getCountry() {
    return country;
}
public void setCountry(String country) {
    this.country = country;
}
public String getZipCode() {
    return zipCode;
}
public void setZipCode(String zipCode) {
    this.zipCode = zipCode;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}
```

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Address other = (Address) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}
```

4. Create another class named `Customer` under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;

public class Customer implements Serializable{

    private static final long serialVersionUID = 228404048222162898L;
    private Long customerId;
    private String name;
    private Address billingAddress;
    private String phoneNumber;
    public Customer() {
        super();
        this.billingAddress = new Address();
    }
    public Customer(Long customerId, String name) {
        this();
        this.customerId = customerId;
        this.name = name;
    }

    public Long getCustomerId() {
        return customerId;
    }

    public void setCustomerId(long customerId) {
        this.customerId = customerId;
    }
}
```

```
}

public String getName() {
    return name;
}

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

public Address getBillingAddress() {
    return billingAddress;
}

public void setBillingAddress(Address billingAddress) {
    this.billingAddress = billingAddress;
}

public String getPhoneNumber() {
    return phoneNumber;
}

public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
}

public static long getSerialversionuid() {
    return serialVersionUID;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((customerId == null) ? 0 :
    customerId.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Customer other = (Customer) obj;
```

```
        if (customerId == null) {
            if (other.customerId != null)
                return false;
        } else if (!customerId.equals(other.customerId))
            return false;
        return true;
    }
}
```

5. Create one more domain class named `ShippingDetail` under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;
import java.util.Date;
import org.springframework.format.annotation.DateTimeFormat;

public class ShippingDetail implements Serializable{

    private static final long serialVersionUID = 6350930334140807514L;
    private Long id;
    private String name;
    @DateTimeFormat(pattern = "dd/MM/yyyy")
    private Date shippingDate;
    private Address shippingAddress;
    public Long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public ShippingDetail() {
        this.shippingAddress = new Address();
    }

    public String getName() {
        return name;
    }

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

    public Date getShippingDate() {
        return shippingDate;
    }
}
```

```
public void setShippingDate(Date shippingDate) {
    this.shippingDate = shippingDate;
}

public Address getShippingAddress() {
    return shippingAddress;
}

public void setShippingAddress(Address shippingAddress) {
    this.shippingAddress = shippingAddress;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    ShippingDetail other = (ShippingDetail) obj;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    return true;
}
}
```

6. Similarly, create our final domain class, named `Order`, under the same package, and add the following code to it:

```
package com.packt.webstore.domain;

import java.io.Serializable;

public class Order implements Serializable{

    private static final long serialVersionUID =
```

```
-3560539622417210365L;
private Long orderId;
private Cart cart;
private Customer customer;
private ShippingDetail shippingDetail;
public Order() {
    this.customer = new Customer();
    this.shippingDetail = new ShippingDetail();
}

public Long getOrderId() {
    return orderId;
}

public void setOrderId(Long orderId) {
    this.orderId = orderId;
}

public Cart getCart() {
    return cart;
}

public void setCart(Cart cart) {
    this.cart = cart;
}

public Customer getCustomer() {
    return customer;
}

public void setCustomer(Customer customer) {
    this.customer = customer;
}

public ShippingDetail getShippingDetail() {
    return shippingDetail;
}

public void setShippingDetail(ShippingDetail shippingDetail) {
    this.shippingDetail = shippingDetail;
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((orderId == null) ? 0 :
    orderId.hashCode());
```

```
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Order other = (Order) obj;
        if (orderId == null) {
            if (other.orderId != null)
                return false;
        } else if (!orderId.equals(other.orderId))
            return false;
        return true;
    }
}
```

7. Next, create an exception class called `InvalidCartException` under the `com.packt.webstore.exception` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.exception;

public class InvalidCartException extends RuntimeException {

    private static final long serialVersionUID =
    -5192041563033358491L;
    private String cartId;

    public InvalidCartException(String cartId) {
        this.cartId = cartId;
    }

    public String getCartId() {
        return cartId;
    }
}
```

8. Open the `CartRepository` interface from the `com.packt.webstore.domain.repository` package, and add one more method declaration to it as follows:

```
void clearCart(String cartId);
```

9. Now open the `InMemoryCartRepository` implementation class from the `com.packt.webstore.domain.repository.impl` package and add the following method definition to it:

```
@Override  
public void clearCart(String cartId) {  
    String SQL_DELETE_CART_ITEM = "DELETE FROM CART_ITEM WHERE CART_ID  
= :id";  
    Map<String, Object> params = new HashMap<>();  
    params.put("id", cartId);  
    jdbcTemplate.update(SQL_DELETE_CART_ITEM, params);  
}
```

10. Now open the `CartService` interface from the `com.packt.webstore.service` package in the `src/main/java` source folder and add two more method declarations to it as follows:

```
Cart validate(String cartId);  
void clearCart(String cartId);
```

11. Next, open the `CartServiceImpl` implementation class from the `com.packt.webstore.service.impl` package in the `src/main/java` source folder, and add the following method implementations to it:

```
@Override  
public Cart validate(String cartId) {  
    Cart cart = cartRepository.read(cartId);  
    if(cart==null || cart.getCartItems().size()==0) {  
        throw new InvalidCartException(cartId);  
    }  
    return cart;  
}  
  
@Override  
public void clearCart(String cartId) {  
    cartRepository.clearCart(cartId);  
}
```

12. Next, create an interface named `OrderRepository` under the `com.packt.webstore.domain.repository` package in the `src/main/java` source folder, and add a single method declaration to it as follows:

```
package com.packt.webstore.domain.repository;  
  
import com.packt.webstore.domain.Order;
```

```
public interface OrderRepository {  
    long saveOrder(Order order);  
}
```

13. Create an implementation class called `InMemoryOrderRepository` under the `com.packt.webstore.domain.repository.impl` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.domain.repository.impl;  
  
import java.util.HashMap;  
import java.util.Map;  
  
import org.springframework.beans.factory  
.annotation.Autowired;  
import org.springframework.jdbc.core.  
namedparam.MapSqlParameterSource;  
import org.springframework.jdbc.core  
.namedparam.NamedParameterJdbcTemplate;  
import org.springframework.jdbc.core  
.namedparam.SqlParameterSource;  
import org.springframework.jdbc.support.GeneratedKeyHolder;  
import org.springframework.jdbc.support.KeyHolder;  
import org.springframework.stereotype.Repository;  
  
import com.packt.webstore.domain.Address;  
import com.packt.webstore.domain.Customer;  
import com.packt.webstore.domain.Order;  
import com.packt.webstore.domain.ShippingDetail;  
import com.packt.webstore.domain.repository.OrderRepository;  
import com.packt.webstore.service.CartService;  
  
@Repository  
public class InMemoryOrderRepository implements OrderRepository {  
  
    @Autowired  
    private NamedParameterJdbcTemplate jdbcTemplate;  
    @Autowired  
    private CartService cartService;  
  
    @Override  
    public long saveOrder(Order order) {  
        Long customerId = saveCustomer(order.getCustomer());  
        Long shippingDetailId =  
            saveShippingDetail(order.getShippingDetail());  
        order.getCustomer().setCustomerId(customerId);  
        order.getShippingDetail().setId(shippingDetailId);  
    }  
}
```

```
        long createdOrderId = createOrder(order);
        CartService.clearCart(order.getCart().getId());
        return createdOrderId;
    }
    private long saveShippingDetail(ShippingDetail shippingDetail) {
        long addressId =
            saveAddress(shippingDetail.getShippingAddress());
        String SQL = "INSERT INTO
SHIPPING_DETAIL(NAME, SHIPPING_DATE, SHIPPING_ADDRESS_ID) "
            + "VALUES (:name, :shippingDate, :addressId)";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("name", shippingDetail.getName());
        params.put("shippingDate", shippingDetail.getShippingDate());
        params.put("addressId", addressId);

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTemplate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }

    private long saveCustomer(Customer customer) {
        long addressId = saveAddress(customer.getBillingAddress());
        String SQL = "INSERT INTO
CUSTOMER(NAME, PHONE_NUMBER, BILLING_ADDRESS_ID) "
            + "VALUES (:name, :phoneNumber, :addressId)";

        Map<String, Object> params = new HashMap<String, Object>();
        params.put("name", customer.getName());
        params.put("phoneNumber", customer.getPhoneNumber());
        params.put("addressId", addressId);

        SqlParameterSource paramSource = new
        MapSqlParameterSource(params);
        KeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTemplate.update(SQL, paramSource, keyHolder, new
        String[]{"ID"});
        return keyHolder.getKey().longValue();
    }

    private long saveAddress(Address address) {
        String SQL = "INSERT INTO
ADDRESS(DOOR_NO, STREET_NAME, AREA_NAME, STATE, COUNTRY, ZIP) "
            + "VALUES (:doorNo, :streetName, :areaName, :state,
:country, :zip)";
    }
}
```

```
Map<String, Object> params = new HashMap<String, Object>();
params.put("doorNo", address.getDoorNo());
params.put("streetName", address.getStreetName());
params.put("areaName", address.getAreaName());
params.put("state", address.getState());
params.put("country", address.getCountry());
params.put("zip", address.getZipCode());

SqlParameterSource paramSource = new
MapSqlParameterSource(params);
KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplete.update(SQL, paramSource, keyHolder, new
String[]{"ID"});
return keyHolder.getKey().longValue();
}

private long createOrder(Order order) {

String SQL = "INSERT INTO
ORDERS(CART_ID,CUSTOMER_ID,SHIPPING_DETAIL_ID) "
+ "VALUES (:cartId, :customerId, :shippingDetailId)";

Map<String, Object> params = new HashMap<String, Object>();
params.put("id", order.getOrderId());
params.put("cartId", order.getCart().getId());
params.put("customerId", order.getCustomer().getCustomerId());
params.put("shippingDetailId",
order.getShippingDetail().getId());

SqlParameterSource paramSource = new
MapSqlParameterSource(params);
KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplete.update(SQL, paramSource, keyHolder, new
String[]{"ID"});
return keyHolder.getKey().longValue();
}
}
```

14. Create an interface named `OrderService` under the `com.packt.webstore.service` package in the `src/main/java` source folder and add the following method declarations to it as follows:

```
package com.packt.webstore.service;

import com.packt.webstore.domain.Order;

public interface OrderService {
```

```
        Long saveOrder(Order order);  
    }
```

15. Create an implementation class named `OrderServiceImpl` for the previous interface under the `com.packt.webstore.service.impl` package in the `src/main/java` source folder and add the following code to it:

```
package com.packt.webstore.service.impl;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
import com.packt.webstore.domain.Order;  
import com.packt.webstore.domain.repository.OrderRepository;  
import com.packt.webstore.service.OrderService;  
  
@Service  
public class OrderServiceImpl implements OrderService{  
    @Autowired  
    private OrderRepository orderRepository;  
    @Override  
    public Long saveOrder(Order order) {  
        return orderRepository.saveOrder(order);  
    }  
}
```

What just happened?

I guess what we have done so far must already be familiar to you: we have created some domain classes (`Address`, `Customer`, `ShippingDetail`, and `Order`), an `OrderRepository` interface, and its implementation class, `InMemoryOrderRepositoryImpl`, to store processed `Order` domain objects. And finally, we also created the corresponding `OrderService` interface and its implementation class `OrderServiceImpl`.

On the surface, it looks the same as usual, but there are some minute details that need to be explained. If you notice, all the domain classes that we created from steps 1 to 4 have just implemented the `Serializable` interface; not only that, we have even implemented the `Serializable` interface for other existing domain classes as well, such as `Product`, `CartItem`, and `Cart`. This is because later we are going to use these domain objects in Spring Web Flow, and Spring Web Flow is going to store these domain objects in a session for state management between page flows.

Session data can be saved onto a disk or transferred to other web servers during clustering. So when the session object is re-imported from a disk, Spring Web Flow de-serializes the domain object (that is, the form backing bean) to maintain the state of the page. That's why it is a must to serialize the domain object/form backing bean. Spring Web Flow uses a term called **Snapshot** to mention these states within a session.

The remaining steps, steps 6 to 13, are self-explanatory. We have created the `OrderRepository` and `OrderService` interfaces and their corresponding implementations, `InMemoryOrderRepositoryImpl` and `OrderServiceImpl`. The purpose of these classes is to save the `Order` domain object. The `saveOrder` method from `OrderServiceImpl` just deletes the corresponding `CartItem` objects from `CartRepository`, after successfully saving the `order` domain object. Now we have successfully created all the required backend services and domain objects, in order to kick off our Spring Web Flow configuration and definition.

