Spring Essentials

Build mission-critical enterprise applications using Spring Framework and Aspect Oriented Programming

Shameer Kunjumohamed
Hamidreza Sattari

[PACKT] open source*
PUBLISHING
community experience distilled

Spring Essentials

Table of Contents

Spring Essentials

Spring Essentials

First published: February 2016

Production reference: 1230216

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78398-234-9

www.packtpub.com

Credits

Authors

Shameer Kunjumohamed

Hamidreza Sattari

Reviewer

Jarosław Krochmalski

Commissioning Editor

Julian Ursell

Acquisition Editors

Larissa Pinto

Pratik Shah

Content Development Editor

Aishwarya Pandere

Technical Editor

Siddhi Rane

Copy Editors

Kevin McGowan

Madhusudan Uchil

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Hemangini Bari

Graphics

Kirk D'Penha

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

About the Authors

Shameer Kunjumohamed is a software architect specializing in Java-based enterprise application integrations, SOA, and the cloud. Besides Java, he is well-versed in the Node.js and Microsoft .NET platforms. He is interested in JavaScript MVC frameworks such as EmberJS, AngularJS, and ReactJS.

Shameer has co-authored another book, *Spring Web Services 2 Cookbook, Packt Publishing* with Hamidreza Sattari, who is the co-author of this book as well.

Based in Dubai, UAE, Shameer has over 15 years of experience in various functional domains. He currently works as a principal applications architect for a major shipping company in Dubai.

I would like to extend my thanks to a number of people who have inspired and influenced me throughout my technical career. The Java, Spring, and Ember communities gave me the knowledge and confidence to write this book. I thank my parents; my wife, Shehida; and my daughters, Shireen, Shahreen, and Safa, who supported me and put up with me when I was busy writing the chapters; it was their precious time I was taking for this book. Special thanks to my friend Hamidreza, who is a great friend and colleague, and to the coordinators and reviewers at Packt Publishing, who made this book a wonderful resource for learning Spring.

Hamidreza Sattari is an IT professional and has worked in several areas of software engineering, from programming to architecture as well as management. He holds a master's degree in software engineering from Herriot Watt University, UK. In recent years, his areas of interest have been software architecture, data science, and machine learning. He co-authored the book *Spring Web Services 2 Cookbook*, *Packt Publishing* in 2012. He maintains the blog http://justdeveloped-blog.blogspot.com/.

First, I should thank the members of the open source community, who are far too many to name. I have been able to write this book by using their products, ideas, articles, and blogs. I would like to give special thanks to my friend Shameer P.K. for his significant role in writing this book.

About the Reviewer

Jarosław Krochmalski is a passionate software designer and developer who specializes in the financial business domain. He has over 12 years of experience in software development. He is a clean code and software craftsmanship enthusiast. He is a Certified ScrumMaster and a fan of Agile. His professional interests include new technologies in web application development, design patterns, enterprise architecture, and integration patterns. He has been designing and developing software professionally since 2000 and has been using Java as his primary programming language since 2002. In the past, he worked for companies such as Kredyt Bank (KBC) and Bank BPS on many large-scale projects, such as international money orders, express payments, and collection systems. He currently works as a consultant at the Danish company 7N as an IT architect for Nykredit Bank. You can reach him via Twitter at `@jkroch` or by e-mail at <`jarek@finsys.pl`>.

I would like to say hello to my friends at 7N and Nykredit; keep up the great job!

www.PacktPub.com

Preface

There are a lot of books written about Spring Framework and its subprojects. A multitude of online references are also available. Most of these massive resources discuss Spring in a lot of detail, which makes learning Spring a very time-consuming and sometimes tedious effort. The idea of this book is to allow novice Java developers or architects to master Spring without spending much time and effort and at the same time provide them with a strong foundation on the topic in order to enable them to design high-performance systems that are scalable and easily maintainable.

We have been using Spring Framework and its subprojects for more than a decade to develop enterprise applications in various domains. While the usage of Spring quickly raises the design quality of projects with its smart templates and subframeworks abstracting many error-prone and routine programming tasks, a developer needs a thorough understanding of its concepts, features, best practices, and above all, the Spring programming model in order to utilize Spring to its best.

We have seen Spring used wrongly inside many projects mainly because the developer either didn’t understand the right use of a particular Spring component or didn’t bother to follow the design approach Spring suggests for that component. Often, developers didn’t appear to have the right knowledge of Spring Framework; when asked, their complaint mostly was the uphill task of learning such a vast framework from huge documentation. Most of this category of developers find Spring a mammoth framework that is difficult to learn, which is not really true.

Spring, if the basics are understood correctly, is very easy to conquer further. A developer needs to understand the Spring style of programming and architecting applications, and the result will be a piece of art. The design will look simple, pretty straightforward, and easily understandable, which is very important for the evolution of applications in the long run. This book is an attempt to fill that gap and provide a very solid foundation in Spring, its concepts, design styles, and best practices, in a very quick and easy way.

This book tries to engage the reader by providing the feeling of developing a realistic, modern enterprise application using Spring and its necessary features while giving him or her a solid understanding of its concepts, benefits, and usage with real-life examples. It covers the most important concepts and features of Spring Framework and a few of its critical subprojects that are necessary for building modern web applications.

The goal of Spring is to simplify enterprise application development. We hope this book simplifies mastering Spring so that developers can build smarter systems that make the world a better place.

What this book covers

Chapter 1, *Getting Started with Spring Core*, introduces the core Spring Framework, including its core concepts, such as POJO-based programming, Dependency Injection, and Aspect Oriented Programming, to the reader. It further explains the Spring IoC container, bean configurations, Spring Expression Language (SpEL), resource management, and bean definition profiles, all which become the foundation for the advanced topics.

Chapter 2, *Building the Web Layer with Spring Web MVC*, gives in-depth coverage of the Spring MVC web framework with its features and various different ways of configuring and tuning web applications using Spring. The chapter covers the building of both view-based web applications and REST APIs with many available options, including asynchronous request processing.

Chapter 3, *Accessing Data with Spring*, discusses the different data-access and persistence mechanisms that Spring offers, including the Spring Data family of projects, such as Spring Data JPA and Spring Data Mongo. This chapter enables the reader to design an elegant data layer for his or her Spring application, delegating all the heavylifting to Spring.

Chapter 4, *Understanding WebSocket*, discusses the WebSocket technology, which is gaining wider usage inside modern web applications, where low latency and high frequency of communication are critical. This chapter explains how to create a WebSocket application and broadcast a message to all subscribed clients as well as send a message to a specific client, and shows how a broker-based messaging system works with STOMP over WebSocket. It also shows how Spring's WebSocket fallback option can tackle browser incompatibility.

Chapter 5, *Securing Your Applications*, teaches the reader how to secure his or her Spring applications. It starts with authentication and explains Spring flexibility on authorization. You learn how to integrate your existing authentication framework with Spring. On authorization, it shows how to use Spring EL expressions for web, method, and domain object authorization. It also explains the OAuth 2.0 Authorization Framework and how to allow third-party limited access to user's protected resources on a server without sharing user's username and password.

Chapter 6, *Building a Single-Page Spring Application*, demonstrates how Spring can be used as the API server for modern single-page applications (SPAs) with an example of an Ember JS application. At first, it introduces the concept of SPAs, and then it explores using Ember JS to build the SPA. Finally, it covers building the backend API that processes requests asynchronously using Spring MVC and implements persistence using Spring Data JPA.

Chapter 7, *Integrating with Other Web Frameworks*, demonstrates how Spring can be integrated with Java web frameworks such as JSF and Struts so that even web applications not based on Spring MVC can leverage the power of Spring.

What you need for this book

In order to execute the sample projects used in this book, you need the following software installed on your computer:

For all chapters, in general, you need the following software:

- Java version 8 onwards
- Spring Framework 4.x
- Apache Maven 3.x.x
- Apache Tomcat 8.x

For Chapter 3, *Accessing Data with Spring*, you need the following databases:

- PostgreSQL 8 onwards
- MongoDB 3.x

Additionally, for Chapter 6, *Building a Single-Page Spring Application*, you need the following software:

- Node.js version 4.x
- Bower JS in the latest version

Who this book is for

If you are a Java developer who is looking to master enterprise Java development using Spring Framework, then this book is ideal for you. A prior understanding of core Java programming and a high-level understanding of Spring Framework is recommended. Having sound knowledge of Servlet-based web development in Java and basic database concepts would be an advantage but not a requirement.

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: "Spring provides mock classes for both client and server sides inside the `org.springframework.mock.http` and `org.springframework.mock.http.client` packages."

A block of code is set as follows:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework.version}</version>
  </dependency>
  ...
</dependencies>
```

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

```
mvn clean package spring-boot:run -Dserver.contextPath=/myapp -
Dserver.port=9090
```

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: "STS and Eclipse allow you to run Java web apps from the IDE just by right-clicking **Run As** and then **Run on Server**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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**.

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

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/SpringEssentials_ColorImages.pd

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.

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.

Chapter 1. Getting Started with Spring Core

Spring Framework is the most trusted and widely used application development framework in Enterprise Java. Originally introduced as a simple and lightweight alternative for the complex J2EE, Spring has now grown to become a truly modern application development platform. Spring and its subprojects provide an excellent foundation for end-to-end application development, with features beyond even those provided by the latest Java EE, such as mobile development, social networking, and big data, besides traditional Java web, server-side, or even standalone applications. After more than a decade since its inception, Spring continues to inspire technologies and technologists across the globe.

Although Spring simplifies Java development drastically, software developers and architects are still required to gain a thorough understanding of its core concepts and features in order to deduce the best use of the Spring family. The simplicity Spring offers to otherwise complex Java development is the result of smart abstractions that it provides in the form of excellent APIs and modules. Spring components relieve the developer of all the technical complexity and heavy lifting of common technical and infrastructure plumbing tasks. As the official Spring documentation says, Spring provides comprehensive infrastructure support so that you can focus on your application.

This book is an attempt to make your Spring learning even easier and a more enjoyable experience.

This chapter gives you a solid foundation of the core Spring Framework, guiding you through its core concepts, components, and modules accompanied by relevant sample code snippets that illustrate the best and most practical usage of each feature in order to solve your everyday programming problems.

In this chapter, we will cover the following topics:

- The Spring landscape
- Setting up the development environment
- Your first Spring application
- Core concepts
- The IoC (Inversion of Control) container
- Beans in detail
- Working with bean definition profiles
- Handling resources
- SpEL (Spring Expression Language)
- Aspect Oriented Programming

The Spring landscape

Spring covers a wide variety of technological aspects handled by applications of different types, ranging from a simple standalone Java application up to the most complex, mission critical distributed enterprise systems you can imagine. Unlike most other open source or proprietary frameworks that focus on a specific technology concern such as Web, Messaging, or Remoting, Spring successfully covers almost all the technical aspects of business applications. In most cases, instead of reinventing solutions, Spring utilizes and integrates proven existing frameworks to achieve this end-to-end coverage. Spring is highly modular; hence, it noninvasively allows you to cherry-pick just the modules or features you require in order to become a one-stop shop for all your development needs on JVM.

The whole Spring Framework portfolio is organized into three major elements:

- Spring Framework
- Spring Tool Suite
- Spring subprojects

Spring is constantly improving and becoming more and more modular with every new version so that you can use just the required modules.

Note

This book is based on Spring version 4.

The Spring Framework modules

The core Spring Framework provides basic infrastructure for Java development on top of its core **Inversion of Control** (**IoC**) container. The IoC container is an infrastructure that provides **Dependency Injection** (**DI**) for applications. Both the concepts of Dependency Injection and IoC containers are explained in detail later in this chapter. The core Spring Framework is divided into the following modules, providing a range of services:

Module	**Summary**
Core container	Provides the IoC and Dependency Injection features.
AOP and instrumentation	Provides AOP Alliance compliant features for weaving cross-cutting concerns in Spring applications.
Messaging	Provides messaging abstraction over the Spring Integration project for messaging-based applications.
Data access/integration	The data-access/integration layer consists of JDBC, ORM, OXM, JMS, and transaction modules.
Web	Web technology abstraction over Spring MVC, web socket, and portlet APIs.
Test	Unit testing and integration testing support with JUnit and TestNG frameworks.

Spring Tool Suite (STS)

STS is an Eclipse-based **IDE** (short for **Integrated Development Environment**) for Spring development. You can download the pre-bundled STS from http://spring.io/tools/sts/all or update your existing Eclipse installation from the update site found at the same location. STS provides various high-productivity features for Spring development. In fact, a Java developer can use any IDE of their choice. Almost all the Java IDEs support Spring development, and most of them have got plugins available for Spring.

Spring subprojects

Spring has many subprojects that solve various application infrastructure needs. From configuration to security, web apps to big data, productivity to **enterprise application integration** (**EAI**), whatever your technical pain point be, you will find a Spring project to help you in your application development. Spring projects are located at http://spring.io/projects.

Some notable projects you may find useful right away are Spring Data (JPA, Mongo, Redis, and so on), Spring Security, Spring Web Services, Spring Integration, Spring for Android, and Spring Boot.

Design concepts behind Spring Framework

The design of Spring Framework is motivated by a set of design patterns and best practices that have evolved in the industry to address the complexity of Object Oriented Programming, including:

- Simple, noninvasive, and lightweight **POJO** (**Plain Old Java Objects**) programming, without having a need for complex application servers
- Loosely-coupled dependencies, achieved by applying the concepts of *program to interfaces* and *composition over inheritance*, which are the underlying design principles of design patterns and frameworks
- Highly configurable systems composed of objects with externalized Dependency Injection
- Templated abstractions to eliminate repetitive, boilerplate code
- Declarative weaving of cross-cutting aspects without polluting business components

Spring implements established design principles and patterns into its elegant components and promotes their use as the default design approach in applications built using Spring. This noninvasive approach lets you engineer robust and highly maintainable systems composed of loosely coupled components and objects written in clean and modular code. Spring Framework components, templates, and libraries realize the goals and concepts explained earlier in the chapter, leaving you to focus on your core business logic.

Setting up the development environment

Spring projects are usually created as Java projects based in Maven, Gradle, or Ivy (which are build automation and dependency management tools). You can easily create a Maven-based Spring project using STS or Eclipse with Spring Tools support. You need to make sure your `pom.xml` (Maven configuration) file contains, at the minimum, a dependency to `spring-context`:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring-framework.version}</version>
  </dependency>
  ...
</dependencies>
```

Of course, you should add further dependencies to modules such as `spring-tx`, `spring-data-jpa`, `spring-webmvc`, and `hibernate`, depending on your project type and requirements.

Unless you explicitly specify the repository location, your project works with Maven’s central repository. Alternatively, you can point to Spring’s official Maven repository (for example, for milestones and snapshots) by specifying it in your `pom.xml` file:

```
<repositories>
    <repository>
        <id>io.spring.repo.maven.milestone</id>
        <url>http://repo.spring.io/milestone/</url>
        <snapshots><enabled>false</enabled></snapshots>
    </repository>
</repositories>
```

You can use the Spring `release`, `milestone`, and `snapshot` repositories as required.

If you are using Gradle as your build system, you can declare your dependencies (typically in the `build.gradle` file) as follows:

```
dependencies {
    compile('org.springframework:spring-context')
    compile('org.springframework:spring-tx')
    compile('org.hibernate:hibernate-entitymanager')
    testCompile('junit:junit')
}
```

If you prefer using the Ivy dependency management tool, then your Spring dependency configuration will look like this:

```
<dependency org="org.springframework"
    name="spring-core" rev="4.2.0.RC3" conf="compile->runtime"/>
```

Your first Spring application

Let's start with a very simple Spring application now. This application simply greets the user with a welcome message. Technically, it demonstrates how you configure a Spring `ApplicationContext` (IoC container) with just a single bean in it and invoke that bean method in your application. The application has four artifacts in it (besides the project build file, of course):

- `GreetingService.java`: A Java interface—just a single method
- `GreetingServiceImpl.java`: A simple implementation of `GreetingService`
- `Application.java`: Your application with a `main` method
- `application-context.xml`: The Spring configuration file of your application

The following are the service components of your application. The service implementation just prints a greeting message to the logger:

```
interface GreetingService {
   void greet(String message);
}

public class GreetingServiceImpl implements GreetingService {
   Logger logger = LoggerFactory.getLogger(GreetingService.class);

   public void greet(String message) {
      logger.info("Greetings! " + message);
   }
}
```

Now let's take a look at the `application-context.xml` file, which is your Spring configuration file, where you register `GreetingService` as a Spring bean in the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
   <bean id="Greeter"
      class="com.springessentialsbook.chapter1.GreetingServiceImpl">
   </bean>
</beans>
```

Finally, you invoke the `GreetingService.greet()` method from your Spring application, as given in the following code:

```
public class Application {

   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext(new
String[] {"application-context.xml"});
      GreetingService greeter = (GreetingService)
context.getBean("Greeter");
     greeter.greet("I am your first Spring bean instance, configured purely
with XML metadata. I am resolved by name.");
   }
}
```

We will explore and conquer the mighty Spring Framework right from this very simple and pretty much self-explanatory application. We will discuss and elaborate the concepts behind this application, and more, in the following sections.

Inversion of Control explained

IoC is a design principle that decouples objects of an object-oriented program from their dependencies (collaborators), that is, the objects they work with. Usually, this decoupling is achieved by externalizing the responsibility of object creation and Dependency Injection to an external component, such as an IoC container.

This concept is often compared to the Hollywood principle, “Don’t call us, we will call you.” In the programming world, it recommends the main program (or a component) not to instantiate its dependencies by itself but let an assembler do that job.

This immediately decouples the program from the many problems caused by tightly coupled dependencies and relieves the programmer to let them quickly develop their code using abstract dependencies (*program to interfaces*). Later, at runtime, an external entity, such as an IoC container, resolves their concentrate implementations specified somewhere and injects them at runtime.

You can see this concept implemented in the example we just saw. Your main program (`Application.java`) is not instantiating the `GreetingService` dependency; it just asks the `ApplicationContext` (IoC container) to return an instance. While writing `Application.java`, the developer doesn’t need to think about how the `GreetingService` interface is actually implemented. The Spring `ApplicationContext` takes care of object creation and injects any other functionality transparently, keeping the application code clean.

Objects managed by an IoC container do not control the creation and resolution of their dependencies by themselves; rather, that control is inverted by moving it away to the container itself; hence the term “Inversion of Control”.

The IoC container assembles the components of the application as specified in the configuration. It handles the life cycles of the managed objects.

Dependency Injection

Dependency Injection is a specific form of Inversion of Control. It is a more formalized design pattern, whereby dependencies of an object are injected by an assembler. DI is generally performed in three major styles: constructor injection, property (setter) injection, or, sometimes, interface injection. IoC and DI are often used interchangeably.

DI offers several benefits, including effective decoupling of dependencies, cleaner code, and increased testability.

The Spring IoC container

The core Spring modules, `spring-core`, `spring-beans`, `spring-context`, `spring-context-support`, and `spring-expression`, together make up the core container. The Spring IoC container is designed as an implementation of the following interfaces:

- `org.springframework.beans.factory.BeanFactory`
- `org.springframework.context.ApplicationContext`

The `BeanFactory` interface provides the configuration framework and basic functionality, while `ApplicationContext`, an extension of `BeanFactory`, adds more enterprise-specific functionality, such as easier integration with Spring's AOP features, message resource handling (for internationalization), and event publication.

Spring provides several concrete implementations of `ApplicationContext` out of the box for various contexts. The following table lists the most popular ones among them:

Application context	**Typical application type**
`ClassPathXmlApplicationContext`	Standalone
`AnnotationConfigApplicationContext`	Standalone
`FileSystemXmlApplicationContext`	Standalone
`GenericWebApplicationContext`	Web
`XmlWebApplicationContext`	Web
`XmlPortletApplicationContext`	Web portlet

In Spring, objects managed by the IoC container are called **beans**. The IoC container handles the assembly and lifecycles of Spring beans. Beans are defined in the configuration metadata consumed by the container, which instantiates and assembles them in order to compose your application.

Configuration metadata

Spring supports three forms of configuration metadata to configure your beans:

- XML-based configuration metadata
- Annotation-based configuration metadata
- Java-based configuration metadata

The example code listing you saw earlier used XML-based configuration metadata. You can always mix and match different forms of metadata in a single application. For example, you may define the primary metadata to be a root XML file that combines a set of annotation-based metadata that in turn defines beans from different layers.

XML-based configuration metadata

The `application-context.xml` file we saw in the previous Spring application sample is a very minimal example for XML-based configuration metadata. Beans are configured as `<bean/>` elements inside a top-level `<beans>` element.

Classes representing the service layer (core business logic, also known as **Service** classes), **Data Access Objects** (**DAOs**), managed web backing beans (such as Struts action instances and JSF managed beans), infrastructure objects (such as Hibernate session factories and JMS queues), and so forth, are excellent candidates for Spring beans. Fine-grained domain objects are not generally configured as Spring beans, because it is usually the responsibility of DAOs and the business logic to create and load domain objects—Hibernate entities are typical examples.

You can create a consolidated (root) `ApplicationContext` XML file that imports other XML files representing various layers of the application:

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

   <import resource="/xml-data-access-objects.xml"/>
   <import resource="/xml-services.xml"/>
   <import resource="/web-beans.xml"/>
   <import resource="/rest-endpoints.xml"/>
...
   <bean id="systemSettings" class="com…SystemSettings">
</beans>
```

Annotation-based configuration metadata

This method relies on bytecode metadata to wire up components instead of XML-based angle bracket declarations. Configuration of a bean is defined at the source level of the bean itself, in the form of annotations at class, field, or method levels.

Let’s take a look at the simplest Spring bean configured by source-level annotation:

```
@Component("Greeter")
public class GreetingServiceImpl implements GreetingService {

   Logger logger = LoggerFactory.getLogger(GreetingService.class);
   public void greet(String message) {
      logger.info("Greetings! " + message);
   }
}
```

This is just an annotated version of the same `GreetingServiceImpl` shown in the *Your first Spring application* section, where it was configured in the `application-context.xml` file purely in XML form. In this preceding listing, the annotation `@Component` makes it a Spring bean. Now, it doesn't require to be defined in XML, but you should instruct your `ApplicationContext` to consider annotations, as given in the following code:

```
<context:component-scan base-package="com.springessentialsbook"/>
```

This code snippet in your `application-context.xml` file forces `ApplicationContext` to scan the entire application, including all its dependencies—even inside JAR files—for components annotated as Spring beans of various stereotypes, such as `@Component`, `@Service`, `@Repository`, and `@Controller`. In addition to component scanning, the `ApplicationContext` looks for all the annotations in that bean at the class, property, constructor, and method levels (including setter methods) in order to inject dependencies and other behaviors into your beans at startup.

Beware, component scanning can be time consuming if you provide a broader package name to the `base-package` attribute; it is advised to provide more specific package names to scan (for example, a set of comma-separated package names) so that you have more control. You can narrow down your component scanning even further using `<context:include-filter/>` and `<context:exclude-filter/>`.

Another simple instruction to enable annotation configuration is `<context:annotation-config/>`. It simply looks for annotations on beans registered in the application context and will not detect the components, whereas if you use `<context:component-scan/>`, it handles both component scanning and other annotations, which will be covered later in this chapter, so you do not need to explicitly declare `<context:annotation-config/>`. So, the best method for annotation-based configuration is to use `<context:annotation-config/>`.

XML-based versus annotation-based configuration

XML-based configuration has some advantages over its annotation-based counterpart. The biggest one is that all your bean definitions are in one place and not scattered in many classes or even JAR dependencies. XML allows you to split your metadata files and then combine them using `<import/>`. Using XML, you can configure any class, including third-party ones such as Spring beans, and inject dependencies and other services into it, which is impossible in the case of annotation. Also, you can define the same class as multiple different beans, each with a different name, dependencies, configuration, and so on.

Annotation-based metadata too has some advantages over XML configuration. It is more concise and much easier to develop and maintain, as your annotation and DI are right inside your source code. All information about a class is in one place.

For bigger applications, the best option would be a mixed approach where the more reusable beans (libraries shared between multiple projects) and third-party components are configured in XML and those with a smaller scope are annotated.

Component stereotype annotations

Spring provides further component stereotypes for beans that represent various roles. The primary stereotype is `@Component`, and all the others are its specializations for more specific use cases:

Stereotype	**Description**
`@Component`	A generic type for all Spring-managed components (beans).
`@Service`	Marker meta-annotation for service layer components. Currently, Spring treats this the same as `@Component`, with no special function.
`@Repository`	Used as DAOs in your persistence layer. Spring Data libraries provide additional functionality.
`@Controller`	Handles Web MVC endpoints in order to process HTTP requests mapped to specific URLs.
`@RestController`	A specialized controller for RESTful web services, part of Web MVC. It is a meta-annotation that combines `@Controller` and `@ResponseBody`.

Custom stereotypes can be created by defining meta-annotations from scratch or combining existing annotations.

Java-based configuration metadata

Starting with Spring 3.0, you can configure Spring metadata purely inside Java classes, completely avoiding any XML configuration while enhancing annotation-based metadata. You annotate any Java class with `@Configuration` annotation at the class level and have methods annotated as `@Configuration` annotation on a factory method that instantiates an `@Component` annotation, or any other specialized bean, to define your application context. Let’s see a simple example:

```
@Configuration
@ComponentScan(basePackages = "com.springessentialsbook")
public class SpringJavaConfigurator {

    @Autowired
    private GreetingService greeter;

    @Autowired
    private BannerService banner;

    @Bean
    public BannerService createBanner() {
        return new BannerService();
    }

    public BannerService getBanner() {
        return this.banner;
    }

    public void run() {
        this.banner.displayBanner();
        this.greeter.greet("I am the Greeter Spring bean, configured with
Java Configuration.");
    }
}
```

In `SpringJavaConfigurator.java`, the Java configuration class configures the Spring beans, replacing the `application-context.xml` file. Your Spring application can directly depend on this `Configuration` class for loading `ApplicationContext`.

Typically, you use an `AnnotationConfigApplication` instance for instantiating your application context:

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(
  SpringJavaConfigurator.class);
SpringJavaConfigurator app = ctx.getBean(SpringJavaConfigurator.class);
app.run();
BannerService banner = ctx.getBean(BannerService.class);
banner.displayBanner();
```

When `@Configuration` classes are provided as the constructor argument, the `@Configuration` class itself is registered as a bean definition and so are all declared `@Bean` methods within the class. Spring will scan for the entire project and its dependencies for `@Component` or its specializations (the other stereotypes listed previously), matching the argument values provided in `@ComponentScan(basePackages = "…")` with all other relevant annotations and building the application context.

The advantage of JavaConfig metadata is that you have programmatic control over Spring configuration while separating out the entire DI and bean configuration into a separate Java class. Using JavaConfig, you eliminate the complexity of managing many XML files. You detect any configuration issues during development at the earliest, as JavaConfig fails during compilation itself, while in the case of XML, you will know about the configuration issues only on application startup.

JSR 330 standard annotations

Besides Spring-specific annotations, Spring supports JSR 330 standard annotations for DI, starting from Spring 3.0. You just need to include `javax.inject` artifacts in your Maven or Gradle configuration.

JSR 330 standard annotations have the following equivalents in Spring:

Spring	**JSR-330 (javax.inject.*)**	**Target level / Usage**
`@Component`	`@Named`	Type (class)
`@Autowired`	`@Inject`	Property and setter methods
`@Qualifier`	`@Named`	Type, property and setter methods

`@Scope("singleton")`	`@Singleton`	Meta-annotation for bean declarations

While the default scope of Spring beans is `singleton`, the JSR 330 default scope is like Spring's `prototype`. However, for consistency, Spring treats JSR 330 annotated beans inside Spring as `singleton`, unless declared prototype explicitly using `@Scope("..")`.

JSR 330 has no equivalents for some Spring-based DI annotations, such as `@Value`, `@Required`, and `@Lazy`. We will discuss more about bean scopes later in this chapter.

Beans in detail

A Spring application is composed of a set of beans that perform functionality specific to your application layers and are managed by the IoC container. You define your beans with configuration metadata in the form of XML, annotation, or JavaConfig.

Note

The default scope of a Spring bean is `singleton`. This means that a single instance is shared between clients anywhere in the application. Beware of keeping state (class level data) in `singleton` classes, as a value set by one client will be visible to all others. The best use case for such `singleton` classes are stateless services.

Beans are uniquely identified by an `id` attribute, any of the values supplied to the (comma, semicolon, or space separated) `name` attribute of the bean definition, or even as an `alias` definition. You can refer to a bean anywhere in the application with `id` or any of the names or aliases specified in the bean definition.

It’s not necessary that you always provide an `id` or name to the bean. If one isn’t provided, Spring will generate a unique bean name for it; however, if you want to refer to it with a name or an `id`, then you must provide one.

Spring will try to autowire beans by type if `id` or name is not provided. This means that `ApplicationContext` will try to match the bean with the same type or implementation in case it is an interface.

You can refer to a bean by type if it is either the only bean registered of that type or marked as `@Primary` (`primary="true"` for XML). Generally, for nested bean definitions and autowire collaborators, you don’t need to define a name unless you refer to it outside the definition.

You can alias a bean outside the bean definition using the `<alias/>` tag, as follows:

```
<alias name="fromName" alias="toName"/>
```

Bean definition

A bean definition object that you define to describe a bean has the following metadata:

Property	**Description**
`class`	The fully qualified class name of the bean.
`id`	The unique identifier of the bean.
`name`	One or more unique names separated by commas, semicolons, or whitespace. Typically, `id` and name would be the same, and you supply either of these. Other names in the list become aliases.
`parent`	The parent bean for inheriting configuration data from a parent bean definition.
`scope`	This decides the scope of the objects. The default scope of a Spring bean is `singleton`. This means that a single instance is shared between calls. We will discuss more about bean scopes later.
`constructor args`	Bean references or names for constructor-based DI.
`properties`	Values or references for setter-based DI.
`autowire` mode	Instructs the bean whether or how to autowire relationships with collaborators. Autowiring will be discussed later.
`primary`	This indicates that the bean should be considered as the primary autowiring candidate in case of multiple matches being found.
`depends-on`	This forces instantiation of dependent beans prior to this bean.
`lazy-init`	If true, this creates a bean instance when it is first requested.
`init-method`	Initialization callback method. This has no `args void` method and will be invoked post instance creation.
`destroy-method`	Destruction callback method. This has no `args void` method and will be invoked before `destroy`.
`factory-method`	Static instance factory method on the bean itself, unless `factory-bean` is provided.
`factory-bean`	Another bean reference that is acting as an instance factory for this bean. Usually comes along with the `factory-method` property.

Let’s take a look at a sample bean definition in XML form:

```
<bean id="xmlTaskService" class="com…XmlDefinedTaskService"
init-method="init" destroy-method="cleanup">
   <constructor-arg ref="userService"/>
   <constructor-arg>
      <bean class="com…TaskInMemoryDAO"></bean>
   </constructor-arg>
</bean>
```

In this sample `application-context` file, the bean, `xmlTaskService`, is autowired via a constructor, that is, dependencies are injected via a constructor. The first constructor argument refers to an existing bean definition, and the second one is an inline bean definition without an `id`. The bean has `init-method` and `destroy-method` pointed to its own methods.

Now, let’s take a look at an annotated bean with slightly different features:

```
@Service
public class AnnotatedTaskService implements TaskService {

...
   @Autowired
   private UserService userService;

   @Autowired
   private TaskDAO taskDAO;

   @PostConstruct
   public void init() {
      logger.debug(this.getClass().getName() + " started!");
   }

   @PreDestroy
   public void cleanup() {
      logger.debug(this.getClass().getName() + " is about to destroy!");
   }

   public Task createTask(String name, int priority, int createdByuserId,
int assigneeUserId) {
      Task task = new Task(name, priority, "Open",
         userService.findById(createdByuserId), null,
         userService.findById(assigneeUserId));
      taskDAO.createTask(task);
      logger.info("Task created: " + task);
      return task;
   }
...
}
```

This `@Service` bean autowires its dependencies on its fields (properties) using an `@Autowired` annotation. Note the `@PostConstruct` and `@PreDestroy` annotations, the equivalents of `init-method` and `destroy-method` in the previous XML bean definition example. These are not Spring specific but are JSR 250 annotations. They work pretty well with Spring.

Instantiating beans

Bean definitions are recipes for instantiating bean instances. Depending on metadata attributes such as `scope`, `lazy`, and `depends-on`, Spring Framework decides when and how an instance is created. We will discuss it in detail later. Here, let’s look at the “how” of instance creation.

With constructors

Any bean definition with or without constructor arguments but without a `factory-method` is instantiated via its own constructor, using the `new` operator:

```
<bean id="greeter" class="com…GreetingBean"></bean>
```

Now let’s see an annotated `@Component` with a default constructor-based instantiation:

```
@Component("greeter")
public class GreetingService {
...
}
```

With a static factory-method

A static method within the same class, marked as `factory-method`, will be invoked to create an instance in this case:

```
<bean id="Greeter" class="...GreetingBean" factory-method="newInstance">
</bean>
```

With Java configuration, you can use an `@Bean` annotation instead of factory methods:

```
@Configuration
@ComponentScan(basePackages = "com.springessentialsbook")
public class SpringJavaConfigurator {
...
   @Bean
   public BannerService createBanner() {
      return new BannerServiceImpl();
   }
...
}
```

With an instance factory-method

In this case, bean definition does not need a class attribute, but you specify the `factory-bean` attribute, which is another bean, with one of its non-static methods as `factory-method`:

```
<bean id="greeter"  factory-bean="serviceFactory" factory-
method="createGreeter"/>
<bean id="serviceFactory"  class="...ServiceFactory">
<!— ... Dependencies… -->
</bean>
```

Injecting bean dependencies

The main purpose of an IoC container is to resolve the dependencies of objects (beans) before they are returned to the clients who called for an instance (say, using the `getBean` method). Spring does this job transparently based on the bean configuration. When the client receives the bean, all its dependencies are resolved unless specified as not required (`@Autowired(required = false)`), and it is ready to use.

Spring supports two major variants of DI—constructor-based and setter-based DI—right out of the box.

Constructor-based Dependency Injection

In constructor-based DI, dependencies to a bean are injected as constructor arguments. Basically, the container calls the defined constructor, passing the resolved values of the arguments. It is best practice to resolve mandatory dependencies via a constructor. Let’s look at an example of a simple POJO `@Service` class, a ready candidate for constructor-based DI:

```
public class SimpleTaskService implements TaskService {
...
   private UserService userService;
   private TaskDAO taskDAO;

   public SimpleTaskService(UserService userService, TaskDAO taskDAO) {
      this.userService = userService;
      this.taskDAO = taskDAO;
   }
...
}
```

Now, let’s define this as a Spring bean in XML:

```
<bean id="taskService" class="com…SimpleTaskService"">
   <constructor-arg ref="userService" />
   <constructor-arg ref="taskDAO"/>
</bean>
```

The Spring container resolves dependencies via a constructor based on the argument’s type. For the preceding example, you don’t need to pass the index or type of the arguments, since they are of complex types.

However, if your constructor has simple types, such as primitives (`int`, `long`, and `boolean`), primitive wrappers (`java.lang.Integer`, `Long`, and so on) or `String`, ambiguities of type and index may arise. In this case, you can explicitly specify the type and index of each argument to help the container match the arguments, as follows:

```
<bean id="systemSettings" class="com…SystemSettings">
   <constructor-arg index="0" type="int" value="5"/>
   <constructor-arg index="1" type="java.lang.String" value="dd/mm/yyyy"/>
   <constructor-arg index="2" type="java.lang.String" value="Taskify!"/>
</bean>
```

Remember, index numbers start from zero. The same applies to setter-based injection as well.

Setter-based Dependency Injection

The container calls the setter methods of your bean in the case of setter-based DI after the constructor (with or without `args`) is invoked. Let’s see how the bean definition for the previous `SystemSettings` would look if the dependencies were injected via setter methods, assuming the `SystemSettings` now has a `no-args` constructor:

```
<bean id="systemSettings" class="com…SystemSettings">
   <property name="openUserTasksMaxLimit" value="5"/>
   <property name="systemDateFormat" value="dd/mm/yyyy"/>
   <property name="appDisplayName" value="Taskify!"/>
</bean>
```

Spring validates the bean definitions at the startup of the `ApplicationContext` and fails with a proper message in case of a wrong configuration. The string values given to properties with built-in types such as `int`, `long`, `String`, and `boolean` are converted and injected automatically when the bean instances are created.

Constructor-based or setter-based DI – which is better?

Which of these DI methods is better purely depends on your scenario and some requirements. The following best practices may provide a guideline:

1. Use constructor-based DI for mandatory dependencies so that your bean is ready to use when it is first called.
2. When your constructor gets stuffed with a large number of arguments, it’s the figurative bad code smell. It’s time to break your bean into smaller units for maintainability.
3. Use setter-based DI only for optional dependencies or if you need to reinject dependencies later, perhaps using JMX.
4. Avoid circular dependencies that occur when a dependency (say, bean B) of your bean (bean A) directly or indirectly depends on the same bean again (bean A), and all beans involved use constructor-based DI. You may use setter-based DI here.
5. You can mix constructor-based and setter-based DI for the same bean, considering mandatory, optional, and circular dependencies.

In a typical Spring application, you can see dependencies injected using both approaches, but this depends on the scenario, considering the preceding guidelines.

Cleaner bean definitions with namespace shortcuts

You can make your bean definitions cleaner and more expressive using `p:(property)` and `c:(constructor)` namespaces, as shown here. While the `p` namespace enables you to use the `<bean/>` element's attributes instead of the nested `<property/>` elements in order to describe your property values (or collaborating bean refs), the `c` namespace allows you to declare the constructor `args` as the attributes of the `<bean/>` element:

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

   <bean id="p-taskService" class="com…SimpleTaskService" c:userService-
ref="userService" c:taskDAO-ref="taskDAO"/>

   <bean id="p-systemSettings" class="com…SystemSettings"
      p:openUserTasksMaxLimit="5"
      p:systemDateFormat"dd/mm/yyyy"
      p:appDisplayName="Taskify!"/>
</beans>
```

The bean definitions in the preceding listing are cleaner but more expressive. Both `c:` and `p:` namespaces follow the same conventions. You need to declare both at the XML root element (`<beans/>`) before using them with the `<bean/>` elements. Note that you use the `-ref` suffix for bean references.

Wiring a List as a dependency

On occasion, we will need to inject static collections of data as bean dependencies. Spring provides a natural method to wire lists. See this example:

```
<bean id="systemSettings" class="com…SystemSettings">
. . .
  <constructor-arg>
    <list>
      <value>admin@taskify.ae</value>
      <value>it@taskify.ae</value>
      <value>devops@taskify.ae</value>
    </list>
  </constructor-arg>
</bean>
```

The preceding example wires a `java.util.List<String>` for simplicity. If your list contains a collection of beans, you can replace `<value>` with `<ref>` or `<bean>`.

Wiring a Map as a dependency

You can inject `java.util.Map` instances too in a similar fashion. Look at this example:

```
<bean id="systemSettings" class="com…SystemSettings">
. . .
  <property name="emails">
    <map>
      <entry key="admin" value="admin@taskify.ae"></entry>
      <entry key="it" value="it@taskify.ae"></entry>
      <entry key="devops" value="devops@taskify.ae"></entry>
    </map>
  </property>
</bean>
```

You can inject beans as values, replacing `<value>` with `<ref>` or `<bean>`.

Autowiring dependencies

Spring can autowire dependencies of your beans automatically by inspecting the bean definitions present in the `ApplicationContext` if you specify the autowire mode. In XML, you specify the `autowire` attribute of the `<bean/>` element. Alternatively, you can annotate a bean with `@Autowired` to autowire dependencies. Spring supports four autowiring modes: `no`, `byName`, `byType`, and `constructor`.