Time for action – implementing the checkout flow

We will now add Spring Web Flow support to our project and define the checkout flow for our shopping cart:

1. Open `pom.xml`; you can find `pom.xml` under the root directory of the project.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter **Group Id** as `org.springframework.webflow`, **Artifact Id** as `spring-webflow`, **Version** as `2.4.2.RELEASE`, select **Scope** as `compile`, click on the **OK** button, and save `pom.xml`.
4. Create a directory structure `flows/checkout/` under the `src/main/webapp/WEB-INF/` directory, create an XML file called `checkout-flow.xml` in `flows/checkout/`, add the following content into it, and save it:

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

    <var name="order" class="com.packt.webstore.domain.Order"
    />

    <action-state id="addCartToOrder">
```

```
<evaluate expression="cartServiceImpl.validate
(requestParameters.cartId)"
    result="order.cart" />
<transition to="invalidCartWarning"
    on-
exception="com.packt.webstore.exception
.InvalidCartException" />
    <transition to="collectCustomerInfo" />
</action-state>

<view-state id="collectCustomerInfo"
view="collectCustomerInfo.jsp" model="order">
    <transition on="customerInfoCollected"
to="collectShippingDetail" />
</view-state>

<view-state id="collectShippingDetail" model="order">
    <transition on="shippingDetailCollected"
to="orderConfirmation" />
    <transition on="backToCollectCustomerInfo"
to="collectCustomerInfo" />
</view-state>

<view-state id="orderConfirmation">
    <transition on="orderConfirmed" to="processOrder" />
    <transition on="backToCollectShippingDetail"
to="collectShippingDetail" />
</view-state>
<action-state id="processOrder">
    <evaluate expression="orderServiceImpl.saveOrder(order) "
result="order.orderId"/>
    <transition to="thankCustomer" />
</action-state>
<view-state id="invalidCartWarning">
    <transition to="endState"/>
</view-state>
<view-state id="thankCustomer" model="order">
    <transition to="endState"/>
</view-state>

<end-state id="endState"/>

<end-state id="cancelCheckout" view = "checkOutCancelled.jsp"/>
<global-transitions>
    <transition on = "cancel" to="cancelCheckout" />
</global-transitions>
</flow>
```

5. Now, create a web flow configuration class called `WebFlowConfig` under the `com.packt.webstore.config` package in the `src/main/java` source folder, and add the following code to it:

```
package com.packt.webstore.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.webflow.config
    .AbstractFlowConfiguration;
import org.springframework.webflow.definition
    .registry.FlowDefinitionRegistry;
import org.springframework.webflow.executor.FlowExecutor;
import org.springframework.webflow.mvc.servlet.FlowHandlerAdapter;
import org.springframework.webflow.mvc.servlet.FlowHandlerMapping;

@Configuration
public class WebFlowConfig extends AbstractFlowConfiguration {

    @Bean
    public FlowDefinitionRegistry flowRegistry() {
        return getFlowDefinitionRegistryBuilder()
            .setBasePath("/WEB-INF/flows")
            .addFlowLocationPattern("/**/*-flow.xml")
            .build();
    }
    @Bean
    public FlowExecutor flowExecutor() {
        return getFlowExecutorBuilder(flowRegistry()).build();
    }
    @Bean
    public FlowHandlerMapping flowHandlerMapping() {
        FlowHandlerMapping handlerMapping = new FlowHandlerMapping();
        handlerMapping.setOrder(-1);
        handlerMapping.setFlowRegistry(flowRegistry());
        return handlerMapping;
    }

    @Bean
    public FlowHandlerAdapter flowHandlerAdapter() {
        FlowHandlerAdapter handlerAdapter = new FlowHandlerAdapter();
        handlerAdapter.setFlowExecutor(flowExecutor());
        handlerAdapter.setSaveOutputToFlashScopeOnRedirect(true);
        return handlerAdapter;
    }
}
```

What just happened?

From steps 1 to 3, we just added the Spring Web Flow dependency to our project through Maven configuration. It will download and configure all the required web flow-related JARs for our project. In step 4, we created our first flow definition file, called `checkout-flow.xml`, under the `/src/main/webapp/WEB-INF/flows/checkout/` directory.

Spring Web Flow uses the flow definition file as a basis for executing the flow. In order to understand what has been written in this file, we need to get a clear idea of some of the basic concepts of Spring Web Flow. We will learn about those concepts in a little bit, and then we will come back to `checkout-flow.xml` to understand it better.

Understanding flow definitions

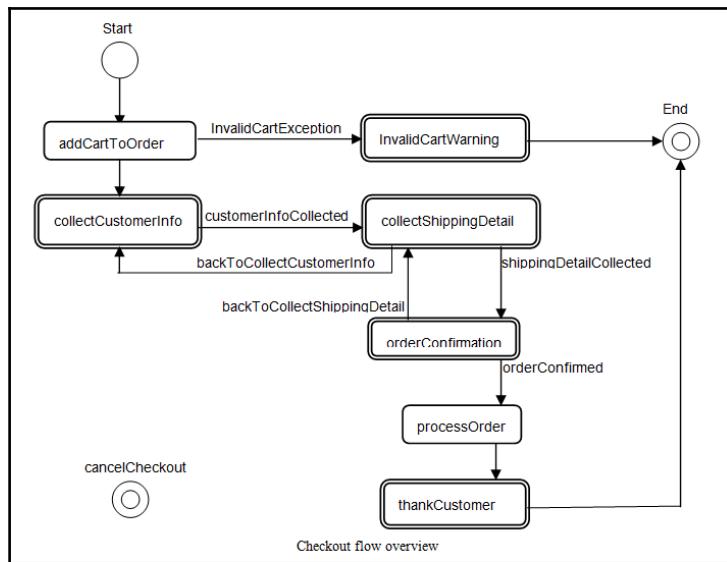
A flow definition is composed of a set of states. Each state will have a unique ID in the flow definition. There are five types of state available in Spring Web Flow:

- **start-state**: Each flow must have a single start state, which helps in creating the initial state of the flow. Note that if the `start-state` is not specified, the very first defined state within the flow definition file becomes the start state.
- **action-state**: A flow can have many action states; an `action-state` executes a particular action. An action normally involves interacting with backend services, such as executing some methods in Spring managed beans; Spring Web Flow uses the **Spring Expression Language** to interact with the backend service beans.
- **view-state**: A `view-state` defines a logical View and Model to interact with the end user. A web flow can have multiple `view-states`. If the `View` attribute is not specified, then the ID of the `view-state` acts as the logical View name.
- **decision-state**: This is used to branch the flow; based on a test condition it routes the transition to the next possible state.
- **subflow-state**: It is an independent flow that can be reused from inside another flow. When an application enters a subflow, the main flow is paused until the subflow completes.
- **end-state**: This state denotes the end of a flow execution. A web flow can have multiple end states; through the `view` attribute of an `end-state`, we can specify a View that will be rendered when its end state is reached.

We have just learned that a flow definition is composed of a set of states, but in order to move from one state to another, we need to define transitions in states. Each state in a web flow (except for the start and end states) defines a number of transitions to move from one state to another. A transition can be triggered by an event signaled by the state.

Understanding checkout flow

Okay, we just got the minimum required introduction to Spring Web Flow concepts; there are plenty of advanced concepts out there to master in Spring Web Flow. We are not going to see all those things, because that itself deserves a separate book. As of now, this is enough to understand the `checkout-flow.xml` flow definition file. But before that, we will provide a quick overview of our checkout flow. The following diagram will give you the overall idea of the checkout flow that we just implemented:



Our checkout flow diagram has a start state and an end state; each rounded rectangle in the diagram defines an action state and each double-line-bordered rounded rectangle defines a view state. Each arrowed line defines transition, and the name associated with it defines the event that causes that particular transition. The `checkout-flow.xml` file just contains this flow in an XML representation.

If you open the `checkout-flow.xml` file, the first tag you encounter within the `<flow>` tag is the `<var>` tag:

```
<var name="order" class="com.packt.webstore.domain.Order" />
```

The `<var>` tag creates a variable in a flow. This variable will be available to all states in a flow, which means we can reference and use this variable inside any state within the flow. In the preceding `<var>` tag, we just created a new instance of the `Order` class and stored it in a variable called `order`.

The next thing we defined within the `checkout-flow.xml` file is the `<action-state>` definition. As we already learned, action states are normally used to invoke backend services, so in the following `<action-state>` definition we have just invoked the `validate` method of the `cartServiceImpl` object and stored the result in the `order.cart` object:

```
<action-state id="addCartToOrder">
    <evaluate expression =
"cartServiceImpl.validate(requestParameters.cartId)"
        result="order.cart" />
    <transition to="invalidCartWarning" on-exception =
"com.packt.webstore.exception.InvalidCartException" />
    <transition to="collectCustomerInfo" />
</action-state>
```

As we already defined the `order` variable at the start of the flow, it will be available in every state of this flow. So we have used that variable (`order.cart`) in the `<evaluate>` tag to store the result of this evaluated expression: `cartServiceImpl.validate(requestParameters.cartId)`.

The `validate` method of `cartServiceImpl` tries to read a `cart` object based on the given `cartId`. If it finds a valid `cart` object, then it returns that. Otherwise, it will throw an `InvalidCartException`; in such a case we route the transition to another state whose ID is `invalidCartWarning`:

```
<transition to="invalidCartWarning" on-exception =
"com.packt.webstore.exception.InvalidCartException" />
```

If no such exception is thrown from the expression evaluation, we naturally transit from the `addCartToOrder` state to the `collectCustomerInfo` state:

```
<transition to="collectCustomerInfo" />
```

If you notice the `collectCustomerInfo` state, it is nothing but a view state in `checkout-flow.xml`. We defined the View that needs to be rendered via the `view` attribute and the Model that needs to be attached via the `model` attribute:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
```

```
model="order">
    <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Upon reaching this view state, Spring Web Flow renders the `collectCustomerInfo` View and waits for the user to interact; once the user has entered the customer info details and pressed the **submit** button, it will resume its transition to the `collectShippingDetail` view state. As we already learned, a transition can be triggered via an event, so here the transition to the `collectShippingDetail` state would get triggered when the `customerInfoCollected` event is triggered. How do we fire this event (`customerInfoCollected`) from the View? We will see later in this chapter:

```
<transition on="customerInfoCollected" to="collectShippingDetail" />
```

The next state defined within the checkout flow is `collectShippingDetail`; again this is also a view state, and it has two transitions back and forth: one is to go back to the `collectCustomerInfo` state and the next is to go forward to the `orderConfirmation` state:

```
<view-state id="collectShippingDetail" model="order">
    <transition on="shippingDetailCollected" to="orderConfirmation" />
    <transition on="backToCollectCustomerInfo" to="collectCustomerInfo" />
</view-state>
```

Note that here in the `collectShippingDetail` state, we haven't mentioned the `view` attribute; in that case Spring Web Flow would consider the `id` of the view state to be the View name.

The `orderConfirmation` state definition doesn't need much explanation. It is more like the `collectShippingDetail` view state, where we have furnished all the order-related details and we ask the user to confirm them; upon confirmation, we move to the next state, which is `processOrder`:

```
<view-state id="orderConfirmation">
    <transition on="orderConfirmed" to="processOrder" />
    <transition on="backToCollectShippingDetail" to =
    "collectShippingDetail" />
</view-state>
```

Next, the `processOrder` state is an action state that interacts with the `orderServiceImpl` object to save the `order` object. Upon successfully saving the `order` object, it stores the `order ID` in the flow variable (`order.orderId`) and transits to the next state, which is `thankCustomer`:

```
<action-state id="processOrder">
```

```
<evaluate expression="orderServiceImpl.saveOrder(order)"  
result="order.orderId"/>  
    <transition to="thankCustomer" />  
</action-state>
```

The `thankCustomer` state is a view state that simply shows a thank you message with the confirmed order ID to the end user, and transits to the end state:

```
<view-state id="thankCustomer" model="order">  
    <transition to="endState"/>  
</view-state>
```

In our checkout flow, we have two end states; one is the normal end state where the flow execution arrives naturally after the flow ends, and the other one is the end state when the user presses the **Cancel** button in any of the Views:

```
<end-state id="endState"/>  
<end-state id="cancelCheckout" view="checkOutCancelled.jsp"/>
```

Note in the `cancelCheckout` end state, we have specified the name of the landing page via the `view` attribute. The transition to the `cancelCheckout` end state happened through the global transitions configuration:

```
<global-transitions>  
    <transition on = "cancel" to="cancelCheckout" />  
</global-transitions>
```

A global transition is for sharing some common transitions between states. Rather than repeating the transition definition every time within the state definition, we can define it within one global transition so that that transition will be available implicitly for every state in the flow. In our case, the end user may cancel the checkout process in any state; that's why we have defined the transition to the `cancelCheckout` state in `global-transitions`.

Okay, we have totally understood the checkout flow definition (`checkout-flow.xml`). Now our Spring MVC should read this file during the boot up of our application, so that it can be ready to dispatch any flow-related request to the Spring Web Flow framework. We are able to do this via some web flow configurations, as mentioned in step 5.

In step 5, we have created beans for `FlowExecutor` and the `FlowDefinitionRegistry`. As its name implies, the `flowExecutor` executes a flow based on the given flow definition. The `flowExecutor` gets its flow definition from a `flowDefinitionRegistry` bean. We can configure as many flow definitions in a `flowDefinitionRegistry` as we want.

A `flowDefinitionRegistry` is a collection of flow definitions. When a user enters a flow, the flow executor creates and launches an exclusive flow instance for that user based on the flow definition:

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
    return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF/flows")
        .addFlowLocationPattern("/**/*-flow.xml")
        .build();
}
```

In the preceding web flow configuration, we created the `flowDefinitionRegistry` bean, whose base-path is `/WEB-INF/flows`, so we need to put all our flow definitions under the `/WEB-INF/flows` directory in order to get picked up by the `flowDefinitionRegistry`. That's why in step 4 we created our `checkout-flow.xml` under the `src/main/webapp/WEB-INF/flows/checkout/` directory. As I already mentioned, a `flowDefinitionRegistry` can have many flow definitions; each flow definition is identified by its ID within the `flowDefinitionRegistry`. In our case, we have added a single flow definition, whose ID is `checkout` and whose relative location is `/checkout/checkout-flow.xml`.

One important thing to understand before we wind up web flow configuration is the ID of a flow definition forms the relative URL to invoke the flow. By this what I mean is that in order to invoke our `checkout` flow via a web request, we need to fire a GET request to the `http://localhost:8080/webstore/checkout` URL, because our flow ID is `checkout`. Also, in our flow definition (`checkout-flow.xml`), we haven't configured any start-state, so the first state definition (which is the `addCartToOrderaction-state`) will becomes the start-state, and `addCartToOrderaction-state`, expecting a `cartId`, should be present in the request parameter of the invoking URL:

```
<action-state id="addCartToOrder">
    <evaluate expression =
    "cartServiceImpl.validate(requestParameters.cartId)" result="order.cart" />
    <transition to="invalidCartWarning" on-
    exception="com.packt.webstore.exception.InvalidCartException" />
    <transition to="collectCustomerInfo" />
</action-state>
```

So the actual URL that can invoke this flow would be something similar to `http://localhost:8080/webstore/checkout?cartId=55AD1472D4EC`, where the part after the question mark (`cartId=55AD1472D4EC`) is considered as a request parameter.

It is good that we have defined our checkout flow and configured it with the Spring Web Flow, but we need to define two more beans named `flowHandlerMapping` and `flowHandlerAdapter` in our `WebFlowConfig.java`, to dispatch all flow-related requests to the `flowExecutor`. We did that as follows:

```
@Bean
public FlowHandlerMapping flowHandlerMapping() {
    FlowHandlerMapping handlerMapping = new FlowHandlerMapping();
    handlerMapping.setOrder(-1);
    handlerMapping.setFlowRegistry(flowRegistry());
    return handlerMapping;
}

@Bean
public FlowHandlerAdapter flowHandlerAdapter() {
    FlowHandlerAdapter handlerAdapter = new FlowHandlerAdapter();
    handlerAdapter.setFlowExecutor(flowExecutor());
    handlerAdapter.setSaveOutputToFlashScopeOnRedirect(true);
    return handlerAdapter;
}
```

`flowHandlerMapping` creates and configures handler mapping, based on the flow ID for each defined flow from `flowRegistry`. `flowHandlerAdapter` acts as a bridge between the dispatcher servlet and Spring Web Flow, in order to execute the flow instances.

Pop quiz – web flow

Consider the following web flow registry configuration; it has a single flow definition file, namely `validate.xml`. How will you form the URL to invoke the flow?

```
@Bean
public FlowDefinitionRegistry flowRegistry() {
    return getFlowDefinitionRegistryBuilder()
        .setBasePath("/WEB-INF/flows")
        .addFlowLocation("/customer/validate.xml", "validateCustomer")
        .build();
}
```

1. `http://localhost:8080/webstore/customer/validate`
2. `http://localhost:8080/webstore/validate`
3. `http://localhost:8080/webstore/validateCustomer`

Consider the following flow invoking URL:

`http://localhost:8080/webstore/validate?customerId=C1234`

In a flow definition file, how will you retrieve the `customerId` HTTP request parameter?

1. `<evaluate expression = "requestParameters.customerId" result = "customerId" />`
2. `<evaluate expression = "requestParameters(customerId)" result = "customerId" />`
3. `<evaluate expression = "requestParameters[customerId]" result = "customerId" />`

Time for action – creating Views for every view state

We have done everything to roll out our checkout flow, but one last thing is pending: creating all the Views that need to be used in the view states of our checkout flow. In total, we have six view states in our flow definition (`collectCustomerInfo`, `collectShippingDetail`, `orderConfirmation`, `invalidCartWarning`, `thankCustomer`, and `cancelCheckout`), so we need to create six JSP files. Let's create all of them:

1. Create a JSP View file called `collectCustomerInfo.jsp` under the `src/main/webapp/WEB-INF/flows/checkout/` directory, add the following code snippet to it, and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<link rel="stylesheet"
      href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
      /bootstrap.min.css">
<title>Customer</title>
</head>
<body>
  <section>
    <div class="jumbotron">
      <div class="container">
```

```
<h1>Customer</h1>
<p>Customer details</p>
</div>
</div>
</section>
<section class="container">
<form:form modelAttribute="order.customer" class="form-
horizontal">
<fieldset>
<legend>Customer Details</legend>

<div class="form-group">
<label class="control-label col-lg-2"
for="name">Name</label>
<div class="col-lg-10">
<form:input id="name" path="name"
type="text" class="form:input-large" />
</div>
</div>

<div class="form-group">
<label class="control-label col-lg-2"
for="doorNo">Door No</label>
<div class="col-lg-10">
<form:input id="doorNo"
path="billingAddress.doorNo" type="text"
class="form:input-large" />
</div>
</div>

<div class="form-group">
<label class="control-label col-lg-2"
for="streetName">Street Name</label>
<div class="col-lg-10">
<form:input id="streetName"
path="billingAddress.streetName."
type="text"
class="form:input-large" />
</div>
</div>

<div class="form-group">
<label class="control-label col-lg-2"
for="areaName">Area Name</label>
<div class="col-lg-10">
<form:input id="areaName"
path="billingAddress.areaName" type="text"
class="form:input-large" />
```

```
</div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2" for="state">State</label>
    <div class="col-lg-10">
        <form:input id="state" path="billingAddress.state" type="text" class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2" for="country">country</label>
    <div class="col-lg-10">
        <form:input id="country" path="billingAddress.country" type="text" class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2" for="zipCode">Zip Code</label>
    <div class="col-lg-10">
        <form:input id="zipCode" path="billingAddress.zipCode" type="text" class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2" for="phoneNumber">Phone Number</label>
    <div class="col-lg-10">
        <form:input id="phoneNumber" path="phoneNumber" type="text" class="form:input-large" />
    </div>
</div>

<input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}"/>
<div class="form-group">
    <div class="col-lg-offset-2 col-lg-10">
        <input type="submit" id="btnAdd" class="btn btn-primary" />
    </div>
</div>
```

```
        value="Add"
        name="_eventId_customerInfoCollected" />
    <button id="btnCancel" class="btn btn-
        default" name="_eventId_cancel">Cancel
    </button>
</div>
</div>

</fieldset>
</form:form>
</section>
</body>
</html>
```

2. Similarly, create one more JSP View file called `collectShippingDetail.jsp` under the same directory, add the following code snippet to it, and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet"
      href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css
      /bootstrap.min.css">
<title>Customer</title>
</head>
<body>
<section>
    <div class="jumbotron">
        <div class="container">
            <h1>Shipping</h1>
            <p>Shipping details</p>
        </div>
    </div>
</section>
<section class="container">
    <form:form modelAttribute="order.shippingDetail"
               class="form-horizontal">
        <fieldset>
            <legend>Shipping Details</legend>
            <div class="form-group">
                <label class="control-label col-lg-2 col-lg-2"
                      for="name" />Name</label>
```

```
<div class="col-lg-10">
    <form:input id="name" path="name"
        type="text" class="form:input-large" />
</div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2 col-lg-2"
        for="shippingDate" />shipping
    Date (dd/mm/yyyy) </label>
    <div class="col-lg-10">
        <form:input id="shippingDate"
            path="shippingDate" type="text"
            class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="doorNo">Door No</label>
    <div class="col-lg-10">
        <form:input id="doorNo"
            path="shippingAddress.doorNo" type="text"
            class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="streetName">Street Name</label>
    <div class="col-lg-10">
        <form:input id="streetName"
            path="shippingAddress.streetName."
            type="text"
            class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="areaName">Area Name</label>
    <div class="col-lg-10">
        <form:input id="areaName"
            path="shippingAddress.areaName"
            type="text"
            class="form:input-large" />
    </div>
</div>
```

```
<div class="form-group">
    <label class="control-label col-lg-2"
        for="state">State</label>
    <div class="col-lg-10">
        <form:input id="state"
            path="shippingAddress.state" type="text"
            class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="country">country</label>
    <div class="col-lg-10">
        <form:input id="country"
            path="shippingAddress.country" type="text"
            class="form:input-large" />
    </div>
</div>

<div class="form-group">
    <label class="control-label col-lg-2"
        for="zipCode">Zip Code</label>
    <div class="col-lg-10">
        <form:input id="zipCode"
            path="shippingAddress.zipCode" type="text"
            class="form:input-large" />
    </div>
</div>

<input type="hidden" name="_flowExecutionKey"
    value="${flowExecutionKey}">

<div class="form-group">
    <div class="col-lg-offset-2 col-lg-10">
        <button id="back" class="btn btn-default"
            name="_eventId_backToCollectCustomerInfo">
            back</button>
        <input type="submit" id="btnAdd" class="btn
            btn-primary"
            value="Add"
            name="_eventId_shippingDetailCollected"/>
        <button id="btnCancel" class="btn btn-
            default"
            name="_eventId_cancel">Cancel</button>
    </div>
</div>
```

```
</fieldset>
</form:form>
</section>
</body>
</html>
```

3. Create one more JSP View file called `orderConfirmation.jsp` to confirm the order by the user, under the same directory, then add the following code snippet to it and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt"
uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet"
      href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css
      /bootstrap.min.css">
<title>Order Confirmation</title>
</head>

<body>

<section>
  <div class="jumbotron">
    <div class="container">
      <h1>Order</h1>
      <p>Order Confirmation</p>
    </div>
  </div>
</section>
<div class="container">
  <div class="row">
    <form:form modelAttribute="order" class="form-horizontal">
      <input type="hidden" name="_flowExecutionKey"
            value="${flowExecutionKey}" />

      <div
        class="well col-xs-10 col-sm-10 col-md-6 col-
        xs-offset-1 col-sm-offset-1 col-md-offset-3">
        <div class="text-center">
```

```
        <h1>Receipt</h1>
    </div>
    <div class="row">
        <div class="col-xs-6 col-sm-6 col-md-6">
            <address>
                <strong>Shipping Address</strong> <br>
                ${order.shippingDetail.name}<br>
                ${order.shippingDetail.shippingAddress.doorNo},
                ${order.shippingDetail.shippingAddress.streetName}
                <br>
                ${order.shippingDetail.shippingAddress.areaName},
                ${order.shippingDetail.shippingAddress.state}
                <br>
                ${order.shippingDetail.shippingAddress.country},
                ${order.shippingDetail.shippingAddress.zipCode}
                <br>
            </address>
        </div>
        <div class="col-xs-6 col-sm-6 col-md-6 text-right">
            <p>
                <em>Shipping Date</em>
                <fmt:formatDate type="date"
value ="${order.shippingDetail.shippingDate}" /></em>
            </p>
        </div>
    </div>
    <div class="row">
        <div class="col-xs-6 col-sm-6 col-md-6">
            <address>
                <strong>Billing Address</strong> <br>
                ${order.customer.name}<br>
                ${order.customer.billingAddress.doorNo},
                ${order.customer.billingAddress.streetName}
                <br>
                ${order.customer.billingAddress.areaName},
                ${order.customer.billingAddress.state}
                <br>
                ${order.customer.billingAddress.country},
                ${order.customer.billingAddress.zipCode}
                <br> <abbr>P:</abbr>
                ${order.customer.phoneNumber}
            </address>
        </div>
    </div>
    <div class="row">
        <table class="table table-hover">
```

```
<thead>
    <tr>
        <th>Product</th>
        <th>#</th>
        <th class="text-center">Price</th>
        <th class="text-center">Total</th>
    </tr>
</thead>
<tbody>
    <c:forEach var="cartItem"
        items="${order.cart.cartItems}">
        <tr>
            <td class="col-md-9">
                <em>${cartItem.product.name}</em></td>
                <td class="col-md-1"
                    style="text-align: center">
                    ${cartItem.quantity}</td>
                <td class="col-md-1 text-
center">${cartItem.product.unitPrice}
                </td>
                <td class="col-md-1 text-
center">${cartItem.totalPrice}
                </td>
            </tr>
        </c:forEach>

        <tr>
            <td> </td>
            <td> </td>
            <td class="text-right"><h4>
                <strong>Grand Total:
                </strong>
            </h4></td>
            <td class="text-center text-
danger"><h4>
                <strong>${order.cart.grandTotal}
                </strong>
            </h4></td>
        </tr>
    </tbody>
</table>
<button id="back" class="btn btn-default"
name="_eventId_backToCollectShippingDetail">
back</button>

<button type="submit" class="btn btn-
success"
name="_eventId_orderConfirmed">
```

```
        Confirm    <span class="glyphicon glyphicon-chevron-right"></span>
    </button>
    <button id="btnCancel" class="btn btn-default"
            name="_eventId_cancel">Cancel</button>
    </div>
    </div>
    </form:>form>
    </div>
</div>
</body>
</html>
```

4. Next, we need to create another JSP View file called `invalidCartWarning.jsp` to show an error message in case the cart is empty at the checkout; add the following code snippet to `invalidCartWarning.jsp` and save it:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet"
      href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger"> Empty Cart</h1>
            </div>
        </div>
    </section>

    <section>
        <div class="container">
            <p>
                <a href=<spring:url value="/market/products" />">
                    <span class="glyphicon-hand-left glyphicon">
                        </span> products
                </a>
            </p>
        </div>
    </section>
</body>
```

```
        </div>
    </section>
</body>
</html>
```

5. To thank the customer after a successful checkout flow, we need to create one more JSP View file, called `thankCustomer.jsp`, as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
    <section>
        <div class="jumbotron">
            <div class="container">
                <h1 class="alert alert-danger"> Thank you</h1>
                <p>Thanks for the order. your order will be
                    delivered to you on
                <fmt:formatDate type="date"
                    value="${order.shippingDetail.shippingDate}" />!
                </p>
                <p>Your Order Number is ${order.orderId}</p>
            </div>
        </div>
    </section>

    <section>
        <div class="container">
            <p>
                <a href=<spring:url value="/market/products" />
                    class="btn btn-primary">
                    <span class="glyphicon-hand-left glyphicon">
                    </span> products
                </a>
            </p>
        </div>
    </section>
```