Note

The default autowiring of Spring beans is `byType`. If you are autowiring an interface, Spring will try to find an implementation of that interface configured as a Spring bean. If there are multiple, Spring will look for the `primary` attribute of the configuration to resolve; if not found, it will fail, complaining about an ambiguous bean definition.

Here is an example of autowiring constructor arguments:

```
@Service
public class AnnotatedTaskService implements TaskService {
...
   @Autowired
   public AnnotatedTaskService(UserService userService, TaskDAO taskDAO) {
      this.userService = userService;
      this.taskDAO = taskDAO;
   }
...
}
```

Alternatively, you can autowire at the field level, as follows:

```
@Service
public class AnnotatedTaskService implements TaskService {
...
   @Autowired
   private UserService userService;
   @Autowired
   private TaskDAO taskDAO;
...
}
```

Autowiring can be fine-tuned with an `@Qualifier` annotation and required attribute:

```
@Autowired(required = true)
@Qualifier("taskDAO")
private UserService userService;
```

You can use `@Qualifier` at the constructor level too:

```
@Autowired
public AnnotatedTaskService(@Qualifier("userService") UserService
userService, @Qualifier("taskDAO") TaskDAO taskDAO) {
   this.userService = userService;
   this.taskDAO = taskDAO;
}
```

Bean scoping

When defining a bean with its dependencies and other configuration values, you can optionally specify the scope of a bean in the bean definition. The scope determines the life span of the bean. Spring comes up with six built-in scopes out of the box and supports the creation of custom scopes too. If not explicitly specified, a bean will assume the `singleton` scope, which is the default scope. The following table lists the built-in Spring scopes:

Scope	**Description**
`singleton`	This ensures a single instance inside the container. This is the default scope.
`prototype`	A new instance is created for every request for the bean.
`request`	Scopes an instance with the life cycle of every new HTTP request.
`session`	Scopes with the life cycle of every new HTTP session.
`globalSession`	Scopes with an HTTP session inside a portlet context.
`application`	Scopes with the life cycle of a `ServletContext`. It's `singleton` for `ServletContext`.

While `singleton` and `prototype` work in all environments, request, session, and application work only in web environments. The `globalSession` scope is for portlet environments.

In an XML bean definition, the scope is set via the `scope` attribute of the `<bean/>` element:

```
<bean id="userPreferences" class="com…UserPreferences" scope="session">...
</bean>
```

You can annotate the bean scope as a meta-annotation to `@Component` or its derivations, such as `@Service` and `@Bean`, as shown in the following listing:

```
@Component
@Scope("request")
public class TaskSearch {...}
```

Generally, service classes and Spring data repositories are declared as `singleton`, since they are built stateless according to best practice.

Dependency Injection with scoped beans

Beans of different scopes can be wired up as collaborators in your configuration metadata. For example, if you have a session-scoped bean as a dependency to `singleton` and face an inconsistency problem, the first instance of the session-scoped bean will be shared between all users. This can be solved using a scoped proxy in place of the scoped bean:

```
<bean id="userPreferences" class="com…UserPreferences" scope="session">
   <aop:scoped-proxy />
</bean>
<bean id="taskService" class="com…TaskService">
   <constructor-arg ref="userPreferences"/>
</bean>
```

Every time the scoped bean is injected, Spring creates a new AOP proxy around the bean so that the instance is picked up from the exact scope. The annotated version of the preceding listing would look like this:

```
@Component
@Scope(value = "session", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class UserPreferences { ... }

public class AnnotatedTaskService implements TaskService {
...
   @Autowired
   private UserPreferences userPreferences;
...
}
```

Creating a custom scope

At times, the scopes supplied by Spring are not sufficient for your specific needs. Spring allows you to create your own custom scope for your scenario. For example, if you want to keep some business process level information throughout its life, you will want to create a new process scope. The following steps will enable you to achieve this:

1. Create a Java class extending `org.springframework.beans.factory.config.Scope`.
2. Define it in your application context (XML or annotation) as a Spring bean.
3. Register the scope bean with your `ApplicationContext` either programmatically or in XML with `CustomScopeConfigurer`.

Hooking to bean life cycles

Often, in enterprise application development, developers will want to plug in some extra functionality to be executed just after the construction and before the destruction of a business service. Spring provides multiple methods for interacting with such stages in the life cycle of a bean.

Implementing InitializingBean and DisposableBean

The Spring IoC container invokes the callback methods `afterPropertiesSet()` of `org.springframework.beans.factory.InitializingBean` and `destroy()` of `org.springframework.beans.factory.DisposableBean` on any Spring bean and implements them:

```
public class UserServiceImpl implements UserService, InitializingBean,
DisposableBean {
...
   @Override
   public void afterPropertiesSet() throws Exception {
      logger.debug(this + ".afterPropertiesSet() invoked!");
      // Your initialization code goes here..
   }

   @Override
   public void destroy() throws Exception {
      logger.debug(this + ".destroy() invoked!");
      // Your cleanup code goes here..
   }
...
}
```

Annotating @PostConstruct and @PreDestroy on @Components

Spring supports JSR 250 `@PostConstruct` and `@PreDestroy` annotations on any Spring bean in an annotation-supported environment, as shown here. Spring encourages this approach over implementing Spring-specific interfaces, as given in the previous section:

```
@Service
public class AnnotatedTaskService implements TaskService {
...
   @PostConstruct
   public void init() {
      logger.debug(this.getClass().getName() + " started!");
   }

   @PreDestroy
   public void cleanup() {
      logger.debug(this.getClass().getName() + " is about to destroy!");
   }
...
}
```

The init-method and destroy-method attributes of <bean/>

If you are using XML-only bean configuration metadata, then your best option is to declare `init-method` and `destroy-method` attributes on your `<bean/>` tags:

```
<bean id="xmlTaskService" class="com…XmlDefinedTaskService" init-
method="init" destroy-method="cleanup">
...
</bean>
```

Container-level default-init-method and default-destroy-method

You can even set container-level default `init` and `destroy` methods so that you don’t need to set it for each bean. The container invokes these methods on beans only if they are present:

```
<beans default-init-method="init" default-destroy-method="cleanup">
...
</beans>
```

Working with bean definition profiles

For commercial projects, it is a common requirement to be able to maintain two or more environment-specific configurations and beans, activated selectively only in the corresponding environment. For example, objects such as data sources, e-mail servers, and security settings could be different for development, testing, and production environments. You would want to switch them declaratively without touching the application code, keeping it externally. Developers traditionally write complex scripts and property files with separate builds to do this job. Spring comes to your rescue here with environment abstraction using bean definition profiles and properties.

Bean definition profiles are a mechanism by which application context is configured differently for different environments. You group bean definitions under named profiles in XML or using annotation and activate one or more profiles in each environment. You can set a default profile to be enabled if you do not specify one explicitly.

Let's take a look the following sample listing that configures data sources for development and production environments:

```
@Configuration
@ComponentScan(basePackages = "com.springessentialsbook")
public class ProfileConfigurator {

   @Bean
   @Profile("dev")
   public DataSource devDataSource() {
      return new EmbeddedDatabaseBuilder()
         .setType(EmbeddedDatabaseType.HSQL) .addScript("scripts/tasks-
system-schema.sql") .addScript("scripts/tasks-master-data.sql") .build();
   }
   @Bean
   @Profile("prod")
   public DataSource productionDataSource() throws Exception {
      Context ctx = new InitialContext();
      return (DataSource)
ctx.lookup("java:comp/env/jdbc/datasource/tasks");
   }
}
```

Practically, for production environments, externalizing this profile config in XML would be a better idea, where you allow your DevOps team to modify it for different environments and forbid them to touch your Java code. XML configuration would look like the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xsi:schemaLocation="...">
  <!-- other bean definitions -->
  <beans profile="dev">
```

```
    <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:scripts/tasks-system-schema.sql"/>
      <jdbc:script location="classpath:scripts/tasks-master-data.sql"/>
    </jdbc:embedded-database>
  </beans>

  <beans profile="production">
    <jee:jndi-lookup id="dataSource" jndi-
name="java:comp/env/jdbc/datasource"/>
  </beans>
</beans>
```

You may create as many profiles as required; it is common for each developer to maintain their own configurations, with profiles named after themselves, say `@Profile("mary")`. You can have multiple profiles active at the same time too; it depends on how well you organize them without having conflicts or duplicate bean definitions across profiles.

Now you can activate one or more profiles as you need in each (`dev`, `test`, or `prod`) environment using any one of the following methods:

1. Programmatically invoking `ctx.getEnvironment().setActiveProfiles("p1", "p2", ..)`.
2. Setting the property `spring.profile.active`—with comma-separated profile names as value—as an environment variable, JVM system property, or Servlet context param in `web.xml`.
3. Add `-Dspring.profile.active="p1,p2, .."` as a command-line or Java argument while starting up your application.

Injecting properties into the Spring environment

Besides the separation of environment-specific configuration using profiles, you would still need to externalize many properties, such as database URLs, e-mails, and date formats in a property file for easier handling. These properties would then either be injected directly into the beans or read from environment by the beans at runtime. Spring's environment abstraction, together with `@PropertySource` annotation, makes this possible in Spring applications.

The `@PropertySource` annotation provides a convenient and declarative mechanism for adding a `PropertySource` to Spring's environment:

```
@Configuration
@PropertySource("classpath:application.properties")
@ComponentScan(basePackages = "com.springessentialsbook")
public class SpringJavaConfigurator {
...
   @Autowired
   @Lazy
   private SystemSettings systemSettings;

   @Autowired
   private Environment env;

   @Bean
   public SystemSettings getSystemSettings() {
      String dateFormat = env.getProperty("system.date-format");
      String appDisplayName = env.getProperty("app.displayname");

      return new SystemSettings(dateFormat, appDisplayName);
   }
…
}
```

Externalizing properties with PropertyPlaceholderConfigurer

`PropertyPlaceholderConfigurer` is another convenient utility to externalize property values from a bean definition into a separate file that uses the standard `java.util.Properties` format. It replaces placeholders in XML bean definitions with matching property values in the configured property file, as shown here. This is the best way to externalize profile or environment-specific information such as datasource config, e-mail settings, and so on. The DevOps team will just edit these property files and never mess with your code:

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

<bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

Here is another simpler declaration of `PropertyPlaceholder`:

```
<context:property-placeholder location="classpath:datasource.properties"/>
```

Handling resources

Spring Framework provides excellent support for accessing low-level resources, thus solving many limitations of Java's standard `java.net.URL` and standard handlers. The `org.springframework.core.io.Resource` package and its many concrete implementations form a solid foundation for Spring Framework's robust resource handling.

Resource abstraction is used extensively in Spring itself, inside many implementations of `ApplicationContext`—it's actually very useful to use as a general utility class by itself in your own code in order to access resources. You will find the following resource implementations that come supplied right out of the box in Spring:

Resource Implementation	**Description**
`UrlResource`	It wraps `java.net.URL` and is useful for accessing anything that can be accessed via a URL, such as files (`file:///`), HTTP targets (`http://`), and FTP targets (`ftp://`).
`ClassPathResource`	It is used for accessing any resource from classpath using the prefix `classpath:`
`FileSystemResource`	This is the resource implementation of `java.io.File`.
`ServletContextResource`	This is the parent bean for inheriting configuration data from a parent bean definition.
`InputStreamResource`	This is the resource implementation for a given `InputStream`.

Generally, you do not directly instantiate any of these resources; rather, you use a `ResourceLoader` interface to do that job for you. All `ApplicationContext` implement a `ResourceLoader` interface; therefore, any `ApplicationContext` can be used to obtain resource instances. The code for this is as follows:

```
ApplicationContext context = new ClassPathXmlApplicationContext(new
String[] {"application-context.xml"});
Resource classPathResource = ctx.getResource("classpath:scripts/tasks-
schema.sql");

Resource fileResource = ctx.getResource("file:///scripts/master-data.sql");

Resource urlResource = ctx.getResource("http://country.io/names.json");
```

You can inject resources into your beans by simply passing the filename or URL of your resource as an argument, as shown here. `ApplicationContext`, which is a `ResourceLoader` interface, will create an instance of an appropriate resource implementation based on the URL you supply:

```
@Value("http://country.io/names.json")
private Resource countriesResource;
```

Here is the XML version of injecting a resource:

```
<property name="countriesResource" value="http://country.io/names.json"/>
```

Spring Expression Language

Expression languages are generally used for simple scripting to manipulate object graphs in a non object-oriented context. For example, if we want to read data or call a method of a Java object from a JSP, XML, or XHTML page, JSP EL and **Unified Expression Language (UEL)** come to the rescue. These expression languages allow page authors to access external data objects in a simple and easy-to-use way, compatible with tag-based languages such as XML and HTML.

The **Spring Expression Language** (**SpEL**), with a language syntax similar to UEL, is a powerful expression language built for querying and manipulating an object graph at runtime. It offers additional features, most notably method invocation and basic string-templating functionality.

SpEL can be used inside a wide variety of technologies that come under the Spring family of projects as well as many technologies that integrate with Spring. It can be used directly in the Spring configuration metadata files, both in XML as well as Java annotations in the form `#{expression-string}`. You can use SpEL inside many view technologies, such as JSP, XML, and XHTML, when integrated with the corresponding technologies, such as JSF, JSP, and Thymeleaf.

SpEL features

The SpEL expression language supports the following functionalities out of the box:

- Boolean, relational, and ternary operators
- Regular expressions and class expressions
- Accessing properties, arrays, lists, and maps
- Method and constructor invocations
- Variables, assignments, and bean references
- Array construction, inline lists, and maps
- User-defined functions and templated expressions
- Collection, projection, and selection

SpEL annotation support

SpEL can be used to specify default values for fields, methods and method or constructor arguments using the `@Value` annotation. The following sample listing contains some excellent usage of SpEL expressions at the field level:

```
@Component
@Scope("prototype")
public class TaskSnapShot {

   Value("#{taskService.findAllTasks().size()}")
   private String totalTasks;

   @Value("#{taskService.findAllTasks()}")
   private List<Task> taskList;

   @Value("#{ new java.util.Date()}")
   private Date reportTime;

   @Value("#{taskService.findAllTasks().?[status == 'Open']}")
   private List<Task> openTasks;
...

}
```

The same approach can be used for XML bean definitions too.

The SpEL API

Generally, most users use SpEL to evaluate expressions embedded in XML, XHTML, or annotations. While SpEL serves as the foundation for expression evaluation within the Spring portfolio, it can be used independently in non-Spring environments using the SpEL API. The SpEL API provides the bootstrapping infrastructure to use SpEL programmatically in any environment.

The SpEL API classes and interfaces are located in the (sub)packages under `org.springframework.expression`. They provide the specification and default SpEL implementations which can be used directly or extended.

The following interfaces and classes form the foundation of the SpEL API:

Class/Interface	**Description**
`Expression`	The specification for an expression capable of evaluating itself against context objects independent of any language such as OGNL or UEL. It encapsulates the details of a previously parsed expression string.
`SpelExpression`	A SpEL-compliant, parsed expression that is ready to be evaluated standalone or in a specified context.
`ExpressionParser`	Parses expression strings (templates as well as standard expression strings) into compiled expressions that can be evaluated.
`SpelExpressionParser`	SpEL parser. Instances are reusable and thread-safe.
`EvaluationContext`	Expressions are executed in an evaluation context, where references are resolved when encountered during expression evaluation.
`StandardEvaluationContext`	The default `EvaluationContext` implementation, which uses reflection to resolve properties/methods/fields of objects. If this is not sufficient for your use, you may extend this class to register custom `ConstructorResolver`, `MethodResolver`, and `PropertyAccessor` objects and redefine how SpEL evaluates expressions.
`SpelCompiler`	Compiles a regular parsed expression instead of the interpreted form to a class containing bytecode for evaluation. A far faster method, but still at an early stage, it does not yet support every kind of expression as of Spring 4.1.

Let’s take a look at an example that evaluates an expression using the SpEL API:

```
@Component
public class TaskSnapshotBuilder {

   @Autowired
   private TaskService taskService;

   public TaskSnapShot buildTaskSnapShot() {
      TaskSnapShot snapshot = new TaskSnapShot();

      ExpressionParser parser = new SpelExpressionParser();
      EvaluationContext context = new
StandardEvaluationContext(taskService);
```

```
        Expression exp = parser.parseExpression("findAllTasks().size()");
        snapshot.setTotalTasks(exp.getValue(context).toString());

        exp = parser.parseExpression("findAllTasks()");
        snapshot.setTaskList((List<Task>)exp.getValue(context));

        exp = parser.parseExpression("new java.util.Date()");
        snapshot.setReportTime((Date)exp.getValue(context));

        exp = parser.parseExpression("findAllTasks().?[status == 'Open']");
        snapshot.setOpenTasks((List<Task>)exp.getValue(context));

        return snapshot;
    }

}
```

In normal scenarios, you would not need to directly use the SpEL API in a Spring application; SpEL with annotation or XML bean definitions would be better candidates. The SpEL API is mostly used to load externalized business rules dynamically at runtime.

Aspect Oriented Programming

Most software applications usually have some secondary—but critical—features, such as security, transaction, and audit-logging, spanned across multiple logical modules. It would be a nice idea not to mix these cross-cutting concerns in your core business logic. **Aspect Oriented Programming** (**AOP**) helps you achieve this.

Object Oriented Programming (**OOP**) is about modularizing complex software programs, with objects as the fundamental units that hold your core business logic and data. AOP complements OOP to add more complex functionality transparently across modules of your application without polluting the original object structure. AOP stitches (weaves) cross-cutting concerns into your program, either at compile time or runtime, without modifying the base code itself. AOP lets the object-oriented program stay clean and just have the core business concerns.

Static and dynamic AOP

In AOP, the framework weaves the cross-cutting concerns into the main program transparently. This weaving process comes in two different flavors: static and dynamic. In the case of static AOP, as the name implies, Aspects are compiled directly into static files, that is, to the Java bytecode, on compilation. This method performs better, as there is no special interception at runtime. But the drawback is that you need to recompile the entire application every time you change anything in the code. AspectJ, one of the most comprehensive AOP implementations, provides compile-time weaving of Aspects.

In the case of dynamic AOP, the weaving process is performed dynamically at runtime. Different frameworks implement this differently, but the most general way of achieving this is using proxies or wrappers for the advised objects, allowing the Advice to be invoked as required. This is a more flexible method as you can apply AOP with varying behavior at runtime depending on data, which is not possible in the case of static AOP. There is no need for recompiling the main application code if you use XML files for defining your AOP constructs (schema-based approach). The disadvantage of dynamic AOP is a very negligible performance loss due to the extra runtime processing.

Spring AOP is proxy based, that is, it follows the dynamic flavor of AOP. Spring provides the facility to use static AOP by integrating with AspectJ too.

AOP concepts and terminology

Understanding AOP concepts and terms gives you an excellent starting point for AOP; it helps you visualize how and where AOP can be applied in your application:

- **Aspect**: The concern that cuts across multiple classes or modules. Transaction and security are examples. Spring Transaction is implemented as Aspects.
- **Join point**: A point during the execution of the program at which you want to insert additional logic using AOP. A method execution and a class instantiation are examples.
- **Advice**: The action taken by (the code or method that executes) the Aspect at a particular join point. Different types of advices include `before`, `after`, and `around` advices. Typically, an Aspect has one or more Advices.
- **Pointcut**: An expression that defines or matches a set of join points. The Advice associated with a pointcut executes at any join point it matches. Spring supports the AspectJ pointcut expression language by default. An example is `execution(* com.xyz.service.*.*(..))`.
- **Target object**: The advised object. If you use dynamic AOP, this would be a proxied object.
- **Weaving**: Inserting Aspects into a target object to make it advised at compile time, load time or runtime. AspectJ supports compile-time weaving and Spring weaves at runtime.
- **Introduction**: The process by which you add a new method or field to an advised object, with or without making it implement an interface.

Spring AOP – definition and configuration styles

Spring provides a proxy-based dynamic implementation of AOP, developed purely in Java. It neither requires a special compilation process like AspectJ nor controls the class loader hierarchy, hence it can be deployed inside any Servlet container or application server.

Although not a full-blown AOP framework like AspectJ, Spring provides a simple and easy-to-use abstraction of most of the common features of AOP. It supports only method execution join points; field interception is not implemented. Spring provides tight integration with AspectJ, in case you want to advise very fine-grained Aspect orientation that Spring AOP doesn't cover by adding more AspectJ-specific features without breaking the core Spring AOP APIs.

Spring AOP uses standard JDK dynamic proxies for Aspect orientation by default. JDK dynamic proxies allow any interface (or set of interfaces) to be proxied. If you want to proxy classes rather than interfaces, you may switch to CGLIB proxies. Spring automatically switches to use CGLIB if a target object does not implement an interface.

Starting from Spring 2.0, you can follow either a schema-based approach or an `@AspectJ` annotation style to write custom Aspects. Both of these styles offer fully typed Advice and use of the AspectJ pointcut language while still using Spring AOP for weaving.

XML schema-based AOP

When using schema-based AOP, you need to import `aop` namespace tags into your `application-context` file, as follows:

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

@AspectJ annotation-based AOP

`@AspectJ` refers to a style of declaring Aspects as regular Java classes that are annotated. Spring interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. Spring AOP has no dependency on the AspectJ compiler or weaver, though.

When using the `@AspectJ` annotation style, you first need to enable `@AspectJ` support in your Spring configuration, whether or not it is in the XML or Java configuration. Additionally, you need to make sure you add `aspectjweaver.jar` in your classpath. Adding an `@EnableAspectJAutoProxy` annotation to your Java `@Configuration` annotation will enable `@AspectJ` support in your project:

```
@Configuration
@ComponentScan(basePackages = "com.springessentialsbook")
@EnableAspectJAutoProxy
public class AOPJavaConfigurator {
...
}
```

Alternatively, if you use XML-based configuration, `@AspectJ` support can be enabled by adding the `<aop:aspectj-autoproxy/>` element in your `application-context` file.

Declaring an @Aspect annotation

Your Aspect is a simple POJO, either annotated with `@Aspect` (`org.aspectj.lang.annotation.Aspect`) or declared as `<aop:aspect/>` under the `<aop:config>` section of your `application-context` XML file. Remember, the class marked as `@Aspect` should be declared as a Spring bean using either an annotation or `<bean/>` declaration in your application context XML file.

Here is an annotated Aspect, a Spring component annotated as `@Aspect`:

```
@Component("auditLoggerAspect")
@Aspect
public class AuditLoggerAspect {
...
}
```

Note that `@Aspect` is a Spring bean too. It can be any of the specializations of `@Component`.

Now, let's take a look at the XML alternative for Aspect declaration:

```
<aop:config>
   <aop:aspect id="audLogAspect" ref="auditLoggerAspect">
</aop:config>
<bean id="auditLoggerAspect" class="com…AuditLoggerAspect"/>
```

Aspects may have methods and fields, just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations. Aspects themselves cannot be the target of Advice from other Aspects; they are excluded from auto-proxying.

Pointcuts

A pointcut comprises two parts, as shown in the following code snippet: a method signature (an empty method with a `void` return type inside the `Aspect` class) with any parameters and an expression that matches the exact method executions we are interested in. Remember, Spring AOP only supports method execution join points:

```
@Pointcut("execution(*
com.springessentialsbook.service.TaskService.createTask(..))") //Pointcut
expression
private void createTaskPointCut() {} //Signature
```

The pointcut expression follows the standard AspectJ format. You may refer to the AspectJ pointcut expression reference for the detailed syntax. The following section gives you a strong foundation for constructing pointcuts for Spring AOP.

Pointcut designators

Spring AOP supports just a subset of the original AspectJ **pointcut designators** (**PCDs**) for use in pointcut expressions, as given in the following table:

PCD	**Description**
`execution`	Method execution join point; the default PCD for Spring AOP

`within`	Matches methods in a range of types, packages, and so on
`this`	Matches proxy instances of a given type
`target`	Matches target object with a given type
`args`	Matches methods with the given argument types
`@target`	Matches methods of classes with the given annotation
`@args`	Matches methods having argument (s) with the given annotation (s)
`@within`	Matches methods within types that have a given annotation
`@annotation`	Matches methods with the given annotation

In addition to the preceding table, Spring supports an extra non-AspectJ PCD, `bean`, which is useful to directly refer to a Spring bean or a set of beans with a comma-separated list of beans using `bean(idsOrNamesOfBean)`.

Note that the pointcuts intercept only `public` methods due to the proxy nature of Spring AOP. If you want to intercept `protected` and `private` methods or even constructors, consider using AspectJ weaving (integrated with Spring itself) instead.

Pointcut examples

Pointcut expressions can be combined using `&&`, `||`, and `!`. You can refer to pointcut expressions by name, too. Let's see a few examples:

```
@Pointcut("execution(* com.taskify.service.*.*(..))")
private void allServiceMethods() {}

@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}

@Pointcut("anyPublicOperation() && allServiceMethods()")
private void allPublicServiceMethods() {}

@Pointcut("within(com.taskify.service..*)")
private void allServiceClasses() {}

@Pointcut("execution(* set*(..))")
private void allSetMethods() {}

@Pointcut("execution(* com.taskify.service.TaskService.*(..))")
private void allTaskServiceMethods() {}

@Pointcut("target(com.taskify.service.TaskService)")
private void allTaskServiceImplMethods() {}

@Pointcut("@within(org.springframework.transaction.annotation.Transactional
)")
private void allTransactionalObjectMethods() {}

@Pointcut("@annotation(org.springframework.transaction.annotation.Transacti
onal)")
private void allTransactionalAnnotatedMethods() {}

@Pointcut("bean(simpleTaskService)")
private void allSimpleTaskServiceBeanMethods() {}
```

An XML version of a pointcut definition goes like this:

```
<aop:config>
  ...
   <aop:pointcut id="allTaskServicePointCut"
         expression="execution(*com.taskify.service..TaskService.*(..))"/>
</aop:config>
```

Advices

An Advice is the action that gets injected before, after, or around the method executions matched by the pointcut expression. The pointcut expression associated with an Advice could be a named or defined pointcut, as listed in the above examples, or a pointcut expression declared in place, that is, advices and pointcuts can be declared together.

Let's see an example for an Advice that refers to a pointcut expression named `Pointcut`:

```
@Pointcut("execution(* com.taskify.service.TaskService.*(..))")
private void allTaskServiceMethods() {}

@Before("allTaskServiceMethods()")
private void logBeforeAllTaskServiceMethods() {
  logger.info("*** logBeforeAllTaskServiceMethods invoked ! ***");
}
```

The following code listing combines both a join point and Advice in one go. This is the most common approach:

```
@After("execution(* com.taskigy.service.TaskService.*(..))")
private void logAfterAllTaskServiceMethods() {
  logger.info("***logAfterAllTaskServiceMethods invoked ! ***");
}
```

The following table lists the available Advice annotations:

Advice annotation	**Description**
`@Before`	Runs before method execution.
`@After`	Runs after method exit (finally).
`@AfterReturning`	Runs after the method returns without an exception. You can bind the return value with the Advice as the method argument.
`@AfterThrowing`	Runs after the method exits by throwing an exception. You can bind the exception with the Advice as the method argument.
`@Around`	The target method actually runs inside this Advice. It allows you to manipulate the method execution inside your Advice method.

The @Around Advice

The `@Around` Advice gives you more control over method execution, as the intercepted method essentially runs inside your Advice method. The first argument of the Advice must be `ProceedingJoinPoint`. You need to invoke the `proceed()` method of `ProceedingJoinPoint` inside the Advice body in order to execute the target method; else, the method will not get called. After the method execution returns to you with whatever it returns back to your advice, do not forget to return the result in your Advice method. Take a look at a sample `@Around` advice:

```
@Around("execution(* com.taskify.service.**.find*(..))")
private Object profileServiceFindAdvice(ProceedingJoinPoint jPoint) throws
Throwable {
    Date startTime = new Date();
    Object result = jPoint.proceed(jPoint.getArgs());
    Date endTime = new Date();
    logger.info("Time taken to execute operation: " + jPoint.getSignature()
+ " is " + (endTime.getTime() - startTime.getTime()) + " ms");
    return result;
}
```

Accessing Advice parameters

There are two distinct ways of accessing the parameters of the method you are advising in the Advice method:

- Declaring a join point as the first argument
- Binding `args` in the pointcut definition

Let’s see the first approach:

```
@Before("execution(* com.taskify.service.TaskService.createTask(..)")
private void logBeforeCreateTaskAdvice(JoinPoint joinpoint) {
   logger.info("***logBeforeCreateTaskAdvice invoked ! ***");
   logger.info("args = " + Arrays.asList(joinpoint.getArgs()));
}
```

You can see that `joinpoint.getArgs()` returns `Object[]` of all the arguments passed to the intercepted method. Now, let’s see how to bind named arguments to the Advice method:

```
@Before("createTaskPointCut() and args(name, priority, createdByuserId,
assigneeUserId)")
private void logBeforeCreateTaskAdvice(String name, int priority, int
createdByuserId, int assigneeUserId) {

  logger.info("name = " + name + "; priority = " + priority + ";
  createdByuserId = " + createdByuserId);
}
```

Note that the `joinpoint` expression matches the arguments by name. You can have a `joinpoint` object as an optional first argument in the method signature without specifying it in the expression: you will have both `joinpoint` and arguments, enabling more manipulation.

Testing with Spring

The degree of testability shows the elegance and maturity of any framework. A more testable system is more maintainable. Spring Framework provides comprehensive support for end-to-end testing of applications for both unit testing as well as integration testing. Spring promotes **test-driven development** (**TDD**), facilitates integration testing, and advocates a set of best practices for the unit testing of beans. This is another compelling reason for using Spring to build serious applications.

The POJO-based programming model and loosely coupled nature of Spring beans make it easier to participate in JUnit and TestNG tests even without Spring in the middle. On top of this, Spring provides many testing support components, utilities, and mock objects to make the testing easier.

Mock objects

Spring provides mock implementations of many container-specific components so that the beans can be tested outside a server or container environment. `MockEnvironment` and `MockPropertySource` are useful for testing environment-dependent beans. To test beans that depend on HTTP communications, Spring provides mock classes for both client and server sides inside the `org.springframework.mock.http` and `org.springframework.mock.http.client` packages.

Another set of useful classes can be found under `org.springframework.mock.jndi` to run test suites that depend on JNDI resources. The `org.springframework.mock.web` package contains mock objects for web components based on Servlet 3.0, such as web contexts, filters, controllers, and asynchronous request processing.

Unit and integration testing utilities

Spring ships certain general-purpose and context-specific utilities for unit and integration testing. The `org.springframework.test.util` package contains a set of utility classes for various testing purposes, including reflection, AOP, JSON, and XML manipulations. Classes under `org.springframework.test.web` and its nested subdirectories contain a comprehensive set of utility classes to test beans dependent on the web environment. Another set of useful classes for usages specific to `ApplicationContext` can be found under `org.springframework.test.context` and its child packages. Their support includes the loading and caching of web, portlet, or application contexts in the test environment; resolving profiles; loading property sources and SQL scripts; managing transactions for test environments; and so on.

The support classes and annotations under the packages listed earlier facilitate the easy and natural testing of Spring applications. A comprehensive discussion over Spring test support is beyond the scope of this book. However, gaining a good understanding of Spring's comprehensive support for unit and integration tests is vital in order to develop elegant code and maintainable applications using Spring.

Summary

We have successfully covered all the major technologies and concepts of core Spring Framework in this chapter. We are now capable of developing robust, standalone Spring applications composed of loosely-coupled beans inside the powerful Spring IoC container. We know how to apply cross-cutting concerns transparently across different layers of an application using the very flexible pointcut expressions of Spring AOP. We can manipulate Spring beans using Spring Expression Language, which helps keep the code clean and highly maintainable. We learned how to maintain multiple environment-specific bean configurations and property files using bean definition profiles. Now, we are all set for professional Spring development.

The source code available with this chapter contains multiple Spring projects that demonstrate the different ways of configuring Spring as well as usage scenarios. The examples listed in this chapter have been extracted from them.

In the next chapter, we will explore Spring Web module, leveraging all that we learned in this chapter in a web-based environment. The topics we have learned in this chapter are going to be the foundation for all the advanced topics that will be covered in the following chapters.

Chapter 2. Building the Web Layer with Spring Web MVC

Web application development is a major focus area for enterprise systems. In this age of cloud and big data, web applications are under a tremendous load of an ever-increasing number of concurrent users accessing them from multiple devices such as mobiles and tablets, as well as traditional desktop web browsers. Modern web applications have to address a newer set of nonfunctional requirements, such as scalability, performance, productivity, responsiveness, and multi-device support.

Spring MVC is a web framework from Spring, perfectly built from the ground up to address the concerns of modern web applications. A lightweight and high-performance web framework, Spring MVC is designed to be highly productive from day one, flexible, and adaptable with a wide variety of view technologies. Sitting on top of the mighty Spring Framework, it integrates well with all Java EE technologies and other open source frameworks. Just like any technology under the Spring portfolio, Spring MVC also promotes POJO programming with the help of a well-defined set of annotations, namespace XML tags, and web-support components.

This chapter introduces Spring MVC and its powerful features to you, describes how to set it up, and guides you on its advanced usages, configurations, and optimizations with relevant examples. We will mostly use annotations in these examples for simplicity. At the end of this chapter, you will be able to build web applications with Spring MVC that have HTML-based user interfaces as well as RESTful APIs with JSON and XML formats.

Features of Spring MVC

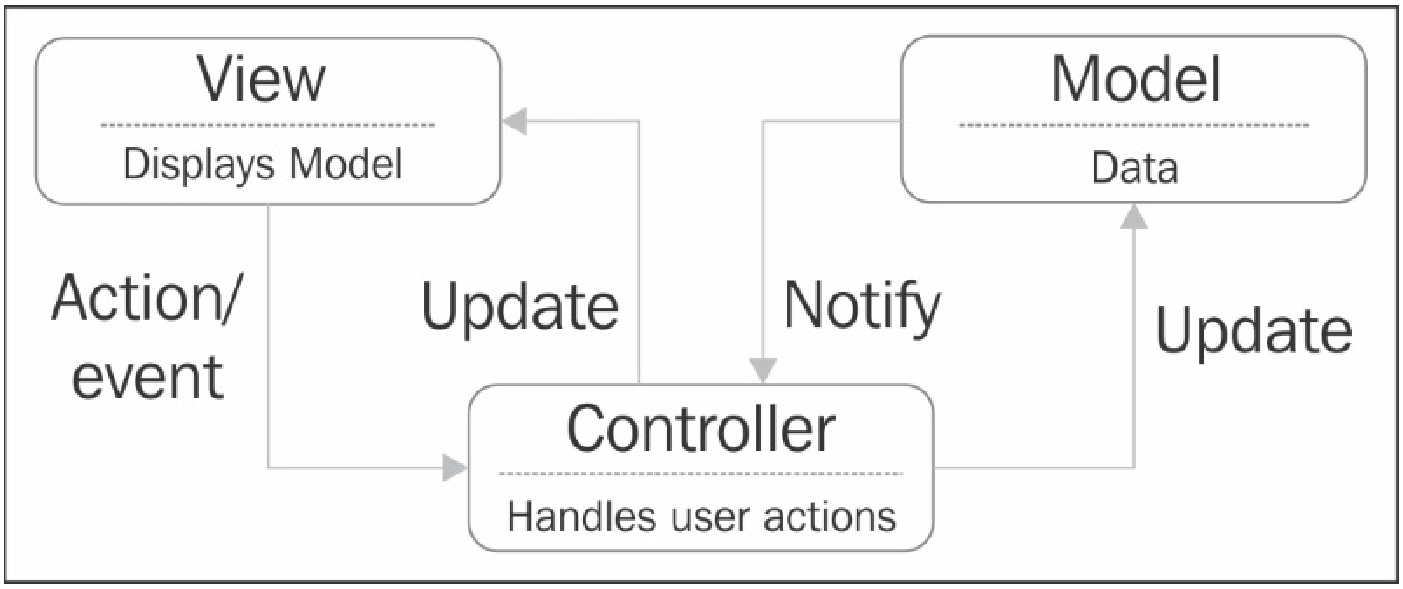
Spring MVC bundles a compelling set of features and advantages over rival web technologies. Knowledge of these will help you decide on choosing Spring MVC for your requirements. The following list covers most of them:

- Simple configuration and native integration with Spring Framework, leveraging the powerful features of Spring and other open source libraries
- Built on top of Java web technologies such as Servlets, JSP, and JSTL and can be deployed into any standard Servlet container, such as Tomcat
- Implemented based on the **Model-View-Controller** (**MVC**) architecture pattern, with clear separation of concerns using simple annotations and namespace XML tags
- Explicit support for convention over configuration for MVC components
- Supports a big set of view technologies, such as JSP, Thymeleaf, Handlebars, Velocity, FreeMarker, PDF, Excel, and JasperReports
- Declarative input validation, data binding, and exception handling
- Flexible URL mapping with automatic request and response transformation into various formats such as JSON, XML, and HTML
- Support for non-blocking asynchronous request processing and HTTP streaming
- Support for internationalization, themes, and multipart file uploads
- Easy integration with Spring Security and thorough testability
- HTTP caching for increased performance
- A simple, yet powerful, JSP tag library

What makes Spring MVC outstanding is its simple programming model, a feature inherited from the core Spring Framework itself. A developer familiar with any standard web framework will find Spring MVC very familiar and easy to learn.

The Model-View-Controller pattern

MVC is a well-established architectural pattern popularly used for building interactive web and desktop applications. There are numerous frameworks implementing this pattern in most software platforms. MVC divides the application into three core elements that actually represent layers, separates concerns between these three core elements, and defines how they communicate with each other.



Model represents data, **View** displays the **Model**, and **Controller** handles user actions. Model can be any data, including that stored in a database. It usually represents a collection of domain objects with clearly defined relationships to each other. A **Model** can be displayed in multiple views depending on how the application is designed.

Controller acts as an intermediary between **View** and **Model**. It often has a set of handlers for each event generated by the view as the user interacts with it. **Controller** delegates user actions to appropriate handlers and then finally redirects to another view for displaying the result of that action.

There are so many implementations of the MVC pattern as frameworks across technology platforms use it in different ways. Spring MVC has implemented it in the simplest and least invasive fashion, while naturally integrating it with the core Spring Framework.

Your first Spring MVC application

Let’s jump to creating a very simple Spring MVC web application. For the purpose of learning, we will develop the web version of *Taskify*, the task management system we started in Chapter 1, *Getting Started with Spring Core*. The samples in this chapter use **Spring Tool Suite** (**STS**) as the IDE, but you can use your favorite IDE, such as IntelliJ and NetBeans. Almost all Java IDEs support Spring development; most of them have plugins to manage Spring projects and artifacts.

To begin with, follow these steps; then, we will explore the code:

1. Open STS or Eclipse → create a new project → type a project name → select a template, either **Spring MVC Project** or **Simple Spring Web Maven** → specify the top-level package name → finish. Your project structure will be generated.
2. Make sure your `pom.xml` file contains Maven dependencies for the `spring-context`, `spring-mvc`, `servlet-api`, `jsp-api`, and `jstl` libraries. Note that `jsp-api` and `jstl` are required only if you are using JSP as the view technology.
3. If it hasn’t been generated, create `web.xml` under `WEB-INF`, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
<!-- \ Root Container shared by Servlets and Filters -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

<!-- Loads Spring Container shared by all Servlets and Filters -->
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <!-- Processes application requests -->
  <servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/servlet-context.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
```

```
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
</web-app>
```

4. If it hasn't been generated, create a `root-context.xml` file, with the following content:

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

  <!-- Root Context: defines all shared beans go here -->
</beans>
```

5. If it hasn't been generated, create a `servlet-context.xml` file, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/mvc" ...>

  <!-- Enables the Spring MVC @Controller programming model -->
  <annotation-driven />
  <context:component-scan base package="com.taskify"/>

  <!-- Handles HTTP GET requests for /resources/** by serving up static
resources in ${webappRoot}/resources directory -->
  <resources mapping="/resources/**" location="/resources/" />

  <!-- Resolves views selected for rendering by @Controllers to
      .jsp resources in the /WEB-INF/views directory -->
  <beans:bean class=
  "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name="prefix" value="/WEB-INF/views/" />
    <beans:property name="suffix" value=".jsp" />
  </beans:bean>
</beans:beans>
```

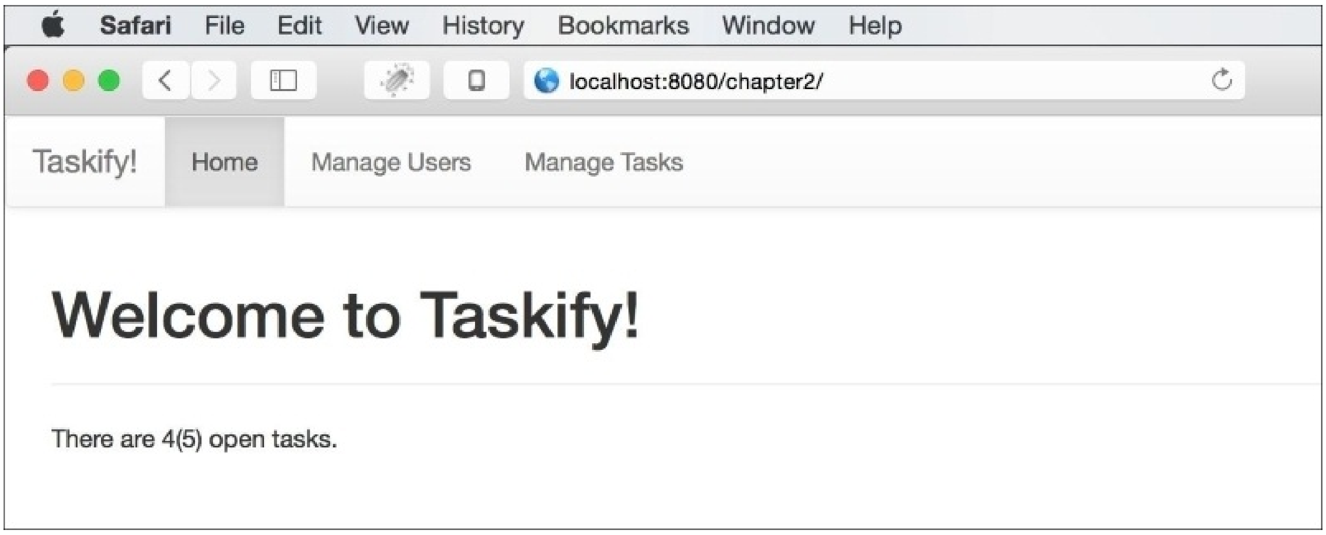
6. Now, create a Java class, `HomeController`, under the package `com.taskify.web.controllers`, with the following content:

```
@Controller
public class HomeController {
  private static final Logger logger =
LoggerFactory.getLogger(HomeController.class);
  @Autowired
  private TaskService taskService;
  // Simply selects the home view to render by returning // name.
  @RequestMapping(value = "/", method = RequestMethod.GET)
  public String home(Locale locale, Model model) {
    logger.info("Welcome to Taskify! Locale is {}.", locale);
    model.addAttribute("totalTasks",
    taskService.findAllTasksCount() );
    model.addAttribute("totalOpenTasks",
taskService.findAllOpenTasksCount() );
    return "home";
  }
}
```

7. Create a JSP view, `home.jsp`, under `~WEB-INF/views`, with the following content:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ page session="false"%>
<html>
  <head>
    <jsp:include page="/WEB-INF/views/theme.jsp"></jsp:include>
    <title>Taskify :: Home</title>
  </head>
  <body>
    <jsp:include page="/WEB-INF/views/navbar.jsp"></jsp:include>
    <div class="container">
      <h1>Welcome to Taskify!</h1><hr />
      <P>There are ${totalOpenTasks}(${totalTasks}) open tasks.</P>
    </div>
  </body>
</html>
```

8. Make sure you have the `TaskService` class (copy it from Chapter 1, *Getting Started with Spring Core*) and its concrete implementation in your project, with the methods `findAllTasksCount()` and `findAllOpenTasksCount()` implemented.
9. Now that your project is ready, make sure you have an Apache Tomcat (or any other) server installed and configured with your IDE. You can download Tomcat from http://tomcat.apache.org/ and install on your PC.
10. STS and Eclipse allow you to run Java web apps from the IDE just by right-clicking **Run As** → **Run on Server**. Resolve all errors, if any, and run again.
11. You should see the home screen of your web app (at `http://localhost:8080/chapter2/`), as seen here:



Welcome to Taskify!

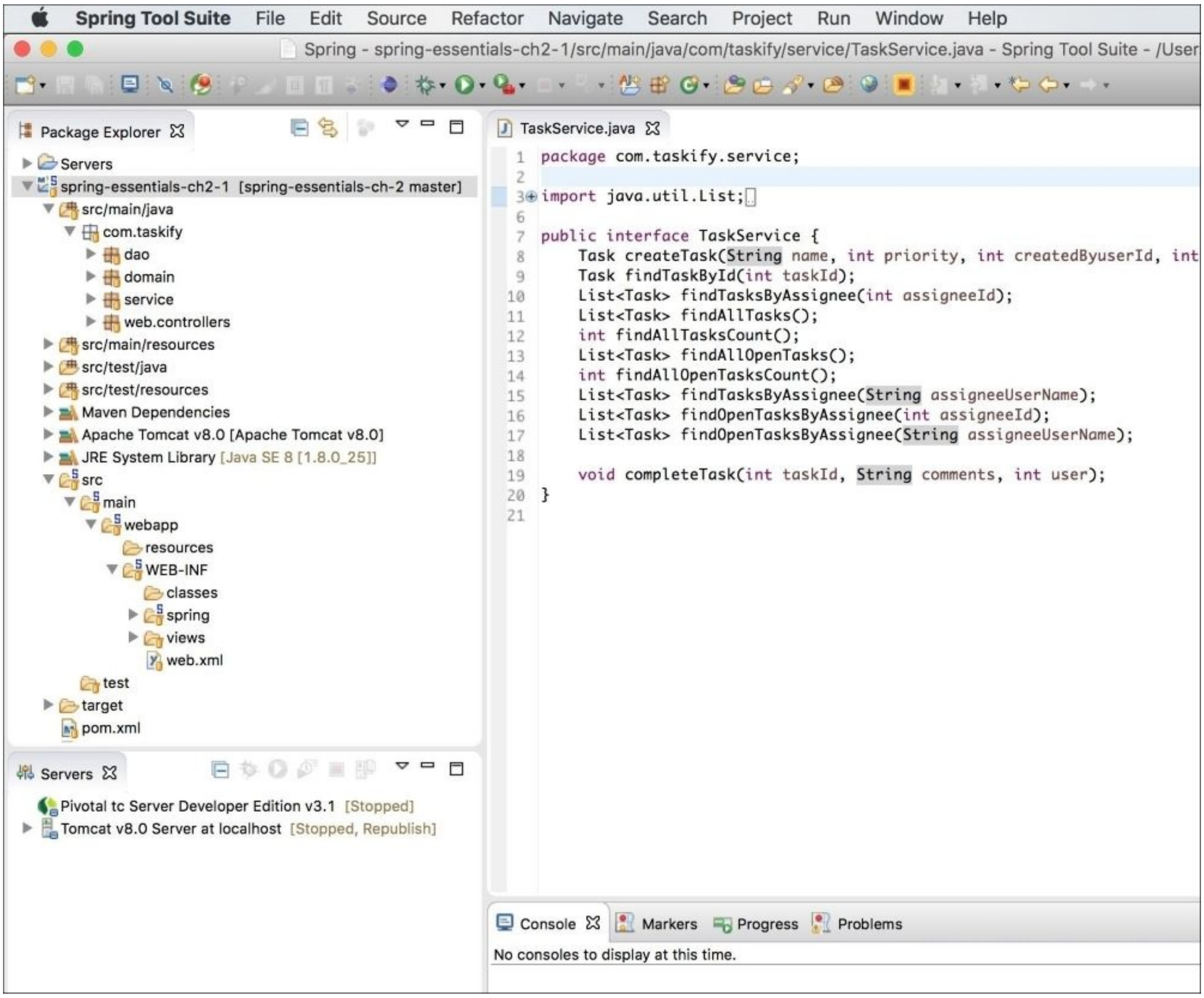
There are 4(5) open tasks.

Setting up a Spring MVC application

Let’s figure out how a Spring MVC web application is configured by analyzing the application artifacts listed in the previous section, *Your first Spring MVC application*. It contains all the necessary artifacts for building a Spring MVC web app.

The project structure of a Spring MVC application

The easiest way to create the project structure and the necessary artifacts is using STS to create a Spring MVC project, as described in the previous section. Alternatively, you may use one of the Maven archetypes available in various repositories online. STS uses such a bundled archetype. Here is the typical project structure of a Spring MVC application as viewed in STS:


Spring Tool Suite File Edit Source Refactor Navigate Search Project Run Window Help
Spring - spring-essentials-ch2-1/src/main/java/com/taskify/service/TaskService.java - Spring Tool Suite - /User
Package Explorer
Servers
spring-essentials-ch2-1 [spring-essentials-ch-2 master]
src/main/java
com.taskify
dao
domain
service
web.controllers
src/main/resources
src/test/java
src/test/resources
Maven Dependencies
Apache Tomcat v8.0 [Apache Tomcat v8.0]
JRE System Library [Java SE 8 [1.8.0_25]]
src
main
webapp
resources
WEB-INF
classes
spring
views
web.xml
test
target
pom.xml
Servers
Pivotal tc Server Developer Edition v3.1 [Stopped]
Tomcat v8.0 Server at localhost [Stopped, Republish]
TaskService.java
Console
Markers
Progress
Problems

```
package com.taskify.service;

import java.util.List;

public interface TaskService {
    Task createTask(String name, int priority, int createdByuserId, int
    Task findTaskById(int taskId);
    List<Task> findTasksByAssignee(int assigneeId);
    List<Task> findAllTasks();
    int findAllTasksCount();
    List<Task> findAllOpenTasks();
    int findAllOpenTasksCount();
    List<Task> findTasksByAssignee(String assigneeUserName);
    List<Task> findOpenTasksByAssignee(int assigneeId);
    List<Task> findOpenTasksByAssignee(String assigneeUserName);

    void completeTask(int taskId, String comments, int user);
}
```

```
No consoles to display at this time.
```

This structure represents a single-WAR web application where all the services and data access components are collocated with the web controllers. In the case of bigger applications, many such components could be part of a different JAR library project, to be shared between multiple web apps and then added as Maven dependencies to the consuming web apps and beans imported to the web application context XML files using an `<import/>` tag or annotation config.

Now, let's examine each artifact listed in *Your first Spring MVC application* in detail.

The web.xml file – Springifying the web app

The `web.xml` file is the standard Java web descriptor in which the fundamental web components that make up a Java web application are registered with the Servlet container. `ServletContextListener`, and `ServletFilter` components are configured here.

A Spring MVC application is also configured and bootstrapped in `web.xml`. `ContextLoaderListener`, registered as a `ServletContextListener` in the `web.xml` sample, bootstraps Spring's root `WebApplicationContext`. In the previous chapter, we saw how a simple console application bootstraps the Spring context from inside the main method using `ClassPathXmlApplicationContext`. In the case of a web application, following `ContextLoaderListener` loads the `WebApplicationContext`. Remember, a Spring MVC application is not just another Servlet-based application but rather Spring integrated within a web context.

```
   <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
   </listener>
```

The following listener looks for a `context-param` tag, `contextConfigLocation`, which is the location of the Spring root bean definition XML file, as seen in the `web.xml` file earlier:

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/spring/root-context.xml</param-value>
</context-param>
```

The next very important Spring component configured in the `web.xml` file is `DispatcherServlet`, the centralized entry point into the Spring MVC application which maps every request with appropriate handlers. `DispatcherServlet` is an implementation of the Front Controller design pattern, which is a single, centralized entry-point for all HTTP requests that come into the application. This internally delegates them to the actual handler of the request type. Here is an excerpt from the earlier `web.xml` listing:

```
<servlet>
   <servlet-name>appServlet</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
   </servlet-class>
   <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
         /WEB-INF/spring/appServlet/servlet-context.xml
      </param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
```

```
   <servlet-name>appServlet</servlet-name>
   <url-pattern>/</url-pattern>
</servlet-mapping>
```

The preceding Servlet registration of `DispatcherServlet` maps the root URL to `DispatcherServlet` so that every HTTP request that comes into the server will be first handled by `DispatcherServlet`. Additionally, it specifies where the Spring application context file for this Servlet will be.

Note

Your application can have any number of `DispatcherServlet` definitions with unique Servlet names, depending on how you want to divide your URL subcontexts logically based on your functional modules. Most applications would have just one `DispatcherServlet` defined.

ApplicationContext files in a Spring MVC application

A Spring MVC application is nothing but a Servlet-based Web MVC implementation with Spring integrated natively. Hence, it requires Spring bean definitions like any other Spring application, as we have seen in the previous chapter. In the case of a Spring MVC application, there would be some framework-specific beans in addition to application-specific beans registered in the context.

For the sake of organizing beans under different logical categories, such as web-specific (`DispatcherServlet`) as well as shared beans, multiple bean definitions can be used. For example, `DispatcherServlet` can have its own application context file with beans helping its processing (just for the web context), and there could be a root application context file, where beans that are not specific to the web layer but shared between many layers are defined.

Inside the sample listed in the earlier section as part of *Your first Spring MVC application*, you can see two Spring bean definition XML files, `root-context.xml` and `servlet-context.xml`. The `root-context.xml` file represents your root `WebApplicationContext` loaded via `ContextLoaderListener`. This is the place where you define or import your shared beans, such as service beans, and data access objects.

The `servlet-context.xml` file is loaded by `DispatcherServlet` on startup. The filename, `servlet-context.xml`, is an explicit filename given in the sample listing. By default, `DispatcherServlet` looks for an XML bean definition file with the pattern `[servlet-name]-servlet.xml`, that is, if it wasn't specified explicitly, `appServlet` would look for a file with the name `appServlet-servlet.xml` at the root of the `WEB-INF` folder. Typically, this file contains the bean definitions controlling the behavior of this Servlet. For example, you can see the resources and view resolver bean definitions in the file `servlet-context.xml`. You can see that the view resolver configured in the sample listing only supports JSP views.

HomeController – @Controller for the home screen

`HomeController` is a simple POJO with the `@Controller` annotation. This annotation registers it as a web controller with a set of annotated handler methods inside it. It can have an `@RequestMapping` annotation at the class level to indicate the root URL of all handler methods in it. The method `home()` is the request handler for the root URL, `http://<context-root>/`.

Since the `@Controller` annotation is yet another Spring bean, you can inject any other bean into it as a dependency. The sample listing shows that `HomeController` has an autowired dependency to `TaskService`. In the `home()` method, `TaskService` methods are invoked, and finally, the return values are set as `Model` attributes for the consumption of the latter parts of the request and to be used in a view.

Your application will have many such controllers that serve groups of related URL endpoints; consider `HomeController` as your basic example. One controller can have multiple request-handling methods that serve different URLs.

The home.jsp file – the landing screen

The `home.jsp` file is your view artifact for the root (`/`) URL. Notice how the `Model` attributes are bound inside the JSP view:

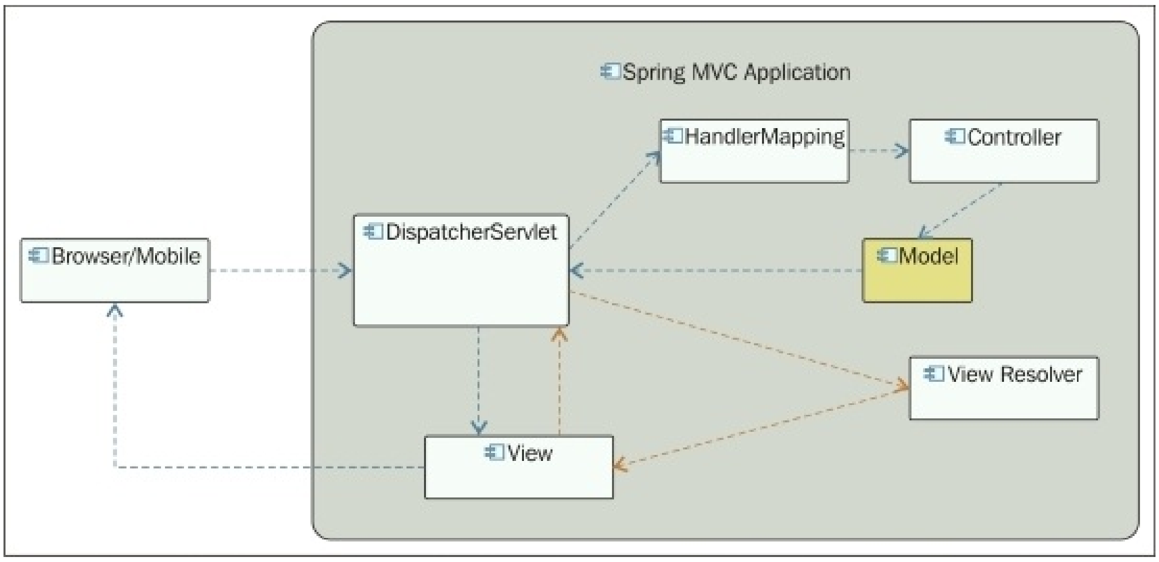
```
<P>There are ${totalOpenTasks}(${totalTasks}) open tasks.</P>
```

Handling incoming requests

Any request that hits the root URL of the app is first received by `DispatcherServlet`, which delegates it to `HomeController.home()`, which processes the request and returns the view name (`home`, in this case). `DispatcherServlet` then picks up the `home.jsp` file based on the resource and view configurations specified in `servlet-context.xml` and renders it, passing the attributes of the model instance created inside `HomeController.home()`.

The architecture and components of Spring MVC

Having gone through your first Spring MVC application, it is now time to look at Spring MVC applications from an architectural perspective:



Spring MVC components

As the name implies, Spring MVC follows the renowned MVC architectural pattern. This pattern ensures the separation of concerns by dividing responsibilities into three major roles:

- **Model**: This represents data and business logic
- **View**: This represents presentation
- **Controller**: This processes client requests and delegates them to the view for rendering back to the client

The **Model** we are talking about here is not necessarily persistent data (a data model) as such; rather, it represents the information passed back and forth between the client and different layers of the server application, which form the building blocks of any system.

Besides the **Model**, **View**, and **Controller** components, `DispatcherServlet` too plays a crucial role in the Spring MVC architecture. It acts as the Front Controller, a popular J2EE design pattern adopted by many MVC Frameworks. In fact, `DispatcherServlet` does much more than just a Front Controller. It will be explained in detail in the next section.

In a Spring MVC application, `DispatcherServlet` first receives a client request hitting the server via HTTP with a URL. With the help of the `HandlerMapping` configuration, `DipatcherServlet` finds the appropriate **Controller** method for the request based on the URL pattern and delegates the request to it. The Controller processes the request, optionally fills in the **Model** object, and returns the name of the **View** to be rendered. `DispatcherServlet` then picks the **View** up and renders it back on the client after applying the attributes of the **Model** to the placeholders in the **View**.

What’s mentioned in the previous paragraph is simply the typical request processing flow of Spring MVC. However, it is extremely flexible, with a great many options to support different types of view technologies and input and output structures and formats, including files, streams, and so on. We will explore them more in the following sections.

DispatcherServlet explained

`DispatcherServlet` is the gateway to any Spring MVC application. Inherited from `javax.servlet.http.HttpServlet`, it is typically configured declaratively in the `web.xml` file. While you can have multiple definitions of `DispatcherServlet` with unique URL patterns, most Spring MVC applications only have single `DispatcherServlet` with the `context-root` URL(`/`), that is, all requests coming to that domain will be handled by `DispatcherServlet`.

Starting from Servlet 3.0, in addition to declarative configuration in the `web.xml` file, `DispatcherServlet` can be configured programmatically by implementing or extending either of these three support classes provided by Spring:

- The `WebAppInitializer` interface
- The `AbstractDispatcherServletInitializer` abstract class
- The `AbstractAnnotationConfigDispatcherServletInitializer` abstract class

The following code listing demonstrates how to implement a `WebAppInitializer` directly in your application:

```
public class ApplicationInitializer implements WebApplicationInitializer {

  private static final Logger logger =
LoggerFactory.getLogger(ApplicationInitializer.class);

  @Override
  public void onStartup(ServletContext servletContext) throws
ServletException {

    logger.info("===== Application is starting up! ========");
    XmlWebApplicationContext appContext = new XmlWebApplicationContext();
    appContext.setConfigLocation("/WEB- INF/spring/appServlet/servlet-
context.xml");

    ServletRegistration.Dynamic registration =
servletContext.addServlet("rootDispatcher", new
DispatcherServlet(appContext));
    registration.setLoadOnStartup(1);
    registration.addMapping("/");
  }
```

WebApplicationContext – ApplicationContext for the Web

`DispatcherServlet` uses a specialized `ApplicationContext` called `WebApplicationContext` that has many web request processing capabilities. It is aware of which `ServletContext` it is associated with and is also capable of resolving themes. This interface has concrete implementations for specific contexts such as XML, `@Configuration` annotated classes, and portlets. By default, `DispatcherServlet` uses `XMLWebApplicationContext`. When `DispatcherServlet` is loaded, it looks for the bean configuration file of `WebApplicationContext` and initializes it.

`WebApplicationContext` objects are hierarchical. Every Spring MVC application has root `ApplicationContext` (configurable with a `context-param` tag called `contextConfigLocation` in the `web.xml` file), and each Servlet, including `DispatcherServlet`, has its own child context (configurable by its own `init-param`, `contextConfigLocation`). Ideally, Servlet-specific child contexts have beans customizing that Servlet, and root `ApplicationContext` has all shared beans.

Beans supporting DispatcherServlet and their roles

Upon receiving a web request, `DispatcherServlet` performs a set of operations in sequence as part of the request processing, with the help of a set of supporting beans. This table lists these special beans and their responsibilities:

Bean	**Responsibilities**
`HandlerMapping`	Maps incoming web requests to handlers and pre- and post-processors
`HandlerAdapter`	Invokes the handler which resolves arguments and dependencies, such as annotated arguments for URL-mapped controller method endpoints
`HandlerExceptionResolver`	Allows programmatic handling of exceptions and maps exceptions to views
`ViewResolver`	Resolves logical view names to view instances
`LocaleResolver`	Resolves the client’s locale in order to enable internationalization
`LocaleContextResolver`	A richer extension of `LocaleResolver`, with timezone information
`ThemeResolver`	Resolves themes configured in your app for enhanced user experience
`MultipartResolver`	Handles multipart file uploads as part of HTTP requests
`FlashMapManager`	Manages FlashMap instances that store temporary Flash attributes between requests redirected from one another

`DispatcherServlet` is extremely flexible; we can even create and configure custom implementations for all these beans. However, Spring MVC provides a set of nice implementations by default so that you don’t need to customize or provide your own implementations unless absolutely required. These default implementations can be found inside `org.springframework.web.servlet.DispatcherServlet.properties`. If you override them with your own implementation of any of these beans, yours will override the defaults.

Controllers in detail

Controllers, with their methods annotated with `@RequestMapping`, handle web requests. They accept input data in multiple forms and transform them into `Model` attributes to be consumed by views that are displayed back to the client. They connect the user to service-layer beans, where your application behavior is defined.

A Controller in Spring MVC has the following signature:

```
public interface Controller {

   ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception;
}
```

A Controller is designed as an interface, allowing you to create any kind of implementation. Starting from Spring version 2.5, you can turn any class into a Controller just by annotating it with `@Controller`. It relieves you from implementing any specific interface or extending a framework-specific class:

```
@Controller
public class HomeController {

   @RequestMapping(value = "/", method = RequestMethod.GET)
   public String home(Model model) {
      logger.info("Welcome to Taskify", locale);
      return "home";
   }
}
```

The `@Controller` annotation assigns the role of a Controller to the given class. A Spring MVC application autodetects all the controllers in its classpath and registers them with `WebApplicationContext` if you enable component scanning, as shown here:

```
<context:component-scan base-package="com.taskify" />
```

`@Controller`, `@RequestMapping`, and a set of other annotations form the basis of Spring MVC. These annotations allow flexible method names and signatures for controllers. We will explore them in detail in the following section.

Mapping request URLs with @RequestMapping

The `@RequestMapping` annotation maps request URLs onto an entire `@Controller` class or its handler methods. It can be applied at the class as well as the method levels. Typically, you apply class-level `@RequestMapping` annotation to map a group of related URLs, such as a form with many actions, and method-level `@RequestMapping` annotation for specific actions, such as create, read, update, delete, upload, and download. Let’s take a look at a typical form-based Controller with various actions in a pure REST model (`GET`, `POST`, `PUT`, and `DELETE`):

```
@Controller
@RequestMapping("/users")
public class UserController {

   @Autowired
   private UserService userService;

   @RequestMapping(method = RequestMethod.GET)
   public String listAllUsers(Locale locale, Model model) {
      model.addAttribute("users", userService.findAllUsers());
      return "user/list";
   }

   @RequestMapping(path = "/new", method = RequestMethod.GET)
   public String newUserForm(Model model) {
      User user = new User();
      user.setDateOfBirth(new Date());
      model.addAttribute("user", user);
      return "user/new";
   }

   @RequestMapping(path = "/new", method = RequestMethod.POST)
   public String saveNewUser(@ModelAttribute("user") User user, Model
model) {
      userService.createNewUser(user);
      return "redirect:/user";
   }
   @RequestMapping(path = "/{id}", method = RequestMethod.GET)
   public ModelAndView viewUser(@PathVariable("id") Long id) {
      return new ModelAndView("user/view").addObject("user",
userService.findById(id));
   }

   @RequestMapping(path = "/{id}/edit", method = RequestMethod.GET)
   public String editUser(@PathVariable("id") Long id, Model model) {
      model.addAttribute("user", userService.findById(id));
      return "user/edit";
   }

   @RequestMapping(path = "/{id}", method = RequestMethod.PUT)
   public String updateUser(@PathVariable("id") Long id,
@ModelAttribute("user") User user, Model model) {
      userService.updateUser(user);
      model.addAttribute("user", userService.findById(user.getId()));
      return "redirect:/user/" + id;
   }

   @RequestMapping(path = "/{id}", method = RequestMethod.DELETE)
   public String deleteUser(@PathVariable("id") Long id, Model model) {
      User existingUser = userService.findById(id);
      userService.deleteUser(existingUser);
      return "redirect:/user";
   }
}
```

`UserController`, listed in the preceding code, has methods that serve as request handlers for URLs representing CRUD operations on user entities with the help of `UserService`, which is injected as a dependency into the Controller. Since this Controller is based on web views, the handler methods fill up the Model and returns either a view name or `ModelAndView` object for further display. The final two handler methods, `updateUser()` and `deleteUser()`, redirect the requests at the end. They perform URL redirection after returning the response to the client.

Notice that `UserController` has a root URL (`/user`) and handler methods have a more narrow mapping with a combination of HTTP methods. They are invoked by the exact URLs seen in the following table:

URL	**Handler method**	**HTTP method**	**Matching URL (sample)**
`/`	`listAllUsers`	`GET`	`http://localhost:8080/user`
`/new`	`newuserForm`	`GET`	`http://localhost:8080/user/new`
`/new`	`saveNewUser`	`POST`	`http://localhost:8080/user/new`
`/{id}`	`viewUser`	`GET`	`http://localhost:8080/user/123`
`/{id}/edit`	`editUser`	`GET`	`http://localhost:8080/user/123/edit`
`/{id}`	`updateUser`	`PUT`	`http://localhost:8080/user/123`
`/{id}`	`deleteUser`	`DELETE`	`http://localhost:8080/user/123`

The HTTP methods `GET` and `POST` are supported by default, in line with the limited HTML (hence browser) support for the other two. However, for `PUT` and `DELETE` to work, you need to register `HiddenHttpMethodFilter` in your `web.xml` file. Use this code:

```
<filter>
   <filter-name>httpMethodFilter</filter-name>
   <filter-class>org.springframework.web.filter.
HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>httpMethodFilter</filter-name>
   <servlet-name>rootDispatcher</servlet-name>
</filter-mapping>
```

`HiddenHttpMethodFilter` works even without Spring MVC; you can use it with any Java web framework or even a plain Servlet application.

URI template patterns with the @PathVariable annotation

In the sample `UserController` listing in the preceding code, you might have noticed templated URL patterns with variable names replaced by values when handling requests. See this, for example:

```
@RequestMapping(path = "/{id}/edit", method = RequestMethod.GET)
public String editUser(@PathVariable("id") Long id, Model mdl) { … }
```

Here, the templated variable, `id`, is mapped to an `@PathVariable` annotation. It is enclosed inside curly braces and annotated as a method argument for mapping. A URL can have any number of path variables. They support regular expressions as well as path patterns in the Apache Ant style. They help you build perfect URI endpoints in the classic REST model.

Binding parameters with the @RequestParam annotation

Request parameters that are inline with URI strings can be mapped with method arguments using the `@RequestParam` annotation. See the following excerpt from `TaskController`:

```
@Controller
public class TaskController {
...
   @RequestMapping(path = "/tasks", method = RequestMethod.GET)
   public String list(@RequestParam(name = "status", required = false)
String status, Model model) {
      model.addAttribute("status", status);
      model.addAttribute("tasks", taskService.findAllTasks(status));
      return "task/list";
   }
...
}
```

A typical URL invoking the above handler is `http:<context-root>/tasks?status=Open`.

`@RequestParam` has four attributes: `name`, `required`, `value`, and `defaultValue`. While `name` is a mandatory attribute, all the others are optional. By default, all request parameters are required to be set to `true`, unless you specify them as `false`. Values of `@RequestParam` are automatically type-converted to parameter types by Spring.

Request handler method arguments

The `@RequestMapping` methods can have flexible method signatures; a mix of frameworks, custom objects, and annotations are supported. They are injected automatically during request processing if found as method arguments. Here is a list of a few supported framework classes and annotations; refer to the Spring official documentation or the Javadoc of `RequestMapping` for the complete list.

Supported classes	**Annotations**
`javax.servlet.ServletRequest`	`@PathVariable`
`javax.servlet.ServletRequest`	`@RequestVariable`
`javax.servlet.http.HttpSession`	`@RequestParam`
`org.springframework.ui.Model`	`@RequestHeader`
`org.springframework.validation.BindingResult`	`@RequestBody`
`Java.util.Map`	`@RequestPart`
`Java.io.InputStream`	`@InitBinder`

While the framework classes do not need any specific annotation, custom classes often need to accompany one of the supported annotations for the handler adapters in order to convert/format from the incoming web request object into the class instances.

Request handler method return types

Similar to flexible argument types, methods annotated by `@RequestMapping` can have either custom types (often annotated as `@ResponseBody`) or one of the many supported framework classes. The following list contains some of the many supported types:

- `org.springframework.web.servlet.ModelAndView`
- `org.springframework.ui.Model`
- `java.util.Map`
- `org.springframework.web.servlet.View`
- `java.lang.String`
- `void`
- `java.util.concurrent.Callable<?>`
- `org.springframework.http.HttpEntity`

Setting Model attributes

`Model` attributes are for the consumption of the view for display and binding with form elements. They can be set at both the controller and handler method level.

Any method with a non-void return type can be annotated as `@ModelAttribute` to make the method return type a `Model` attribute for all views resolved by the declared Controller. See an example:

```
@ModelAttribute(value = "users")
public List<User> getUsersList() {
   return userService.findAllUsers();
}
```

Model attributes specific to a view are set inside the handler method from where the view was resolved. Here is an example:

```
@RequestMapping(path = "/tasks/new", method = RequestMethod.GET)
public String newTaskForm(Model model) {
   model.addAttribute("task", new Task());
   return "task/new";
}
```

Building RESTful services for JSON and XML media

A web application often needs to expose some of its services as web APIs with the XML or JSON data formats, or both, for the consumption of AJAX requests from browsers as well as other devices, such as mobile and tablets.

REpresentational State Transfer (**REST**), is an established architectural style for building web APIs that align with native web protocols and methods. With REST, data is represented as resources that can be accessed and manipulated using a URI over the stateless protocol of HTTP. REST insists on the mapping of the create, read, update, and delete operations (CRUD) around a resource with the HTTP methods `POST`, `GET`, `PUT`, and `DELETE`, respectively.

Spring MVC makes it extremely easy to build simple API endpoints that consume and produce different media types such as text, JSON, and XML. A request handler method in an `@Controller` annotation can accept JSON, XML, or any other media type using the following two steps:

1. Set the attribute `consumes` to the appropriate media type(s) at the `RequestMapping` method, for example, `consumes = {"text/plain", "application/json"})`.
2. Annotate the method argument of the required type with `@RequestBody`. The web request is expected to contain the data in the format mentioned in step 1 (`consumes`; JSON, XML, and so on) and is resolved to this type by `HttpMessageConverter` during handling.

Similarly, the request handler method can produce JSON, XML, or any other media type using the following two steps:

1. Set the attribute `produces` with the appropriate media type(s) at the `RequestMapping` method, for example, `consumes = {"text/plain", "application/json"})`.
2. Annotate the return type of the handler method or the method declaration itself (next to `@RequestMapping`) with `@ResponseBody`. The handler will transform the return value into the data format specified in the `produces` attribute of `RequestMapping`.

The `consumes` and `produces` attributes of `RequestMapping` narrow down the primary mapping to the given media type (for example, `consumes = "application/xml"`) or a sequence of media types (for example, `consumes = {"text/plain", "application/json"}`).

In addition to the attributes, make sure the following library exists in the `pom.xml` file:

```
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.6.2</version>
</dependency>
```

Here is an example handler method that accepts a JSON request and returns a JSON response:

```
@RequestMapping(path = "/tasks/new.json", method=RequestMethod.POST,
consumes = "application/json", produces = "application/json")
@ResponseBody
public CreateTaskResponse createNewTaskJSON(@RequestBody CreateTaskRequest
createRequest) {
   Task task = new Task();
   task.setName(createRequest.getTaskName());
   ...
   return new CreateTaskResponse(taskService.createTask(task));
}
```

This handler method expects a web request with JSON content:

```
{
   "taskName":"Project estimation",
   "priority": 2,
   "creatorId": 1,
   "assigneeId": 2,
   "comments": "Involve the team in the process"
}
```

Now, the same method could be modified slightly to support XML content, `consumes` as well as `produces`. Look at the following listing:

```
@RequestMapping(path = "/tasks/new.xml", method = RequestMethod.POST,
consumes = "application/xml", produces = "application/xml")
@ResponseBody
public CreateTaskResponse createNewTaskXML(@RequestBody CreateTaskRequest
createRequest) {
   Task task = new Task()
   task.setName(createRequest.getTaskName());
   . . .
   return new CreateTaskResponse(taskService.createTask(task));
}
```

Make sure you have the JAXB annotation `@XmlRootElement` at the root of both `RequestBody` and `ResponseBody` types (`CreateTaskRequest` and `CreateTaskResponse` in this case).

You can invoke the preceding XML handler by sending the following content with the web request to the handler URI:

```
<CreateTaskRequest>
   <taskName>Estimate the project</taskName>
   <priority>2</priority>
   <creatorId>1</creatorId>
   <assigneeId>2</assigneeId>
   <comments>Involve the team in the process</comments>
</CreateTaskRequest>
```

Building a RESTful service with RestController

`RestController` is a convenient stereotype provided for building REST API endpoints that serve custom media types such as JSON or XML. It combines `@Controller` with `@ResponseBody`, that is, you do not need to annotate `@ResponseBody` in the handler methods. `@RequestMapping` methods assume `@ResponseBody` semantics by default.

Let’s see what the JSON handler method looks like when it becomes part of an `@RestController` annotation:

```
@RestController
public class TaskRestController {
   . . .
  @RequestMapping(path="/api/tasks/new.json", method=RequestMethod.POST,
consumes="application/json",produces= "application/json")
  public CreateTaskResponse createNewTaskJSON(@RequestBody
CreateTaskRequest createRequest) {
    Task task = new Task();
    task.setName(createRequest.getTaskName());
    . . .
    return new CreateTaskResponse(taskService.createTask(task));
  }}
}
```

Notice that the only difference in the mapping is the missing `@ResponseBody` annotation. It is best practice to define your REST APIs inside REST controllers.

Asynchronous request processing in Spring MVC

In an age of APIs, AJAX clients, and devices, web servers are under exponentially growing traffic. Figuring out ways to make servers more scalable is an ongoing challenge for server vendors. The traditional **one thread per HTTP connection** strategy does not scale well for a bigger number of concurrent user access. In this model, every request blocks a thread from the thread pool allocated by the Servlet container until the request is completely processed (the examples shown so far follow this model). When AJAX clients —where a single screen frequently fires multiple concurrent connection requests—join the traditional, blocking I/O model of web servers with long-running processes, servers easily get exhausted due to the thread starvation problem, since no free thread is available in the pool. This makes the application unavailable on increased load.

Asynchronous HTTP request processing is a technique that utilizes the non-blocking I/O capability of the Java platform's NIO API. In this model, a server thread is not constantly attached to a persistent HTTP connection during the whole request processing. The Servlet container releases the container thread as soon as the request is received and further processing is delegated to a thread managed by another application (Spring, in this case) so that the container thread is free to serve new incoming requests. This non-blocking request processing model saves a lot of server resources and steadily increases the scalability of the server.

Servlet 3.0 introduced asynchronous processing support, and Spring has implemented this support starting from Spring 3.2. As of 4.2, Spring provides two easy ways of defining asynchronous request handlers:

- Returning a `java.util.concurrent.Callable` instance instead of a value and producing the actual return value form inside the `call` method of `Callable`, that is, a thread managed by Spring, instead of a Servlet container
- Returning an instance of the Spring-specific `DeferredResult` type and producing the actual return value form inside any other thread or external event, such as JMS or a Quartz scheduler

Both these methods release the container thread at the earliest possible opportunity and use external threads to continue long-running transactions asynchronously. Let's look at an example for the first option, that is, using `Callable`:

```
@RequestMapping(path="/tasks/new.xml",method= RequestMethod.POST, consumes
= "application/xml", produces = "application/xml")
@ResponseBody
public Callable<CreateTaskResponse> createNewTaskXMLAsyncCallable(
@RequestBody CreateTaskRequest createRequest) {
   return new Callable<CreateTaskResponse>() {

      @Override
      public CreateTaskResponse call() throws Exception {
         Task task = new Task();
         task.setName(createRequest.getTaskName());
         . . .
         Task persistedTask = taskService.createTask(task);
         // Send an email here…
         // Send some push notifications here…
         . . .
         return new CreateTaskResponse(persistedTask);
      }
   };
}
```

In this method, you can see that the handler method returns the `Callable` object immediately after receiving the request and without waiting for the `Callable.call()` method to execute. Spring MVC invokes the `call()` method in another thread using `TaskExecutor`, and the response is dispatched back to the Servlet container once the `call()` method returns the value.

The following is an example of how to use `DeferredResult`:

```
@RequestMapping(path = "/tasks/new-async-deferred.json", method =
RequestMethod.POST, consumes = "application/json", produces =
"application/json")
@ResponseBody
public DeferredResult<CreateTaskResponse>
createNewTaskJSONAsyncDeferredResult( @RequestBody CreateTaskRequest
createRequest) {

   DeferredResult<CreateTaskResponse> defResult = new DeferredResult<>();
   CompletableFuture.runAsync(new Runnable() {
      @Override
      public void run() {
         Task task = new Task();
         task.setName(createRequest.getTaskName());
         . . .
         Task persistedTask = taskService.createTask(task);
         // Send an email here…
         // Send some push notifications here…
         defResult.setResult(newCreateTaskResponse(persistedTask));
      }
   });
   return deferredResult;
}
```

Remember, you must enable asynchronous processing support in `DispatcherServlet` as well as for all Servlet filters declared in the `web.xml` file (or wherever you are defining them—maybe in the JavaConfig class) for it to work. The following code shows how you set it in `web.xml`:

```
<servlet>
   <servlet-name>appServlet</servlet-name>
   <servlet-class>
      org.springframework.web.servlet.DispatcherServlet</servlet-class>
   . . .
   <async-supported>true</async-supported>
```

```
</servlet>
```

You may choose any of the preceding approaches as per your convenience to enable asynchronous processing. Consider designing all your non-trivial services to work asynchronously for high scalability and performance.