```
</body>
</html>
```

6. If the user cancels the checkout in any of the Views, we need to show the checkout cancelled message; for that we need to have a JSP file called checkOutCancelled.jsp as follows:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

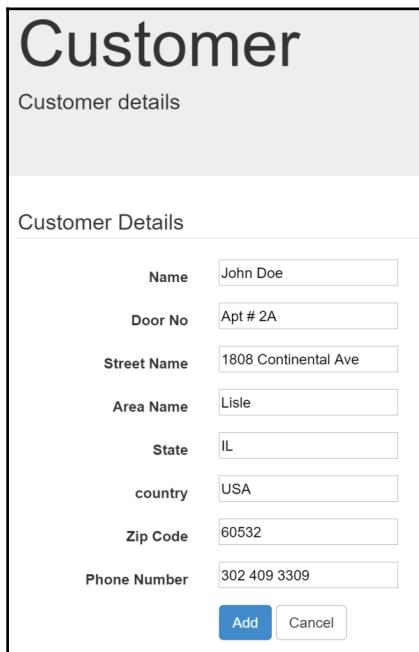
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<link rel="stylesheet"
href="//netdna.bootstrapcdncdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">
<title>Invalid cart </title>
</head>
<body>
<section>
<div class="jumbotron">
<div class="container">
<h1 class="alert alert-danger">check out cancelled</h1>
<p>Your Check out process cancelled! you may
continue shopping..</p>
</div>
</div>
</section>

<section>
<div class="container">
<p>
<a href="
```

7. As a last step, open cart.jsp from `src\main\webapp\WEB-INF\views\` and assign the value `<spring:url value="/checkout?cartId=${cartId}" />` to the href attribute of the checkout link, as follows:

```
<a href= "<spring:url value="/checkout?cartId=${cartId}"/>"  
class="btn btn-success pull-right">  
    <span class="glyphicon-shopping-cart glyphicon">  
    </span> Check out  
</a>
```

8. With all these steps executed, now run the application and enter the `http://localhost:8080/webstore/market/products` URL. Next, click on the **Details** button of any of the products and click on the **Order Now** button from the product details page to add products to the shopping cart. Now go to the cart page by clicking the **View Cart** button; you will be able to see our **Checkout** button on that page. Just click on the **Checkout** button; you will be able to see a web page as follows to collect the customer info:



The image shows a modal dialog titled "Customer" with a sub-section "Customer details". Inside, there's a heading "Customer Details" followed by a list of input fields for customer information:

Name	John Doe
Door No	Apt # 2A
Street Name	1808 Continental Ave
Area Name	Lisle
State	IL
Country	USA
Zip Code	60532
Phone Number	302 409 3309

At the bottom are two buttons: "Add" (blue) and "Cancel".

Customer details collection form

9. After furnishing all the customer details, if you press the **Add** button, Spring Web Flow will take you to the next view state, which is to collect shipping details, and so on up to confirming the order. Upon confirming the order, Spring Web Flow will show you the thank you message View as the end state.

What just happened?

What we have done from steps 1 to 6 is a repeated task, creating JSP View files for each view state. We defined the `model` attribute for each view state in `checkout-flow.xml`:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
model="order">
    <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

That Model object gets bound to the View via the `modelAttribute` attribute of the `<form:form>` tag, as follows:

```
<form:form modelAttribute="order.customer" class="form-horizontal">
    <fieldset>
        <legend>Customer Details</legend>

        <div class="form-group">
            <label class="control-label col-lg-2"
for="name">Name</label>
            <div class="col-lg-10">
                <form:input id="name" path="name" type="text"
class="form-input-large" />
            </div>
        </div>
    </fieldset>
</form:form>
```

In the preceding snippet of `collectCustomerInfo.jsp`, you can see that we have bound the `<form:input>` tag to the `name` field of the `customer` object, which comes from the Model object (`order.customer`). Similarly, we have bound the `shippingDetail` and `order` objects to `collectShippingDetail.jsp` and `orderConfirmation.jsp` respectively.

It's good that we have bound the `Order`, `Customer`, and `ShippingDetail` objects to the Views, but what will happen after clicking the **submit** button in each View, or say, the **Cancel** or **back** buttons? To know the answer, we need to investigate the following code snippet from `collectCustomerInfo.jsp`:

```
<input type="submit" id="btnAdd" class="btn btn-primary" value="Add"
name="_eventId_customerInfoCollected" />
```

On the surface, the preceding `<input>` tag just acts as a submit button, but the real difference comes from the `name` attribute (`name="_eventId_customerInfoCollected"`). We have assigned the value `_eventId_customerInfoCollected` to the `name` attribute of the `<input>` tag for a purpose. The purpose is to instruct Spring Web Flow to raise an event on submission of this form.

When this form is submitted, Spring Web Flow raises an event based on the `name` attribute. Since we have assigned a value with the `_eventId_` prefix (`_eventId_customerInfoCollected`), Spring Web Flow recognizes it as the event name and raises an event with the name `customerInfoCollected`.

As we already learned, transitions from one state to another state happen with the help of events, so on submitting the `collectCustomerInfo` form, Spring Web Flow takes us to the next view state, which is `collectShippingDetail`:

```
<view-state id="collectCustomerInfo" view="collectCustomerInfo.jsp"
model="order">
    <transition on="customerInfoCollected" to="collectShippingDetail" />
</view-state>
```

Similarly, we raise events when clicking the **Cancel** or **back** buttons; see the following code snippet from `collectShippingDetail.jsp`:

```
<button id="back" class="btn btn-default"
name="_eventId_backToCollectCustomerInfo"name="_eventId_cancel"
```

Okay, so we understand how to raise Spring Web Flow events from a View to directly transition from one view state to another state. However, we need to understand one more important concept regarding Spring Web Flow execution—each flow execution is identified by the flow execution key at runtime. During flow execution, when a view state is entered, the flow execution pauses and waits for the user to perform some action (such as entering some data in the form). When the user submits the form or chooses to cancel the form, the flow execution key is sent along with the form data, in order for the flow to resume where it left off. We can do that with the help of the hidden `<input>` tag, as follows:

```
<input type="hidden" name="_flowExecutionKey" value="${flowExecutionKey}" />
```

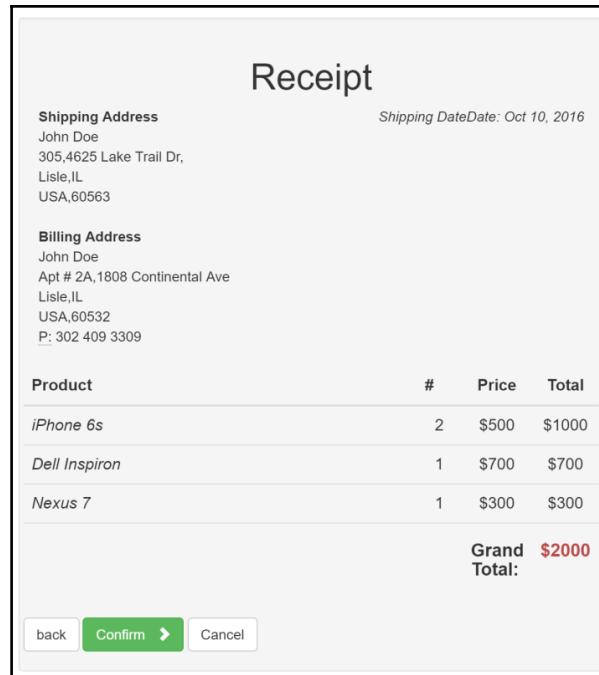
If you look carefully, we have this tag in every flow-related View file, such as `collectCustomerInfo.jsp`, `collectShippingDetail.jsp`, and so on. Spring Web Flow stores a unique flow execution key under the model attribute name `flowExecutionKey` in every flow-related View; we need to store this value in the form variable called `_flowExecutionKey` in order for it to be identified by Spring Web Flow.

So that's all about the View files associated with our checkout flow definition. But we need to invoke the flow upon clicking the **Checkout** button from the cart page. As we already learned to invoke our checkout flow, we need to fire a web request with the cart ID as the request parameter, so in step 7 we have changed the `href` attribute of the checkout link, to

form a request URL something similar to

`http://localhost:8080/webstore/checkout?cartId=55AD1472D4EC.`

So now, if you click on the **Checkout** button after selecting some products and placing them in the shopping cart, you will able to initiate the checkout flow; the following screenshot shows the order confirmation page that will be shown as an outcome of reaching the `orderConfirmation` state:



Order confirmation view state

Have a go hero – adding a decision state

Although we have finished our checkout flow, there is still a bit of room to improve the flow. Every time the checkout flow starts, it collects the customer details, but what if a returning customer makes an order—they probably don't want to fill in their details each time. You can autofill returning customer details from existing records. You can also update the inventory of products upon confirmation.

Here are some of the improvements you can make to avoid collecting customer details for returning customers:

- Create a customer repository and service layer to store, retrieve, and find customer objects. You can probably have methods such as the following in your `CustomerRepository` and `CustomerService` interfaces, and in their corresponding implementation classes:
 - `public void saveCustomer(Customer customer)`
 - `public Customer getCustomer(String customerId)`
 - `public Boolean isCustomerExist(String customerId)`
- Define a view state in `checkout-flow.xml` to collect customer IDs. Don't forget to create the corresponding JSP View file to collect customer IDs.
- Define a decision state in `checkout-flow.xml` to check whether a customer exists in `CustomerRepository`, through `CustomerService`. Based on the returning Boolean value, direct the transition to collect the customer details view state or prefill the `order.customer` object from `CustomerRepository`. The following is the sample decision state:

```
<decision-stateid="checkCustomerExist">
<if test="customerServiceImpl.isCustomerExist(order.customer.
customerId)">
then=" collectShippingDetail"
else=" collectCustomerInfo"/>
</decision-state>
```

- After collecting customer details, don't forget to store them in `CustomerRepository` through an action state. Similarly, fill in the `order.customer` object after the decision state.

Summary

We only saw the very minimum of the concepts required to get a quick overview of the Spring Web Flow framework in this chapter. At the start of this chapter, we learned some of the basic concepts of the Spring Web Flow framework, and then we created the checkout flow for our web store application. We also learned how to fire a Web Flow event from a View. In the next chapter, we will learn more about how to incorporate the Apache Tiles framework into Spring MVC.

11

Template with Tiles

When it comes to web application development, reusability and maintenance are two important factors that need to be considered. Apache Tiles is another popular open source framework that encourages reusable template-based web application development.

In this chapter, you are going to see how to incorporate the Apache Tiles framework within a Spring MVC application, so that we can obtain maximum reusability of frontend templates with the help of Apache Tiles. Apache Tiles are mostly used to reduce redundant code in the frontend by leveraging frontend templates. After finishing this chapter, you will have a basic idea about decomposing pages using reusable Apache Tile templates.

Enhancing reusability through Apache Tiles

In the past, we developed a series of web pages (Views) as part of our [webstore](#) application, such as a page to show products, another page to add products, and so on. Although every View has served a different purpose, all of them share a common visual pattern; each page has a header, a content area, and so on. We hardcoded and repeated those common elements in every JSP View page. But this is not a good idea because, in future, if we want to change the look and feel of any of these common elements, we have to change every page in order to maintain a consistent look and feel across all the web pages.

To address this problem, modern web applications use template mechanisms; Apache Tiles is one such template composition framework. Tiles allow developers to define reusable page fragments (tiles), which can be assembled into a complete web page at runtime. These fragments can have parameters to allow dynamic content. This increases the reusability of templates and reduces code duplication.

Time for action – creating Views for every View state

Okay, enough introduction, let's dive into Apache Tiles by defining a common layout for our web application and let the pages extend the layout:

1. Open pom.xml. You can find pom.xml under the project root directory itself.
2. You will be able to see some tabs under pom.xml; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter org.apache.tiles as **Group Id**, tiles-extras as **Artifact Id**, 3.0.5 as **Version**, and select **Scope** as **compile**. Then click on the **OK** button and save pom.xml.
4. Now create a directory structure called layouts/definitions/ under the src/main/webapp/WEB-INF/ directory and create an XML file called tiles.xml. Add the following content to it and save it:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache
Software Foundation//DTD Tiles Configuration 3.0//EN"
"http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
    <definition name="baseLayout" template="/WEB-
INF/layouts/template/baseLayout.jsp">
        <put-attribute name="title" value="Sample
Title" />
        <put-attribute name="heading" value="" />
        <put-attribute name="tagline" value="" />
        <put-attribute name="navigation" value="/WEB-
INF/layouts/template/navigation.jsp" />
        <put-attribute name="content" value="" />
        <put-attribute name="footer" value="/WEB-
INF/layouts/template/footer.jsp" />
    </definition>
    <definition name="welcome" extends="baseLayout">
        <put-attribute name="title" value="Products" />
        <put-attribute name="heading" value="Products"
/>
        <put-attribute name="tagline" value="All the
available products in our store" />
        <put-attribute name="content" value="/WEB-
INF/views/products.jsp" />
    </definition>
    <definition name="products" extends="baseLayout">
        <put-attribute name="title" value="Products" />
        <put-attribute name="heading" value="Products"
```