Working with views

Spring MVC provides a very flexible view resolution mechanism that is fully decoupled from the other elements of the MVC framework. It does not force you to use a particular view technology; rather, it makes it easier to use your own favorite technology. It even allows you to mix and match multiple technologies at the view tier. Spring MVC provides out-of-the-box support for JPS, XSLT, and Velocity views.

Resolving views

In a typical Spring MVC application, the developer chooses a view technology of his choice and accordingly uses a `ViewResolver` that resolves views built using that technology.

The component responsible for resolving views in a Spring MVC application is `org.springframework.web.servlet.ViewResolver`. It maps logical view names with physical view resources and the chosen view technology.

All request-handling methods of controllers must resolve a logical view name by either returning a view name, a view object, or a `ModelAndView` object. The `org.springframework.web.servlet.View` object prepares `HttpRequest` for the consumption of the chosen view technology.

Spring MVC comes with a set of convenient view resolvers out of the box:

ViewResolver	**Description**
`AbstractCachingViewResolver`	This is a convenient base class for `ViewResolver` implementations. For better performance, it caches view objects once they are resolved.
`XmlViewResolver`	This uses bean definitions from a dedicated XML file to resolve view definitions. The file is specified by a resource location. By default, it is located at `WEB-INF/views.xml`.
`ResourceBundleViewResolver`	This uses bean definitions in `ResourceBundle` specified by the bundle basename in order to define views. The default basename is `views.properties`.
`UrlBasedViewResolver`	This resolves view names with physical resources in the matching URL. Its two supporting properties, prefix and suffix, help locate the resource.
`InternalResourceViewResolver`	This resolves Servlets and JSPs with JSTL support. It is a subclass of `UrlBasedViewResolver`.
`VelocityViewResolver`	This resolves Velocity templates and is a subclass of `UrlBasedViewResolver`.
`FreeMarkerViewResolver`	This resolves FreeMarker templates. It is a subclass of `UrlBasedViewResolver`.
`JasperReportsViewResolver`	This resolves JasperReport views for different formats, such as CSV, HTML, XLS, and XLSX.
`TilesViewResolver`	This resolves Tiles views for both version 2 and 3.

The sample application in this chapter uses `UrlBasedViewResolver` for resolving JSP views. When you use multiple view technologies in a web application, you may use `ResourceBundleViewResolver`.

Resolving JSP views

Java Server Pages (**JSP**), the primary web templating technology for Java EE, is a simple and easy tool for the rapid development of dynamic web content based on JVM. Built on top of Servlet technology, JSP has direct access to the entire Java API. JSP makes a web page author’s life a lot easier by allowing him to design web pages in natural HTML format and then embed the required Java code inside scriptlet blocks.

Java Server Pages Tag Library (**JSTL**) is a set of standardized HTML-style tags highly useful for JSP pages. JSTL eliminates the need to mix Java code inside JSP pages, thus making JSP pages much cleaner and easier to author.

Spring MVC resolves JSP pages using `InternalResourceViewResolver`. In an earlier section, *Your first Spring MVC application*, we already configured the `ViewResolver` class for JSP, as follows:

```
<beans:bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <beans:property name="prefix" value="/WEB-INF/views/" />
  <beans:property name="suffix" value=".jsp" />
</beans:bean>
```

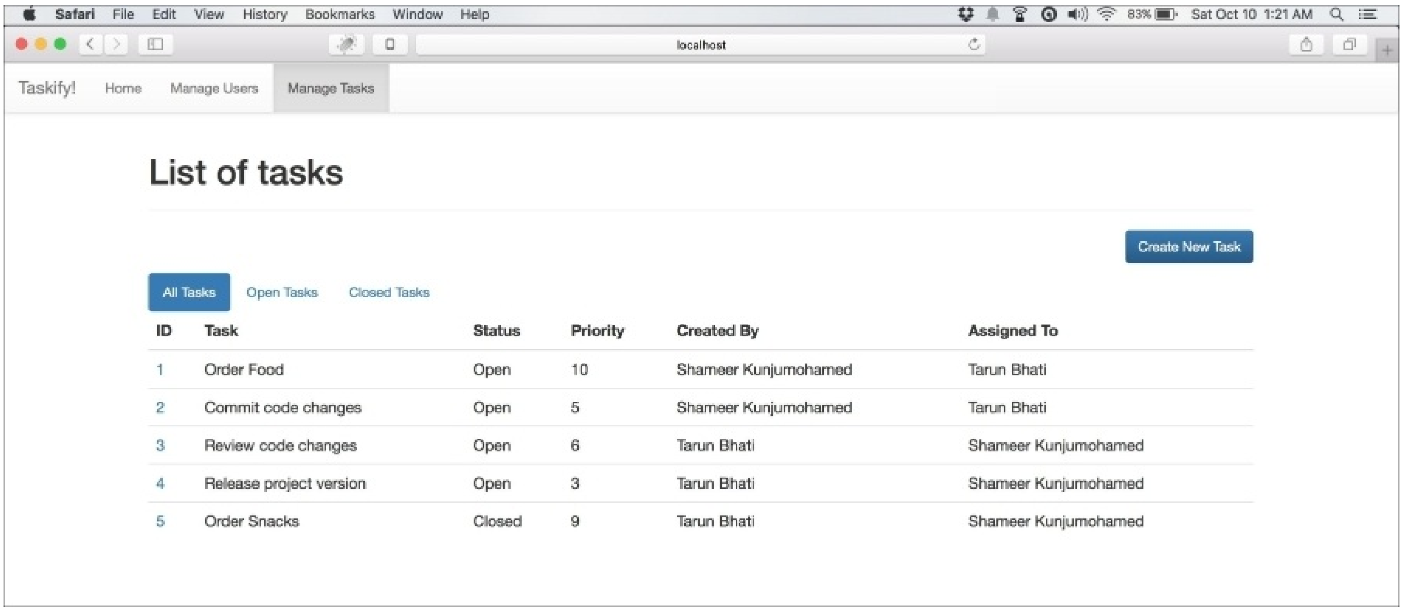
Spring MVC recommends keeping your view files (JSP in this case) under the `WEB-INF` directory to avoid direct client access. `ViewResolver` discovers the view files from the physical location and caches them by default once resolved, which helps performance.

Binding Model attributes in JSP pages using JSTL

Views have access to `Model` attributes set from associated handler methods and controllers. These `Model` attributes can be displayed in JSP views with the help of JSTL. In the following example, the `Model` attribute `tasks` is listed using JSTL:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<table class="table table-hover">
  <thead>
    <tr>
      <th>ID</th>
      <th>Task</th>
      <th>Status</th>
      <th>Priority</th>
      <th>Created By</th>
      <th>Assigned To</th>
    </tr>
  </thead>
  <tbody>
    <c:if test="${not empty tasks}">
      <c:forEach var="task" items="${tasks}">
        <tr>
          <td>
            <a href='<c:url value="/tasks/${task.id}"/>'>${task.id}</a>
          </td>
          <td>${task.name}</td>
          <td>${task.status}</td>
          <td>${task.priority}</td>
          <td>${task.createdBy.name}</td>
          <td>${task.assignee.name}</td>
        </tr>
      </c:forEach>
    </c:if>
  </tbody>
</table>
...
```

You may have noticed the declaration and usage of JSTL tags in the preceding JSP extract of the `/tasks/list.jsp` view. Here is how it would be rendered with proper styling in a browser:



List of tasks

ID	Task	Status	Priority	Created By	Assigned To
1	Order Food	Open	10	Shameer Kunjumohamed	Tarun Bhati
2	Commit code changes	Open	5	Shameer Kunjumohamed	Tarun Bhati
3	Review code changes	Open	6	Tarun Bhati	Shameer Kunjumohamed
4	Release project version	Open	3	Tarun Bhati	Shameer Kunjumohamed
5	Order Snacks	Closed	9	Tarun Bhati	Shameer Kunjumohamed

Spring and Spring form tag libraries

Spring bundles a set of tags for the easier authoring of plain JSP pages and JSP forms, defined in `spring.tld` and `spring-form.tld` respectively. `spring.tld` describes general-purpose JSP tags commonly used in JSP pages, listed in the following table:

Spring tag	**Description**
`<spring:bind/>`	This allows the data binding of an attribute given in the bind path of a locally declared bean or a `Model` attribute and provides a `BindStatus` object to the enclosed body content.
`<spring:escapeBody/>`	This applies HTML escaping and JavaScript escaping for the body.
`<spring:hasBindErrors/>`	This provides an error instance if there are bind errors.
`<spring:htmlEscape/>`	This sets an HTML escape value for the current JSP page.
`<spring:message/>`	This displays a message for a given code, usually resolved from a resource bundle.
`<spring:nestedPath/>`	This sets a nested path of `ModelAttribute` to the `<spring:bind/>` tags enclosed inside.
`<spring:theme/>`	This loads the theme resource using the given code.
`<spring:transform/>`	This transforms properties inside the `<spring:bind/>` tag and exports them to a variable in a given scope.
`<spring:url/>`	This creates a URL with URI template variables. It is modeled after the JSTL `c:url` tag.
`<spring:eval/>`	This evaluates SpEL expressions.

Spring `form` tags provide data binding for HTML forms. They have tight integration with request handlers in controllers. Generally, they represent similarly named HTML `form` elements and share common attributes:

Form tag	**Sample**
`<form:input/>`	`<form:input path="name" placeholder="Task Name"/>`
`<form:textarea/>`	`<form:textarea path="comments" id="txtComments" rows="5" cols="30" />`

`<form:select/>`
`<form:option/>` and
`<form:options/>`
	`<form:select path="createdBy" id="selectCreatedBy">`
	`<form:option value="-1" label="Select"/>`
	`<form:options items="${users}" itemValue="id" itemLabel="name" />`
	`</form:select>`

`<form:label/>`	`<form:label for="txtTaskName" path="name">Task-names</form:label>`
`<form:hidden/>>`	`<form:hidden path="taskId" id="hdnTaskId"/>`
`<form:password/>`	`<form:password path="userPassword"/>`
`<form:radiobutton/>`	`<form:radiobutton path="sex" value="Male"/>`

`<form:radiobuttons/>`	`<form:radiobuttons path="sex" items="${sexOptions}"/>`
`<form:checkbox/>`	`<form:checkbox path="task.priority" value="1"/>`
`<form:checkboxes/>`	`<form:checkboxes path="task.priority" value="${taskPriorities}"/>`
`<form:password/>`	`<form:password path="password" />`
`<form:errors/>`	`<form:errors path="createdBy.id" />`

Composing a form in JSP

Spring forms can be composed in JSP pages using the `<spring>` and `<form>` tags. For the purpose of illustration, let's take a look at a JSP form that uses both the Spring and form tag libraries along with JSTL. The following is a stripped-down version of `views/task/new.jsp`:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
...
<form:form action="new" method="post" commandName="task">
  <spring:bind path="name">
    <div class="form-group${status.error ? ' has-error' : ''}">
      <label for="txtTaskName">Task-name</label>
      <form:input path="name" class="form-control" id="txtTaskName"
placeholder="Task Name" />
      <form:errors path="name" cssClass="control-label" />
    </div>
  </spring:bind>
  <div class="form-group">
    <label for="txtComments">Comments</label>
    <form:textarea path="comments" class="form-control" id="txtComments"
placeholder="Comments" rows="5" cols="30" />
  </div>
  <spring:bind path="createdBy.id">
    <div class="form-group${status.error ? ' has-error' : ''}">
      <label for=" slCrtBy ">Created By</label>
      <form:select path="createdBy" id="slCrtBy" class="form-control">
        <form:option value="-1" label="-- Select --">
        </form:option>
        <form:options items="${users}" itemValue="id" itemLabel="name" />
      </form:select>
      <form:errors path="createdBy.id" cssClass="control-label" />
    </div>
  </spring:bind>
  <button type="submit" class="btn btn-success">Save</button>
  <a href='<c:url value="/tasks"/>' class="btn btn-primary">Cancel</a>
</form:form>
```

As you can see in the preceding code listing, you must declare the JSTL, Spring, and form tag library directives at the top of your JSP page before you use Spring form tags.

All `<form>` elements should be inside a `<form:form/>` container element. The `commandName` attribute value of `<form:form/>` binds the `Model` attribute with the name in the handler method of the Controller. The handler method from which the preceding JSP form is resolved would look like the following code:

```
@RequestMapping(path = "/tasks/new", method = RequestMethod.GET)
public String newTaskForm(Model model) {
   model.addAttribute("task", new Task());
   model.addAttribute("priorities", priorities); // This is a collection
   model.addAttribute("users", userService.findAllUsers());
   return "task/new";
}
```

Notice the `Model` attribute, `task`, which is bound with the `<form:form/>` tag in the JSP page. The form is submitted to the following handler method, which again serializes the `Task` object back for persistence:

```
@RequestMapping(path = "/tasks/new", method = RequestMethod.POST)
public String createNewTask(@ModelAttribute("task") @Valid Task task,
BindingResult result, Model model) {
   if(result.hasErrors()) {
      return "task/new";
   } else {
      taskService.createTask(task);
      return "redirect:/tasks";
   }
}
```

Validating forms

Spring MVC makes form validation a lot easier using Spring's `Validator` framework. You might have noticed the `@Valid` annotation and the usage of the `BindingResult.hasErrors()` method call inside handler methods listed in the previous section. They are part of the validation framework.

Let's create a validator for a `Task` object by following these steps:

1. Add the Validation API's Maven dependency, `javax.validation` (build file: `pom.xml`).
2. Make sure you have defined `MessageSourceBean` for the `validation-errors` properties file in your bean definition:

   ```
   <beans:bean id="messageSource" class="org.springframework.context.
   support.ReloadableResourceBundleMessageSource">
      <beans:property name="defaultEncoding" value="UTF-8" />
      <beans:property name="basenames" value="classpath:validation-errors"
   />
   </beans:bean>
   ```

3. Make sure there is a `validation-errors.properties` file with the following sample content in your root resources location. You may add as many error messages into it as you like.

   ```
   error.invalid={0} is in invalid format.
   error.required={0} is required.
   ```

4. Create a `Validator` class, `TaskValidator`:

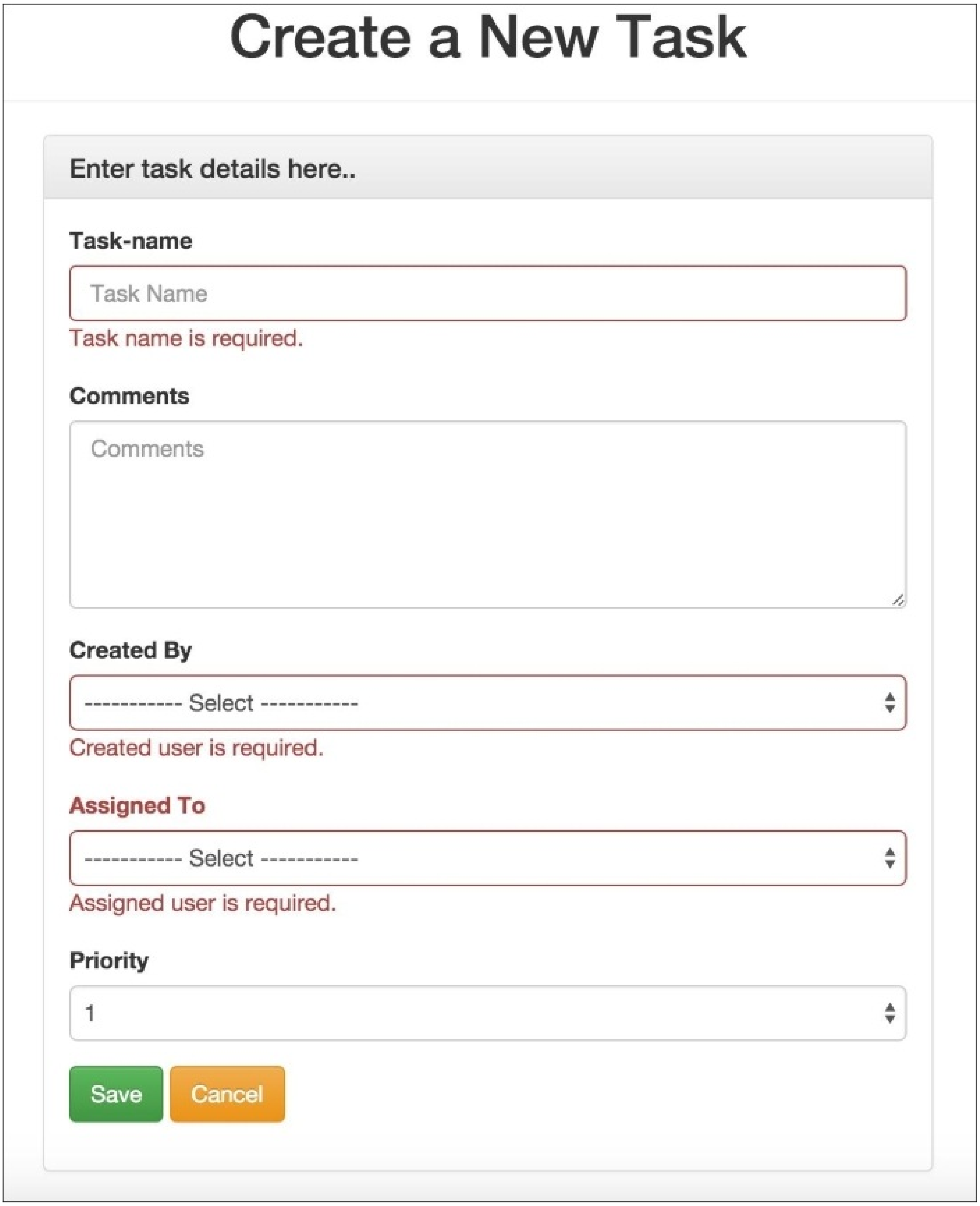
   ```
   public class TaskValidator implements Validator {

      @Override
      public boolean supports(Class<?> clazz) {
         return clazz.isAssignableFrom(Task.class);
      }

      @Override
      public void validate(Object target, Errors errors) {
         Task task = (Task) target;
         ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name",
   "error.required", new Object[] { "Task name" });
         ValidationUtils.rejectIfEmpty(errors, "createdBy.id",
   "error.required", new Object[] { "Created user" });
      }
   }
   ```

5. Register the `TaskValidator` class inside the `TaskController` class using `InitBinder`:

   ```
   @InitBinder("task")
   public void initBinder(WebDataBinder binder) {
      binder.addValidators(new TaskValidator());
   }
   ```

6. Annotate `ModelAttribute` with the `@Valid` annotation of `javax.validation.Valid` in the handler method.
7. Handle validation errors in the request handler method, as given in the `createNewTask()` method listed in the previous section.
8. Add a `<form:errors/>` tag for each form field you are validating—as seen in the `/tasks/new.jsp` file.

The form will look like this in case of validation errors:



Create a New Task

Enter task details here..

Task-name

Task Name

Task name is required.

Comments

Comments

Created By

----------- Select -----------

Created user is required.

Assigned To

----------- Select -----------

Assigned user is required.

Priority

1

Handling file uploads

Most web applications require multipart file upload functionality. Spring MVC makes it extremely easy to handle this otherwise cumbersome feature. It provides two built-in implementations of `MultiPartResolvers`: `CommonsMulipartResolver` for Apache Commons FileUpload and `StandardServletMultipartResolver` for the Servlet 3.0 API.

Since most modern web applications use a Servlet 3.0 container, let's see how the FileUpload functionality can be handled using `StandardServletMultipartResolver` with the help of following example:

1. Register your `MultipartResolver` in your `servlet-context.xml` file (or add it programmatically if you are using a Java configuration):

   ```
   <beans:bean id="multipartResolver"
   class="org.springframework.web.multipart.support.StandardServletMultipa
   rtResolver">
   </beans:bean>
   ```

2. Add multipart configuration support to your `DispatcherServlet` in your `web.xml` (or JavaConfig) file:

   ```
   <servlet>
     <servlet-name>appServlet</servlet-name>
   ...
     <multipart-config>
       <location>/tmp/servlet-uploads</location>
       <max-file-size>20848820</max-file-size>
       <max-request-size>418018841</max-request-size>
       <file-size-threshold>1048576</file-size-threshold>
     </multipart-config>
   </servlet>
   ```

3. Make sure that the location you provided in the previous section really exists. Create the directory if it doesn't.
4. Create the web form with an input file element in it. This sample JSP snippet uploads a user's profile image:

   ```
   <form:form action="../${user.id}/profileForm" method="post"
   enctype="multipart/form-data">
      <div class="form-group">
         <label for="txtUserName">Choose File</label>
         <input type="file" name="profileImage"/>
      </div>
      <button type="submit" class="btn btn-success">Upload</button>
      <a href="../${user.id}" class="btn btn-primary">Cancel</a>
   </form:form>
   ```

5. Create the request handler method in your Controller:

   ```
   @RequestMapping(path = "/{userId}/profileForm", method =
   RequestMethod.POST)
   public String uploadProfileImage(@PathVariable("userId") Long userId,
   @RequestParam("profileImage") MultipartFile file) throws IOException {
   ```

```
   User user = userService.findById(userId);
   String rootDir = FILE_ROOT_DIR + "/" + user.getId();

   if (!file.isEmpty()) {
      java.io.File fileDir = new java.io.File(fileSaveDirectory);
      if (!fileDir.exists()) {
         fileDir.mkdirs();
      }
      FileCopyUtils.copy(file.getBytes(), new java.io.File(rootDir  +
"/" + file.getOriginalFilename()));

      File profileImageFile = this.userService.addProfileImage(userId,
file.getOriginalFilename());
   }
   return "redirect:/users/" + userId;
}
```

Resolving Thymeleaf views

Thymeleaf is a Java-based XML/HTML/HTML5 template engine library to build web applications. It allows faster processing of templates and increased performance due to the intelligent caching of parsed view files. Please refer to the official Thymeleaf documentation for Thymeleaf page authoring.

You need Thymeleaf and Spring (`org.thymeleaf`) in your Maven dependencies in order to use Thymeleaf in your projects. Thymeleaf views can be resolved in your project with the following snippet:

```
<beans:bean class="org.thymeleaf.spring4.view.ThymeleafViewResolver">
   <beans:property name="templateEngine" ref="templateEngine" />
   <beans:property name="order" value="1" />
   <beans:property name="viewNames" value="*.html,*.xhtml" />
</beans:bean>

<beans:bean id="templateResolver" class=
"org.thymeleaf.templateresolver.ServletContextTemplateResolver">
   <beans:property name="prefix" value="/WEB-INF/templates/" />
   <beans:property name="suffix" value=".html" />
   <beans:property name="templateMode" value="HTML5" />
</beans:bean>

<beans:bean id="templateEngine"
class="org.thymeleaf.spring4.SpringTemplateEngine">
   <beans:property name="templateResolver" ref="templateResolver" />
</beans:bean>
```

More view technologies

Spring MVC supports an impressive set of view technologies; you can use any of these after adding the right Maven dependencies in your project. Spring provides view resolvers out of the box for most of the view technologies. Here is a list of other view technologies supported by Spring MVC:

- Velocity and FreeMarker
- Groovy markup templates
- JavaScript templates (on Nashhorn): Handlebars, Mustache, ReactJS, and EJS
- ERB templates on JRuby and String templates on Jython
- XML views and XSLT (built in)
- Tiles
- PDF (iText) and Excel (Apache POI)
- JasperReports
- Feed views

In most cases, you will need to mix and match view technologies in the same application. For example, you may use JSP for normal HTML screens, but you will still need JasperReports to report screens and may need to download some reports as PDF and Excel files. Using Spring MVC ensures that all these features can be easily integrated.

Summary

In this chapter, we learned how to build highly scalable and dynamic web applications using Spring MVC. Starting from setting up the project and configuring `WebApplicationContext` with proper layering, we explored different ways of designing controllers and map request handlers for both web and API endpoints—that too including asynchronous processing and multipart file uploads—using easily configurable components. Now, we can compose beautiful JSP pages using `<spring>` and `<form>` tags and also enable form validation using the Validation API.

So far, we have been holding data in memory without bothering about making it persistent somewhere. In the next chapter, we will dive one level deeper into the data layer of enterprise application development, learning various data access and persistence mechanisms with and without ACID transactions. We are going to deal with more serious concerns from this point.

Chapter 3. Accessing Data with Spring

Data access or persistence is a major technical feature of data-driven applications. This is a critical area where careful design and expertise is required. Modern enterprise systems use a wide variety of data storage mechanisms ranging from traditional relational databases such as Oracle, SQL Server, and Sybase to more flexible, schema-less NoSQL databases such as MongoDB, Cassandra, and Couchbase. Spring Framework provides comprehensive support for data persistence in multiple flavors of mechanism, ranging from convenient template components to smart abstractions over popular **ORM** (**Object Relational Mapping**) tools and libraries, making them much easier to use. Spring's data access support is another great reason for choosing it for developing Java applications.

Spring Framework offers the following primary approaches for data persistence mechanisms for developers to choose from:

- Spring JDBC
- ORM Data Access
- Spring Data

Furthermore, Spring standardizes the preceding approaches under a unified **DAO** (**Data Access Object**) notation called `@Repository`.

Another compelling reason for using Spring is its first class transaction support. Spring provides consistent transaction management, abstracting different transaction APIs such as JTA, JDBC, JPA, Hibernate, JDO, and other container-specific transaction implementations.

In order to make development and prototyping easier, Spring provides embedded database support, smart abstractions (`DataSource`), and excellent test integration. This chapter explores various data access mechanisms provided by Spring Framework and its comprehensive support for transaction management in both standalone and web environments, with relevant examples.

Note

Why use Spring Data Access when we have JDBC?

JDBC (**Java Database Connectivity**), the Java Standard Edition API for data connectivity from Java to relational databases, is a very a low-level framework. Data access via JDBC is often cumbersome; the boiler-plate code the developer needs to write makes the code error-prone. Moreover, JDBC exception handling is not sufficient for most use cases; there exists a real need for simplified but extensive and configurable exception handling for data access. Spring JDBC encapsulates the often repeating code, simplifying the developer code tremendously, and lets him/her focus directly on his business logic. Spring Data Access components abstract the technical details including the lookup and management of persistence resources such as connections, statements, and resultsets, and accept the specific SQL statements and relevant parameters to perform the operation. Spring Data Access components use the same JDBC API under the hood, while exposing simplified, straightforward interfaces for the client’s use. This approach makes for a much cleaner and hence maintainable data access layer for Spring applications.

Configuring DataSource

The first step to connect to a database from any Java application is to obtain a connection object specified by JDBC. `DataSource`, a part of Java SE, is a generalized factory of `java.sql.Connection` objects that represents the physical connection to the database and is the preferred means of producing a connection. `DataSource` handles transaction management, connection lookup, and pooling functionalities, relieving the developer of those infrastructural issues.

`DataSource` objects are often implemented by database driver vendors and typically looked up via JNDI. Application servers and Servlet engines provide their own implementations of `DataSource` (and) or connectors to `DataSource` objects provided by the database vendor. Typically configured inside XML-based server descriptor files, server-supplied `DataSource` objects generally provide built-in connection pooling and transaction support. As a developer, you just configure your data sources inside the server configuration files declaratively in XML and look them up from your application via JNDI.

In a Spring application, you configure your `DataSource` reference as a Spring bean, and inject it as a dependency into your DAOs or other persistence resources. The Spring `<jee:jndi-lookup/>` tag (of http://www.springframework.org/schema/jee namespace) allows you to look up and construct JNDI resources easily, including a `DataSource` object defined from inside an application server. For applications deployed in a J2EE application server, a JNDI `DataSource` object provided by the container is recommended.

```
<jee:jndi-lookup id="taskifyDS" jndi-
name="java:jboss/datasources/taskify"/>
```

For standalone applications, you need to create your own `DataSource` implementation or use third-party implementations such as Apache Commons DBCP, C3P0, or BoneCP. The following is a sample `DataSource` configuration using Apache Commons DBCP2. It provides configurable connection pooling features too.

```
<bean id="taskifyDS" class="org.apache.commons.dbcp2.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="${driverClassName}" />
    <property name="url" value="${url}" />
    <property name="username" value="${username}" />
    <property name="password" value="${password}" />
    <property name="initialSize" value="3" />
    <property name="maxTotal" value="50" />
    ...
</bean>
```

Make sure you add the corresponding dependency to your `DataSource` implementation in your build file. The following is for DBCP2:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
</dependency>
```

Spring provides `DriverManagerDataSource`, a simple implementation of `DataSource`, which is only meant for testing and development purposes, not for production use. Note that it does not provide connection pooling. Here is how you configure it in your application.

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

It can also be configured using the Java-based configuration, as shown in the following code:

```
@Bean
DataSource getDatasource() {
    DriverManagerDataSource dataSource = new
DriverManagerDataSource(pgDsProps.getProperty("url"));
    dataSource.setDriverClassName(
pgDsProps.getProperty("driverClassName"));
    dataSource.setUsername(pgDsProps.getProperty("username"));
    dataSource.setPassword(pgDsProps.getProperty("password"));
    return dataSource;
}
```

Note

Never use `DriverManagerDataSource` on production environments. Use third-party data sources such as DBCP, C3P0, and BoneCP for standalone applications, and JNDI `DataSource` provided by the container, for the J2EE container instead. They are more reliable and provide efficient connection pooling functionality off the shelf.

Using embedded databases

For prototyping and test environments, it is a good idea to use Java-based embedded databases to quickly start up the project and configure easily. They are lightweight and easily testable. Spring supports the HSQL, H2, and Derby database engines for that purpose natively. Here is a sample `DataSource` configuration for an embedded HSQL database:

```
@Bean
DataSource getHsqlDatasource() {
    return new
        EmbeddedDatabaseBuilder().setType(EmbeddedDatabaseType.HSQL)
         .addScript("db-scripts/hsql/db-schema.sql")
         .addScript("db-scripts/hsql/data.sql")
         .addScript("db-scripts/hsql/storedprocs.sql")
         .addScript("db-scripts/hsql/functions.sql")
         .setSeparator("/").build();
}
```

The XML version of this would look like the following code:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:db-scripts/hsql/ db-schema.sql" />
    . . .
</jdbc:embedded-database>
```

Handling exceptions in the Spring Data layer

With traditional JDBC-based applications, exception handling is based on `java.sql.SQLException`, which is a checked exception. It forces the developer to write `catch` and `finally` blocks carefully for proper handling and to avoid resource leakages such as leaving a database connection open. Spring, with its smart exception hierarchy based on `RuntimeException`, spares the developer from this nightmare. Having `DataAccessException` as the root, Spring bundles a bit set of meaningful exceptions, translating the traditional JDBC exceptions. Spring also covers Hibernate, JPA, and JDO exceptions in a consistent manner.

Spring uses `SQLErrorCodeExceptionTranslator`, which inherits `SQLExceptionTranslator` for translating `SQLException` to `DataAccessExceptions`. We can extend this class to customize the default translations. We can replace the default translator with our custom implementation by injecting into the persistence resources (such as `JdbcTemplate`, to be covered later). See the following code listing for how we define a `SQLExceptionTranslator` class in your code:

```
String userQuery = "select * from TBL_NONE where name = ?";
SQLExceptionTranslator excTranslator = new SQLExceptionTranslator() {

  @Override
  public DataAccessException translate(String task, String sql,
SQLException ex) {
    logger.info("SUCCESS --- SQLExceptionTranslator.translate invoked !!");
    return new BadSqlGrammarException("Invalid Query", userQuery, ex){};
  }
};
```

The preceding code snippet catches any `SQLException` and converts it into a Spring-based `BadSqlGrammarException` instance. Then, this custom `SQLExceptionTranslator` needs to be passed to the `Jdbctemplate` before use, as shown in the following code:

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate.setExceptionTranslator(excTranslator);
Map<String, Object> result = jdbcTemplate.queryForMap(userQuery, new
Object[] {"abc"});
```

Now, any invalid query will invoke the custom `SQLExceptionTranslator` class. You can customize its behavior according to your requirements.

DAO support and @Repository annotation

The standard way of accessing data is via specialized DAOs that perform persistence functions under the data access layer. Spring follows the same pattern by providing DAO components and allowing developers to mark their data-access components as DAOs, using the annotation `@Repository`. This approach ensures consistency over various data access technologies such as JDBC, Hibernate, JPA, and JDO, and project-specific repositories. Spring applies `SQLExceptionTranslator` across all these methods consistently.

Spring recommends your data-access components to be annotated with stereotype, `@Repository`. The term, repository, was originally defined in *Domain-Driven Design, Eric Evans, Addison Wesley* as “a mechanism for encapsulating storage, retrieval, and search behavior which emulates a collection of objects.” This annotation makes the class eligible for `DataAccessException` translation by Spring Framework.

Spring Data, another standard data-access mechanism provided by Spring, revolves around `@Repository` components. We will discuss this more in later sections.

Spring JDBC abstraction

Spring JDBC components simplify JDBC-based data access by encapsulating the boilerplate code and hiding the interaction with JDBC API components from the developer with a set of simple interfaces. These interfaces handle the opening and closing of JDBC resources (connections, statements, resultsets) as required. They prepare and execute statements, extract results from resultsets, provide callback hooks for converting, mapping and handling data, handle transactions, and translate SQL exceptions into the more sensible and meaningful hierarchy of `DataAccessException`.

Spring JDBC provides three convenient approaches for accessing relational databases:

- `JdbcTemplate`
- `SimpleJDBC` classes
- RDBMS `Sql*` classes

Each of these Spring JDBC categories has multiple flavors of components under them which you can mix-and-match based on your convenience and technical choice. You may explore them under the `org.springframework.jdbc` package and its subpackages.

JdbcTemplate

`JdbcTemplate` is the core component under Spring JDBC abstraction. This powerful component executes almost all of the possible JDBC operations with its simple, meaningful methods, accepting parameters for an impressive set of flavors of data access. It belongs to the package, `org.springframework.jdbc.core`, which contains many other supporting classes that help `JdbcTemplate` to complete its JDBC operations. A `DataSource` instance is the only dependency for this component. All other Spring JDBC components use `JdbcTemplate` internally for their operations.

Usually, you configure `JdbcTemplate` as yet another Spring bean, and inject it into your DAOs or into any other bean where you want to invoke its methods.

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <constructor-arg ref="dataSource"/>
</bean>
<bean id="userDAO"
class="com.springessentials.chapter3.dao.impl.UserJdbcDAO">
  <constructor-arg ref="jdbctemplate"/>
</bean>
```

Note

`JdbcTemplate` is one of the implementations of Template Pattern in Spring. Template Pattern is a behavioral pattern listed in the *Gang of Four* design pattern catalog. It defines the skeleton of an algorithm in a method or operation called **Template Method**, deferring some steps into the subclasses, without changing the algorithm's structure. `JdbcTemplate` is a collection of these Template Methods; the user can extend it and override some of the behaviors based on specific requirements. `JMSTemplate` and `JpaTemplate` are also examples of Template Pattern implementations.

`JdbcTemplate` executes SQL queries (`SELECT`), update statements (`INSERT`, `UPDATE`, and `DELETE`), stored procedure and function calls, returns extracted results (for `SELECT` queries), and invokes call-back methods for result-set extraction and mapping rows with domain objects. It has a comprehensive set of query and execute methods for different methods of result-set extraction. The following table introduces a few very useful `JdbcTemplate` methods:

Method	**Description**
`execute`	A set of overloaded methods for executing a SQL update (`INSERT`, `UPDATE`, and `DELETE`) statement, with different parameter sets including the SQL statement to be executed, bind parameters, a statement creator, and callback methods.
`query`	A set of overloaded methods for querying `PreparedStatement` for a given SQL `SELECT` statement with a multitude of parameter sets including bind parameters, argument types, `RowMapper`, `ResultSetExtractor`, `PreparedStatementCreator`, `RowCallbackHandler`, and so on. While methods with callbacks are void methods, the others return a list of objects of type `<T>` specified with the corresponding `RowMapper`, `ResultSetExtractor`, or a populated instance of type `<T>`.
	A set of overloaded query methods executing a `SELECT` query returns a list of objects of type `<T>`

queryForList	specified as an argument, `Class<T> elementType`. Those methods not specifying the `elementType` return `List<Map<String, Object>>`.
queryForMap	Executes a (`SELECT`) query and returns the result as `Map<String, Object>`.
queryForObject	A set of overloaded methods querying a given SQL `SELECT` statement with parameter sets including bind parameters, argument types, `RowMapper`, and the required return type `<T>`.
update	A set of overloaded methods issuing an update (`INSERT`, `UPDATE`, or `DELETE`) statement with parameter sets including bind parameters, argument types, `PreparedStatementCreator`, and so on. It returns an integer, which is the count of records affected.
batchUpdate	A set of overloaded methods for executing multiple SQL updates (`INSERT`, `UPDATE`, and DELETE) with different parameter sets including an array of SQL statements and many combinations of `PreparedStatementSetter` and other arguments.
execute	A set of overloaded methods for executing a SQL update (either `INSERT`, `UPDATE`, or `DELETE`) statement, with different parameter sets including the SQL statement to be executed, bind parameters, `StatementCreator`, and callback methods.
query	A set of overloaded methods for querying `PreparedStatement` for a given SQL `SELECT` statement with several parameter sets including bind parameters, argument types, `RowMapper`, `ResultSetExtractor`, `PreparedStatementCreator`, `RowCallbackHandler`, and so on. While those methods with callbacks are void methods, the others return a list of objects of type `<T>` specified with the corresponding `RowMapper`, `ResultSetExtractor`, or a populated instance of type `<T>`.

Behind the super capabilities of `JdbcTemplate` is a set of callback interfaces being passed as arguments for the methods listed in the preceding table. These execution hooks help `JdbcTemplate` to deal with relational data in a pure object-oriented and reusable fashion. A good understanding of these interfaces is critical for the right usage of `JdbcTemplate`. See the following table for these callback interfaces:

Callback interface	**Callback method(s)**	**Responsibilities**
CallableStatementCreator	execute	Constructs `java.sql.CallableStatement`, which is used to execute stored procedures inside its `createCallableStatement(Connection)`method:.
PreparedStatementCreator	execute, update, query	Constructs `java.sql.PreparedStatement,` given a connection, inside the method, `createPreparedStatement (Connection)`.
PreparedStatementSetter	update, query	Sets values to `PreparedStatement` before execution, inside `JdbcTemplate.setValues (PreparedStatement)`.
CallableStatementCallback	execute	Prepares `CallableStatement`. Usually sets the `IN` and `OUT` parameters of a stored procedure or function, before the actual execution, inside `JdbcTemplate.doInCallableStatement(CallableStatement)`.
PreparedStatementCallback	execute	Used by `JdbcTemplate` execute methods for preparing `PreparedStatement`. Usually sets the bind parameters, before the actual execution, inside the `doInPreparedStatement(PreparedStatement)`method:.
ResultSetExtractor	query	Extracts results from `ResultSet` and returns a domain object, inside the `extractData(ResultSet)`method:.

`RowCallbackHandler`	`query`	Processes each row of a `ResultSet` in a stateful manner, inside the `processRow(Resultset)`method, which doesn't return anything.
`RowMapper`	`query`	Maps each row of a `ResultSet` into a domain object, inside the `mapRow(Resultset, int rowNum)`method, returning the created domain object.

Now let's try some nice realistic usages of `JdbcTemplate`. The following is a simple method executing a count query using `JdbcTemplate`.

```
@Override
public int findAllOpenTasksCount() {
  return jdbcTemplate.queryForObject("select count(id) from tbl_user where
status = ?", new Object[]{"Open"}, Integer.class);
}
```

Do you see how this straightforward one-liner code saves you from all the boilerplate and exception-handling code you would otherwise need to write in typical JDBC code?

The following code snippet is a bit more complex and illustrates how to query a unique row from a table and map it with a domain object (`User`, in this case) using `RowMapper`:

```
public User findByUserName(String userName) {
  return jdbcTemplate.queryForObject("SELECT ID, NAME, USER_NAME, PASSWORD,
DOB, PROFILE_IMAGE_ID, PROFILE_IMAGE_NAME FROM TBL_USER WHERE USER_NAME =
?", new Object[] { userName },
    new RowMapper<User>() {
      @Override
      public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new User(rs.getLong("ID"),
        rs.getString("NAME"),
        userName,
        rs.getString("PASSWORD"),
        rs.getDate("DOB"));
    }
  });
}
```

It is so much easier to deal with collections of data using `JdbcTemplate`. The following code snippet illustrates the query method of `JdbcTemplate` with bind parameters and a `RowMapper` that converts `ResultSet` into a list of type: `<Task>`.

```
@Override
public List<Task> findCompletedTasksByAssignee(Long assigneeId) {
  String query = "SELECT * FROM TBL_TASK WHERE STATUS = ? AND
  ASSIGNEE_USER_ID = ? ";

  return this.jdbcTemplate.query(query, new Object[] {"Complete",
    assigneeId }, new RowMapper<Task>() {
    @Override
    public Task mapRow(ResultSet rs, int rowNum) throws SQLException{
      Task task = new Task();
      task.setId(rs.getLong("id"));
      Long assigneeId = rs.getLong("assignee_user_id");
      if (assigneeId != null)
        task.setAssignee(userDAO.findById(assigneeId));
      task.setComments(rs.getString("comments"));
      task.setName(rs.getString("name"));
      ...
      return task;
    }
  });
}
```

`JdbcTemplate` takes care of all the repeating code for you and you just need to write the specific code, which is about how you map the data of a row with your domain object.

Another variation of row mapping that uses a `ResultSetExtractor` interface that extracts a single row from `ResultSet` is illustrated in the following code:

```
@Transactional(readOnly = true)
public User findUserById(Long userId) {
  return jdbcTemplate.query("SELECT NAME, USER_NAME, PASSWORD, DOB,
PROFILE_IMAGE_ID, PROFILE_IMAGE_NAME FROM TBL_USER WHERE ID = ?",
    new Object[] { userId }, new ResultSetExtractor<User>() {
    @Override
    public User extractData(ResultSet rs) throws SQLException,
DataAccessException {
      if (rs.next()) {
        return new User(userId, rs.getString("NAME"),
rs.getString("USER_NAME"), rs.getString("PASSWORD"), rs.getDate("DOB"));
      } else {
        return null;
      }
    }
  });
}
```

Now let’s take a look at some update statements. The following is the execution of a simple `INSERT` statement as one-liner code. The SQL `UPDATE` and `DELETE` statements follow the same pattern.

```
@Override
public void createUser(User user) {
  jdbcTemplate.update("INSERT INTO TBL_USER(NAME, USER_NAME, PASSWORD, DOB)
VALUES(?,?,?,?)", new Object[] { user.getName(), user.getUserName(),
user.getPassword(), user.getDateOfBirth()});
}
```

The preceding method has a drawback. Although it inserts the new user record into the table, the generated ID (probably by a database sequence) is not returned back; you would need to issue another query to retrieve it separately. However, `JdbcTemplate` offers a nice way to solve this problem: using a `KeyHolder` class. It is another `variation` of the `update` method which was explained in the following code; you can retrieve the generated key (ID in this case) in a single execution, using a `KeyHolder` class in combination with `PreparedStatementCreator`:

```
public void createUser(User user) {
  KeyHolder keyHolder = new GeneratedKeyHolder();
  jdbcTemplate.update( new PreparedStatementCreator() {
    public PreparedStatement createPreparedStatement(Connection connection)
throws SQLException {
      PreparedStatement ps = connection.prepareStatement(
      "INSERT INTO TBL_USER(NAME,USER_NAME,PASSWORD,DOB) VALUES(?,?,?,?)",
new String[]{"ID"});

        ps.setString(1, user.getName());
        ps.setString(2, user.getUserName());
        ps.setString(3, user.getPassword());
        ps.setDate(4, new java.sql.Date(user.getDateOfBirth().getTime()));
        return ps;
    }
  }, keyHolder);

  user.setId(keyHolder.getKey().longValue());
}
```

`JdbcTemplate` makes batch updates easy, following the same pattern as shown earlier. Take a look at the following code: it executes a single `PreparedStatement` over a collection of data:

```
@Override
public void createUsers(List<User> users) {
    int[] updateCounts = jdbcTemplate.batchUpdate("INSERT INTO
TBL_USER(NAME, USER_NAME, PASSWORD, DOB) VALUES(?,?,?,?)", new
BatchPreparedStatementSetter() {
        public void setValues(PreparedStatement ps, int idx) throws
SQLException {
            ps.setString(1, users.get(idx).getName());
            ps.setString(2, users.get(idx).getUserName());
            ps.setString(3, users.get(idx).getPassword());
            ps.setDate(4, new java.sql.Date(users.get(idx)
.getDateOfBirth().getTime()));
        }

        public int getBatchSize() {
            return users.size();
        }
    });
}
```

NamedParameterJdbcTemplate

So far, we have used `JdbcTemplate` with bind parameters using `?` placeholders. When it comes to a bigger number of parameters, a named parameter is a better choice for readability and maintainability. `NamedParameterJdbcTemplate`, a specialized version of `JdbcTemplate`, supports using named parameters rather than traditional `?` placeholders. Instead of extending from `JdbcTemplate`, `NamedParameterJdbcTemplate` uses the underlying `JdbcTemplate` for its operations.

You can define `NamedParameterJdbcTemplate` in the same way as the classic `JdbcTemplate`, passing a `DataSource` object as a mandatory dependency. Then, you can use it just like `JdbcTemplate`, but using named parameters instead of bound parameters

(?). The following code snippet illustrates the use of the `NamedParameterJdbcTemplate` query method that uses `RowMapper` for object-relational mapping.

```
public User findByUserName(String userName, DataSource dataSource) {

  NamedParameterJdbcTemplate jdbcTemplate = new
NamedParameterJdbcTemplate(dataSource);
  SqlParameterSource namedParameters = new
MapSqlParameterSource("USER_NAME", userName);

  return jdbcTemplate.queryForObject("SELECT ID, NAME, USER_NAME, PASSWORD,
DOB, PROFILE_IMAGE_ID, PROFILE_IMAGE_NAME FROM TBL_USER WHERE USER_NAME =
:USER_NAME", namedParameters, new RowMapper<User>() {

    @Override
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
      return new User(rs.getLong("ID"), rs.getString("NAME"), userName,
rs.getString("PASSWORD"), rs.getDate("DOB"));
    }
  });
}
```

SimpleJdbc classes

`SimpleJdbc` classes are another nice approach to accessing data in a more object-oriented fashion, but still using the same `JdbcTemplate` internally. They belong to the `org.springframework.jdbc.core.simple` package. There are two classes in it:

- `SimpleJdbcCall`
- `SimpleJdbcInsert`

`SimpleJdbcCall` handles calls to stored procedures and functions and `SimpleJdbcInsert` deals with SQL `INSERT` commands to database tables. Both are `DatabaseMetadata`-aware, hence they auto-detect or map similarly named fields of domain objects. Both of them act as templates for performing JDBC operations around a relational entity (a stored procedure or function and a database table respectively), accepting parameters that determine the behavior of the operation once (declared globally), and then reusing it repeatedly with a dynamic set of data at runtime.

A `SimpleJdbcCall` class is declared as follows:

```
SimpleJdbcCall createTaskStoredProc = new SimpleJdbcCall(dataSource)
    .withFunctionName("CREATE_TASK")
    .withSchemaName("springessentials")
    .declareParameters(new SqlOutParameter("v_newID", Types.INTEGER),
        new SqlParameter("v_name", Types.VARCHAR),
        new SqlParameter("v_STATUS", Types.VARCHAR),
        new SqlParameter("v_priority", Types.INTEGER),
        new SqlParameter("v_createdUserId", Types.INTEGER),
        new SqlParameter("v_createdDate", Types.DATE),
        new SqlParameter("v_assignedUserId", Types.INTEGER),
        new SqlParameter("v_comment", Types.VARCHAR));
```

The preceding code declares `SimpleJdbcCall`, which invokes a stored procedure (in PostgreSQL, stored procedures are also called functions) and all its parameters. Once this is declared, it can be reused any number of times at runtime. Usually, you declare it at the class level (of your DAO). The following code illustrates how we invoke it at runtime:

```
@Override
public void createTask(Task task) {
    SqlParameterSource inParams = new
        MapSqlParameterSource().addValue("v_name", task.getName())
        .addValue("v_STATUS", task.getStatus())
        .addValue("v_priority", task.getPriority())
        .addValue("v_createdUserId", task.getCreatedBy().getId())
        .addValue("v_createdDate", task.getCreatedDate())
        .addValue("v_assignedUserId", task.getAssignee() == null ?
null : task.getAssignee().getId())
        .addValue("v_comment", task.getComments());

    Map<String, Object> out = createTaskStoredProc.execute(inParams);
    task.setId(Long.valueOf(out.get("v_newID").toString()));
}
```

`SimpleJdbcInsert` is typically declared as shown in the following code:

```
SimpleJdbcInsert simpleInsert = new SimpleJdbcInsert(dataSource)
  .withTableName("tbl_user")
  .usingGeneratedKeyColumns("id");
```

Note the declaration of the generated key column beside the table name in the following code snippet. Again, this is usually declared at the class level for better reuse. Now, take a look at how this is invoked at runtime.

```
public void createUser(User user) {
   Map<String, Object> parameters = new HashMap<>(4);
   parameters.put("name", user.getName());
   parameters.put("user_name", user.getUserName());
   parameters.put("password", user.getPassword());
   parameters.put("dob", user.getDateOfBirth());

   Number newId = simpleInsert.executeAndReturnKey(parameters);
   user.setId(newId.longValue());
}
```

You can see that the generated key is returned after the execution, which is set back to the `User` object. `SimpleJdbcCall` and `SimpleJdbcInsert` are convenient alternatives to the vanilla `JdbcTemplate`; you can use any of these solutions consistently or you can mix-and-match them in the same application.

JDBC operations with Sql* classes

A set of classes belonging to the `org.springframework.jdbc.object` package offers another method of performing JDBC operations in a more object-oriented manner. The following table lists the most common of them:

Component	**Responsibilities**
`MappingSqlQuery`	Concrete representation of a SQL query, supporting a `RowMapper`, and having a wide variety of convenient `execute` and `find*` methods. Supports named parameters too.
`SqlUpdate`	Executes an SQL update (`INSERT`, `UPDATE`, and `DELETE`) operation, with support for named parameters and keyholders (for retrieving generated keys).
`SqlCall`	Performs SQL-based calls for stored procedures and functions with support for named-parameters and keyholders (for retrieving generated keys).

The following code illustrates the use of `MappingSqlQuery`:

```
public Task findById(Long taskId) {
   MappingSqlQuery<Task> query = new MappingSqlQuery<Task>() {

      @Override
      protected Task mapRow(ResultSet rs, int rowNum) throws SQLException {
         return new RowMapper<Task>() {
            @Override
            public Task mapRow(ResultSet rs, int rowNum) throws
SQLException {
               Task task = new Task();
               task.setId(rs.getLong("id"));
               ...
               return task;
            }
         }.mapRow(rs, rowNum);
      }
   };

   query.setJdbcTemplate(jdbcTemplate);
   query.setSql("select id, name, status, priority, created_user_id," + "
created_date, assignee_user_id, completed_date, comments " + "from tbl_task
where id = ?");
   query.declareParameter(new SqlParameter("id", Types.INTEGER));

   return query.findObject(taskId);
}
```

SQL updates (`INSERT`, `UPDATE`, and `DELETE`) can be performed using `SqlUpdate` with a more descriptive code, as the example in the following code illustrates:

```
@Override
public void deleteTask(Task task) {
   SqlUpdate sqlUpdate = new SqlUpdate(this.jdbcTemplate.getDataSource(),
"DELETE FROM TBL_TASK WHERE ID = ?");
   sqlUpdate.declareParameter(new SqlParameter("ID", Types.NUMERIC));
```

```
    sqlUpdate.compile();
    sqlUpdate.update(task.getId());
}
```

`SqlUpdate` provides a variety of convenient update methods, suitable for many parameter combinations. You can mix-and-match any of the preceding listed Spring JDBC components according to your convenience and preferred programming style.

Spring Data

Spring Data is an umbrella project under the Spring portfolio, designed to provide consistent data access across a number of different data stores including relational and NoSQL Databases, and other types of data stores such as REST (HTTP), search engines, and Hadoop. Under Spring Data, there are subprojects for each specific approach and data store, put together by companies or developers of those technologies. Spring Data significantly simplifies the building of the data layer regardless of the underlying database and persistence technology.

The following table lists a few Spring Data subprojects with a short description of each:

Project	**Description**
Spring Data Commons	Contains a core Spring Data repository specification and supporting classes for all Spring Data projects. Specifies concepts such as repository, query, auditing, and history.
Spring Data JPA	Deals with JPA-based repositories.
Spring Data MongoDB	Provides easy integration with MongoDB, including support for query, criteria, and update DSLs.
Spring Data Redis	Integrates with the Redis in-memory data structure store, from Spring applications.
Spring Data Solr	Provides integration with Apache Solr, a powerful, open source search platform based on Apache Lucene.
Spring Data Gemfire	Provides easy integration with Pivotal Gemfire, a data management platform that provides real-time data access, reliable asynchronous event notifications, and guaranteed message delivery.
Spring Data KeyValue	Deals with key value-based data stores.
Spring Data REST	Exposes repositories with REST APIs.

The Spring Data portfolio contains community modules for more data stores that are not covered by the official Spring Data projects. Communities of several very popular open source and proprietary databases are contributing to these projects, which makes Spring Data an excellent source of proven solutions for building the data-access layer of enterprise applications regardless of the underlying data store. Cassandra, Neo4J, Couchbase, and ElasticSearch are some examples of community projects based on Spring Data.

Spring Data Commons

Spring Data standardizes data-access via all its store-specific modules (subprojects) through a consistent API called Spring Data Commons. Spring Data Commons is the foundational specification and a guideline for all Spring Data Modules. All Spring Data subprojects are store-specific implementations of Spring Data Commons.

Spring Data Commons defines the core components and general behaviors of Spring Data modules.

- Spring Data repository specification
- Query derivation methods
- Web support
- Auditing

We will examine each of these components, their setup, and usage in the following sections.

Spring Data repository specification

`org.springframework.data.repository.Repository` is the central interface of Spring Data abstraction. This marker interface is a part of Spring Data Commons and has two specialized extensions, `CrudRepository` and `PagingAndSortingRepository`.

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
    ...
}
```

A repository manages a domain entity (designed as a POJO). `CrudRepository` provides CRUD with the following CRUD operations for an entity.

- `save(One)`, `save(List)`
- `find`, `findOne`, `findAll`
- `delete`, `deleteAll`
- `count`
- `exists`

`PagingAndSortingRepository` adds pagination and sorting features over `CrudRepository`. It has the following two methods:

- `Page<T> findAll(Pageable)`
- `Iterable<T> findAll(Sort)`

Now is time to jump ahead and discuss the technology and store-specific modules of Spring Data. We are covering Spring Data JPA and Spring Data MongoDB to illustrate two totally different worlds in the database universe: relational and NoSQL. When we use a specific implementation, we use an implementation-specific repository but your method interfaces remain the same; hence, theoretically, a switch from a specific Spring Data implementation to another would not affect your client programs (service, controller, or test cases).

Spring Data JPA

Spring Data JPA is the **JPA** (**Java Persistence Architecture**)-based implementation of Spring Data, dealing with object-relational data access. For a developer, most of the programming is based on what is described in Spring Data Commons, whereas Spring Data JPA allows for some extra customizations specific to relational SQL and JPA. The main difference is in the repository setup and the query optimization using the `@Query` annotation.

Enabling Spring Data JPA

Enabling Spring Data JPA in your project is a simple two-step process:

1. Add the `spring-data-jpa` dependency to your `maven/gradle` build file.
2. Declare enable JPA repositories in your bean configuration.

In Maven, you can add a `spring-data-jpa` dependency as shown in the following code:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-jpa</artifactId>
  <version>${spring-data-jpa.version}</version>
</dependency>
```

You can enable JPA repositories, as shown in the following line, if you are using XML:

```
  <jpa:repositories base-package="com.taskify.dao" />
```

In the case of Java configuration, you just annotate to enable JPA repositories.

```
@Configuration
@ComponentScan(basePackages = {"com.taskify"})
@EnableJpaRepositories(basePackages = "com.taskify.dao")
public class JpaConfiguration {
  ...
}
```

JpaRepository

After enabling JPA repositories, Spring scans the given package for Java classes annotated with `@Repository`, and creates fully-featured proxy objects ready to be used. These are your DAO, where you just define the methods, Spring gives you proxy-based implementations at runtime. See a simple example:

```
public interface TaskDAO extends JpaRepository<Task, Long>{

  List<Task> findByAssigneeId(Long assigneeId);

  List<Task> findByAssigneeUserName(String userName);
}
```

Spring generates smart implementations that actually perform the required database operations for these methods inside the proxy implementation, looking at the method names and arguments.

Spring Data MongoDB

MongoDB is one of the most popular document-oriented NoSQL databases. It stores data in **BSON** (**Binary JSON**) format, allowing you to store an entire complex object in nested structures, avoiding the need to break data into a lot of relational tables. Its nested object structure maps directly to object-oriented data structures and eliminates the need for any object-relational mapping, as is the case with JPA/Hibernate.

Spring Data MongoDB is the Spring Data module for MongoDB. It allows Java objects to be mapped directly into MongoDB documents. It also provides a comprehensive API and infrastructural support for connecting to MongoDB and manipulating its document collections.

Enabling Spring Data MongoDB

Spring Data MongoDB can be enabled with the following steps:

1. Add `spring-data-mongodb` to your build file (`maven/gradle`).
2. Register a Mongo instance in your Spring metadata configuration.
3. Add a `mongoTemplate` Spring Bean to your Spring metadata.

Adding the `spring-data-mongodb` dependency with Maven should look like this:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${spring.framework.version}</version>
</dependency>
```

You can register a Mongo instance in your XML metadata, as shown in the following line:

```
<mongo:mongo host="192.168.36.10" port="27017" />
```

This Mongo instance is a proxy of your actual MongoDB instance running on a server.

A simplistic `mongoTemplate` looks like the listing given in the following code:

```
<bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg ref="mongo" />
  <constructor-arg name="databaseName" value="Taskify" />
</bean>
```

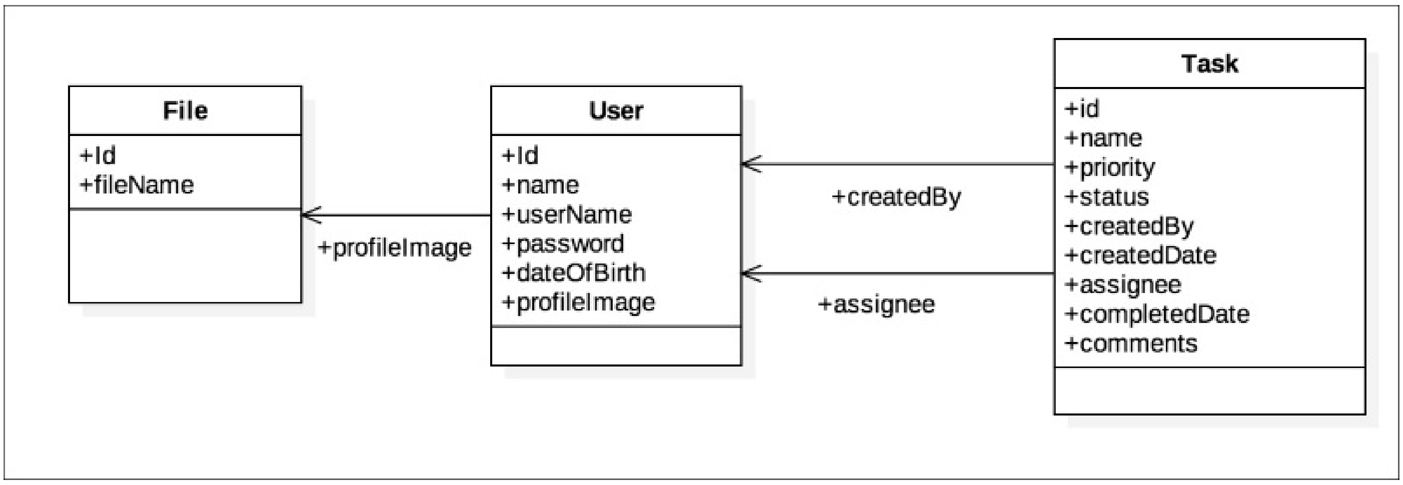
MongoRepository

MongoRepository is the MongoDB-specific repository for Spring Data MongoDB. It looks very similar to `JpaRepository`. Take a look at a sample `MongoRepository` class:

```
public interface TaskDAO extends MongoRepository<Task, String>{

  List<Task> findByAssigneeId(String assigneeId);

  @Query("{ 'status' : 'Complete' }")
  List<Task> findCompletedTasks();
  @Query(value = "{ 'status' : 'Open', assignee.id: ?0 }")
  List<Task> findOpenTasksByAssigneeId(String assigneeId);
  ...
}
```

Domain objects and entities

Data-driven applications often design domain objects as entities and then persist them into databases either as relational tables or document structures of key-value pairs at runtime. Spring Data deals with domain entities like any other persistence framework. In order to illustrate the usage of a repository, we will refer to the following three related entities, designed as **Plain Old Java Objects** (**POJOs**) in your program.



The following are the Java representations. The first one is annotated for JPA and the other two for MongoDB. JPA entities are annotated with `@Entity`. Columns are mapped against each field. Remember that, instead of annotations, you can use XML-based mapping for JPA entities too. XML mapping offers several benefits including centralized control and maintainability. This example uses annotations for simplicity, assuming that the reader is already familiar with JPA or Hibernate mappings.

```
@Entity
@Table(name = "TBL_USER", uniqueConstraints = @UniqueConstraint(name =
"UK_USER_USERNAME", columnNames = {"USER_NAME" }) )
public class User {

  @Id
  @SequenceGenerator(name = "SEQ_USER", sequenceName = "SEQ_USER",
allocationSize = 1, initialValue=1001)
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"SEQ_USER")
  private Long id;

  @Column(name = "NAME", length = 200)
  private String name;

  @Column(name = "USER_NAME", length = 25)
  private String userName;

  @Column(name = "PASSWORD", length = 20)
  private String password;

  @Column(name = "DOB")
  @Temporal(TemporalType.TIMESTAMP)
  private Date dateOfBirth;

  @ManyToOne(optional = true)
  @JoinColumn(name = "FILE_ID", referencedColumnName = "ID")
  private File profileImage;

  public User() {}

  public User(Long id, String name, String userName, String password, Date
dateOfBirth) {
    super();
    this.id = id;
    this.name = name;
    this.userName = userName;
    this.password = password;
    this.dateOfBirth = dateOfBirth;
  }

  public Long getId() {
    return id;
  }
  ...
}
```

The following is the task entity, annotated as a MongoDB document. Mongo entities are annotated with `@Document`. It requires an ID field, either annotated with `@Id` or with the name `id`.

```
@Document(collection = "tasks")
public class Task {

  @Idprivate String id;
  private String name;
  private int priority;
  private String status;
  private User createdBy;
  private Date createdDate;
  private User assignee;
  private Date completedDate;
  private String comments;

  public Task() {}
  ...
}
```

The file entity is annotated as a JPA entity.

```
@Entity
@Table(name = "TBL_FILE")
public class File {

  @Id
  @SequenceGenerator(name = "SEQ_FILE", sequenceName = "SEQ_FILE",
allocationSize = 1)
  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"SEQ_FILE")
  private Long id;

  @Column(name = "FILE_NAME", length = 200)
  private String fileName;
  ...
}
```

Query resolution methods

In addition to the declared query (`find`, `count`, `delete`, `remove`, and `exists`) methods at interface level, `CrudRepository` supports declared queries using the `@Query` annotation methods with any name, which helps to derive the actual SQL queries from the **SpEL** (**Spring Expression Language**) expression given as a parameter. Of these two query deriving options, Spring Data adopts one based on the following query lookup strategies:

Query Lookup Strategy	**Description**
`CREATE`	Generates module-specific queries from the method name.
`USE_DECLARED_QUERY`	Uses a query declared by an annotation or some other means.
`CREATE_IF_NOT_FOUND`	This strategy combines the first two. This is the default strategy.

The query lookup strategy is normally set while enabling JPA repositories.

```
<jpa:repositories base-package="com.taskify.dao" query-lookup-
strategy="create-if-not-found"/>
```

The query generation strategy (`CREATE`) works around the properties of the entity, including their dependencies, in a nested direction. As a developer, you define method names based on a specific format that can be interpreted and realized by Spring Data. The general structure of the query method is shown here:

```
[return Type] [queryType][limitKeyword]By[criteria][OrderBy][sortDirection]
```

- `return type` can be the entity `<T>` itself (in the case of a unique result), a list `<T>`, a stream `<T>`, page `<T>`, primitive numbers, Java wrapper types, void, future `<T>`, `CompletableFuture<T>`, `ListenableFuture<T>`, and so on. The last three are for Spring's asynchronous method execution and should be annotated with `@Async`.
- `queryType` can be `find`, `read`, `query`, `count`, `exists`, `delete`, and so on.
- `limitKeyword` supports `distinct`, `First[resultSize]`, and `Top[resultSize]`. An example is `First5`.
- `criteria` is built by combining one or more property expressions (using camel-casing) with standard operators such as `Or`, `And`, `Between`, `GreaterThan`, `LessThan`, `IsNull`, `StartsWith`, and `Exists`. Criteria can be suffixed by `IgnoreCase` or `AllIgnoreCase`, to apply case insensitivity.
- `OrderBy` is used as it is, suffixed by property expressions.
- `sortDirection` can be either of `Asc` or `Desc`. This is used only with `OrderBy`.

Let's see some examples for better clarity. The following sample code illustrates how to construct query (or delete) methods so that Spring Data can generate the actual SQL query at runtime.

```
public interface UserDAO extends JpaRepository<User, Long> {

  // Returns unique user with given user-name
  User findByUserName(String userName);

  // Returns a paginated list of users whose name starts with // given
value
  Page<User> findByNameStartsWith(String name, Pageable pageable);

  // Returns first 5 users whose name starts with given value,
  // order by name descending
  List<User> findTop5ByNameStartsWithOrderByNameDesc(String name);

  // Returns number of users whose birth date is before the given // value
  Long countUsersDateOfBirthLessThan(Date dob);

  // Deletes the User of given id
  void deleteById(Long userId);

  // Asynchronously returns a list of users whose name contains // the
given value
  @Async
  Future<List<User>> findByNameContains(String name);
}
```

The preceding example showing `JpaRepository` and `MongoRepository` works in the same way; you just need to extend from it, without changing the method signatures. You have seen the constraining query and filter methods traversing root-level properties of the entity, combining operators appropriately. Besides root-level properties, you can traverse and filter by nested properties as well, to define query constraints, in other words, limiting the result. Take a look at the following example:

```
public interface TaskDAO extends MongoRepository<Task, String>{

  List<Task> findByAssigneeId(Long assigneeId);

  List<Task> findByAssigneeUserName(String userName);
}
```

The methods listed in the preceding example are traversing nested properties of the task entity:

- `findByAssigneeId = task.assignee.id`
- `findByAssigneeUserName = task.assignee.userName`

You can traverse into any level of nested elements of your entity, depending on how complex your entity and requirements are.

Using the @Query annotation

Besides the autogeneration of queries based on method names as demonstrated in the previous section, Spring Data allows you to declare queries for entities locally, directly in the repository itself, over the method names. You declare the query using SpEL, and Spring Data interprets it at runtime and (the proxy repository) generates the queries for you. This is an implementation of the query resolution strategy: `USE_DECLARED_QUERY`.

Let’s take a look at some self-explanatory examples:

```
public interface TaskDAO extends JpaRepository<Task, Long>{

  @Query("select t from Task t where status = 'Open'")
  List<Task> findOpenTasks();

  @Query("select t from Task t where status = 'Complete'")
  List<Task> findCompletedTasks();

  @Query("select count(t) from Task t where status = 'Open'")
  int findAllOpenTasksCount();

  @Query("select count(t) from Task t where status = 'Complete'")
  int findAllCompletedTasksCount();

  @Query("select t from Task t where status = 'Open' and assignee.id = ?1")
  List<Task> findOpenTasksByAssigneeId(Long assigneeId);

  @Query("select t from Task t where status = 'Open' and assignee.userName
= ?1")
  List<Task> findOpenTasksByAssigneeUserName(String userName);

  @Query("select t from Task t where status = 'Complete' and assignee.id =
?1")
  List<Task> findCompletedTasksByAssigneeId(Long assigneeId);

  @Query("select t from Task t where status = 'Complete' and
assignee.userName = ?1")
  List<Task> findCompletedTasksByAssigneeUserName(String userName);
}
```

You can see from the preceding example that we can traverse into nested properties to constrain the queries, in the criteria part of it. You can also have both query generation strategies (`CREATE` and `USE_DECLARED_QUERY`) in the same repository.

The preceding example was based on Spring Data JPA; the Spring Data MongoDB equivalent is given in the following code. You can see how the `@Query` annotation values differ in comparison to the MongoDB structure.

```
public interface TaskDAO extends MongoRepository<Task, String>{

  @Query("{ 'status' : 'Open' }")
  List<Task> findOpenTasks();

  @Query("{ 'status' : 'Complete' }")
  List<Task> findCompletedTasks();

  @Query(value = "{ 'status' : 'Open' }", count = true)
  int findAllOpenTasksCount();
  @Query(value = "{ 'status' : 'Complete' }", count = true)
  int findAllCompletedTasksCount();

  @Query(value = "{ 'status' : 'Open', assignee.id: ?0 }")
  List<Task> findOpenTasksByAssigneeId(String assigneeId);

  @Query(value = "{ 'status' : 'Open', assignee.userName: ?0 }")
  List<Task> findOpenTasksByAssigneeUserName(String userName);

  @Query(value = "{ 'status' : 'Complete', assignee.id: ?0 }")
  List<Task> findCompletedTasksByAssigneeId(String assigneeId);

  @Query(value = "{ 'status' : 'Open', assignee.userName: ?0 }")
  List<Task> findCompletedTasksByAssigneeUserName(String userName);
}
```

Spring Data web support extensions

Spring Data provides a smart extension called `SpringDataWebSupport` to Spring MVC applications, integrating a few productivity components automatically if you enable it. It primarily resolves domain entities as `Pageable` and `Sort` instances with request-mapping controller methods directly from request parameters, if you are using Spring Data repository programming model for data access.

You need to enable `SpringDataWebSupport` for your project before you can use the features. You can annotate `@EnableSpringDataWebSupport`, as shown in the following code, if you are using a Java configuration:

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = {"com.taskify"})
@EnableSpringDataWebSupport
@EnableJpaRepositories(basePackages = "com.taskify.dao")
public class ApplicationConfiguration {
 ...
}
```

In the case of XML metadata, you can register `SpringDataWebConfiguration` as a Spring bean, as shown in the following code:

```
<bean
class="org.springframework.data.web.config.SpringDataWebConfiguration" />
```

Once you set up `SpringDataWebSupport`, you can start using Spring Data entities as request arguments with request-mapping methods, as shown in the following code:

```
@RestController
@RequestMapping("/api/v1/user")
@CrossOrigin
public class UserController {

  @RequestMapping(path = "/{id}", method = RequestMethod.GET)
  public User getUser(@PathVariable("id") User user) {
    return user;
  }
  ...
}
```

In the preceding method, you can see that Spring Data loads the `User` entity data using `UserRepository` transparently for you. Similarly, you can accept `Pageable` and `Sort` instances against JSON or XML post requests. Wise usage of the `SpringDataWebSupport` extension makes your code cleaner and more maintainable.

Auditing with Spring Data

Tracking data modifications is a critical feature of serious business applications. Administrators and managers are anxious to know when and who changed certain business information saved in the database. Spring Data provides smart and easy methods for auditing data entities transparently. Spring Data ships the following meaningful annotations for capturing modified user and time data entities in the system:

Annotation	**Expected type**
`@CreatedBy`	The principal user who created the entity. Typically, it is another entity that represents the domain user.
`@CreatedDate`	Records when the entity is created. Supported types: `java.util.Date`, calendar, JDK 8 date/time types, `Joda DateTime`.
`@LastModifiedBy`	The user principal who last updated the entity. It is the same type as `@CreatedBy`.
`@LastModifiedDate`	Records when the entity was last updated. Supported types are the same as for `@CreatedDate`.

A typical JPA entity should look like the following code:

```
@Entity
@Table(name = "tbl_task")
public class Task {

  @Id
  private Long id;
  ...
  @ManyToOne(optional = true)
  @JoinColumn(name = "CREATED_USER_ID", referencedColumnName = "ID")
  @CreatedBy
  private User createdBy;

  @Column(name = "CREATED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  @CreatedDate
  private Date createdDate;

  @ManyToOne(optional = true)
  @JoinColumn(name = "MODIFIED_USER_ID", referencedColumnName = "ID")
  @LastModifiedBy
  private User modifiedBy;

  @Column(name = "MODIFIED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  @LastModifiedDate
  private Date modifiedDate;
  ...
}
```

If you are using XML instead of annotations to map your entities, you can either implement an auditable interface, which forces you to implement the audit metadata fields, or extend `AbstractAuditable`, a convenient base class provided by Spring Data.

Since you are recording the information of the user who is creating and modifying entities, you need to help Spring Data to capture that user information from the context. You need to register a bean that implements `AuditAware<T>`, where `T` is the same type of field that you annotated with `@CreatedBy` and `@LastModifiedBy`. Take a look at the following example:

```
@Component
public class SpringDataAuditHelper implements AuditorAware<User> {

  ...
  @Override
  public User getCurrentAuditor() {
    // Return the current user from the context somehow.
  }

}
```

If you are using Spring Security for authentication, then the `getCurrentAuditor` method should get and return the user from the `SecurityContextHolder` class, as follows:

```
@Component
public class SpringDataAuditHelper implements AuditorAware<User> {

  ...
  @Override
  public User getCurrentAuditor() {
    Authentication authentication =
SecurityContextHolder.getContext().getAuthentication();

    if (authentication == null || !authentication.isAuthenticated()) {
      return null;
    }
    return ((User) authentication.getPrincipal()).getUser();
  }
}
```

Now your auditing infrastructure is ready, any modification you make in your auditable entities will be tracked transparently by Spring Data.

So far you have mastered the mighty Spring Data and you know how to create elegant and clean yet really powerful data access layers with Spring Data repositories, so now it is time to think about how to ensure the data integrity and reliability of your application. Spring Transaction is the answer; let’s explore it in the next section.

Spring Transaction support

Data-driven enterprise systems consider data integrity as paramount, hence transaction management is a critical feature supported by major databases and application servers. Spring framework provides comprehensive transaction support, abstracting any underlying infrastructure. Spring Transaction support includes a consistent approach across different transaction choices such as JTA, JPA, and JDO. It integrates well with all Spring data-access mechanisms. Spring Transaction supports both declarative and programmatic transaction management.

Note

A **transaction** can be defined as an atomic unit of data exchange, typically SQL statements in the case of relational databases, which should be either committed or rolled back as a block (all or nothing). A transactional system or a transaction management framework enforces **ACID** (**Atomic**, **Consistent**, **Isolated**, **Durable**) properties across the participating systems or resources (such as databases and messaging queues).

Relevance of Spring Transaction

Enterprise Java application servers natively provide **JTA** (**Java Transaction API**) support, which enables distributed transaction, which is also known as global transaction, spanning multiple resources, applications and servers. Traditionally, **Enterprise Java Beans** (**EJB**) and **Message Driven Beans** (**MDB**) were used for **container-managed transactions** (**CMT**), which is based on JTA and JNDI. JTA transaction management is resource-intensive; its exception handling is based on checked exceptions and so is not developer-friendly. Moreover, unit testing is hard with EJB CMT.

For those who do not want to use resource-intensive JTA transactions, a local transaction is another available option, and one that allows you to programmatically enforce resource-specific transactions using APIs such as JDBC. Although relatively easy to use, it is limited to a single resource, as multiple resources cannot participate in a single transaction. Moreover, local transactions are often invasive, hence they pollute your code.

Spring Transaction abstraction solves the problems of global and local transactions by providing a consistent transaction model that can run in any environment. Although it supports both declarative and programmatic transaction management, the declarative model is sufficient for most cases. Spring Transaction eliminates the need for an application server such as JBoss or WebLogic just for transactions. You can start with local transactions using Spring on a simple Servlet engine such as Tomcat and scale it up later to distributed transactions on an application server without touching your business code, just by changing the transaction manager in your Spring metadata.

Most applications just need local transactions since they do not deal with multiple servers or transactional resources such as databases, JMS, and JCA; hence, they do not need a full-blown application server. For distributed transactions spanned across multiple servers over remote calls, you need JTA, necessitating an application server, as JTA needs JNDI to look up the data source. JNDI is normally available only in an application server. Use `JTATransactionManager` inside application servers for JTA capabilities.

Note

When you deploy your Spring application inside an application server, you can use server-specific transaction managers to utilize their full features. Just switch the transaction manager to use server-specific `JtaTransactionManager` implementations such as `WebLogicJTATransactionManager` and `WebSphereUowTransactionManager` inside your Spring metadata. All your code is completely portable now.

Spring Transaction fundamentals

Spring Transaction Management abstraction is designed around an interface named `PlatformTransactionManager`, which you need to configure as a Spring bean in your Spring metadata. `PlatformTransactionManager` manages the actual transaction instance that performs the transaction operations such as commit and rollback, based on a `TransactionDefinition` instance that defines the transaction strategy. `TransactionDefinition` defines the critical transaction attributes such as isolation, propagation, transaction timeout, and the read-only status of a given transaction instance.

Note

Transaction attributes determine the behavior of transaction instances. They can be set programmatically as well as declaratively. Transaction attributes are:

Isolation level: Defines how much a transaction is isolated from (can see) other transactions running in parallel. Valid values are: `None`, `Read committed`, `Read uncommitted`, `Repeatable reads`, and `Serializable`. `Read committed` cannot see dirty reads from other transactions.

Propagation: Determines the transactional scope of a database operation in relation to other operations before, after, and nested inside itself. Valid values are: `REQUIRED`, `REQUIRES_NEW`, `NESTED`, `MANDATORY`, `SUPPORTS`, `NOT_SUPPORTED`, and `NEVER`.

Timeout: Maximum time period that a transaction can keep running or waiting before it completes. Once at timeout, it will roll back automatically.

Read-only status: You cannot save the data read in this mode.

These transaction attributes are not specific to Spring, but reflect standard transactional concepts. The `TransactionDefinition` interface specifies these attributes in the Spring Transaction Management context.

Depending on your environment (standalone, web/app server) and the persistence mechanism you use (such as plain JDBC, JPA, and Hibernate), you choose the appropriate implementation of `PlatformTransactionManager` and configure it as required, in your Spring metadata. Under the hood, using Spring AOP, Spring injects `TransactionManager` into your proxy DAO (or `EntityManager`, in the case of JPA) and executes your transactional methods, applying transaction semantics declared in your Spring configuration, either using the `@Transactional` annotation or the equivalent XML notations. We will discuss the `@Transactional` annotation and its XML equivalent later on in this chapter.