```
/>
    <put-attribute name="tagline" value="All the
available products in our store" />
    <put-attribute name="content" value="/WEB-
INF/views/products.jsp" />
</definition>
<definition name="product" extends="baseLayout">
    <put-attribute name="title" value="Product" />
    <put-attribute name="heading" value="Product"
/>
    <put-attribute name="tagline" value="Details"
/>
    <put-attribute name="content" value="/WEB-
INF/views/product.jsp" />
</definition>
<definition name="addProduct"
extends="baseLayout">
    <put-attribute name="title" value="Products" />
    <put-attribute name="heading" value="Products"
/>
    <put-attribute name="tagline" value="Add
Product" />
    <put-attribute name="content" value="/WEB-
INF/views/addProduct.jsp" />
</definition>
<definition name="login" extends="baseLayout">
    <put-attribute name="title" value="Login" />
    <put-attribute name="heading" value="Welcome to
Web Store!" />
    <put-attribute name="tagline" value="The one
and only amazing web store" />
    <put-attribute name="content" value="/WEB-
INF/views/login.jsp" />
</definition>
<definition name="cart" extends="baseLayout">
    <put-attribute name="title" value="Shopping
Cart" />
    <put-attribute name="heading" value="Cart" />
    <put-attribute name="tagline" value="All the
selected products in your cart" />
    <put-attribute name="content" value="/WEB-
INF/views/cart.jsp" />
</definition>
</tiles-definitions>
```

5. Now create a directory called `template` under the `src/main/webapp/WEB-INF/layouts/` directory and create a JSP file called `baseLayout.jsp`. Add the following content to it and save it:

```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="tiles"
uri="http://tiles.apache.org/tags-tiles"%>

<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width,
initial-scale=1.0">

<title><tiles:insertAttribute name="title" /></title>

<link rel="stylesheet"
href="//netdna.bootstrapcdn.com/bootstrap/3.0.0/css
/bootstrap.min.css">

<script
src="https://ajax.googleapis.com/ajax/libs
/angularjs/1.5.1/angular.min.js"></script>

<script
src="/webstore/resources/js/controllers.js"></script>

</head>

<body>
    <section class="container">
        <div class="pull-right" style="padding-right:
50px">
            <a href="?language=en">English</a> | <a
href="?language=nl">Dutch</a>
            <a href=<c:url value="/logout
/>">Logout</a>
        </div>
    </section>

    <div class="container">
```

```
<div class="jumbotron">
    <div class="header">
        <ul class="nav nav-pills pull-right">
            <tiles:insertAttribute
name="navigation" />
        </ul>
        <h3 class="text-muted">Web Store</h3>
    </div>

    <h1>
        <tiles:insertAttribute name="heading" />
    </h1>
    <p>
        <tiles:insertAttribute name="tagline" />
    </p>
</div>

<div class="row">
    <tiles:insertAttribute name="content" />
</div>

<div class="footer">
    <tiles:insertAttribute name="footer" />
</div>

</div>
</body>
</html>
```

6. Under the same directory (`template`), create another template JSP file called `navigation.jsp` and add the following content to it:

```
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<li><a href=<spring:url
value="/market/products"/>">Home</a></li>
<li><a href=<spring:url
value="/market/products/" />">Products</a></li>
<li><a href=<spring:url
value="/market/products/add"/>">Add Product</a></li>
<li><a href=<spring:url
value="/cart/" />">Cart</a></li>
```

7. Similarly, create one last template JSP file called `footer.jsp` and add the following content to it:

```
<p>&copy; Company 2016</p>
```

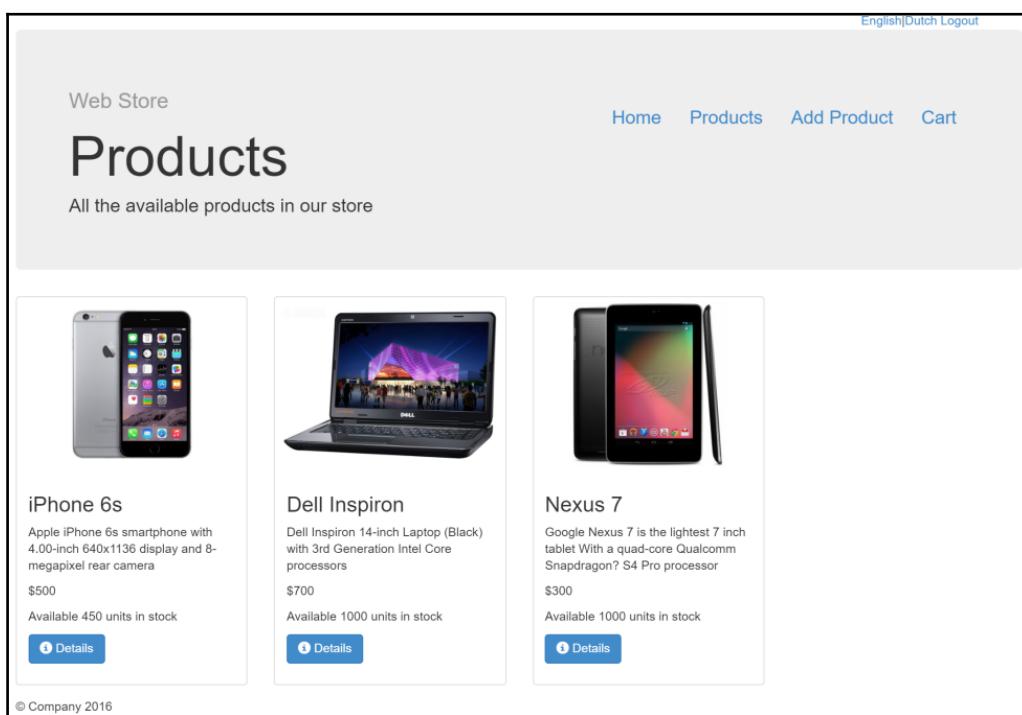
8. Now we have created the common base layout template and the tile definition for all our pages, we need to remove the common page elements from all our JSP View files. For example, if you remove all the common elements, such as the jumbotron section and others, from our `products.jsp` file, and keep only the container section, it would look as follows:

Do not remove the tag lib references and link references.



```
<%@ taglib prefix="c"
uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="spring"
uri="http://www.springframework.org/tags"%>

<section class="container">
    <div class="row">
        <c:forEach items="${products}" var="product">
            <div class="col-sm-6 col-md-3"
style="padding-bottom: 15px">
                <div class="thumbnail">
                    . You will be able to see our regular products page with an extra navigation bar at the top and a footer at the bottom. You can add a product by clicking on the **Add Product** link:



Products page with the Apache Tiles View

## What just happened?

To work with Apache Tiles, we need Apache Tiles related JARs, so from steps 1 to 3 we added those JARs via Maven dependencies. Step 4 is very important because we created our tiles definition file (`tiles.xml`) in that step. Understanding the tiles definition file is crucial to developing Apache Tiles-based applications, so you need to understand our tile definition file.

A tile definition file is a collection of definitions, where each definition can be associated with a template via the `template` attribute for the layout:

```
<definition name="baseLayout" template="/WEB-
INF/layouts/template/baseLayout.jsp">
 <put-attribute name="title" value="Sample Title" />
 <put-attribute name="heading" value="Sample Heading" />
 <put-attribute name="tagline" value="Sample Tagline" />
 <put-attribute name="navigation" value="/WEB-
INF/layouts/template/navigation.jsp" />
 <put-attribute name="content" value="" />
 <put-attribute name="footer" value="/WEB-
INF/layouts/template/footer.jsp" />
</definition>
```

Within each definition, we can define many attributes. These attributes can be a simple text value or a full-blown markup file. These attributes would be available in the template file via the `<tiles:insertAttribute>` tag. For example, if you open the base layout (`baseLayout.jsp`) template, you can see the following snippet under the `jumbotron<div>` tag:

```
<h1>
 <tiles:insertAttribute name="heading" />
</h1>
<p>
 <tiles:insertAttribute name="tagline" />
</p>
```

So at runtime, Apache Tiles would replace the `<tiles:insertAttribute name="heading" />` tag with the value Sample Heading and similarly the `<tiles:insertAttribute name="tagline" />` tag with the value Sample Tagline.

So the `baseLayout` definition is associated with the template `/WEB-INF/layouts/template/baseLayout.jsp`, and we can insert the defined attributes such as `title`, `heading`, `tagline`, and more in the template using the `<tiles:insertAttribute>` tag.

Apache Tiles allows us to extend a definition just like how we extend a Java class, so that the defined attributes would be available for the derived definition, and we can even override those attributes values if we want. For example, look at the following definition from `tile-definition.xml`:

```
<definition name="products" extends="baseLayout">
 <put-attribute name="title" value="Products" />
 <put-attribute name="heading" value="Products" />
```

```
<put-attribute name="tagline" value=" All the available products in our
store" />
<put-attribute name="content" value="/WEB-INF/views/products.jsp" />
</definition>
```

This definition is an extension of the `baseLayout` definition. We have only overridden the `title`, `heading`, `tagline`, and `content` attributes, and since we have not defined any template for this definition, it uses the same template that we configured for the `baseLayout` definition.

Similarly, we defined the tile definition for every possible logical View name that can be returned from our Controllers. Note that each definition name (except the `baseLayout` definition) is a Spring MVC logical View name.

From steps 5 to 7, we just created the templates that can be used in the tile definition. First, we created the base layout template (`baseLayout.jsp`), then the navigation template (`navigation.jsp`), and finally the footer template (`footer.jsp`).

Steps 8 and 9 explained how to remove the existing redundant content such as the `jumbotron<div>` tag from every JSP View page. Note you have to be careful while doing this—don't accidentally remove the `taglib` references.

In step 10, we defined our `UrlBasedViewResolver` for `TilesView` in order to resolve logical View names into the tiles View and also configured the `TilesConfigurer` to locate the tiles definition files by the Apache Tiles framework.

That's it; if you run our application and enter the URL

`http://localhost:8080/webstore/products`, you will be able to see our regular products page with an extra navigation bar at the top and a footer at the bottom, as mentioned in step 11. You can add a product page by clicking on the **Add Product** link. Previously, every time a logical View name was returned by the Controller method, the `InternalResourceViewResolver` comes into action and finds the corresponding `.jsp` View for the given logical View name. Now for every logical View name, the `UrlBasedViewResolver` will come into action and compose the corresponding View based on the template definition.

## Pop quiz – Apache Tiles

Which of the following statements are true according to Apache Tiles?

1. The logical View name returned by the Controller must be equal to the <definition> tag name.
2. The <tiles:insertAttribute> tag acts as a placeholder in the template.
3. A <definition> tag can extend another <definition> tag.
4. All of the above.

## Summary

Apache Tiles is a separate framework. We only provided the bare minimum required concepts to get a quick overview of Apache Tiles. You saw how to use and leverage the Apache Tiles framework in order to ensure maximum reusability in the View files and maintain a consistent look and feel throughout all the web pages of our application.

In the next chapter, you will see how to test our web application using the various APIs provided by Spring MVC.

# 12

## Testing Your Application

*For a web application developer, testing web applications is always a challenging task because getting a real-time test environment for web applications requires a lot of effort. But thanks to the **Spring MVC Test framework**, it simplifies the testing of Spring MVC applications.*

But why do we need to consider putting effort into testing our application? Writing good test cases for our application is a kind of like buying an insurance policy for your application, although it does not add any functional values to your application, it will definitely save you time and effort by detecting functionality failures early. Consider your application growing bigger and bigger in terms of functionality—you need some mechanism to ensure that the existing functionalities were not disturbed by means of introducing new functionalities.

Testing frameworks provide you with such a mechanism, to ensure that your application's behavior is not altered due to refactoring or the addition of new code. They also ensure existing functionalities work as expected.

In this chapter, we are going to look at the following topics:

- Testing the domain object and validator
- Testing Controllers
- Testing RESTful web services

# Unit testing

In software development, unit testing is a software testing method in which the smallest testable parts of source code, called units, are individually and independently tested to determine whether they behave exactly as we expect. To unit test our source code, all we need is a test program that can run a bit of our source code (unit), provide some inputs to each unit, and check the results for the expected output. Most unit tests are written using some sort of test framework set of library code designed to make writing and running tests easier. One such framework is called **JUnit**. It is a unit testing framework for the Java programming language.