For applications that operate on a single `DataSource` object, Spring provides `DataSourceTransactionManager`. The following shows how to configure it in XML:

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="taskifyDS" />
</bean>
```

For multiple `DataSource` objects or transactional resources, you need a `JtaTransactionManager` with JTA capabilities, which usually delegates to a container JTA provider. You need to use `DataSource` objects in Java EE application servers, defined with the server, and looked up via JNDI along with `JtaTransactionManager`. A typical combination should look like the following code fragment:

```
<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />
</bean>
<jee:jndi-lookup id="taskifyDS" jndi-name="java:jboss/datasources/taskify"
expected-type="javax.sql.DataSource/>
```

If you are using Hibernate and just a single `DataSource` (and no other transactional resource), then the best option is to use `HibernateTransactionManager`, which requires you to pass the session factory as a dependency. For JPA, Spring provides `JpaTransactionManager`, which binds a single JPA `EntityManager` instance. However, it is advisable to use `JtaTransactionManager` in application-container environments.

Spring provides specialized transaction managers for application servers for WebLogic and WebSphere in order to leverage full power from container-specific transaction coordinators. Use `WebLogicTransactionManager` and `WebsphereUowTransactionManager` in the respective environments.

Declarative transaction management

Separating Transaction semantics out of your business code into an XML file or annotations above the methods is usually called **declarative transaction management**. Spring Framework allows you to apply transactional behavior into your beans transparently and non-invasively using its declarative transaction management feature.

You can apply Spring Transaction declaratively on any Spring bean, unlike EJB CMT. With Spring Transaction, you can specify transactional advices around your bean methods inside the metadata in an AOP style; then Spring will apply your those advices at runtime using AOP. You can set rollback rules to specify which exceptions around which beans or methods cause automatic rollback or non-rollback.

Transactional modes – proxy and AspectJ

Spring Transactions supports two transactional modes: proxy mode and AspectJ mode. Proxy is the default and most popular mode. In proxy mode, Spring creates an AOP proxy object, wrapping the transactional beans, and applies transactional behavior transparently around the methods using transaction aspects based on the metadata. The AOP proxy created by Spring based on transactional metadata, with the help of the configured `PlatformTransactionManager`, performs transactions around the transactional methods.

If you choose AspectJ mode for transactions, the transactional aspects are woven into the bean around the specified methods modifying the target class byte code during compile-time. There will be no proxying in this case. You will need AspectJ mode in special cases such as invoking transactional methods of the same class with different propagation levels, where proxying would not help.

Defining transactional behavior

Spring offers two convenient approaches for declaratively defining the transactional behavior of your beans:

- AOP configuration for transactions in an XML metadata file
- Using the `@Transactional` annotation

Let’s start with AOP configuration in an XML file. Refer to the *Aspect Oriented Programming* section of Chapter 1, *Getting Started with Spring Core*, for a detailed discussion of configuring AOP, using aspects, pointcuts, advice, and so on.

Typically, you declare transaction advices and pointcuts with pointcut expressions in your XML metadata file. The best approach is to keep the transaction configuration in a separate bean-definition file (for example, `transation-settings.xml`) and import it into your primary application-context file.

Typically, you declare transactional advices and other semantics as shown in the following code:

```
<!-- transactional advices -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <!-- the transactional semantics… -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="find*" read-only="true" />
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*" isolation="DEFAULT" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>

<!-- Applying the above advices to the service layer methods -->
<aop:config>
  <aop:pointcut id="allServiceMethods"
  expression="execution(* com.taskify.service.*.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut- ref="allServiceMethods" />
</aop:config>
```

You can see that this AOP configuration instructs Spring how to weave transactional advices around the methods using pointcuts. It instructs `TransactionManager` to make all find methods of the entire service layer read-only, and to force other methods to have the transaction propagation: `REQUIRED`, which means that, if the caller of the method is already in a transactional context, this method joins the same transaction without creating a new one; otherwise, a new transaction is created. If you want to create a different transaction for this method, you should use the `REQUIRES_NEW` propagation.

Also, note that the transaction isolation level is specified as `DEFAULT`, which means the default isolation of the database is to be used. Most databases default to `READ_COMMITTED`, which means a transactional thread cannot see the data of other transactions in progress (dirty reads).

Setting rollback rules

With Spring transaction, you can set rollback rules declaratively, in the same `<tx:advice>` block, as shown in the following code:

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    ...
    <tx:method name="completeTask" propagation="REQUIRED" rollback-
for="NoTaskFoundException"/>
    <tx:method name="findOpenTasksByAssignee" read-only="true" no-rollback-
for="InvalidUserException"/>
    <tx:method name="*" isolation="DEFAULT" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>
```

You can specify which exceptions should or should not rollback transactions for your business operations using the `rollback-for` and `no-rollback-for` attributes of the `<tx:method>` element.

Note

`TransactionException` thrown by the `PlatformTransactionManager` interface’s methods is the unchecked exception, `RuntimeException`. In Spring, transactions rollback for unchecked exceptions automatically. Checked, or application exceptions are not rolled back unless specified in the metadata, using the `rollback-for` attribute.

Spring Transaction allows you to customize the transactional behavior of your beans to a minute level of granularity using Spring AOP and SpEL. Moreover, you can specify the behavioral attributes of your transaction such as propagation, isolation, and timeout at the method level on the `<tx:method>` element.

Using the @Transactional annotation

The `@Transactional` annotation describes transactional attributes on a method or class. Class-level annotation applies to all methods unless explicitly annotated at method level. It supports all the attributes you otherwise set at the XML configuration. See the following example:

```
@Service
@Transactional
public class TaskServiceImpl implements TaskService {
  ...
  public Task createTask(Task task) {
    if (StringUtils.isEmpty(task.getStatus()))
      task.setStatus("Open");
    taskDAO.save(task);
    return task;
  }

  @Transactional(propagation = Propagation.REQUIRED, rollbackFor =
NoUserFoundException)
  public Task createTask(String name, int priority, Long createdByuserId,
Long assigneeUserId, String comments) {
    Task task = new Task(name, priority, "Open",
userService.findById(createdByuserId), null,
userService.findById(assigneeUserId), comments);
    taskDAO.save(task);
    logger.info("Task created: " + task);
    return task;
  }

  @Transactional(readOnly = true)
  public Task findTaskById(Long taskId) {
    return taskDAO.findOne(taskId);
  }
  ...
}
```

In the preceding example, the transactional method `createTask` with propagation `REQUIRED` rolls back for `NoUserFoundException`. Similarly, you can set no-rollback rules at the same level too.

Note

`@Transactional` can be applied only to public methods. If you want to annotate over protected, private, or package-visible methods, consider using AspectJ, which uses compile-time aspect weaving. Spring recommends annotating `@Transactional` only on concrete classes as opposed to interfaces, as it will not work in most cases such as when you use `proxy-target-class="true"` or `mode="aspectj"`.

Enabling transaction management for @Transactional

You need to first enable transaction management in your application before Spring can detect the `@Transactional` annotation for your bean methods. You enable transaction in your XML metadata using the following notation:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

The following is the Java configuration alternative for the preceding listing:

```
@Configuration
@EnableTransactionManagement
public class JpaConfiguration {
}
```

Spring scans the application context for bean methods annotated with `@Transactional` when it sees either of the preceding settings.

You can change the transaction mode from `proxy`, which is the default, to `aspectj` at this level:

```
<tx:annotation-driven transaction-manager="transactionManager"
mode="aspectj"/>
```

Another attribute you can set at this level is `proxy-target-class`, which is applicable only in the case of the `proxy` mode.

Programmatic transaction management

Spring provides comprehensive support for programmatic transaction management using two components: `TransactionTemplate` and `PlatformTransactionManager`. The following code snippet illustrates the usage of `TransactionTemplate`:

```
@Service
public class TaskServiceImpl implements TaskService {
  @Autowired
  private TransactionTemplate trxTemplate;
  ...
  public Task createTask(String name, int priority, Long createdByuserId,
Long assigneeUserId, String comments) {

    return trxTemplate.execute(new TransactionCallback<Task>() {
      @Override
      public Task doInTransaction(TransactionStatus status) {
        User createdUser = userService.findById(createdByuserId);
        User assignee = userService.findById(assigneeUserId);
        Task task = new Task(name, priority, "Open", createdUser, null,
assignee, comments);
        taskDAO.save(task);
        logger.info("Task created: " + task);
        return task;
      }
    });
  }
}
```

`TransactionTemplate` supports the setting of all transaction attributes, as in the case of XML configuration, which gives you more granular control at the expense of mixing your business code with transactional concerns. Use it only if you need absolute control over a particular feature that cannot be achieved with declarative transaction management. Use declarative transaction management if possible, for better maintainability and management of your application.

Summary

We have so far explored Spring Framework’s comprehensive coverage of all technical aspects around data access and transaction. Spring provides multiple convenient data access methods, which removes much of the hard work for the developer involved in building the data layer and standardizing the business components. The correct usage of Spring data access components makes the data layer of the Spring application clean and highly maintainable. Leveraging Spring Transaction support ensures the data integrity of applications without polluting the business code and makes your application portable across different server environments. Since Spring abstracts much of the technical heavy lifting, building the data layer of your applications becomes an enjoyable piece of software engineering.

Chapter 4. Understanding WebSocket

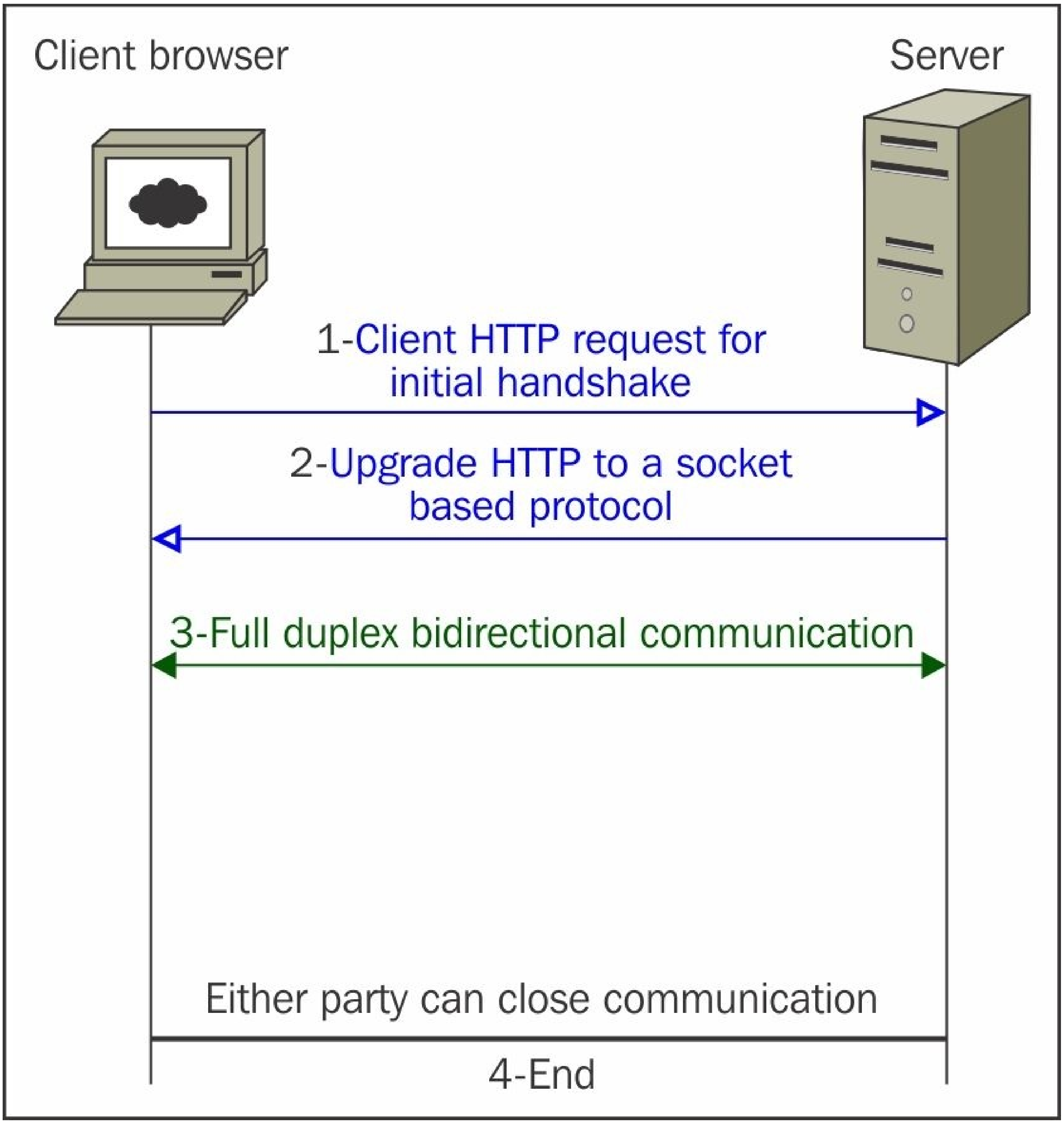
The idea of web applications was built upon a simple paradigm. In a unidirectional interaction, a web client sent a request to a server, the server replied to the request, and the client rendered the server’s response. The communication started with a client-side request and ended with the server’s response.

We built our web applications based on this paradigm; however, some drawbacks existed in the technology: the client had to wait for the server’s response and refresh the browser to render it. This unidirectional nature of the communication required the client to initiate a request. Later technologies such as AJAX and long polling brought major advantages to our web applications. In AJAX, the client initiated a request but did not wait for the server’s response. In an asynchronous manner, the AJAX client-side callback method got the data from the server and the browsers’ new DHTML features rendered the data without refreshing the browser.

Apart from unidirectional behavior, the HTTP dependencies of these technologies required the exchange of extra data in the form of HTTPS headers and cookies. This extra data caused latency and became a bottleneck for highly responsive web applications.

WebSocket reduced kilobytes of transmitted data to a few bytes and reduced latency from 150 milliseconds to 50 milliseconds (for a message packet plus the TCP round trip to establish the connection), and these two factors attracted the Google’s attention (Ian Hickson).

WebSocket (RFC 6455) is a full duplex and bidirectional protocol that transmits data in the form of frames between client and server. A WebSocket communication, as shown in the following figure, starts with an HTTP connection for a handshake process between a client and a server. Since firewalls let certain ports be open to communicate with the outside, we cannot start with the WebSocket protocol:


Client browser
Server
1-Client HTTP request for initial handshake
2-Upgrade HTTP to a socket based protocol
3-Full duplex bidirectional communication
Either party can close communication
4-End

WebSocket communication

During the handshake process, the parties (client and server) decide which socket-based protocol to choose for transmitting data. At this stage, the server can validate the user using HTTP cookies and reject the connection if authentication or authorization fails.

Then, both parties upgrade from HTTP to a socket-based protocol. From this point onward, the server and client communicate on a full duplex and bidirectional channel on a TCP connection.

Either the client or server can send messages by streaming them into frame format. WebSocket uses the heartbeat mechanism using ping/pong message frames to keep the connection alive. This looks like sending a ping message from one party and expecting a pong from the other side. Either party can also close the channel and terminate the communication, as shown in the preceding diagram.

Like a web URI relies on HTTP or HTTPS, WebSocket URI uses `ws` or `wss` schemes (for example, `ws://www.sample.org/` or `wss://www.sample.org/`) to communicate. WebSocket's `ws` works in a similar way to HTTP by transmitting non-encrypted data over TCP/IP. By contrast, `wss` relies on **Transport Layer Security** (**TLS**) over TCP, and this combination brings data security and integrity.

A good question is where to use WebSocket. The best answer is to use it where low latency and high frequency of communication are critical—for example, if your endpoint data changes within 100 milliseconds and you expect to take very quick measures over the data changes.

Spring Framework 4 includes a new Spring WebSocket module with Java WebSocket API standard (JSR-356) compatibility as well as some additional value-adding features.

While using WebSocket brings advantages to a web application, a lack of compatibility in a version of some browser blocks WebSocket communication. To address this issue, Spring 4 includes a fallback option that simulates the WebSocket API in case of browser incompatibility.

WebSocket transmits data in the frame format, and apart from a single bit to distinguish between text and binary data, it is neutral to the message's content. In order to handle the message's format, the message needs some extra metadata, and the client and server should agree on an application-layer protocol, known as a **subprotocol**. The parties choose the subprotocol during the initial handshake.

WebSocket does not mandate the usage of subprotocols, but in the case of their absence, both the client and server need to transmit data in a predefined style standard, framework-specific, or customized format.

Spring supports **Simple Text Orientated Messaging Protocol** (**STOMP**) as a subprotocol —known as STOMP over WebSocket—in a WebSocket communication. Spring's Messaging is built upon integration concepts such as messaging and channel and handler, along with annotation of message mapping. Using STOMP over WebSocket gives message-based features to a Spring WebSocket application.

Using all of these new Spring 4 features, you can create a WebSocket application and broadcast a message to all subscribed clients as well as send a message to a specific user. In this chapter, we start by creating a simple Spring web application, which will show how to set up a WebSocket application and how a client can send and receive messages to or from an endpoint. In the second application, we will see how Spring WebSocket's fallback option can tackle browser incompatibly, how a broker based messaging system works with STOMP over WebSocket, and how subscribed clients can send and receive messages. In the last web application, however, we will show how we can send broker-based messages to a specific user.

Creating a simple WebSocket application

In this section, while developing a simple WebSocket application, we will learn about WebSocket's client and server components. As mentioned earlier, using a subprotocol is optional in a WebSocket communication. In this application, we have not used a subprotocol.

First of all, you need to set up a Spring web application. In order to dispatch a request to your service (called a handler in Spring WebSocket), you need to set up a framework Servlet (dispatcher Servlet). This means that you should register `DispatcherServlet` in `web.xml` and define your beans and service in the application context.

Setting up a Spring application requires you to configure it in XML format. Spring introduced the Spring Boot module to get rid of XML configuration files in Spring applications. Spring Boot aims at configuring a Spring application by adding a few lines of annotation to the classes and tagging them as Spring artifacts (bean, services, configurations, and so on). By default, it also adds dependencies based on what it finds in the classpath. For example, if you have a web dependency, then Spring Boot can configure Spring MVC by default. It also lets you override this default behavior. Covering Spring Boot in complete detail would require a full book; we will just use it here to ease the configuration of a Spring application.

These are the Maven dependencies of this project:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.5.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-websocket</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-messaging</artifactId>
    </dependency>
            <dependency>
        <groupId>org.json</groupId>
        <artifactId>json</artifactId>
        <version>20140107</version>
    </dependency>
</dependencies>
<properties>
    <java.version>1.8</java.version>
</properties>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

As mentioned in the beginning of this section, there is no subprotocol (and, subsequently, no application-layer framework) to interpret WebSocket messages. This means that the client and server need to handle the job and be aware of the message's format.

On the server's side, the handler (endpoint) receives and extracts the message and replies back (based on the business logic) to the client. In Spring, you can create a customized handler by extending either `TextWebSocketHandler` or `BinaryWebSocketHandler`. `TextWebSocketHandler` handles string or text messages (such as JSON data) and `BinaryWebSocketHandler` handles binary messages (such as image or media data). Here is a code listing that uses `TextWebSocketHandler`:

```
public class SampleTextWebSocketHandler extends TextWebSocketHandler {
   @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage
message) throws Exception {
        String payload = message.getPayload();
        JSONObject jsonObject = new JSONObject(payload);
        StringBuilder builder=new StringBuilder();
        builder.append("From Myserver-").append("Your
Message:").append(jsonObject.get("clientMessage"));
        session.sendMessage(new TextMessage(builder.toString()));
   }
}
```

Since we process only JSON data here, the class `SampleTextWebSocketHandler` extends `TextWebSocketHandler`. The method `handleTextMessage` obtains the client's message by receiving its payload and converting it into JSON data, and then it sends a message back to the client.

In order to tell Spring to forward client requests to the endpoint (or handler here), we need to register the handler:

```
@Configuration
@EnableWebSocket
public class SampleEhoWebSocketConfigurer {
    @Bean
    WebSocketConfigurer webSocketConfigurer(final WebSocketHandler
webSocketHandler) {
        return new WebSocketConfigurer() {
            @Override
            public void registerWebSocketHandlers(WebSocketHandlerRegistry
registry) {
                registry.addHandler(new SampleTextWebSocketHandler(),
"/path/wsAddress");
            }
        };
    }
    @Bean
    WebSocketHandler myWebsocketHandler() {
        return new SampleTextWebSocketHandler();
    }
```

`@Configuration` and `@EnableWebsocket` tell Spring this is the WebSocket configurator of the project. It registers our handler (`SampleTextWebSocketHandler`) and sets the request path (in a WebSocket URL, such as `ws://server-ip:port/path/wsAddress`) that will be forwarded to this handler.

And now the question is how to set up a Spring application and glue all of this stuff together. Spring Boot provides an easy way to set up a Spring-based application with a configurable embedded web server that you can "just run":

```
package com.springessentialsbook.chapter4;
...
@SpringBootApplication
public class EchoWebSocketBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(EchoWebSocketBootApplication
        .class, args);
    }
}
```

`@SpringBootApplication` tags the `EchoWebSocketBootApplication` class as a special configuration class of your application and `@SpringBootApplication` behaves like the following annotations:

- `@Configuration`, which declares the class as a bean definition of an application context
- `@EnableAutoConfiguration`, which lets Spring Boot add a dependent bean definition based on the classpath (for example, `spring-webmvc` in the project classpath tells Spring Boot to set up a web application with its `DispatcherServlet` registration in `web.xml`)
- `@ComponentScan`, which is used to scan all annotations (services, controllers, configurations, and so on) within the same package (`com.springessentialsbook.chapter4`) and configure them accordingly

Finally, the `main` method calls `SpringApplication.run` to set up a Spring application within a web application without writing a single line of XML configuration (`applicationContext.xml` or `web.xml`).

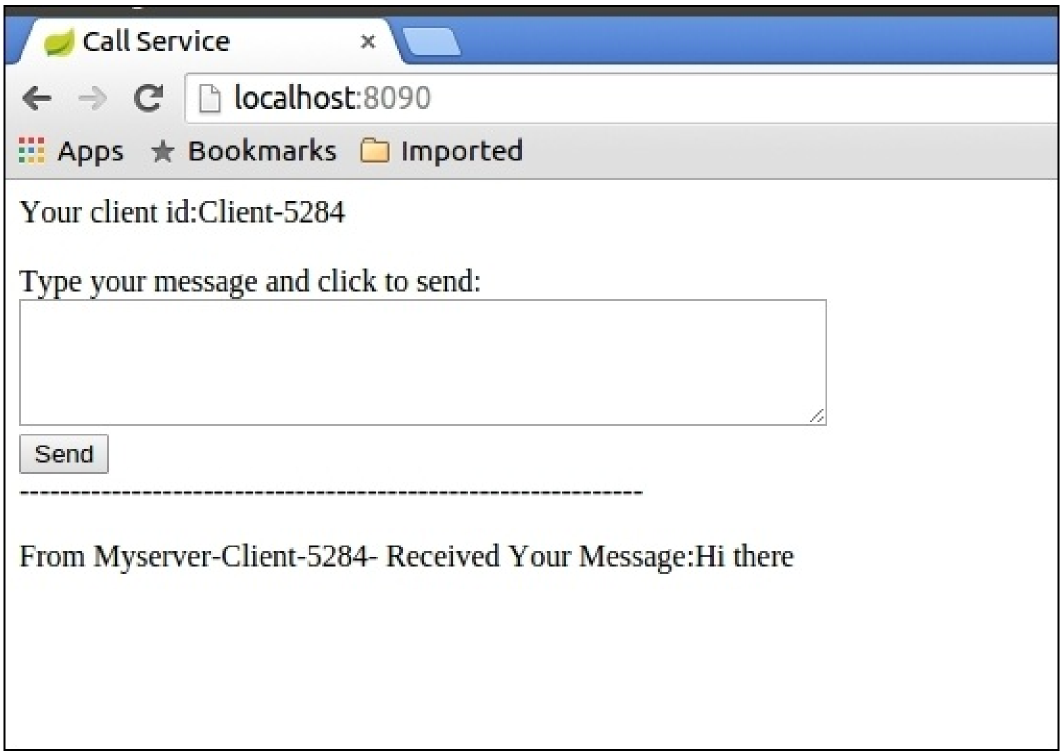
When a client wants to send a WebSocket request, it should create a JavaScript client object (`ws = new WebSocket('ws://localhost:8090/path/wsAddress')`) and pass the WebSocket service address. In order to receive the data, we need to attach a callback listener (`ws.onmessage`) and an error handler (`ws.onerror`), like so:

```
    function openWebSocket(){
        ws = new WebSocket( 'ws://localhost:8090/path/wsAddress');
        ws.onmessage = function(event){
            renderServerReturnedData(event.data);
        };

        ws.onerror = function(event){
            $('#errDiv').html(event);
        };
    }

    function sendMyClientMessage() {
        var myText = document.getElementById('myText').value;
        var message=JSON.stringify({ 'clientName': 'Client-'+randomnumber,
'clientMessage':myText});
        ws.send(message);
        document.getElementById('myText').value='';
    }
```

You can run the application by running this command:

```
mvn spring-boot:run -Dserver.port=8090
```

This runs and deploys the web application on an embedded server on port `8090` (`8080` is not used here as it may conflict with your running Apache service). So, the index page of the application will be accessible at `http://localhost:8090/` (follow the instructions in `read-me.txt` to run the application). It should look like this:



Your client id:Client-5284

Type your message and click to send:

From Myserver-Client-5284- Received Your Message:Hi there

The opening page of the application in a Chrome browser

When a user sends a text in Chrome, it will be handled by `SampleTextWebSocketHandler`, the handler will reply, and the response will be rendered in the browser.

If you try to test this application in a version of Internet Explorer lower than 10, you will get a JavaScript error.

As we discussed earlier, certain versions of browsers do not support WebSocket. Spring 4 provides a fallback option to manage these types of browsers. In the next section, this feature of Spring will be explained.

STOMP over WebSocket and the fallback option in Spring 4

In the previous section, we saw that in a WebSocket application that does not use subprotocols, the client and server should be aware of the message format (JSON in this case) in order to handle it. In this section, we use STOMP as a subprotocol in a WebSocket application (this is known as **STOMP over WebSocket**) and show how this application layer protocol helps us handle messages.

The messaging architecture in the previous application was an asynchronous client/server-based communication.

The `spring-messaging` module brings features of asynchronous messaging systems to Spring Framework. It is based on some concepts inherited from Spring Integration, such as messages, message handlers (classes that handle messages), and message channels (data channels between senders and receivers that provide loose coupling during communication).

At the end of this section, we will explain how our Spring WebSocket application integrates with the Spring messaging system and works in a similar way to legacy messaging systems such as JMS.

In the first application, we saw that in certain types of browsers, WebSocket communication failed because of browser incompatibility. In this section, we will explain how Spring's fallback option addresses this problem.

Suppose you are asked to develop a browser-based chat room application in which anonymous users can join a chat room and any text sent by a user should be sent to all active users. This means that we need a topic that all users should be subscribed to and messages sent by any user should be broadcasted to all. Spring WebSocket features meet these requirements. In Spring, using STOMP over WebSocket, users can exchange messages in a similar way to JMS. In this section, we will develop a chat room application and explain some of Spring WebSocket's features.

The first task is to configure Spring to handle STOMP messages over WebSocket. Using Spring 4, you can instantly configure a very simple, lightweight (memory-based) message broker, set up subscription, and let controller methods serve client messages. The code for the `ChatroomWebSocketMessageBrokerConfigurer` class is:

```
package com.springessentialsbook.chapter4;
…..
@Configuration
@EnableWebSocketMessageBroker
public class ChatroomWebSocketMessageBrokerConfigurer extends
AbstractWebSocketMessageBrokerConfigurer {
   @Override
   public void configureMessageBroker(MessageBrokerRegistry config) {
      config.enableSimpleBroker("/chatroomTopic");
      config.setApplicationDestinationPrefixes("/myApp");
   }
   @Override
   public void registerStompEndpoints(StompEndpointRegistry registry) {
      registry.addEndpoint("/broadcastMyMessage").withSockJS();
   }
}
```

`@Configuration` tags a `ChatroomWebSocketMessageBrokerConfigurer` class as a Spring configuration class. `@EnableWebSocketMessageBroker` provides WebSocket messaging features backed by a message broker.

The overridden method `configureMessageBroker`, as its name suggests, overrides the parent method for message broker configuration and sets:

- `setApplicationDestinationPrefixes`: Specify `/myApp` as the prefix, and any client message whose destination starts with `/myApp` will be routed to the controller's message-handling methods.
- `enableSimpleBroker`: Set the broker topic to `/chatroomTopic`. Any messages whose destinations start with `/chatroomTopic` will be routed to the message broker (that is, broadcasted to other connected clients). Since we are using an in-memory broker, we can specify any topic. If we use a dedicated broker, the destination's name would be `/topic` or `/queue`, based on the subscription model (pub/sub or point-to-point).

The overridden method `registerStompEndpoints` is used to set the endpoint and fallback options. Let's look at it closely:

- The client-side WebSocket can connect to the server's endpoint at `/broadcastMyMessage`. Since STOMP has been selected as the subprotocol, we do not need to know about the underlying message format and let STOMP handle it.
- The `.withSockJS()` method enables Spring's fallback option. This guarantees successful WebSocket communication in any type or version of browser.

As Spring MVC forwards HTTP requests to methods in controllers, the MVC extension can receive STOMP messages over WebSocket and forward them to controller methods. A Spring `Controller` class can receive client STOMP messages whose destinations start with `/myApp`. The handler method can reply to subscribed clients by sending the returned message to the broker channel, and the broker replies to the client by sending the message to the response channel. At the end of this section, we will look at some more information about the messaging architecture. As an example, let's look at the `ChatroomController` class:

```
    package com.springessentialsbook.chapter4;
      ...
@Controller
public class ChatroomController {

    @MessageMapping("/broadcastMyMessage")
    @SendTo("/chatroomTopic/broadcastClientsMessages")
    public ReturnedDataModelBean broadCastClientMessage(ClientInfoBean
message) throws Exception {
        String returnedMessage=message.getClientName() +
":"+message.getClientMessage();
```

```
        return new ReturnedDataModelBean(returnedMessage );
    }
}
```

Here, `@Controller` tags `ChatroomController` as an MVC workflow controller. `@MessageMapping` is used to tell the controller to map the client message to the handler method (`broadCastClientMessage`). This will be done by matching a message endpoint to the destination (`/broadcastMyMessage`). The method's returned object (`ReturnedDataModelBean`) will be sent back through the broker to the subscriber's topic (`/chatroomTopic/broadcastClientsMessages`) by the `@SendTo` annotation. Any message in the topic will be broadcast to all subscribers (clients). Note that clients do not wait for the response, since they send and listen to messages to and from the topic and not the service directly.

Our domain POJOs (`ClientInfoBean` and `ReturnedDataModelBean`), detailed as follows, will provide the communication message payloads (actual message content) between the client and server:

```
package com.springessentialsbook.chapter4;
public class ClientInfoBean {
    private String clientName;
    private String clientMessage;
    public String getClientMessage() {
    return clientMessage;
  }
    public String getClientName() {
        return clientName;
    }
}


package com.springessentialsbook.chapter4;
public class ReturnedDataModelBean {

    private String returnedMessage;
    public ReturnedDataModelBean(String returnedMessage) {
        this.returnedMessage = returnedMessage; }
    public String getReturnedMessage() {
        return returnedMessage;
    }
}
```

To add some sort of security, we can add basic HTTP authentication, as follows (we are not going to explain Spring security in this chapter, but it will be detailed in the next chapter):

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
    }
    @Autowired
    void configureGlobal(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication()
        .withUser("user").password("password").roles("USER");
    }
}
```

The `@Configuration` tags this class as a configuration class and `@EnableGlobalMethodSecurity` and `@EnableWebSecurity` set security methods and web security in the class. In the `configure` method, we set basic authentication, and in `configureGlobal`, we set the recognized username and password as well as the role that the user belongs to.

To add Spring Security features, we should add the following Maven dependencies:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-messaging</artifactId>
    <version>4.0.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
```

As we explained in the previous section, the `@SpringBootApplication` tag sets up a Spring application within a web application without us having to write a single line of XML configuration (`applicationContext.xml` or `web.xml`):

```
package com.springessentialsbook.chapter4;
...
@SpringBootApplication
public class ChatroomBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(ChatroomBootApplication.class, args);
    }
}
```

Finally, you can run the application by running this command:

```
mvn spring-boot:run -Dserver.port=8090
```

This runs and deploys the web application on an embedded web server on port `8090` (`8080` is not used as it may conflict with your running Apache service). So, the index page of the application will be accessible at `http://localhost:8090/` (follow `read-me.txt` to run the application):

```
    <script src="sockjs-0.3.4.js"></script>
    <script src="stomp.js"></script>
    <script type="text/javascript">
...
function joinChatroom() {
    var topic='/chatroomTopic/broadcastClientsMessages';
    var servicePath='/broadcastMyMessage';
    var socket = new SockJS(servicePath);
    stompClient = Stomp.over(socket);
    stompClient.connect('user','password', function(frame) {
        setIsJoined(true);
        console.log('Joined Chatroom: ' + frame);
        stompClient.subscribe(topic, function(serverReturnedData){

renderServerReturnedData(JSON.parse(serverReturnedData.body).returnedMessag
e);
        });
    });
}
...
function sendMyClientMessage() {
    var serviceFullPath='/myApp/broadcastMyMessage';
    var myText = document.getElementById('myText').value;
    stompClient.send(serviceFullPath, {}, JSON.stringify({ 'clientName':
'Client-'+randomnumber, 'clientMessage':myText}));
    document.getElementById('myText').value='';
}
```

On the client side, notice how the browser connects (with `joinChatRoom`) and sends data (in the `sendMyClientMessage` method). These methods use the JavaScript libraries SockJS and Stomp.js.

As you can see, when a client subscribes to a topic, it registers a listener method (`stompClient.subscribe(topic, function(serverReturnedData){.…}`). The listener method will be called when any message (from any client) arrives in the topic.

As discussed earlier, some versions of browsers do not support WebSocket. SockJS was introduced to handle all versions of browsers. On the client side, when you try to connect to the server, the SockJS client sends the `GET/info` message to get some information from the server. Then it chooses the transport protocol, which could be one of WebSocket, HTTP streaming, or HTTP long-polling. WebSocket is the preferred transport protocol; however, in case of browser incompatibility, it chooses HTTP streaming, and in the worse case, HTTP long-polling.

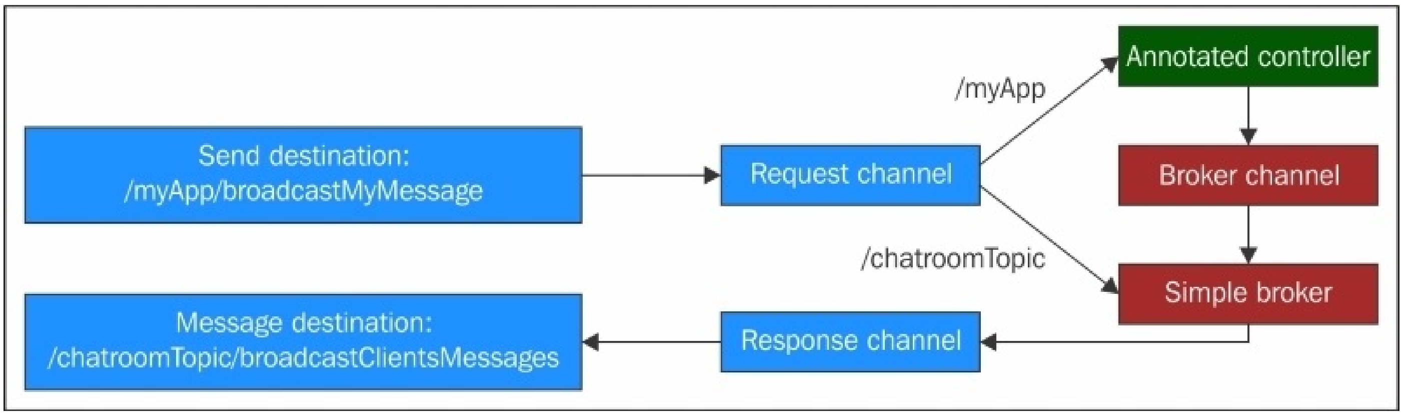
In the beginning of this section, we described how our WebSocket application integrates with the Spring messaging system and works in a way similar to legacy messaging systems.

The overridden method settings of `@EnableWebSocketMessageBroker` and `ChatroomWebSocketMessageBrokerConfigurer` create a concrete message flow (refer to the following diagram). In our messaging architecture, channels decouple receivers and senders. The messaging architecture contains three channels:

- The client inbound channel (**Request channel**) for request messages sent from the client side

- The client outbound channel (**Response channel**) for messages sent to the client side
- The **Broker channel** for internal server messages to the broker

Our system uses STOMP destinations for simple routing by prefix. Any client message whose destination starts with `/myApp` will be routed to controller message-handling methods. Any message whose destination starts with `/chatroomTopic` will be routed to the message broker.


Annotated controller
/myApp
Send destination: /myApp/broadcastMyMessage
Request channel
Broker channel
/chatroomTopic
Simple broker
Message destination: /chatroomTopic/broadcastClientsMessages
Response channel

The simple broker (in-memory) messaging architecture

Here is the messaging flow of our application:

1. The client connects to the WebSocket endpoint (`/broadcastMyMessage`).
2. Client messages to `/myApp/broadcastMyMessage` will be forwarded to the `ChatroomController` class (through the **Request channel**). The mapping controller's method passes the returned value to the Broker channel for the topic `/chatroomTopic/broadcastClientsMessages`.
3. The broker passes the message to the **Response channel**, which is the topic `/chatroomTopic/broadcastClientsMessages`, and clients subscribed to this topic receive the message.

Broadcasting a message to a single user in a WebSocket application

In the previous section, we saw a WebSocket application of the multiple subscriber model, in which a broker sent messages to a topic. Since all clients had subscribed to the same topic, all of them received messages. Now, you are asked to develop an application that targets a specific user in a WebSocket chat application.

Suppose you want to develop an automated answering application in which a user sends a question to the system and gets an answer automatically. The application is almost the same as the previous one (STOMP over WebSocket and the fallback option in Spring 4), except that we should change the WebSocket configurer and endpoint on the server side and subscription on the client side. The code for the `AutoAnsweringWebSocketMessageBrokerConfigurer` class is:

```
@Configuration
@EnableWebSocketMessageBroker
public class AutoAnsweringWebSocketMessageBrokerConfigurer extends
AbstractWebSocketMessageBrokerConfigurer {
   @Override
   public void configureMessageBroker(MessageBrokerRegistry config) {
      config.setApplicationDestinationPrefixes("/app");
      config.enableSimpleBroker("/queue");
      config.setUserDestinationPrefix("/user");
   }
   @Override
   public void registerStompEndpoints(StompEndpointRegistry registry) {
      registry.addEndpoint("/message").withSockJS();
   }
}
```

The `config.setUserDestinationPrefix("/user")` method sets a prefix noting that a user has subscribed and expects to get their own message on the topic. The code for the `AutoAnsweringController` class is:

```
@Controller
public class AutoAnsweringController {
    @Autowired
    AutoAnsweringService autoAnsweringService;
    @MessageMapping("/message")
    @SendToUser
    public String sendMessage(ClientInfoBean message) {
        return autoAnsweringService.answer(message);
    }
    @MessageExceptionHandler
    @SendToUser(value = "/queue/errors", broadcast = false)
    String handleException(Exception e) {
        return "caught ${e.message}";
    }
}
```

```
@Service
public class AutoAnsweringServiceImpl implements AutoAnsweringService {
    @Override
    public String answer(ClientInfoBean bean) {
        StringBuilder mockBuffer=new StringBuilder();
        mockBuffer.append(bean.getClientName())
                .append(", we have received the message:")
                .append(bean.getClientMessage());
        return mockBuffer.toString();
    }
}
```

In the endpoint, we use `@SendToUser` instead of `@SendTo("...")`. This forwards the response only to the sender of the message. `@MessageExceptionHandler` will send errors `(broadcast = false)` to the sender of message as well.

`AutoAnsweringService` is just a mock service to return an answer to the client message. On the client side, we only add the `/user` prefix when a user subscribes to the topic (`/user/queue/message`):

```
function connectService() {
    var servicePath='/message';
    var socket = new SockJS(servicePath);
    stompClient = Stomp.over(socket);
    stompClient.connect({}, function(frame) {

        setIsJoined(true);
        stompClient.subscribe('/user/queue/message', function(message) {
            renderServerReturnedData(message.body);
        });
        stompClient.subscribe('/user/queue/error', function(message) {
            renderReturnedError(message.body);
        });
    });
}
function sendMyClientMessage() {
    var serviceFullPath='/app/message';
    var myText = document.getElementById('myText').value;
    stompClient.send(serviceFullPath, {}, JSON.stringify({ 'clientName':
'Client-'+randomnumber, 'clientMessage':myText}));
    document.getElementById('myText').value='';
}
```

The topic `user/queue/error` is used to receive errors dispatched from the server side.

Note

For more about Spring's WebSocket support, go to http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/websocket.html.

For more about WebSocket communication, refer to *Chapter 8, Replacing HTTP with WebSockets* from the book *Enterprise Web Development, Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, Viktor Gamov, O'Reilly*.

Summary

In this chapter, we explained WebSocket-based communication, how Spring 4 has been upgraded to support WebSocket, and the fallback option to overcome browsers' WebSocket incompatibility. We also had a small sample of adding basic HTTP authentication, which is a part of Spring Security. We will discuss more on security in Chapter 5, *Securing Your Applications*.

Chapter 5. Securing Your Applications

Spring Security provides a wide range of features for securing Java/Spring-based enterprise applications. At first glance, the security features of Servlets or EJB look an alternative of Spring Security; however, these solutions lack certain requirements for developing enterprise applications. The server's environment dependency could be another drawback of these solutions.

Authentication and authorization are the main areas of application security. Authentication is the verification of a user's identity, whereas authorization is the verification of the privileges of a user.

Spring Security integrates with a variety of authentication models, most of which are provided by third-party providers. In addition, Spring Security has developed its own authentication models, based upon major security protocols. Here are some of these protocols:

- Form-based authentication
- HTTP Basic authentication
- LDAP
- JAAS
- Java Open Single Sign On
- Open ID authentication

Since there is a big list of Spring Security models, we can only detail the most popular of them in this chapter.

Spring Security is quite strong on authorization features. We can categorize these features into three groups: web, method, and domain object authorization. Later, in the *Authorization* section, we will explain these categories.

In order to use Spring Security features in a web application, you need to include the following dependencies in your project:

```
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-web</artifactId>
   <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-config</artifactId>
   <version>4.0.2.RELEASE</version>
</dependency>
```

The **open standard for authorization** (**OAuth**) concept, introduced in late 2006, aimed to allow third-party limited access to users' resources on Microsoft, Google, Facebook, Twitter, or similar accounts, without sharing their usernames and passwords.

In 2010, OAuth was standardized as the OAuth 1.0a protocol in RFC 5849. Later in 2012, it evolved to the OAuth 2.0 framework in RFC 6749. In this chapter, we explain Spring's

OAuth 2.0 framework implementation.

The OAuth 2.0 Authorization Framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf (http://tools.ietf.org/html/rfc6749).

Spring provides a separate module (`spring-security-oauth2`) for its OAuth 2.0 implementation, which relies on Spring Security features. In this chapter, we explain authentication and how Spring facilitates the process by providing its own easy-to-use features as well as giving you options to plug in your customized implementation. Authorization is the second topic included in this chapter, in which we explain how to configure separate security models within the same application. In the last section, we explain Spring's OAuth 2.0 feature.

Authentication

In an application's security domain, the first thing that comes to mind is authentication. During the authentication process, an application compares a user's credentials (for example, a username and password or a token) with the information available to it. If these two match, it allows the process to enter the next step. We will follow the next step in the *Authorization* section.

Spring Security provides features to support a variety of security authentication protocols. In this section, we will focus on basis and form-based authentication.

Spring provides a built-in form for the purpose of form-based authentication. In addition, it lets you define your own customized login form.

Spring gives you the option to use in-memory authentication, in which the username and password will be hardcoded in the application.

An alternative option is to use a customized authentication provider that lets you decide how to authenticate users by program, for example, calling a data layer service to validate users. It also lets you integrate Spring Security with your existing security framework.

The first thing you need in order to configure Spring Security to authenticate users is to define a Servlet filter known as `springSecurityFilterChain`. This filter is responsible for applying security measures (for example, validating users, navigating to different pages after login bases on the user's role, and protecting application URLs) in a web application.

`WebSecurityConfigurerAdapter` is a convenient Spring template for configuring `springSecurityFilterChain`:

```
@Configuration
@EnableWebSecurity
@ComponentScan(basePackages = "com.springessentialsbook.chapter5")
public class WebSecurityConfigurator extends WebSecurityConfigurerAdapter {
    @Autowired
    private AuthenticationSuccessHandler authenticationSuccessHandler;

    @Autowired
    public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {

auth.inMemoryAuthentication().withUser("operator").password("password").rol
es("USER");

auth.inMemoryAuthentication().withUser("admin").password("password").roles(
"ADMIN");

auth.inMemoryAuthentication().withUser("accountant").password("password").r
oles("ACCOUNTANT");

    }
```

`@Configuration` registers this class as a configuration class. The method's name, `configureGlobalSecurity`, is not important, as it only configures an `AuthenticationManagerBuilder` instance through autowire. The only important thing is annotating the class with `@EnableWebSecurity`, which registers Spring web security in the application. As you can see, we used in-memory authentication for simplicity, which hardcoded the user's username, password, and role used for user authentication. In real enterprise applications, LDAP, databases or the cloud provide services for validating user credentials.

We don't code all that much in the config class, but it really does a lot behind the scenes. Here are some of the features implemented by the class. Apart from user authentication and role assignment, we will explain other features next in this chapter.

- Protecting all application URLs by asking for authentication first
- Creating a Spring default login form to authenticate the user
- Authenticating users (operator/password, admin/password, accountant/password) and assigning separate roles for each user (user, admin, and accountant) using form-based authentication
- Allowing the user to log out
- CSRF attack prevention

As we explained, in real-world enterprise applications, one never hardcodes user credentials within the application's code. You may have an existing security framework that calls a service in order to validate users. In this case, you can configure Spring Security in a customized service to authenticate the user.

The authentication interface implementation is what carries user credentials within the Spring Security context. You can obtain the authentication object anywhere within the application using `SecurityContextHolder.getContext().getAuthentication()`.

When a user is authenticated, `Authentication` will be populated. If you don't specify `AuthenticationProvider` (for example, if you use in-memory authentication), `Authentication` will be populated automatically. Here, we look at how to customize `AuthenticationProvider` and populate the `Authentication` object.

The following code shows how Spring's `AuthenticationProvider` implementation class integrates with a customized user detail service (which returns user credentials from a data source):

```
@Component
public class MyCustomizedAuthenticationProvider implements
AuthenticationProvider {
  @Autowired
  UserDetailsService userService;
  public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
    User user=null;
    Authentication auth=null;
    String username=authentication.getName();
    String password=authentication.getCredentials().toString();
    user= (User) userService.loadUserByUsername(username);
    if(password ==null ||   ! password.equals(user.getPassword())) throw
new UsernameNotFoundException("wrong user/password");
    if(user !=null){
      auth = new UsernamePasswordAuthenticationToken(user.getUsername(),
user.getPassword(), user.getAuthorities());
    } else throw new UsernameNotFoundException("wrong user/password");
    return auth;

  }
  public boolean supports(Class<?> aClass) {
    return true;
  }
}
```

Your customized authentication provider should implement `AuthenticationProvider` and its `authenticate` method.

Note that the `userService` instance here should implement the Spring `UserDetailsService` interface and its `loadUserByUserName` method. The method returns the data model of a user. Note that you can extend Spring’s `User` object and create your own customized user. We mocked the `UserService` integration part with a data service. In a real application, there could be a service call to fetch and return user data, and your `UserServiceImpl` class will only wrap the user in the `UserDetails` data model, as follows:

```
@Service
public class UserServiceImpl implements UserDetailsService {
    public UserDetails loadUserByUsername(String userName) throws
UsernameNotFoundException {
        // suppose we fetch user data from DB and populate it into // User
object
        // here we just mock the service
        String role=null;
        if(userName.equalsIgnoreCase("admin")){
            role ="ROLE_ADMIN";
        }else if(userName.equalsIgnoreCase("accountant") ){
            role="ROLE_ACCOUNTANT";
        }else if(userName.equalsIgnoreCase("operator")){
            role="ROLE_USER";
        }else{
            throw new UsernameNotFoundException("user not found in DB");
        }
        List<GrantedAuthority> authorities=new ArrayList<GrantedAuthority>
();
        authorities.add(new GrantedAuthorityImpl(role));
        return new User(userName, "password", true, true, true, true,
authorities);
    }
}
```

After this, you can set your customized provider in the configuration class, as shown in the following code. When a user is authenticated, the authentication object should be populated programmatically. Later in this chapter, in the *Authorization* section, we will explain this object.

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled=true)
@ComponentScan(basePackages = "com.springessentialsbook.chapter5")
public class MultiWebSecurityConfigurator   {

    @Autowired
    private AuthenticationProvider authenticationProvider;

    @Autowired
    public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {
        auth.authenticationProvider(authenticationProvider);
    }
```

We defined the `springSecurityFilterChain` filter in the first step. To make it work, we need to register it in the web application, like so:

```
import
org.springframework.security.web.context.AbstractSecurityWebApplicationInit
ializer;
public class SecurityWebApplicationInitializer extends
AbstractSecurityWebApplicationInitializer { }
```

The class doesn't need any code, as the superclass (`AbstractSecurityWebApplicationInitializer`) registers the Spring Security filter. This happens while the Spring context starts up.

If we don't use Spring MVC, we should pass the following to the constructor:

```
super(WebSecurityConfigurator);
```

The class `AnnotatedConfigDispatcherServletInitializer` extends Spring's Servlet initializer `AbstractAnnotationConfigDispatcherServletInitializer`. This class allows Servlet 3 containers (for example, Tomcat) to detect the web application automatically, without needing `web.xml`. This is another step of simplifying the setting up of a web application, and it registers `DispatcherServlet` and Servlet mapping programmatically. By setting the `WebSecurityConfigurator` class in `getRootConfigClasses`, you tell the parent class method that creates the context of the application to use your annotated and customized Spring Security configuration class. The following is the code for the `AnnotatedConfigDispatcherServletInitializer` class:

```
public class AnnotatedConfigDispatcherServletInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
   @Override
   protected Class<?>[] getRootConfigClasses() {
      return new Class[] { MultiWebSecurityConfigurator.class };
   }

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

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

What we have configured so far in Spring Security is for checking whether the username and password are correct. If we want to configure other security features, such as defining a login page and the web application URL request to be authenticated, we need to override the `configure(HttpSecurity http)` method of `WebSecurityConfigurerAdapter`.

In our customized security configurator, we define a login page (`login.jsp`) and an authorization failure page (`nonAuthorized.jsp`), as follows:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfigurator extends WebSecurityConfigurerAdapter {
@Autowired
private AuthenticationSuccessHandler authenticationSuccessHandler;

   ...
   @Override
   public void configure(HttpSecurity http) throws Exception {
   ...

   .and().formLogin()
   .loginPage("/login").successHandler(authenticationSuccessHandler)
   .failureUrl("/nonAuthorized")

.usernameParameter("username").passwordParameter("password").loginProcessin
gUrl("/login")

...
```

This code tells Spring to process a submitted HTTP request form (with the POST method) with the expected username and password as parameters and `"/login"` as the action. Here is the login form:

```
<form role="form" action="/login" method="post">
  <input type="hidden" name="${_csrf.parameterName}"
value="${_csrf.token}"/>
  <div>
    <label for="username">Username</label>
    <input type="text" name="username" id="username" required autofocus>
  </div>
  <div>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
  </div>
  <button type="submit">Sign in</button>
</form>
```

Tip

If you don’t specify a username, password, and `loginProcessingUrl` parameter in the configuration file, Spring Security expects `j_username`, `j_password`, and `j_spring_security_check` from the client browser. By overriding Spring's default values, you can hide the Spring Security implementation from the client browser.

A **cross-site request forgery** (**CSRF**) attack happens, for example, when a malicious link clicked by an authenticated web client performs an unwanted action, such as transferring funds, obtaining contact e-mails, or changing passwords. Spring Security provides a randomly generated CSRF to protect the client from CSRF attacks.

If you omit `.loginPage` in the `configure` method, Spring uses its default login page, which is a very plain HTML login page. In this case, Spring Security uses the expected `j_username`, `j_password`, and `j_spring_security_check` parameters for the username, password, and action, and you should not configure them in the method.

For example, here we ask Spring to provide its own default login form:

```
@Override
public void configure(HttpSecurity http) throws Exception {
   ...
         .and().formLogin()
         .successHandler(authenticationSuccessHandler)
         .failureUrl("/nonAuthorized")
         ...

}
```

Spring Security supports HTTP Basic authentication, in which the client browser opens a popup (for the initial time) when you want to access a resource that matches a pattern (`"/adResources*/**"` in this case):

```
protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/adResources*/**").authorizeRequests().anyRequest().hasRol
e("ADMIN")
        .and()
        .httpBasic();
}
```

Server-side navigation could be the next step after authentication. Even though routing information is provided from the client side in modern client-side frameworks such as AngularJS, you may still want to keep routing logic on the server side. A success handler is a Spring Security feature that lets you define navigation logic after authentication in a web application.

Spring Security lets you configure customized server-side navigation after authentication. You can configure it inside the `configure` method (using `successHandler`):

```
@Override
public void configure(HttpSecurity http) throws Exception {
    ...
    .loginPage("/login").successHandler(authenticationSuccessHandler)
      ....
}
```

Your customized navigation handler should implement the interface

`AuthenticationSuccessHandler`. `OnAuthenticationSuccess` is the method that will be called when a user is authenticated. Within this method, we should define the target URL. In the sample implementation class shown here, the user's role is just used to define the target URL:

```
@Component
public class MyCustomizedAuthenticationSuccessHandler implements
AuthenticationSuccessHandler {
    private RedirectStrategy redirectStrategy = new
DefaultRedirectStrategy();

    public void onAuthenticationSuccess(final HttpServletRequest request,
final HttpServletResponse
    response, final Authentication authentication) throws IOException {
        handle(request, response, authentication);
        final HttpSession session = request.getSession(false);
        if (session != null) {
            session.setMaxInactiveInterval(3600);//1 hour
        }
        clearAttributes(request);
    }

    protected void handle(final HttpServletRequest request, final
HttpServletResponse response, final
    Authentication authentication) throws IOException {
        final String url = getUrl(authentication);
        if (response.isCommitted()) {
           return;
        }
        redirectStrategy.sendRedirect(request, response, url);
    }

    private String getUrl(final Authentication authentication) {
        String url=null;
        final Collection<? extends GrantedAuthority> authorities =
authentication.getAuthorities();
        for (final GrantedAuthority grantedAuthority : authorities) {
            if (grantedAuthority.getAuthority().equals("ROLE_USER")) {
                url= "/user" ;
                break;
            } else if
(grantedAuthority.getAuthority().equals("ROLE_ADMIN")) {
                url= "/admin" ;
                break;
             } else if
(grantedAuthority.getAuthority().equals("ROLE_ACCOUNTANT")) {
                url= "/accountant" ;
                break;
            }else {
                throw new IllegalStateException();
            }
        }
        return url;
    }
    protected void clearAttributes(final HttpServletRequest request) {
        final HttpSession session = request.getSession(false);
        if (session == null) {
            return;
        }
        session.removeAttribute(WebAttributes.AUTHENTICATION_EXCEPTION);
    }

    public void setRedirectStrategy(final RedirectStrategy
redirectStrategy) {
        this.redirectStrategy = redirectStrategy;
    }

    protected RedirectStrategy getRedirectStrategy() {
        return redirectStrategy;
    }
}
```

Spring Security lets you configure your security configuration in multiple methods, and in each method, you can define a different category of resources. Here, we have separated the security configuration for form-based and basic authentication into these two classes:

```
@EnableWebSecurity
@ComponentScan(basePackages = "com.springessentialsbook.chapter5")
public class MultiWebSecurityConfigurator {
   @Autowired
   private AuthenticationProvider authenticationProvider;
   @Autowired
   public void configureGlobalSecurity(AuthenticationManagerBuilder auth)
throws Exception {

      auth.authenticationProvider(authenticationProvider);
   }
   @Configuration
   protected static class LoginFormBasedWebSecurityConfigurerAdapter
extends WebSecurityConfigurerAdapter {
      @Autowired
      private AuthenticationSuccessHandler authenticationSuccessHandler;
      @Override
      public void configure(HttpSecurity http) throws Exception {
         http.authorizeRequests()
               ...
               .permitAll();
      }
   }
   @Configuration
   @Order(1)
   public static class HttpBasicWebSecurityConfigurationAdapter extends
WebSecurityConfigurerAdapter {
      @Override
      protected void configure(HttpSecurity http) throws Exception {

http.antMatcher("/adResources*/**").authorizeRequests().anyRequest().hasRol
e("ADMIN")
         .and()
         .httpBasic();
```

```
        }

}}
```

For example, in one method, we configure resources in the `adResources` path to be viewed by the admin role in an HTTP-based authentication (the browser opens a popup and asks for a username and password). In the second method, we apply form login authorization and limit access to resources based on user roles.

Authorization

In the *Authentication* section, we showed how user-provided credentials (username/password) are compared with application-stored ones, and if they match, the user is authenticated.

To boost security, we can limit the user's access to application resources. This is where authorization comes into the picture—the question of who should access which application's resources.

Spring Security provides very comprehensive authorization features. We can categorize these features into these three authorization groups:

- Web request (who can access which application URL?)
- Method invoking (who can call a method?)
- Domain object access (who can see which data?)

  For example, a customer should be able to see his own order and profile data, whereas an admin should be able to see all the customers' orders plus the data that is not visible to any customer.

Since version 3.0 of Spring Security, Spring has added Spring EL expressions to its authorization features. Spring EL lets you convert complex authorization logic into simple expressions. In this section, we use Spring EL for authorization.

`GrandAuthority` in Spring Security is the object for including a string value that is interchangeably called an authority, right, or permission (refer to the *Authentication* section, where the `AuthenticationProvider` interface is explained, to see how `GrandAuthority` is created). By default, if this string value starts with the prefix `ROLE_` (for example, `ROLE_ADMIN`), it will be considered as a user's role. So, it is also flexible enough to be used as a permission if it does not start with the prefix. Spring Security uses this object for web, method, and domain object authorization.

For web request authorization, we can limit user access based on the user's role in Spring Security, as follows (we will see later in this section how to do this in a controller):

```
public void configure(HttpSecurity http) throws Exception {
   http.authorizeRequests()
      .antMatchers("*.jsp").denyAll()
      .antMatchers("/", "/login").permitAll()
      .antMatchers("/user*//**").access("hasRole('USER') or 
hasRole('ADMIN')")
      .antMatchers("/admin*//**").access("hasRole('ADMIN')")
      .antMatchers("/accountant*//**").access("hasRole('ADMIN') or 
hasRole('ACCOUNTANT')")
      .failureUrl("/nonAuthorized")
      ...
      .permitAll();
}
```

Since we use spring MVC, we deny all URLs that end with `.jsp` (`*.jsp`) and let MVC map the URL to the JSP page. We permit anybody to have access to the login page using (`.antMatchers("/", /login").permitAll()`).

We limit user access to accountant resources to the admin and accountant roles (for example, `antMatchers("/accountant*//**").access("hasRole('ADMIN') or hasRole('ACCOUNTANT')")`). We set an error URL and forward a user to it if he fails authentication or tries to access non-authorized resources with `failureUrl("/nonAuthorized")`.

You need to add `@EnableGlobalMethodSecurity(prePostEnabled=true)` to be able to apply method/domain-level authorization:

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled=true)
@ComponentScan(basePackages = "com.springessentialsbook.chapter5")
public class MultiWebSecurityConfigurator {
```

We already described how to limit access to URLs using a configuration file. You can do the same thing in the controller's methods too:

```
@PreAuthorize("hasRole('ADMIN') or hasRole('ACCOUNTANT')"
@RequestMapping(value = "/accountant", method = RequestMethod.GET)
public String dbaPage(ModelMap model) {
...
}
```

For method-invoking authorization, you can configure Spring Security at the method level and define who can run a particular method in your application's service layer:

```
@PreAuthorize("hasRole('ADMIN') or hasRole('ACCOUNTANT')"
)
public void migrateUsers(id){...};
```

For domain object access, you can apply method-invoking authorization and have a service method to fine-tune who can see which data in the application. For example, in the service layer, you can limit access if the username parameter is equal to the logged-in username or the user has an admin role (refer to `bussinessServiceImpl` in the code):

```
@PreAuthorize("@businessServiceImpl.isEligibleToSeeUserData(principal,
#username)")
@RequestMapping("/userdata/{username}")
public String getUserPage(@PathVariable String username,ModelMap model) {
  {...}
```

The OAuth2 Authorization Framework

The OAuth2 Authorization Framework is simply a way to let third-party applications access your protected resources without you sharing your user credentials (username/password). You will have faced this situation when a website such as LinkedIn asks you to share your e-mail contacts, and when you agree, you are forwarded to your mail provider's login page (for example, Yahoo!).

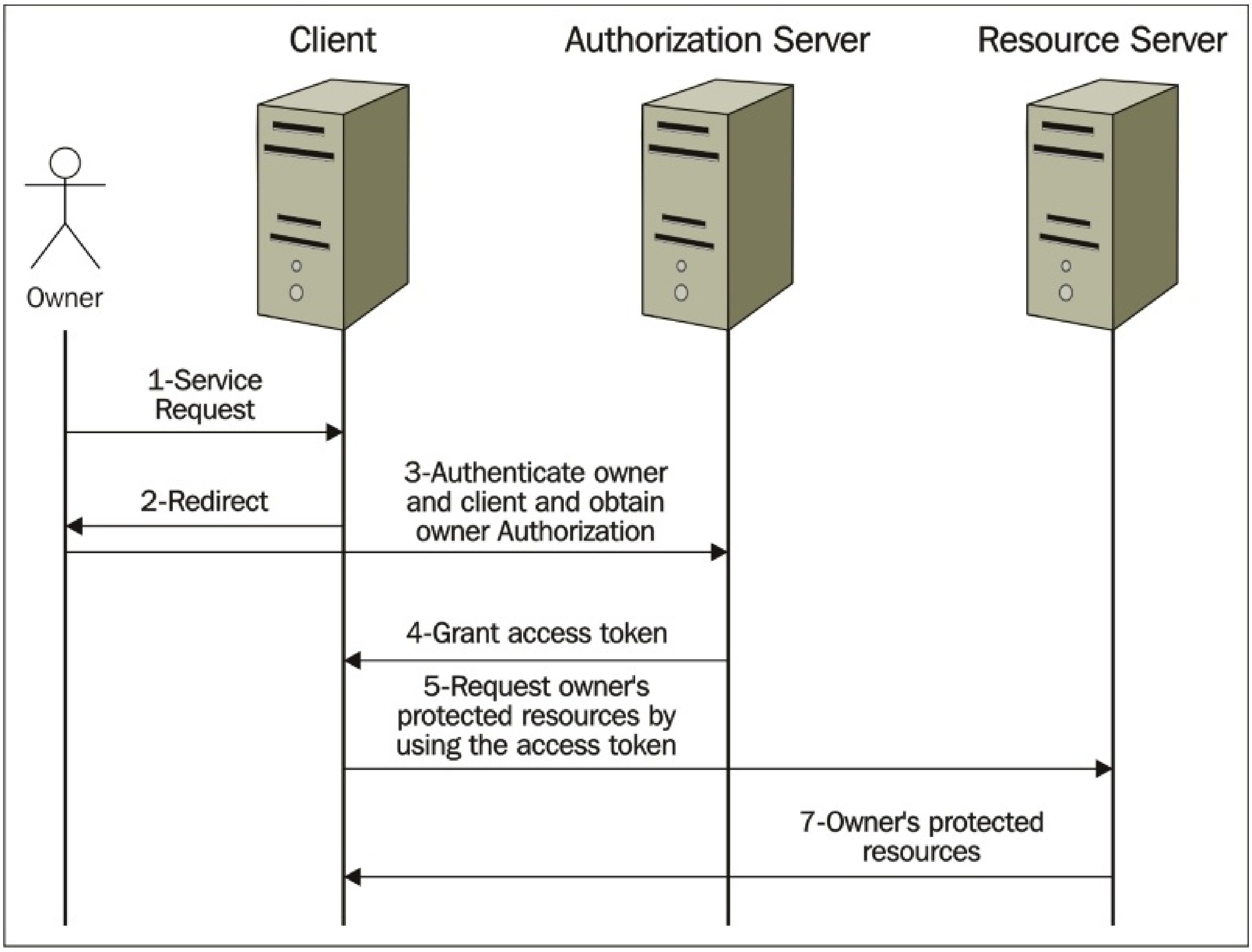
When you log in, the mail provider asks for your permission to share your contacts with LinkedIn. Then, LinkedIn can get the list of your contacts in order to send them an invitation.

OAuth2 relies on the following entities:

- **The resource owner**: This is the user with protected resources, for example, a Yahoo! e-mail user
- **The client or third-party application**: This is an external application that requires access to the owner's protected resources, for example, LinkedIn
- **The authorization server**: This server grants access to the client/third party after authenticating the resource owner and obtaining authorization
- **The resource server**: This server hosts the owner's protected resources, for example, the Yahoo! server

Many leading providers (for example, Google and Facebook) have both authorization and resource servers.

This diagram illustrates how the OAuth2 framework works in a simple form:


Client
Authorization Server
Resource Server
Owner
1-Service Request
2-Redirect
3-Authenticate owner and client and obtain owner Authorization
4-Grant access token
5-Request owner's protected resources by using the access token
7-Owner's protected resources

Spring facilitates the OAuth2 framework by reusing Spring Security concepts for authentication and authorization and includes new features to implement authorization and resource servers. To use Spring OAuth2 in your project, you need the following dependency:

```
<dependency>
   <groupId>org.springframework.security.oauth</groupId>
   <artifactId>spring-security-oauth2</artifactId>
   <version>2.0.8.RELEASE</version>
</dependency>
```

What we explained in the *Authentication* section with respect to validating the user and protecting resources remains the same here. The new things are the authorization and resource server settings.

The OAuth 2.0 service includes authorization and resource servers. Spring Security lets you have separate applications as authorization and resource servers, on which one authorization server could be shared by one or many resource servers, or have both types of servers in a single application. For simplicity, we implement authorization and resource servers within the same application.

In the class `MultiOAuth2ResourceAndAuthorizationConfigurator`, we define resource and authorization servers. `@EnableResourceServer` tags the class `ResourceServerConfiguration` as a resource server, which defines resources with the URL `/public` as non-protected and ones with the `/protected/**` URL as secure resources that require a valid token to access.

`@EnableAuthorizationServer` tags `AuthorizationServerConfiguration` as an authorization server that grants tokens to third-party clients. `TokenStore` is a Spring interface; its implementation classes (`InMemoryTokenStore`, `JdbcTokenStore`, and `JwtTokenStore`) keep track of tokens.

`JdbcTokenStore` uses a database to store tokens and has a Spring-JDBC dependency. `JdbcTokenStore` is suitable when you want to have a history of tokens, recovery after server failure, or the sharing of tokens among several servers.

`JwtTokenStore` encodes token-related data into the token itself. `JwtTokenStore` does not make tokens persistent and requires `JwtAccessTokenConverter` as a translator between a JWT-encoded token and OAuth authentication information.

For simplicity, we use the `InMemoryTokenStore` implementation class, but in real applications, using `JdbcTokenStore`/`JwtTokenStore` is a better practice.

We reuse the `AuthenticationManager` class that was detailed in the *Authentication* section.

The method `configure(ClientDetailsServiceConfigurer clients)` is the location in which we configure token generation settings, as follows:

- `withClient` tells us which client can access resources (this is separate from user authentication)
- `secret` is the client’s password
- `authorities` tells us which user roles are eligible to access the resource
- `authorizedGrantType` specifies which grant type the client has (for example, the refresh and access token)
- `accessTokenValiditySeconds` sets the token’s time to live

The settings are mentioned in the following code:

```
@Configuration
public class MultiOAuth2ResourceAndAuthorizationConfigurator {
    @Configuration
    @EnableResourceServer
    protected static class ResourceServerConfiguration extends
ResourceServerConfigurerAdapter {
        @Override
        public void configure(HttpSecurity http) throws Exception {
            http
                .headers()
                .frameOptions().disable()
                .authorizeRequests()
                .antMatchers("/public/").permitAll()
                .antMatchers("/protected/**").authenticated();
        }
    }
    @Configuration
    @EnableAuthorizationServer
    protected static class AuthorizationServerConfiguration extends
AuthorizationServerConfigurerAdapter implements EnvironmentAware {
        private static final String  CLIENT_ID = "myClientId";
        private static final String  CLIENT_PASSWORD = "myClientPassword";
        private static final int  TOKEN_TIME_TO_LIVE = 1800;
        private static final String  ROLE_USER = "ROLE_USER";
        private static final String  ROLE_ACCOUNTANT = "ROLE_ACCOUNTANT";
        private static final String  READ_ACCESS = "read";
        private static final String  WRITE_ACCESS = "write";
        private static final String  GRANT_TYPE_PASSWORD = "password";
        private static final String  GRANT_TYPE_REFRESH_TOKEN =
"refresh_token";
        @Bean
        public TokenStore tokenStore() {
            return new InMemoryTokenStore();
        }
        @Autowired
        private AuthenticationManager authenticationManager;
        @Override
        public void configure(AuthorizationServerEndpointsConfigurer
endpoints) throws Exception {
            endpoints
                .tokenStore(tokenStore())
                .authenticationManager(authenticationManager);
        }
        @Override
        public void configure(ClientDetailsServiceConfigurer clients)
throws Exception {
            clients
                .inMemory()
                .withClient(CLIENT_ID)
                .secret(CLIENT_PASSWORD)
                .scopes(READ_ACCESS, WRITE_ACCESS)
                .authorities(ROLE_USER, ROLE_ACCOUNTANT)
                .authorizedGrantTypes(GRANT_TYPE_PASSWORD,
GRANT_TYPE_REFRESH_TOKEN)
                .accessTokenValiditySeconds( TOKEN_TIME_TO_LIVE);
        }
        @Override
        public void setEnvironment(Environment environment) {
        }
    }
}
```

The resources we granted access to using the token are included in a controller. Here, we define a very simple resource:

```
@RequestMapping(value = "/protected", method = RequestMethod.GET)
@ResponseBody
public String getProtectedResources(ModelMap model) {
   return "this is from getProtectedResources";
}
@RequestMapping(value = "/public", method = RequestMethod.GET)
```

```
@ResponseBody
public String getPublicResources(ModelMap model) {
   return  "this is from getPublicResources";
}
```

You can run the project with the following command, which builds and runs the resource and authorization server:

```
mvn clean package spring-boot:run -Dserver.contextPath=/myapp -
Dserver.port=9090
```

If you try the following, you can see the resource because this URL is unprotected:

```
curl -i http://localhost:9090/myapp/public
```

However, if you try the next command, you get a “non-authorized” error and you need a valid token to access this resource:

```
curl -i http://localhost:9090/myapp/protected
```

You need to get a token first to be able to access protected resources. Spring MVC exposes an endpoint, `TokenEndpoint`, in order to get the token with the `/oauth/token` URL by default. The following command gives you an authorization token:

```
curl -X POST -vu myClientId:myClientPassword
'http://localhost:9090/myapp/oauth/token?
username=operator&password=password&grant_type=password'
```

Now, you can provide the token and access the secure resource:

```
curl -i -H "Authorization: Bearer [access_token]"
http://localhost:9090/myapp/protected
```

Notice that we set a time to live for the token and we need to refresh the token if it expires. The following command renews the token by calling the `/oauth/token` endpoint and passing `refresh_token` as the `grant_type` parameter:

```
curl  -X POST  -vu  myClientId:myClientPassword
'http://localhost:9090/myapp/oauth/token?
grant_type=refresh_token&refresh_token=[refresh_token]'
```

Summary

In this chapter, we detailed some features of Spring Security. Since Spring Security is a separate module and has a variety of features, in order to get more information about the whole specification, you need to go through https://docs.spring.io/spring-security/site/docs/current/reference/html/index.html and http://projects.spring.io/spring-security-oauth/.

Chapter 6. Building a Single-Page Spring Application

Having mastered many powerful features of Spring Framework while handling all the major technical concerns of enterprise applications, it is time to build a modern web application by putting all the techniques we learned in the previous chapters together. The current trend in web development is to build **single-page applications** (**SPAs**) that offer native-like user experience and an intuitive UI. In this chapter, let's build a responsive SPA powered by a Spring backend.

We will use Ember.js for building the SPA and Bootstrap for styling and responsive behavior. For Ember development, we will use a command-line tool called **Ember CLI**, which runs on Node.js and combines a collection of supporting tools for various critical functions of JavaScript-based modern frontend development.

The motivations behind SPAs

We know that Spring mainly focuses on the server side, that is, the integration, service, and data layers. Spring relies on other web technologies for rendering the presentation layer. Although Spring MVC does facilitate the presentation layer with the help of web technologies such as JSP and Thymeleaf, all of them work based on server-side rendering and full-page refreshes for responding to user interactions. In this traditional approach, the presentation layer of a web application is composed of a bunch of totally independent HTML files served by a server on demand, each representing a single screen, with just one rendered to the client browser at a time, taking a full round trip to the server for each user interaction. This provides a very poor user experience compared to native desktop applications, which gracefully re-render just the specific parts of the screen when required.

Although you can use some AJAX-using frameworks such as jQuery, in order to get data from a server or even for partial rendering of the UI (as in the case of JSF), it requires a lot of server resources for the presentation layer, and server processing is easily exhausted when the number of concurrent users grows. The presentation layer concerns are distributed across both the server and client tiers in this approach. UI developers need both client-side as well as server-side skills in this case, which makes web development harder.

Web developers had always been looking for a smarter method to build the UI of a data-driven application which is developed entirely on the client side, running inside a web browser, which offers a native-like rich user experience without a full refresh to the server for page transitions and navigations. They wanted a way to make their UI dynamic with data purely on the client side, eliminating the need for a server during frontend development, and plugs in to the server only when everything is ready on the client side. And for all these problems and requirements, the SPA paradigm is the answer.

SPAs explained

An SPA is a web application or website composed entirely of static web resources such as HTML, JavaScript, and CSS, loaded just once into the web browser in a single page load. Once booted, it updates itself intelligently as the user starts interacting with it. Unlike traditional web applications that perform a full page refresh for screen navigations, SPA routes and redraws (re-renders) screens without reloading the whole page (or the next page) from the server. It reconstructs the DOM structure with the help of JavaScript and styles itself with CSS in response to user actions and application events in order to represent them on the screen.

After the initial boot, the only time an SPA confers with a server is for dynamic data. SPAs usually rely on AJAX or WebSockets for data access from the server. The data transfer format is mostly JSON and sometimes XML. They contact the server via AJAX over HTTP asynchronously behind the scenes; this gives a smooth, fluid user experience without blocking the screen or keeping the user waiting for server responses. Besides, the server can synchronize its data changes with the client using the WebSocket API to provide a real-time experience.

The architectural benefits of SPAs

Besides the massive productivity gain and prominence of frontend developers, SPA offers many architectural benefits. It is blazingly fast compared to traditional server-rendered web applications, since it works entirely locally to the client. SPA offers a much more smooth and fluid user experience because of its immediate response, without needing us to resubmit the entire page to the server on every user interaction.

Note

JavaScript-intensive web applications run best on modern web browsers with enough memory on the host computer. Most frameworks utilize many HTML5 features and newer JavaScript functionality such as AJAX. SPAs can kill older browsers on slower PCs in no time.

SPAs offload the responsibility of the entire application state to the browser, freeing up server resources to focus on the core business logic (service) and data in terms of stateless web services, often designed as REST APIs. With SPAs, the server just becomes an API server; the entire user interaction is handled by the client, which improves server scalability a lot.

Another advantage, probably the most important one of SPAs, is that both client and server applications can be designed and evolved independently from each other. You can replace either of these without affecting the other as long as the endpoint (API) contracts remain intact. Also, you can let frontend developers build the UI and backend developers provide the data; both teams can focus on their own domain while working around a data contract.

SPA frameworks

Developing an SPA in plain JavaScript is not a smart idea considering the magnitude of responsibility handled by the SPA paradigm. It would be extremely tiring and error-prone if we set out to write all the routing, data binding, screen authoring, and rendering code from scratch in our applications. Fortunately, a set of very impressive frameworks emerged out of the SPA concept. Each of them offers varying levels of abstraction and architecture styles; some of them use powerful templating technologies. Let's take a look at the most popular SPA frameworks:

- **AngularJS**: Maintained by Google and supported by a community of developers and companies, Angular is the most popular and widely used SPA framework. It enhances vanilla HTML with the help of smart directives by adding two-way data binding. Angular supports localization and the building of reusable components.
- **ReactJS**: Backed by Facebook, Instagram, and a community of developers and companies, React is the fastest growing SPA framework at the time of writing. Facebook and Instagram have been developed using React. Its working is based on the concept of virtual DOM, an in-memory representation of displayed DOM that can be rendered either at the client or server (using Node), and manipulated using one-way binding. React screens are authored using JSX, an extension of JavaScript that allows the easy quoting of HTML inside JavaScript functions.
- **Ember.js**: A very powerful JavaScript MVC framework created by Yehuda Katz and contributed to by a strong community of active developers, Ember is used by many popular heavy traffic websites and applications, such as Groupon, Yahoo! (Ad Manager Plus), Zendesk, Square, Discourse, and LivingSocial. Ember can be used for building mobile and desktop applications: Apple Music is a notable desktop application built with Ember. Ember addresses the end-to-end problems of client-side web applications in an opinionated fashion. An early adopter of web and JavaScript standards such as ES6, web components, and promises, Ember comes with a set of powerful productivity tools and components that make it a complete-stack frontend framework.

In this chapter, we will use Ember.js for building an SPA that works as the frontend for a Spring API server. We will explore Ember.js, its core components, and the development tools first and then develop the frontend application using Ember, connecting to a Spring-based API server on the backend. This chapter will make you a full-stack developer with both server-side and client-side skills on the modern technology stack.

Introducing Ember.js

Ember is a comprehensive frontend framework for creating ambitious web applications. It is modeled after the **Model-View-Controller** (**MVC**) architectural pattern for the frontend. Its well-designed components with clearly defined responsibilities and rich capabilities allow developers to develop complex web applications with dramatically less code. In an Ember application, screens are composed using Handlebars templates that update themselves automatically when the underlying data changes.