## Time for action – unit testing domain objects

Let's see how to test one of our domain objects using the JUnit framework to ensure it functions as expected. In an earlier chapter, we created a domain object to represent an item in a shopping cart called `CartItem`. The `CartItem` class has a method called `getTotalPrice` to return the total price of that particular cart item, based on the product and number of items it represents. Let's test whether the `getTotalPrice` method behaves properly:

1. Open `pom.xml`, which you can find under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of the `pom.xml` file. Select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter `junit` as **Group Id**, `junit` as **Artifact Id**, `4.12` as **Version**, select **Scope** as `test`, click on the **OK** button, and save `pom.xml`.
4. Now create a class called `CartItemTest` under the `com.packt.webstore.domain` package in the `src/test/java` source folder, add the following code to it, and save the file:

```
package com.packt.webstore.domain;
import java.math.BigDecimal;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

public class CartItemTest {

 private CartItem cartItem;

 @Before
```

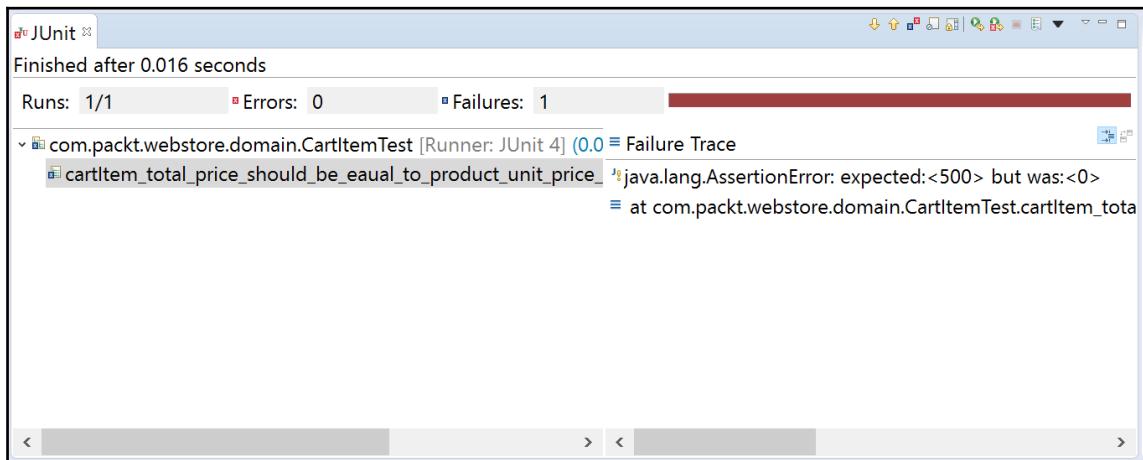
```
public void setup() {
 cartItem = new CartItem("1");
}

@Test
public void cartItem_total_price_should_be_equal_
to_product_unit_price_in_case_of_single_quantity() {
 //Arrange
 Product iphone = new Product("P1234", "iPhone
5s", new BigDecimal(500));
 cartItem.setProduct(iphone);

 //Act
 BigDecimal totalPrice =
cartItem.getTotalPrice();

 //Assert
 Assert.assertEquals(iphone.getUnitPrice(),
totalPrice);
}
}
```

5. Now right-click on `CartItemTest.java` and choose **Run As | JUnit Test**. You will see a failing test case in the **JUnit** window, as shown in the following screenshot:

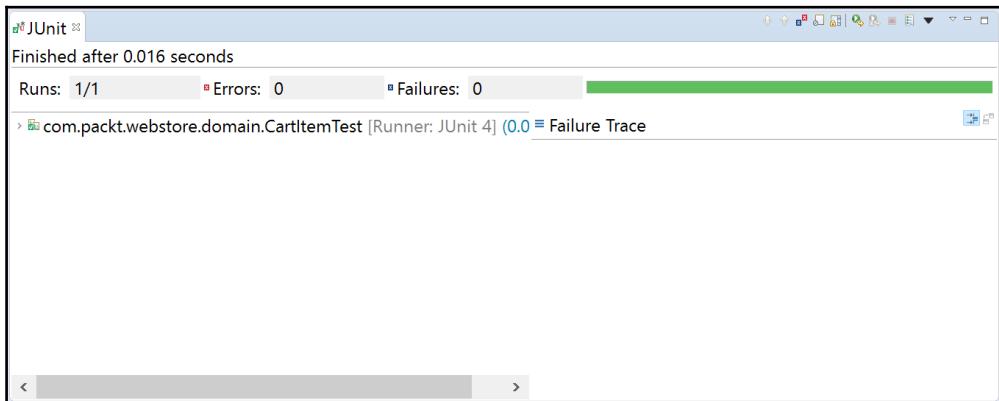


JUnit failing test case in CartItemTest

6. To make the test case pass, assign the value 1 to the `quantity` field of `CartItem` in the `setProduct` method of the `CartItem` class as follows, and save the file:

```
public void setProduct(Product product) {
 this.product = product;
 this.quantity = 1;
 this.updateTotalPrice();
}
```

7. Now right-click on `CartItemTest.java` again and choose **Run As | JUnit Test**. You will see a passing test case in the **JUnit** window, as shown in the following screenshot:



JUnit passing test case in CartItemTest

## What just happened?

As I already mentioned, the `getTotalPrice` method of the `CartItem` class is designed to return the correct total price based on the product and the number of products it represents. But to ensure its behavior, we wrote a test program called `CartItemTest` under the `com.packt.webstore.domain` package in the `src/test/java` source folder, as mentioned in step 4.

In `CartItemTest`, we used some of the JUnit framework APIs such as the `@Test` and `@Before` annotations, and more. So in order to use these annotations in our `CartItemTest` class, prior to that we need to add the JUnit JAR as a dependency in our project. That's what we did from steps 1 to 3.

Now you need to understand the `CartItemTest` class thoroughly. The important method in the `CartItemTest` class is the `@Test` annotated method called `cartItem_total_price_should_be_equal_to_product_unit_price_in_case_of_single_quantity`. The `@Test (org.junit.Test)` annotation marks a particular method as a test method so that the JUnit framework can treat that method as a test method and execute it when we choose the **Run As | JUnit Test** option:

```
@Test
public void
cartItem_total_price_should_be_equal_to_product_unit_price_in_case_of_single_quantity() {
 //Arrange
 Product iphone = new Product("P1234", "iPhone 5s", new
BigDecimal(500));
 cartItem.setProduct(iphone);
 //Act
 BigDecimal totalPrice = cartItem.getTotalPrice();
 //Assert
 Assert.assertEquals(iphone.getUnitPrice(), totalPrice);
}
```

If you notice, this method was divided into three logical parts called **Arrange**, **Act**, and **Assert**:

- Arrange all the necessary preconditions and inputs to perform a test
- Act on the object or method under test
- Assert that the expected results have occurred

In the **Arrange** part, we just instantiated a product domain object (`iphone`) with a unit price value of 500 and added that product object to the `cartItem` object by calling `cartItem.setProduct(iphone)`. Now we have added a single product to the `cartItem`, but we haven't altered the `quantity` of the `cartItem` object. So if we call the `getTotalPrice` method of `cartItem`, we must get 500 (in `BigDecimal`), because the unit price of the domain object (`iphone`) we added in `cartItem` is 500.

In the **Act** part, we just called the method under test, which is the `getTotalPrice` method of the `cartItem` object, and stored the result in a `BigDecimal` variable called `totalPrice`. Later, in the **Assert** part, we used the JUnit API (`Assert.assertEquals`) to assert the equality between the `unitPrice` of the product domain object and the calculated `totalPrice` of `cartItem`:

```
Assert.assertEquals(iphone.getUnitPrice(), totalPrice);
```

The `totalPrice` of `cartItem` must be equal to the `unitPrice` of the product domain object, which we added to `cartItem`, because we added a single product domain object whose `unitPrice` needs to be the same as the `totalPrice` of `cartItem`.

When we run our `CartItemTest` as mentioned in step 5, the JUnit framework tries to execute all the `@Test` annotated methods in the `CartItemTest` class. So based on the Assertion result, a test case may fail or pass. In our case, our test case failed. You can see the failure trace showing an error message saying **expected <500> but was: <0>** in the screenshot for step 5. This is because in the Arrange part, when we added a product domain object to `cartItem`, it doesn't update the `quantity` field of the `cartItem` object. It is a bug, so to fix this bug we default the `quantity` field value to 1 whenever we set the `product` argument using the `setCartItem` method, as mentioned in step 6. Now finally, if we run our test case again, this time it passes as expected.

## Have a go hero – adding tests for Cart

It's good that we tested and verified the `getTotalPrice` method of the `CartItem` class, but can you similarly write a test class for the `Cart` domain object class? In the `Cart` domain object class, there is method to get the grand total (`getGrandTotal`), and write various test case to check whether the `getGrandTotal` method works as expected.

## Integration testing with the Spring Test context framework

When individual program units are combined and tested as a group, then it is known as **integration testing**. The Spring Test context framework provides first-class support for an integration test of a Spring-based application. We have defined lots of Spring managed beans in our web application context, such as services, repositories, view resolvers, and more, to run our application.

These managed beans are instantiated during the startup of an application by the Spring framework. While doing the integration testing, our test environment must also have those beans to test our application successfully. The Spring Test context framework gives us the ability to define a test context that is similar to the web application context. Let's see how to incorporate the Spring Test context to test our `ProductValidator` class.

## Time for action – testing product validator

Let's see how we can boot up our test context using the Spring Test context framework to test our `ProductValidator` class:

1. Open `pom.xml`, which you can find under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of the `pom.xml` file; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; enter `org.springframework` as **Group Id**, `spring-test` as **Artifact Id**, `4.3.0.RELEASE` as **Version**, and select `test` as **Scope**, then click on the **OK** button and save `pom.xml`.
4. Similarly, add one more dependency for `jsp-api`; repeat the same step with **Group Id** as `javax.servlet`, **Artifact Id** as `jsp-api`, and **Version** as `2.0`, but this time, select **Scope** as `test` and then click on the **OK** button and save `pom.xml`.
5. Next create a class called `ProductValidatorTest` under the `com.packt.webstore.validator` package in the `src/test/java` source folder, and add the following code to it:

```
package com.packt.webstore.validator;

import java.math.BigDecimal;

import org.junit.Assert;
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 org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.validation.BindException;
import org.springframework.validation.ValidationUtils;

import com.packt.webstore.config.WebApplicationContextConfig;
import com.packt.webstore.domain.Product;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
WebApplicationContextConfig.class)
```

```
@WebAppConfiguration
public class ProductValidatorTest {
 @Autowired
 private ProductValidator productValidator;
 @Test
 public void
product_without_UnitPrice_should_be_invalid() {
 //Arrange
 Product product = new Product();
 BindException bindException = new
BindException(product, " product");

 //Act
 ValidationUtils.invokeValidator
(productValidator, product, bindException);
 //Assert
 Assert.assertEquals(1,
bindException.getErrorCount());
 Assert.assertTrue(bindException
.getLocalizedMessage().contains("Unit price is
Invalid. It cannot be empty."));
 }
 @Test
 public void
product_with_existing_productId_invalid() {
 //Arrange
 Product product = new Product("P1234","iPhone
5s", new BigDecimal(500));
 product.setCategory("Tablet");
 BindException bindException = new
BindException(product, " product");

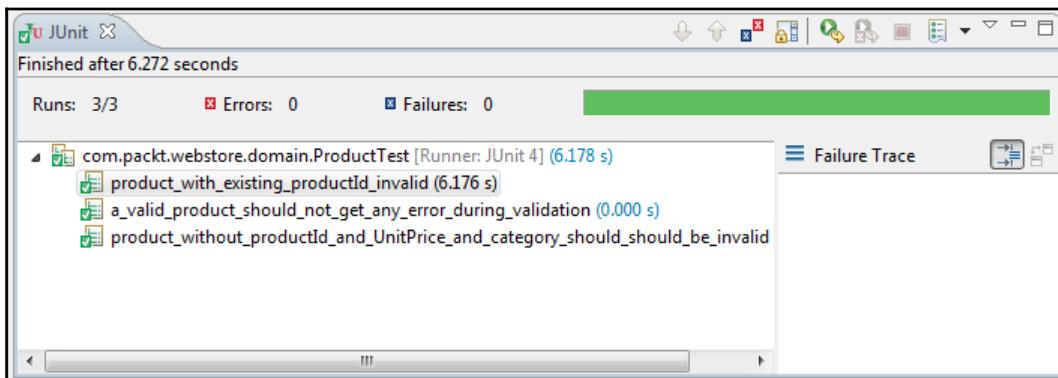
 //Act
 ValidationUtils.invokeValidator
(productValidator, product, bindException);
 //Assert
 Assert.assertEquals(1,
bindException.getErrorCount());
 Assert.assertTrue(bindException.
getLocalizedMessage().contains("A product already
exists with this product id."));
 }
 @Test
 public void a_valid_product_should_not_get
_any_error_during_validation() {
 //Arrange
 Product product = new Product("P9876","iPhone
5s", new BigDecimal(500));
```

```
 product.setCategory("Tablet");
 BindException bindException = new
 BindException(product, " product");

 //Act
 ValidationUtils.invokeValidator
 (productValidator, product, bindException);
 //Assert
 Assert.assertEquals(0,
 bindException.getErrorCount());
 }

}
```

6. Now right-click on `ProductValidatorTest` and choose **Run As | JUnit Test**. You will be able to see passing test cases, as shown in the following screenshot:



Customer details collection form

## What just happened?

As I already mentioned, Spring provides extensive support for integration testing. In order to develop a test case using the Spring Test context framework, we need the required `spring-test` JAR. In step 3, we just added a dependency to the `spring-test` JAR. The Spring Test context framework cannot run without the support of the JUnit JAR.

Step 5 is very important because it represents the actual test class (`ProductValidatorTest`) to test the validity of our `Product` domain object. The goal of the test class is to check whether all the validations (including bean validation and Spring validation) that were specified in the `Product` domain class are working. I hope you remember that we specified some of the bean validation annotations such as `@NotNull`, `@Pattern`, and more in the `Product` domain class.

One way to test whether those validations are taking place is by manually running our application and trying to enter invalid values. This approach is called manual testing. This is a very difficult job, whereas in automated testing we can write some test classes to run test cases in repeated fashion to test our functionality. Using JUnit, we can write this kind of test class.

The `ProductValidatorTest` class contains three test methods in total; we can identify a test method using the `@Test` (`org.junit.Test`) annotation of JUnit. Every test method can be logically separated into three parts, that is, `Arrange`, `Act`, and `Assert`. In the `Arrange` part, we instantiated and instrumented the required objects for testing; in the `Act` part, we invoked the actual functionality that needs to be tested; and finally in the `Assert` part, we compared the expected result and the actual result that is an output of the invoked functionality:

```
@Test
public void product_without_UnitPrice_should_be_invalid() {
 //Arrange
 Product product = new Product();
 BindException bindException = new BindException(product, "product");

 //Act
 ValidationUtils.invokeValidator(productValidator, product,
 bindException);
 //Assert
 Assert.assertEquals(1, bindException.getErrorCount());
 Assert.assertTrue(bindException.getLocalizedMessage().contains("Unit price
 is Invalid. It cannot be empty."));
}
```

In the `Arrange` part of this test method, we just instantiated a bare minimum `Product` domain object. We have not set any values for the `productId`, `unitPrice`, and `category` fields. We purposely set up such a bare minimum domain object in the `Arrange` part to check whether our `ProductValidator` class is working properly in the `Act` part.

According to the `ProductValidator` class logic, the present state of the `product` domain object is invalid. And in the `Act` part, we invoked the `productValidator` method using

the ValidationUtils class to check whether the validation works or not. During validation, productValidator will store the errors in a BindException object. In the Arrange part, we simply check whether the bindException object contains one error using the JUnit Assert APIs, and check that the error message was as expected.

Another important thing you need to understand in our ProductValidatorTest class is that we used a Spring standard @Autowired annotation to get the instance of ProductValidator. The question here is who instantiated the productValidator object? The answer is in the @ContextConfiguration annotation. Yes, if you looked at the classes attribute specified in the @ContextConfiguration annotation, it has the name of our test context file (WebApplicationContextConfig.class).

If you remember correctly, you learned in the past that during the booting up of our application, Spring MVC creates a web application context (Spring container) with the necessary beans, as defined in the web application context configuration file. We need a similar kind of context even before running our test classes, so that we can use those defined beans (objects) in our test class to test them properly. The Spring Test framework makes this possible via the @ContextConfiguration annotation.

Likewise, we need a similar running application environment with all the resource files, and to achieve this we used the @WebAppConfiguration annotation from the Spring Test framework. The @WebAppConfiguration annotation instructs the Spring Test framework to load the application context as WebApplicationContext.

Now you have seen almost all the important things related to executing a Spring integration test, but there is one final configuration you need to understand, how to integrate JUnit and the Spring Test context framework into our test class. The @RunWith(SpringJUnit4ClassRunner.class) annotation just does this job.

So finally, when we run our test cases, we are able to see a green bar in the JUnit window indicating that the tests were successful.

## Time for action – testing product Controllers

Now let's look at how to test our Controllers:

1. Create a class called ProductControllerTest under the com.packt.webstore.controller package in the src/test/java source folder and add the following code to it:

```
package com.packt.webstore.controller;
```

```
import static org.springframework.test.web.servlet.
.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet
.result.MockMvcResultMatchers.model;

import java.math.BigDecimal;

import org.junit.Before;
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 org.springframework.test.context
.web.WebAppConfiguration;
import org.springframework.test.web
.servlet.MockMvc;
import org.springframework.test.web
.servlet.setup.MockMvcBuilders;
import org.springframework.web
.context.WebApplicationContext;

import com.packt.webstore.config
.WebApplicationContextConfig;
import com.packt.webstore.domain.Product;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
WebApplicationContextConfig.class)
@WebAppConfiguration
public class ProductControllerTest {

 @Autowired
 private WebApplicationContext wac;

 private MockMvc mockMvc;

 @Before
 public void setup() {
 this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
 }

 @Test
 public void testGetProducts() throws Exception {
```

```
 this.mockMvc.perform(get("/market/products"))
 .andExpect(model())
 .attributeExists("products"));
 }
 @Test
 public void testGetProductById() throws Exception
 {
 //Arrange
 Product product = new Product("P1234", "iPhone 5s",
new BigDecimal(500));
 //Act & Assert
 this.mockMvc.perform(get("/market/product")
 .param("id", "P1234"))
 .andExpect(model().attributeExists("product"))
 .andExpect(model().attribute("product", product));
 }

}
```

2. Now right-click on the `ProductControllerTest` class and choose **Run As | JUnit Test**. You will be able to see that the test cases are being executed, and you will be able to see the test results in the **JUnit** window.

## What just happened?

Similar to the `ProductValidatorTest` class, we need to boot up the test context and want to run our `ProductControllerTest` class as a Spring integration test. So we used similar annotations on top of `ProductControllerTest` as follows:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = WebApplicationContextConfig.class)
@WebAppConfiguration
public class ProductControllerTest {
```

As well as the two test methods that are available under `ProductControllerTest`, a single setup method is available as follows:

```
@Before
public void setup() {
 this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
}
```

The `@Before` annotation that is present on top the previous method indicates that this method should be executed before every test method. And within that method, we simply build our `mockMvc` object in order to use it in the following test methods. The `MockMvc` class is a special class provided by the Spring Test context framework to simulate browser actions within a test case, such as firing HTTP requests:

```
@Test
public void testGetProducts() throws Exception {
 this.mockMvc.perform(get("/market/products"))
 .andExpect(model().attributeExists("products"));
}
```

This test method simply fires a GET HTTP request to our application using the `mockMvc` object, and as a result, we ensure the returned model contains an attribute named `products`. Remember that our `list` method from the `ProductController` class is the thing handling the previous web request, so it will fill the model with the available products under the attribute name `products`.

After running our test case, you are able to see the green bar in the **JUnit** window, which indicates that the tests passed.

## Time for action – testing REST Controllers

Similarly, we can test the REST-based Controllers as well—just follow these steps:

1. Open `pom.xml`, which you can find `pom.xml` under the root directory of the project itself.
2. You will be able to see some tabs at the bottom of `pom.xml` file; select the **Dependencies** tab and click on the **Add** button of the **Dependencies** section.
3. A **Select Dependency** window will appear; for **Group Id** enter `com.jayway.jsonpath`, for **Artifact Id** enter `json-path-assert`, for **Version** enter `2.2.0`, and for **Scope** select `test`, then click on the **OK** button and save `pom.xml`.
4. Now create a class called `CartRestControllerTest` under the `com.packt.webstore.controller` package in the `src/test/java` source folder, and add the following code to it:

```
package com.packt.webstore.controller;

import static org.springframework.test.web.
.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web
```

```
.servlet.request.MockMvcBuilders.put;
import static org.springframework.test.web
.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web
.servlet.result.MockMvcResultMatchers.status;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory
.annotation.Autowired;
import org.springframework.mock
.web.MockHttpSession;
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.servlet.MockMvc;
import org.springframework.test.web
.MockMvcBuilders;
import org.springframework.web.context
.WebApplicationContext;

import com.packt.webstore.config
.WebApplicationContextConfig;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes =
WebApplicationContextConfig.class)
@WebAppConfiguration
public class CartRestControllerTest {
 @Autowired
 private WebApplicationContext wac;

 @Autowired
 MockHttpSession session;
 private MockMvc mockMvc;

 @Before
 public void setup() {
 this.mockMvc =
MockMvcBuilders.webAppContextSetup(this.wac).build();
 }
 @Test
 public void
read_method_should_return_correct_cart_Json_object ()
```

```
throws Exception {
 //Arrange
 this.mockMvc.perform(put("/rest/cart/add/P1234")
 .session(session))
 .andExpect(status().is(200));
 //Act
 this.mockMvc.perform(get("/rest/cart/" +
 session.getId()).session(session))
 .andExpect(status().isOk())
 .andExpect(jsonPath("$.cartItems[0].
 product.productId").value("P1234"));
}
}
```

5. Now right-click on `CartRestControllerTest` and choose **Run As | JUnit Test**. You will be able to see that the test cases are being executed, and you will be able to see the test results in the **JUnit** window.

## What just happened?

While testing REST Controllers, we need to ensure that the web response for the given web request contains the expected JSON object. To verify that, we need some specialized APIs to check the format of the JSON object. The `json-path-assert` JAR provides such APIs. We added a Maven dependency to the `json-path-assert` JAR from steps 1 to 3.

In step 4, we created our `CartRestControllerTest` to verify that our `CartRestController` class works properly. The `CartRestControllerTest` class is very similar to `ProductControllerTest`—the only difference is the way we assert the result of a web request. In `CartRestControllerTest`, we have one test method to test the `read` method of the `CartRestController` class.

The `read` method of `CartRestController` is designed to return a `cart` object as a JSON object for the given cart ID. In `CartRestControllerTest`, we tested this behavior in the `read_method_should_return_correct_cart_Json_object` test method:

```
@Test
public void read_method_should_return_correct_cart_Json_object () {
 throws Exception {
 //Arrange
 this.mockMvc.perform(put("/rest/cart/add/P1234").session(session))
 .andExpect(status().is(200));
 //Act
 this.mockMvc.perform(get("/rest/cart/" +
```

```
session.getId()).session(session)
 .andExpect(status().isOk())
.andExpect(jsonPath("$.cartItems[0].product.productId").value("P1234"));
}
```

In order to get a cart object for the given cart ID, we need to store the cart object in our cart repository first, through a web request. That is what we did in the `Arrange` part of the previous test method. The first web request we fired in the `Arrange` part adds a product domain object in the cart, whose ID is the same as the session ID.

In the `Act` part of the test case, we simply fired another REST based web request to get the cart object as JSON object. Remember we used the session ID as our cart ID to store our cart object, so while retrieving it, we need to provide the same session ID in the request URL. For this, we can use the mock session object given by the Spring Test framework. You can see that we auto-wired the session object in our `CartRestControllerTest` class:

```
this.mockMvc.perform(get("/rest/cart/" + session.getId()).session(session))
 .andExpect(status().isOk())
 .andExpect(
(jsonPath("$.cartItems[0].product.productId").value("P1234"));
```

After we get the cart domain object as the JSON object, we have to verify whether it contains the correct product. We can do that with the help of the `jsonPath` method of `MockMvcResultMatchers`, as specified in the previous code snippets. After sending the REST web request to go get the `cart` object, we verified that the response status is okay and we also verified that the JSON object contains a product with the ID `P1234`.

Finally, when we run this test case, you can see the test cases being executed, and you can see the test results in the **JUnit** window.

## Have a go hero – adding tests for the remaining REST methods

It's good that we tested and verified the `read` method of `CartRestController`, but we have not tested the other methods of `CartRestController`. You can add tests for the other methods of `CartRestController` in the `CartRestControllerTest` class to get more familiar with the Spring Test framework.

## **Summary**

In this final chapter, you saw the importance of testing a web application and you saw how to test the validator using the Spring Test framework. You also found out how to test a normal Controller using the Spring Test context framework. As a last exercise, you also discovered how to test REST-based Controllers and how to use the mock session object from the Spring Test framework. During that exercise, you also learned how to use the JSON path library for assertion.

## **Thank you readers!**

I am so thankful to all of you readers. I hope this book has helped you to understand Spring MVC from a beginner's perspective. I'm grateful for our connection. I hope all of you will continue to reach out to me at any time with feedback or questions. Thank you once again and all the very best in your exploration of the latest technologies!

Happy coding!

# Using the Gradle Build Tool

*Throughout this book, we have used Apache Maven as our build tool, but there are other popular build tools also used widely in the Java community. One such build tool is Gradle. Instead of XML, Gradle uses a Groovy-based **Domain Specific Language (DSL)** as the base for the build script, which provides more flexibility when defining complex build scripts. Compared to Maven, Gradle takes less time for incremental builds. So, Gradle builds are very fast and effective for large projects.*

In this appendix, we will see how to install and use Gradle as the build tool in our project.

## Installing Gradle

Perform the following steps to install Gradle:

1. Go to the Gradle download page by entering the URL <http://www.gradle.org/downloads> in your browser.
2. Click on the latest Gradle stable release download link; at the time of writing this, the stable release is gradle-2.14.1.
3. Once the download is finished, go to the downloaded directory and extract the ZIP file into a convenient directory of your choice.
4. Create an environment variable called `GRADLE_HOME`. Enter the extracted Gradle ZIP directory path as the value for the `GRADLE_HOME` environment variable.
5. Finally, append the `GRADLE_HOME` variable to `PATH` by simply appending the text; `%GRADLE_HOME%\bin` to the `PATH` variable.

Now that you have installed Gradle on your Windows-based computer, to verify whether the installation was completed correctly, go to the command prompt, type `gradle -v`, and press *Enter*. The output shows the Gradle version and also the local environment configuration.

## The Gradle build script for your project

To configure the Gradle build script for your project, perform the following steps:

1. Go to the root directory of your project from the filesystem, create a file called `build.gradle`, and add the following content into the file and save it:

```
apply plugin: 'war'
apply plugin: 'eclipse-wtp'
repositories {
 mavenCentral() //add central maven repo to your buildfile
}

dependencies {

 compile 'org.springframework:spring-webmvc:4.3.0.RELEASE',
 'javax.servlet:jstl:1.2',
 'org.springframework:spring-jdbc:4.3.0.RELEASE',
 'org.hsqldb:hsqldb:2.3.2',
 'commons-fileupload:commons-fileupload:1.2.2',
 'org.apache.commons:commons-io:1.3.2',
 'org.springframework:spring-oxm:4.3.0.RELEASE',
 'org.codehaus.jackson:jackson-mapper-asl:1.9.10',
 'com.fasterxml.jackson.core:jackson-databind:2.8.0',
 'log4j:log4j:1.2.17',
 'org.springframework.security:spring-security
 -config:4.1.1.RELEASE',
 'org.springframework.security:spring-security
 -web:4.1.1.RELEASE',
 'org.hibernate:hibernate-validator:5.2.4.Final',
 'org.springframework.webflow:spring
 -webflow:2.4.2.RELEASE',
 'org.apache.tiles:tiles-extras:3.0.5'

 providedCompile 'javax.servlet:javax.servlet-api:3.1.0'

 testCompile 'junit:junit:4.12',
 'org.springframework:spring
 -test:4.3.0.RELEASE',
```

```
'javax.servlet:jsp-api:2.0',
'com.jayway.jsonpath:json-path
-assert:2.2.0'
}
```

2. Now go to the root directory of your project from the command prompt and issue the following command:

```
> gradle eclipse
```

3. Next, open a new workspace in your STS, go to **File | Import**, select the **Existing Projects into Workspace** option from the tree list (you can find this option under the **General** node), and then click on the **Next** button.
4. Click on the **Browse** button to select the root directory and locate your project directory. Click on **OK** and then on **Finish**.

Now, you will be able to see your project configured with the right dependencies in your STS.

## Understanding the Gradle script

A task in Gradle is similar to a goal in Maven. The Gradle script supports many in-built plugins to execute build-related tasks. One such plugin is the `war` plugin, which provides many convenient tasks to help you build a web project. We can incorporate these tasks in our build script easily by applying a plugin in our Gradle script as follows:

```
apply plugin: 'war'
```

Similar to the `war` plugin, there is another plugin called `eclipse-wtp` to incorporate tasks related to converting a project into an eclipse project. The `eclipse` command we used in step 2 is actually provided by the `eclipse-wtp` plugin.

Inside the `repositories` section, we can define our remote binary repository location. When we build our Gradle project, we use this remote binary repository to download the required JARs. In our case, we defined our remote repository as the Maven central repository, as follows:

```
repositories {
 mavenCentral()
}
```

All of the project dependencies need to be defined inside of the `dependencies` section grouped under the scope declaration, such as `compile`, `providedCompile`, and `testCompile`. Consider the following code snippet:

```
dependencies {
 compile
 'org.springframework:spring-webmvc:4.3.0.RELEASE',
 'javax.servlet:jstl:1.2'.
}
```

If you look closely at the following dependency declaration line, the `compile` scope declaration, you see that each dependency declaration line is delimited with a : (colon) symbol, as follows:

```
'org.springframework:spring-webmvc:4.3.0.RELEASE'
```

The first part of the previous line is the group ID, the second part is the artifact ID, and the final part is the version information as provided in Maven.

So, it is more like a Maven build script but defined using a Gradle script, which is based on the Groovy language.

# Pop Quiz Answers

This appendix contains answers to all the pop quizzes that appear in the chapters. Now, let's have a look at the answers to the respective questions.

## Chapter 2, Spring MVC Architecture – Architecting Your Web Store

Questions	Answers
Suppose I have a Spring MVC application for library management called <i>BookPedia</i> and I want to map a web request URL <code>http://localhost:8080/BookPedia/category/fiction</code> to a controller's method—how would you form the <code>@RequestMapping</code> annotation?	2. <code>@RequestMapping("/category/fiction")</code>
What is the request path in the following URL: <code>http://localhost:8080/webstore/</code> ?	2. /
Considering the following servlet mapping, identify the possible matching URLs: <code>@Override protected String[] getServletMappings() {     return new String[] { "*.do" }; }</code>	3. <code>http://localhost:8080/webstore/welcome.do</code>
Considering the following servlet mapping, identify the possible matching URLs: <code>@Override protected String[] getServletMappings() {     return new String[] { "/" }; }</code>	4. All the above
In order to identify a class as a controller by Spring, what needs to be done?	4. All of the above.

# Chapter 3, Control Your Store with Controllers

Questions	Answers
If you imagine a web application called <i>Library</i> with the following request mapping on a Controller class level and in the method level, which is the appropriate request URL to map the request to the <i>books</i> method? <pre>@RequestMapping("/books") public class BookController { ... @RequestMapping(value = "/list") public String books(Model model) { ...</pre>	1. <code>http://localhost:8080/library/books/list</code>
Similarly, suppose we have another handler method called <i>bookDetails</i> under <i>BookController</i> as follows, what URL will map to that method? <pre>@RequestMapping public String details(Model model) { ...</pre>	2. <code>http://localhost:8080/library/books</code>
If we have a web application called <i>webstore</i> with the following request mapping on the Controller class level and in the method level, which is the appropriate request URL? <pre>@RequestMapping("/items") public class ProductController { ... @RequestMapping(value = "/type/{type}", method = RequestMethod.GET) public String productDetails(@PathVariable("type") String productType, Model model) {</pre>	2. <code>http://localhost:8080/webstore/items/type/electronics</code>
For the following request mapping annotation, which are the correct methods' signatures to retrieve the path variables? <pre>@RequestMapping(value="/manufacturer/{manufacturerId}/product/{productId}")</pre>	1. <code>public String productByManufacturer(@PathVariable String manufacturerId, @PathVariable String productId, Model model)</code> 4. <code>public String productByManufacturer(@PathVariable("manufacturerId") String manufacturer, @PathVariable("productId") String product, Model model)</code>
For the following request mapping method signature, which is the appropriate request URL? <pre>@RequestMapping(value = "/products", method = RequestMethod.GET) public String productDetails(@RequestParam String rate, Model model)</pre>	2. <code>http://localhost:8080/webstore/products?rate=400</code>

## Chapter 4, Working with Spring Tag Libraries

Questions	Answers
Consider the following data binding customization and identify the possible matching field bindings: <pre>@InitBinder public void initialiseBinder(WebDataBinder binder) {     binder.setAllowedFields("unit*"); }</pre>	2. unitPrice 4. united

## Chapter 5, Working with View Resolver

Questions	Answers
Consider the following customer Controller: <pre>@Controller("/customers") public class CustomerController {     @RequestMapping("/list")     public String list(Model model) {         return "customers";     }     @RequestMapping("/process")     public String process(Model model) {         // return     } }</pre> If I want to redirect the <code>list</code> method from <code>process</code> , how should I form the return statement with the <code>process</code> method?	3. return "redirect:customers/list"
Consider the following resource configuration: <pre>@Override public void addResourceHandlers(ResourceHandlerRegistry registry) {     registry.addResourceHandler("/resources/**")         .addResourceLocations("/pdf/"); }</pre> Under the <code>pdf</code> directory, if I have a sub-directory such as <code>product/manuals/</code> , which contains a PDF file called <code>manual-P1234.pdf</code> , how can I form the request path to access that PDF file?	2. <code>/resources/product/manuals/manual-P1234.pdf</code>

## Chapter 6, Internalize Your Store with Interceptor

Questions	Answers
<p>Consider the following interceptor:</p> <pre>public class SecurityInterceptor extends HandlerInterceptorAdapter{     @Override     public void     afterCompletion(HttpServletRequest     request, HttpServletResponse response,     Object handler, Exception ex)      throws     Exception {         // just some code related to after         completion     } }</pre> <p>Is this <code>SecurityInterceptor</code> class a valid interceptor?</p>	<p>2. It is valid because it extends the <code>HandlerInterceptorAdapter</code> class.</p>
<p>Within the interceptor methods, what is the order of execution?</p>	<p>2. <code>preHandle</code>, <code>postHandle</code>, <code>afterCompletion</code>.</p>

## Chapter 7, Incorporating Spring Security

Questions	Answers
<p>Which URL is the Spring Security default authentication handler listening on for the username and password?</p>	<p>1. <code>/login</code></p>
<p>What is the default logout handler URL for Spring Security?</p>	<p>1. <code>/logout</code></p>

## Chapter 10, Float Your Application with Web Flow

Questions	Answers
<p>Consider the following web flow registry configuration; it has a single flow definition file, namely validate.xml. How will you form the URL to invoke the flow?</p> <pre>@Bean public FlowDefinitionRegistry flowRegistry() {     return getFlowDefinitionRegistryBuilder()         .setBasePath("/WEB-INF/flows") .addFlowLocation("/customer/validate.xml","validateCustomer")         .build(); }</pre>	<p>3. <a href="http://localhost:8080/webstore/validateCustomer">http://localhost:8080/webstore/validateCustomer</a></p>
<p>Consider the following flow invoking URL: <a href="http://localhost:8080/webstore/validate?customerId=C1234">http://localhost:8080/webstore/validate?customerId=C1234</a></p> <p>In a flow definition file, how will you retrieve the customerId HTTP request parameter?</p>	<p>1. &lt;evaluate expression = "requestParameters.customerId" result = "customerId" /&gt;</p>

## Chapter 11, Template with Tiles

Questions	Answers
Which of the following statements are true according to Apache Tiles?	4. All of the above.

# Index

## @

@RequestMapping annotation, for CRUD methods  
reference links 229

## A

Apache Tiles  
about 288, 298  
used, for enhancing reusability 288  
Views, creating for every View state 289

Apache Tomcat  
homepage link 15

Asynchronous JavaScript and XML (Ajax)  
about 236  
REST web services, consuming via 236  
web services, handling 236

## B

Bean Validation annotation  
reference link 188

build tool  
configuring 12  
Maven build tool, installing 13

## C

Cart domain  
tests, adding 304

checkout flow  
about 265  
decision state, adding 286  
Views, creating for every view state 271  
web flow registry configurations 270

ContentNegotiatingViewResolver  
configuring 143  
using 142

controller method 81

Controller, Spring MVC

about 79  
class level request mapping 84  
class-level request mapping, adding 81  
default request mapping method 83  
defining 80

custom validation  
adding, to category 200  
adding, with Java Bean Validation (JSR-303)  
195

Bean Validation support, adding 195

## D

data binding  
customizing 116, 121  
form fields, whitelisting for 117

Data Transfer Objects (DTO) 108

dependency injection (DI) 45, 69

development environment  
configuring 16

Dispatcher servlet configuration  
servlet mapping, examining 42

Dispatcher servlet  
about 38  
configuration 41  
request mapping, examining 39, 41  
servlet mapping, versus request mapping 43

Domain layer, web application architecture  
about 53  
domain object, creating 54

Domain Specific Language (DSL) 317

## F

Flash attributes  
about 130  
using 130

flow definition  
about 264  
action-state 264  
checkout flow 265  
decision-state 264  
end-state 264  
start-state 264  
subflow-state 264  
view-state 264  
form binding 107  
forms  
customer registration form, creating 116  
processing 107  
serving 107  
Front Controller Pattern 32

## G

Google translate service  
reference link 167  
Gradle build script  
configuring 318  
Gradle script 319  
Gradle  
download link 317  
installation, steps 317

## H

handler mapping 85  
handler method 81  
HandlerExceptionResolver  
exception handler, adding 150  
ResponseStatus exception, adding 148  
working with 147  
HandlerInterceptor interface  
afterCompletion method 156  
perHandle method 156  
postHandle method 156  
Hibernate Validator 187  
HTTP Status 404 error 31  
HyperSQL DB 60, 66

## I

integrated development environment (IDE) 16  
integration testing 304

interceptors  
about 155, 160  
configuring 156  
working with 156

## J

Java Bean Validation (JSR-303)  
about 187  
adding, in Add new product page 195  
support, adding 188  
used, for custom validation 195

Java Development Kit (JDK)  
about 9  
environment variables, setting up 11  
installing 10

Java Runtime Environment (JRE) 11

Java SE  
download link 10  
JavaServer Pages (JSP) 27, 106  
JavaServer Pages Standard Tag Library (JSTL)  
about 70, 106  
jsp-api 305

## L

LocaleChangeInterceptor  
about 161  
internationalization, adding 162  
internationalization, adding to products page 167

## M

mapped interceptors  
about 167  
used, for intercepting offer page requests 168  
mapped method 81  
matrix variable  
product based on filters, displaying 92  
using 91  
Maven  
download link 13  
Model View Controller (MVC)  
about 51  
Controller 51  
Model 51

View 51  
multipart request  
  images, adding to product 137  
  product user manuals, uploading to server 142  
  using 136

## P

Persistence layer, web application architecture  
  about 59  
  repository object, creating 60  
Postman  
  about 230  
  reference link 230  
product Controllers  
  testing 309

## R

RedirectView  
  about 127, 129  
  examining 127  
request mapped method 81  
request parameters  
  about 96, 101  
  master detail View, implementing 101  
  multiple filters, adding to list products 104  
  product detail page, adding 97  
REST 211  
REST-based Controllers  
  testing 312  
RESTful web services  
  consuming 230  
  implementing 212

## S

Service layer, web application architecture  
  about 70  
  product domain object, accessing via service 75  
  service object, creating 71  
servlet-api 26  
Spring MVC project  
  configuring 20  
  creating, in Spring Tool Suite (STS) 21  
  dependencies 25  
  deploying 35

Dispatcher servlet 31  
Dispatcher servlet, configuring 32  
Java version properties, adding in pom.xml 22  
running 35

Spring jars, adding 25, 27  
welcome page, adding 29  
working with 29

Spring MVC request flow  
  overview 52

Spring MVC Test framework 299

Spring MVC  
  advantages 106  
  Controller 79

Spring Security  
  users based on roles, authenticating 176  
  using 175

Spring Test context framework  
  product validator, testing 305  
  used, for integration testing 304

Spring Tool Suite (STS)  
  download link 17  
  installing 16  
  Maven, configuring on 17  
  Tomcat, configuring 18

Spring validation  
  adding 201  
  adding, to product image 209  
  combining, with Bean Validation 204  
  using 200

Spring Web Flow (SWF)  
  about 246  
  checkout flow, implementing 261  
  order processing service, implementing 247  
  working with 246

static resources  
  images, adding to product detail page 134  
  serving 131  
  static view 133

## T

text messages  
  all labels, externalizing from all pages 124  
  externalizing 121  
Tomcat 8. 15  
transactional management, Spring

reference link 74  
transitive dependencies 28

## U

unit testing  
  about 300  
  domain objects 300  
URI template patterns  
  products based on category, displaying 86  
  request path variable 90  
  using 86  
user interface (UI) 247

## V

Views  
  resolving 125

## W

web application architecture  
  about 53

customers, listing 76  
Domain layer 53  
overview 76  
Persistence layer 59  
Service layer 70  
web application context configuration  
  @ComponentScan 49  
  @Configuration 49  
  @EnableWebMvc 49  
  about 49  
web application context  
  about 44, 46  
  configuration 49  
  view resolver 45  
web applications  
  testing 299  
Web MVC pattern 52  
web server  
  installing 15  
  Tomcat web server, installing 15  
wiring 69