Ember is productive out of the box, with a comprehensive development stack and a friendly API. The Ember development stack contains the following tools:

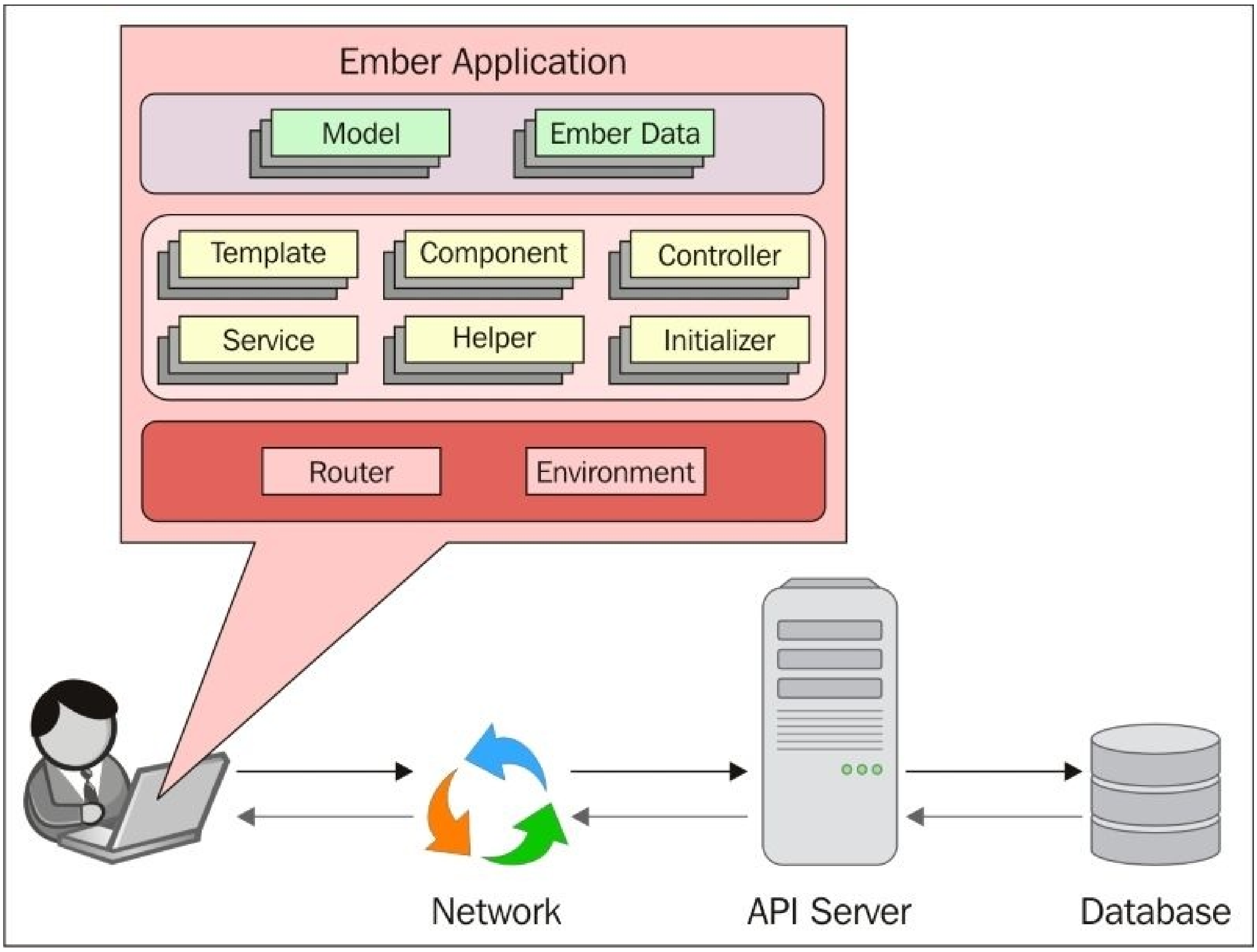
- **Ember CLI**: This is a command-line tool for creating projects, scaffolding, and managing their resources. It provides a development server with live reload, a testing framework, mocking server, and comprehensive asset management support.
- **Ember Inspector**: This is a debugger-cum-inspector tool for Ember applications, shipped as a plugin for Firefox and Chrome browsers. It allows you to evaluate and change Ember objects, elements, and variables while debugging, and provides a visual representation of the running Ember app.
- **Ember Data**: This subproject of Ember is a data-persistence library that can be directly mapped to a remote data source, such as a REST API. It maps Ember model objects with data entities on the server side via channels such as API endpoints. Ember Data provides adapters and serializers for standard REST and JSON API endpoints, and allows you to create your own adapters for any data source, for example, the browser's local storage.
- **Fastboot**: This is a server based on Node.js for the server-side rendering of Ember resources, eliminating the need for downloading JavaScript payloads post the loading of static assets for increased performance.
- **Liquid Fire**: This provides animation support for Ember views.
- **A testing framework**: Ember CLI integrates QUnit for testing Ember resources.

Ember is a very opinionated framework; this means that you are expected to structure the app by its own conventions, and then the framework takes care of the rest. If you follow the guidelines, you will end up writing very little, and very readable, code. Ember CLI generates the Ember project structure and artifacts with simple commands, in the way expected by the framework.

The anatomy of an Ember application

An Ember application is composed of a set of core elements with well-defined responsibilities and properties. They are defined under the Ember and DS namespaces of the Ember API.

This diagram depicts the high-level structure of an Ember application:



Routers

A router manages the application state. It maps a set of logical routes against unique URLs as mapped in the router configuration.

Routes or route handlers

A route handler, also known as a route (defined in `Ember.Route`), represents the handler for an individual route transition. A route can render a template that displays a screen. A route provides a model (data) that can be consumed by its template and controller. It has a corresponding controller that can handle user actions and maintain the state. A route can handle user actions by itself.

Templates

Templates are HTML fragments, usually rendered by routes and components. The user interface of an Ember application is composed of a collection of templates. Templates use the Handlebars syntax, which looks like regular HTML with some Handlebars expressions, which are enclosed in double curly braces (`{{ }}`). These Handlebars expressions bind Ember resources such as properties, objects, helpers, and components.

Components

Components control the behavior of the user interface. They handle user actions and manage many attributes that are used by the templates. A component consists of two parts:

- A JavaScript object that extends `Ember.Component`, where the actions and attributes are defined
- A template that is rendered into the parent view, usually that of a router

Models

Part of the Ember Data project, models represent the state of domain data in an Ember application. An Ember application will typically have a set of models extending from `DS.Model`. Routes usually display model data with the help of templates and modify data from the action handlers. Models are often loaded from a store (`DS.Store`), while Model instances are fetched from the actual persistent storage, mostly an API endpoint on the web server. Models can be persisted to the store; usually, they are sent back to the appropriate API endpoints.

Controllers

Controllers have a limited role in modern Ember applications; they will be deprecated in future versions. Currently, their use is limited to maintaining the state for a route and handling user actions. Since routes and components can handle actions, they are the perfect places for adding action handlers instead of controllers.

Besides these core elements, there are some supporting components that help the application development be easier and more elegant.

Input helpers

These are ready-made components bundled with Ember for taking inputs from users. Most of them are Ember versions of general form controls. Examples are the `{{input}}` and `{{textarea}}` input helpers. Custom-developed components can be used similarly to input helpers.

Custom helpers

Helpers add custom functionality to an application when they are not readily available, for using inside templates. Mostly, they are used for some kind of formatting. Examples are `{{format-date}}` and `{{format-currency}}`.

Initializers

Initializers can perform certain operations on application boot. There are two types of initializers: application initializers, which are executed on application boot, and application instance initializers, which load on application instance boot.

Services

Services are objects that can hold data and functions whose scope is application-wide. They are typically used for encapsulating core business logic spanned across many routes. Services can be injected into controllers, routes, components, and so on, where their methods can be invoked.

Working with Ember CLI

Ember CLI is an integrated, rapid development environment for Ember applications. Based on Broccoli, a fast and reliable asset pipeline that runs on Node.js, Ember CLI is a powerful command-line interface that integrates many productivity tools and optimization utilities necessary for JavaScript development.

Ember CLI provides the following features and tools for Ember development:

- It creates a strong, convention-based project structure for Ember applications
- It generates Ember-specific application resources, such as routes, templates, and components, from the command line
- It supports template authoring in the Handlebars, HTMLBars, and Emblem.js formats
- It supports scripting in ES2015 (ES6) modules, CoffeeScript, and EmberScript syntaxes
- It supports CSS authoring in CSS, Sass, Compass, and Stylus
- It converts Node.js-style ES2015 modules into RequireJS-model AMD modules
- It integrates the npm and Bower package managers for managing dependencies to JS libraries
- It integrates a development server with LiveReload, which automatically rebuilds and updates code changes to all connected browsers
- It performs asset management functions for application resources (combining, minifying, uglifying, versioning, and so on)
- It enables the sharing of code and functionality using add-ons and blueprints

Later in this chapter, we will use Ember CLI as a development tool for building an Ember application and its various artifacts.

Setting up Ember CLI

Ember CLI depends on Node.js. So, the first step is installing Node.js. Follow the instructions given on the website http://nodejs.org to set up Node.js.

Once Node.js is installed, you can install Ember CLI using `npm`, with the following command:

```
npm install -g ember-cli
```

Now, install Bower using the following command:

```
npm install -g bower
```

You may optionally install Watchman for better watching of code changes and the PhantomJS test-running environment.

Getting started with Ember CLI commands

Once Ember CLI is installed, you may start creating Ember applications incrementally using this set of commands to generate the required Ember artifacts:

Command	**Purpose**
`ember`	Prints the available commands.
`ember new <appname>`	Generates a fresh new project root folder with the same name as `<appname>`, the whole project structure, and all the necessary artifacts for a starter Ember application.
`ember init`	Turns the current directory into an Ember application and generates all necessary artifacts.
`ember build`	Builds and generates the deployable to the `dist` directory. Specify the environment using the environment flag, which defaults to `development`.
`ember server (or serve)`	Starts the development server at port `4200`. You may point to another port using the `--port` flag, for example, `ember serve --port 8080`.
`ember generate <generatortype> <name> <options>`	Generates specific generators, such as route, template, and helper, with the given name and options. Type `ember help generate` for the full list of available generators. Use the `--pod` flag for generators in the POD structure (explained later).
`ember destroy <generatortype> <name> <options>`	Removes artifacts created using the `ember generate` command. Remember to use the `--pod` flag if it was used while generating the artifact.
`ember test`	Runs tests written in the application using the Testem test runner.
`ember install <addon-name>`	Installs the given add-on into the application and registers it in the `package.json` file.

The Ember project structure

When you use the `ember new <project-name>` command, Ember CLI generates and organizes files in a specific structure based on convention and then compiles them and performs a set of tasks during building and runtime. The following table describes the folder layout and important files generated by Ember CLI:

File/Folder	**Description**
`app/`	This is the Ember application root. The `index.html` file and all your JavaScript files and templates go inside this, under proper subdirectories. Everything except `index.html` is compiled through the ES6 module transpiler, minified and concatenated to `<app-name>.js`, and then loaded by the `index.html` file at build time.
`app/index.html`	This is the only HTML page loaded from the server, which boots the Ember application on load from `<app-name>.js`, and is loaded using the `<script/>` tag embedded in it. Ember builds the entire DOM structure from inside this foundation HTML document in the browser at runtime.
`app/app.js`	This is the Ember application module. This is the application's entry point, where all the other modules are initialized and injected in order to create the entire application instance based on the resolver and environment-specific configuration.
`app/router.js`	This is the router configuration module of the application.
`app/adapters/`	Adapters for Ember Data modules go here. This folder is generated when the `ember generate adapter <model-name>` command is executed for the first time.
`app/components/`	All components go here, unless the `--pod` option is used.
`app/controllers/`	All controllers go here, unless the `--pod` option is used.
`app/helpers/`	All helpers go here, unless the `--pod` option is used.
`app/models/`	All models go here, unless the `--pod` option is used.
`app/routes/`	All routes go here, unless the `--pod` option is used.
`app/services`	All services go here, unless the `--pod` option is used.
`app/styles/`	Put all your style sheets for the application, whether Sass, LESS, Stylus, Compass, or plain CSS, here. Only plain CSS is supported by default; you can enable other types by installing the appropriate `npm` modules. For Sass, type `ember install ember-cli-sass` in the command line. For LESS, the command is `ember-cli-less`; for Compass, `ember-cli-compass-compiler`, and so on. For the default CSS option, add your styles to `app.css`. You can also organize the styles in different CSS files and import them to your `app.css` file.
`app/templates/`	All templates go here, unless the `--pod` option is used.
`bower.json`	This is the Bower configuration file.
`bower_components/`	Dependencies managed by Bower go here.

`config/`	Application configuration files fall here.
`config/environment.js`	Your environment-specific configurations go inside this file.
`dist/`	The deployable files generated by the build process go here. This is what you need to distribute for release.
`ember-cli-build.js`	This is the Broccoli build file. Include all resources managed by Bower and `npm` here.
`node_modules`	All node dependencies managed by npm go here.
`package.json`	This is the NPM dependency configuration file.
`public/`	This is a directory for uncompiled assets, such as fonts and images. The contents are copied as they are.
`server/`	This is where you can set up a development server for mock APIs and tests.
`tests/`	All your unit and integration tests go here.
`tmp/`	This is a temporary folder for build execution.
`vendor/`	Place your external dependencies that are not managed by npm or Bower here.

At the end of the build process, Ember CLI generates the deployable at `dist/directory`. You need to distribute the contents of this directory for hosting the deployable on a web server on release.

Working with the POD structure

By default, the `ember generate <generator>` command generates artifacts inside specific resource directories directly under the `app` root directory. So, all your routes go under `app/routes`, templates under `app/templates`, and so on. However, this becomes a bit unmaintainable as the application grows. To solve this problem, Ember CLI provides the option of organizing your files in a feature-driven (POD) structure using the `--pod` flag when you generate an artifact using the `ember generate` command.

In order for the POD structure to work, you need to first configure the POD directory in `config/environment.js` as given in the following code:

```
module.exports = function(environment) {
  var ENV = {
    ...
    podModulePrefix: 'my-ember-app/pod-modules',
    ...
    },
    ...
  return ENV;
};
```

The preceding snippet specifies that all the artifacts you generate with the `--pod` flag will be generated inside the `<app-root>/pod-modules` directory.

Once you configure the POD, you can start generating your artifacts with the `--pod` flag.

For example, if you want to generate a route inside the POD structure, use the following command:

```
ember generate route user --pod
```

This will generate the route file at `/app/pod-modules/user/route.js`.

POD modules group all the artifacts related to a feature in one place, thus making it more manageable.

Understanding the Ember object model

Ember comes with a rich API out-of-the-box, extending vanilla JavaScript classes and introducing new structures, providing enhanced capabilities such as two-way data binding, property observation, and so on. It provides smarter replacements for most of the common JavaScript constructs such as objects and arrays.

`Ember.Object` is the main base class of all Ember objects. It provides a class system with advanced features such as mixins and constructor methods. `Ember.Object` provides many special features, such as computed properties, data binding, and property-value change observers.

Declaring types (classes) and instances

You can inherit all the features of `Ember.Object` in your objects; just extend it in a purely object-oriented fashion, as given in the following code:

```
var User = Ember.Object.extend({
   ...
});
```

The preceding snippet is just a declaration of the `User` type. Now, you need to instantiate this class structure in order to use it in your program, as follows:

```
var User = Ember.Object.create();
```

You can either call a no args constructor like the preceding snippet, or you can pass a set of attributes with values as a JS object in order to create an instance of a declared class, as follows:

```
var myUser = User.create({
    firstName: "John",
    lastName: "Smith",
    userName: "jsmith",
    password: "secretp@ss",
    dateOfBirth: new Date(1980, 10, 24);
});
```

Accessing and mutating properties

Once the type is initialized, you can access its properties using a `get` method, as follows:

```
var name = myUser.get("name");
```

Remember to always use the `get` method instead of `object.property`, since Ember objects store managed properties in a different hash, which provides a few special features, unlike a vanilla JS object.

Make sure you use the `set` method for enabling all the special features of Ember objects, such as computed properties and property observation:

```
myUser.set('firstName', "Shameer");
```

Computed properties

A computed property is a virtual property derived from other normal properties, or it is a value returned by a function. `Ember.Object` can have computed properties too, as shown here:

```
var User = Ember.Object.extend({
   ...

   fullName: Ember.computed('firstName', 'lastName', function() {
      return `${this.get('firstName')} ${this.get('lastName')}`;
   }),
   ...
});
```

Once instantiated, you can access computed properties as well in the same manner as normal properties. They update themselves whenever a dependent property changes. You can create mutable computable properties too. The following is an example of a sensible implementation of such a computed property:

```
fullName: Ember.computed('firstName', 'lastName', {
    get(key) {
        return `${this.get('firstName')} ${this.get('lastName')}`;
    },
    set(key, value) {
        var [firstName, lastName] = value.split(/\s+/);
        this.set('firstName', firstName);
        this.set('lastName',  lastName);
        return value;
    }
})
```

Since the computed property is like any other function, you can add any business logic to it.

Property observers

You can observe normal or computed properties for any change in value. Register the property with `Ember.Observer` for this purpose. See the following example:

```
var User = Ember.Object.extend({
   ...

   dateOfBirth: new Date(),
   dobChanged: Ember.observer('dateOfBirth', function() {
      // deal with the change
      console.log(`Date of birth updated. New value is:
${this.get('dateOfBirth')}`);
   })
});
```

In the preceding snippet, the `dobChanged` function will fire whenever the `dateOfBirth` property gets updated. You can bind multiple properties with a single observer method by passing all the properties as arguments into the `Ember.observer` method prior to the function definition.

Note

Computed properties can also be observed. However, the observer method will not be triggered until the computed property is accessed, even if the dependent properties are updated.

Working with collections

Ember makes array manipulation smarter using a set of core collection classes, shown in the following table. Each of these provide many convenient methods that abstract complex array manipulation:

Collection type	Description
`Ember.Array`	This is an abstract implementation of observer-friendly array-like behavior. Concrete implementations are expected to have implemented methods, such as `length()` and `objectAt()`. Notable convenient methods are `any()`, `every()`, `filter()`, `filterBy()`, `find()`, `findBy()`, `forEach()`, `getEach()`, `map()`, `mapBy()`, `objectAt()`, `replace()`, `reverse()`, `sortBy`, `without()`, and so on.
`Ember.ArrayProxy`	`ArrayProxy` wraps objects that implement `Ember.Array` for binding use cases and swapping content while iterating.
`Ember.MutableArray`	This is an extension of `Array`, supporting an array of ordered sets.
`Ember.Enumerable`	This is a mixin for enumerating arrays.
`Ember.NativeArray`	This is the most concrete implementation of all of the above. You would use this in most cases.

Building UI templates using Handlebars

The primary UI authoring technology in Ember.js is Handlebars. Handlebars templates allow HTML fragments to embed dynamic content using Handlebars expressions placed inside double curly braces (`{{ }}`), the dynamic scripting blocks. Handlebars expressions perform data binding with attributes of routes, models, controllers, components, services, utils, and even application instances. Here is a sample Handlebars expression:

```
<h3>Welcome <strong>{{loggedInUser.fullName}}.</strong></h3>
```

This code snippet expects an object (preferably derived from `Ember.Object`, though it binds with normal JS objects too) with the name `loggedInUser`, present somewhere in the context in the parent context hierarchy (template, controller, route, or application). Then, it establishes a one-way data binding with the `fullName` attribute of the `loggedInUser` object; hence, it just displays the value of the bound attribute.

Handlebars helpers

Handlebars relies on helpers for business logic inside the dynamic scripting blocks. Handlebars executes the business logic implemented inside the helpers (if any) placed inside the curly braces, or it simply performs data binding with bound attributes.

Ember ships a set of built-in helpers and provides a nice way of developing custom helpers too. Built-in helpers can be categorized as follows:

- Input helpers
- Control flow helpers
- Event helpers
- Development helpers

Helpers can either be inline or en bloc. Inline helpers are just one-liners, similar to empty HTML and XML tags. See the `action` helper, which is an inline helper that takes parameters for processing:

```
{{action 'editUser' user}}
```

Inline helpers can be nested, embedding more dynamic values inside them:

```
{{action 'editUser' user (format-date today format='MMM DD, YYYY')}}
```

Block helpers have a start and an end construct with the same name, similar to HTML tags:

```
{{#if isLoggedIn}}
    Welcome <strong>{{loggedInUser.fullName}}</strong>
{{/if}}
```

Data binding with input helpers

Templates can establish two-way data binding using input helpers. Input helpers are mostly HTML form elements wrapped inside Ember components or views. Ember ships some built-in input helpers, such as `Ember.TextField`, `Ember.TextArea`, and `Ember.Checkbox`. Let's take a look at an example:

```
{{input placeholder="User Name" value=editingUser.userName}}
```

`{{input}}` is a built-in input helper that wraps HTML input text fields and checkboxes based on the value of the `type` attribute, which defaults to `text`. It allows two-way binding between the generated `<input type="text"/>` tag and the attribute `editingUser.userName`. Whenever either of the values is changed, it updates the other participant of the two-way binding. The `{{input}}` helper supports many useful attributes, such as `readonly`, `required`, `size`, `height`, `name`, `autofocus`, `placeholder`, `tabindex`, and `maxlength`.

Checkboxes are created using the same `{{input}}` helper, but by setting the type attribute to `checkbox`. The `{{textarea}}` helper represents the HTML `<textarea/>` component.

You can create your own input helpers as Ember components, which we will learn later in this chapter.

Using control flow helpers in Handlebars

Like most scripting languages, Handlebars supports the following control flow helpers:

- Conditionals:
  - `{{if}}`
  - `{{#else}}`
  - `{{#else if}}`
  - `{{#unless}}`
- Loops:
  - `{{#each}}`

Here is an example of the `{{if}}`, `{{else}}`, and `{{else if}}` helpers:

```
<div class="container">
{{#if isIdle}}
    You are idle for {{SessionService.idleMinutes}} minutes.
{{else if isLoggedIn}}
    Welcome <strong>{{loggedInUser.fullName}}</strong>
{{else}}
    <a {{action showLoginPopup}}>Please login</a>
{{/if}}
</div>
```

The `{{#each}}` helper is used to loop (iterate) through a collection, display it, and provide event hooks or actions around each element in the collection. A typical `{{#each}}` helper looks like this:

```
{{#each model as |user|}}
<tr>
<td><a {{action 'showUser' user }}>{{user.id}}</a></td>
<td>{{user.userName}}</td>
    ...
</tr>
{{/each}}
```

Using event helpers

Event helpers respond to user-invoked actions. The two primary event helpers in Ember are the `{{action}}` and `{{link-to}}` helpers.

The `{{link-to}}` helper helps in navigating to another route. See the following example:

```
{{link-to "Login here" "login" class="btn btn-primary"}}
```

The `{{action}}` helper is generally added to a normal HTML element in order to attach an event and event handler to it:

```
<a {{action "editTask" _task}} class="btn btn-success">Edit</a>
```

Handling routes

An Ember application transitions its state between a set of routes; each can render a template that displays the current state and a controller to support its state-based data. Routes are registered inside the router configuration, typically inside `router.js`, in the case of an Ember CLI project structure. Routes are defined inside their own JS files.

Routes can be generated and autoconfigured from the command line as follows:

```
ember generate route user --pod
```

This command generates `route.js` and `template.hbs` under `app/<pod-directory>/user/`. Upon generation, both artifacts will have a basic structure and you need to flesh them out according to your specific requirements. A typical route will have a model hook, which prepares its data. See the structure of a typical but minimal route given in the following code:

```
import Ember from 'ember';

export default Ember.Route.extend({

  model: function(args) {
    return this.store.findAll('task');
  }
});
```

In the preceding example, the `model` hook fetches data from `DS.Store`, the Ember Data repository. The route renders the `template.hbs` file in the same directory in the case of an Ember CLI project, unless another template is specified inside the `renderTemplate` method. The model of a route is available to the controller and template (via a controller) for manipulation and rendering.

Handling UI behavior using components

Components are the building blocks of dynamic UI fragments or elements in Ember. They render a template, optionally backed by a class extending `Ember.Component`.

The easiest way to create a component is to create a template file with a dash-separated name in the `app/components/` directory. Then you can embed it in inside other templates by just calling `{{<component-name>}}` and passing the required parameters.

Components are independent and completely isolated from the client context; all required data must be passed as parameters. However, if you use `{{yield}}` inside the template, it essentially becomes a block (or container) component, where you can add any content; this content can access any controller attribute and model.

A component can be generated by the following command:

```
ember generate component <component-name> --pod
```

This command generates two files, `component.js` and `template.hbs`, under the `app/<pod-dir>/components/<component-name>/` directory. If you do not use the `--pod` flag, it generates the `<component-name>.js` and `<component-name>.hbs` files under the directory `app/components/`.

Components insert the content into the DOM structure, where it is invoked, and control the behavior of the inserted content. By default, a component renders a `<div/>` element with the content generated by its template inside the `<div/>` element. You can specify a different HTML element instead of the `<div/>` element by setting the `tagName` attribute inside the `component.js` file. Similarly, you can set CSS class names dynamically using another property, `assNameBindings`.

Components provide some very useful life cycle hooks for manipulating different phases of the component. Some life cycle methods that can be overridden in the component class are `didInsertElement()`, `willInsertElement()`, and `willDestroyElement()`.

Components support standard HTML element events, depending upon which `tagName` is being used. They support all the standard touch events such as `touchStart` and `touchMove`, keyboard events such as `keyDown`, `keyUp`, and `keyPressed`, mouse events such as `mouseDown`, `mouseOver`, `click`, and `doubleClick`, form events such as submit and change, and HTML5 drag and drop events such as `dragStart` and `dragEnd`. You just need to declare the event as a function inside the component class; the component will fire the event and the associated function will get invoked as the user interacts with it.

Besides events, components can respond to action handlers, which are named functions defined inside the `actions` hash of the component class. These actions can be triggered anywhere from the component's template. Action handlers can accept parameters from the client code or templates.

Building a ToggleButton component step by step

Let's learn how to build an Ember component step by step using Ember CLI. We'll build a toggle button that turns off and on when clicked on. The component just changes its label and style based on its status attribute, `isActive`. We use Bootstrap styles for this example.

First, let's generate the component class and template file (`.hbs`) using Ember CLI. Issue this command from the command line at the root of your project:

```
ember generate component toggle-button --pod
```

See the `component.js` and `template.hbs` files generated at `app/<pod-dir>/components/toggle-button/`. Open and see the `component.js` file, it looks as given in the following code:

```
import Ember from 'ember';

export default Ember.Component.extend({
});
```

The generated `template.js` file just has `{{yield}}` inside it. Now you need to add necessary attributes and business logic into these two artifacts in order to make it a proper toggle button component. Here is a modified `component.js` file, with the proper behavior:

```
import Ember from 'ember';

export default Ember.Component.extend({
  tagName: "button",
  attributeBindings: ['type'],
  type: "button",
  classNames: ["btn"],
  classNameBindings: ["isActive:btn-primary:btn-default"],
  activeLabel: "On",
  inactiveLabel: "Off",
  isActive: false,

  currentLabel: Ember.computed('isActive', 'activeLabel', 'inactiveLabel',
function() {
    return this.get(this.get("isActive") ? "activeLabel" :
"inactiveLabel");
  }),

  click: function() {
    var active = this.get("isActive")
    this.set("isActive", !active);
  }
});
```

In the preceding code, notice that you specified the `tagName` attribute as `button`; otherwise, the generated HTML would be `<div/>`. Also, see how CSS class names are bound dynamically based on the `isActive` attribute. The `currentLabel` attribute is a computed attribute that depends on a few other attributes. In effect, the component responds to a click event and actually toggles the `isActive` variable. Everything else will work based on this event.

Now, let’s take a look at the modified `template.js` file to see how it utilizes the attributes and events handled by the `component.js` file:

```
{{currentLabel}}
```

Surprise! This is all the content in the template. It’s so simple to build. All the rest of the heavy lifting is done by the `component.js` file itself. Now the most interesting part is how the component is invoked from the client. Let’s take a look:

```
{{toggle-button}}
```

This is how you add the toggle button component in your client code, it is mostly route’s template. You can start clicking on the button repeatedly and see that it switches on and off.

This component can be customized by overriding its default properties. Let’s try changing its labels when it is on and off from the client side:

```
{{toggle-button activeLabel="Turn me off now :)" inactiveLabel="Turn me On
please.."}}
```

You can see the new active and inactive labels on the screen as you click on the button, toggling it. The toggle button is the simplest example of an Ember component, intended to give you just a taste of Ember components. A typical Ember application will have many complex components. Converting a reusable UI module or portion into a component is the best way to make your application more elegant and maintainable.

Persisting data with Ember Data

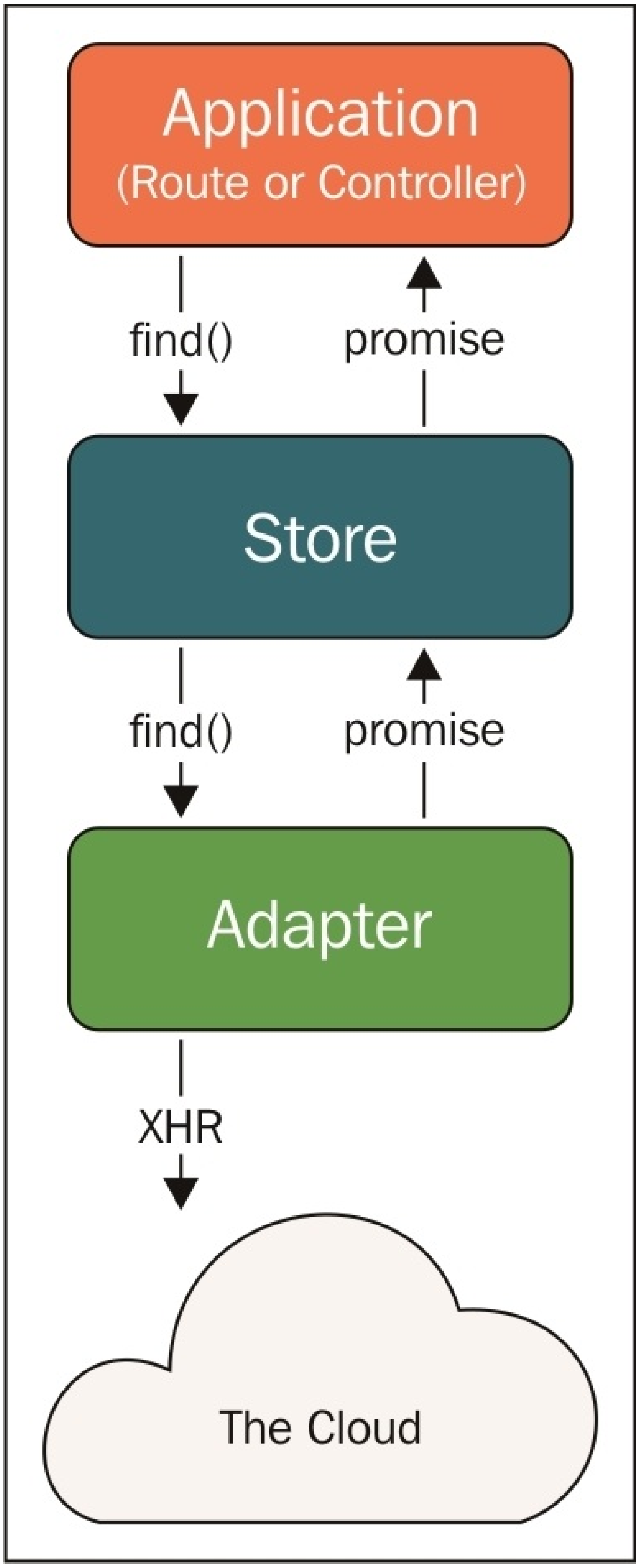
Ember Data is Ember’s data-access mechanism. It provides a simple API to deal with data, abstracting the complexities and protocols of data access and diverse data sources. With Ember Data, clients can deal with data models just as any other Ember object.

Ember Data defines a set of fundamental components that handle various roles and responsibilities in data access. These components are grouped under the namespace `DS`. The following table describes the most important Ember Data components defined under `DS`:

Component	**Purpose**
`DS.Model`	This is the fundamental unit of data and represents a record in a data collection. You need to define your data models by extending this class. It provides methods to save, delete, reload, and iterate properties, relationships, related types, and so on. It provides information about states, attributes, fields, relationships, errors, and so on. Also, it provides life cycle hook events.
`DS.Store`	This is the local repository of all the data created, fetched, and modified by Ember Data. `Store` fetches data with the help of adapters and converts them into appropriate `DS.Model` instances. Using serializers, `Store` serializes model instances into forms suitable for the servers. It provides methods for querying and creating new records.
`DS.Adapter`	This is an abstract implementation that receives various persistence commands from Store and translates them into forms that the actual data source (such as a Server API or a browser local storage) understands. Ember ships two concrete implementations: `DS.RESTAdapter` and `DS.JSONAPIAdapter`. Override the adapters if you want to change the default behaviors or attributes, such as remote URLs and headers.
`DS.Serializer`	This normalizes `DS.Model` instances into payloads for the API (or whichever data source it is) and serializes them back into the model. Two default serializers are `RestSerializer` and `JSONAPISerializer`. Override the serializers to customize the data formats for the server.

Ember Data architecture

Ember Data components communicate with each other asynchronously for data access operations, based on a **promise**. The `query` and `find` methods of both the **Store** and **Adapter** are asynchronous, and essentially return a **promise** object immediately. Once resolved, the model instance is created and returned to the client. The following diagram demonstrates how Ember Data components coordinate a `find` method operation asynchronously:



The clients of Ember Data components, which are typically routes, components, controllers, services, and so on, do not directly deal with adapters and serializers. They talk to the **Store** and model for normal data-access operations. Since the `Route.model` method (hook) supports **promise** objects, the transition will pause until the **promise** is resolved. We do not deal with resolving promises and hence with asynchronicity; rather, Ember handles it smartly.

Defining models

Models represent the domain data of an Ember application. They need to be defined in proper structures and registered with the store before they can be used for data access. An Ember CLI project expects models under the `app/models/` directory, or `app/<pod-dir>/models/` in case you are using the POD directory structure.

Let’s see a sample model definition. The following is the definition of a user model:

```
import DS from 'ember-data';

export default DS.Model.extend({

  name: DS.attr('string'),
  userName: DS.attr('string'),
  password: DS.attr('string'),
  dateOfBirth: DS.attr('date'),
  profileImage: DS.belongsTo('file')
});
```

Model attributes can be of the string, number, Boolean, and date types by default. For custom types, you need to subclass `DS.Transform`. Attributes can have default values too. You can specify default values as shown in the following line:

```
dateOfBirth: DS.attr('date', { defaultValue: new Date() }),
```

Defining model relationships

Models can engage in one-to-one, one-to-many, and many-to-many relationships among themselves:

- A one-to-one relationship is defined using `DS.belongsTo` in both model definitions
- A one-to-many relationship is defined using `DS.belongsTo` in one model and `DS.hasMany` in the other model
- A many-to-many relationship is declared when both models have `DS.hasMany` defined for each other

Building a Taskify application

Hey, it’s time to build our Taskify application end-to-end. First, let’s go back to building a proper API layer using Spring and then revisit Ember to build the frontend SPA. We will use Spring Data to connect to and access data from the API server.

For simplicity, we will not apply any security to the server; we will just focus on performing CRUD operations on two models: `User` and `Task`. Both `User` and `Task` are related to each other: `Task belongsTo User`. We will build models on both (server and client) sides. Let’s see how both technologies work together without having direct dependencies on each other.

Building the API server app

We explored the building of web apps using Spring MVC in Chapter 2, *Building the Web Layer with Spring Web MVC*. In Chapter 3, *Accessing Data with Spring*, we also learned how to persist data using Spring Data JPA. We are going to apply both these techniques again for building an API application for Taskify.

Setting up and configuring the project

Since we have already learned the basics of creating Spring MVC applications with Spring Data JPA, at this point, we will go into detail only about the specifics of the API endpoints. Refer to Chapter 2, *Building the Web Layer with Spring Web MVC* for Spring MVC configuration and Chapter 3, *Accessing Data with Spring* for details about Spring Data JPA. Set up and configure the project with the following steps:

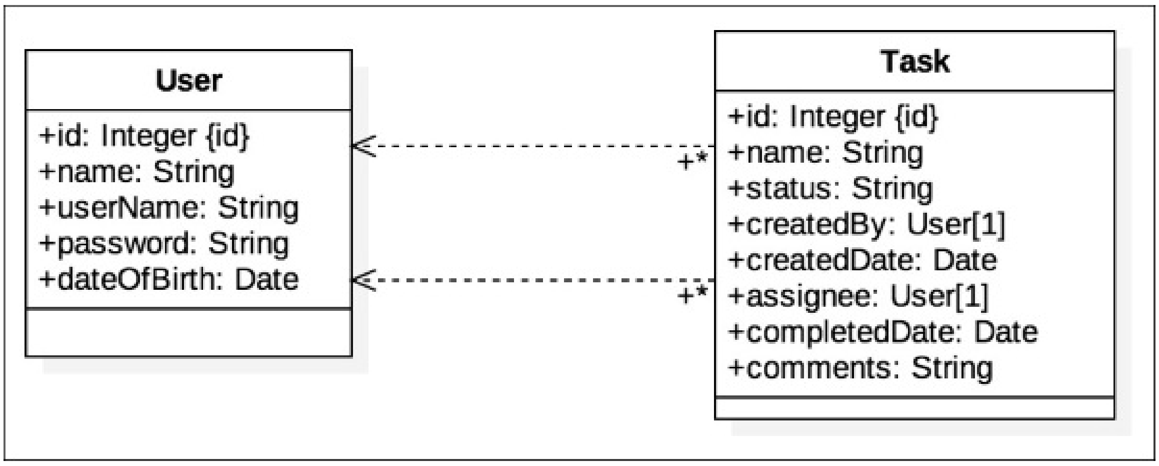
1. Create a Spring MVC application with a dependency on Spring Data JPA and the database of your choice.
2. Enable JPA repositories, specifying the base packages. For JavaConfig, annotate like this:

   ```
   @EnableJpaRepositories(basePackages = "com.taskify.dao")
   ```

3. Configure Spring Data JPA artifacts such as `DataSource`, `JdbcTemplate`, `TransactionManager`, and `EntityManager` with the flavor of your choice.

Defining the model definitions – User and Task

The application has the following two models as domain objects:



Now we need to realize these as Java classes, annotated as JPA entities, so that we can persist them to a database, as follows:

User.java

```
package com.taskify.domain;

import java.util.Date;
...
@Entity
@Table(name = "TBL_USER", uniqueConstraints = @UniqueConstraint(name =
"UK_USER_USERNAME", columnNames = {"USER_NAME" }) )
public class User {

  @Id
  @GeneratedValue
  private Long id;

  @Column(name = "NAME", length = 200)
  private String name;

  @Column(name = "USER_NAME", length = 25)
  private String userName;

  @Column(name = "PASSWORD", length = 20)
  private String password;

  @Column(name = "DOB")
  @Temporal(TemporalType.TIMESTAMP)
  private Date dateOfBirth;
  ...
  //Getters and setters go here..

}
```

Task.java

```
package com.taskify.domain;

import java.util.Date;
...
@Entity
@Table(name = "tbl_task")
public class Task {
  @Id
  @GeneratedValue
  private Long id;

  @Column(name = "NAME", length = 500)
  private String name;

  @Column(name = "PRIORITY")
  private int priority;

  @Column(name = "STATUS")
  private String status;

  @ManyToOne(optional = true)
  @JoinColumn(name = "CREATED_USER_ID", referencedColumnName = "ID")
  private User createdBy;

  @Column(name = "CREATED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  private Date createdDate;

  @ManyToOne(optional = true)
  @JoinColumn(name = "ASSIGNEE_USER_ID", referencedColumnName = "ID")
  private User assignee;

  @Column(name = "COMPLETED_DATE")
  @Temporal(TemporalType.TIMESTAMP)
  private Date completedDate;

  @Column(name = "COMMENTS")
  private String comments;
  ...
  //Getters and setters go here..
}
```

Once the JPA entities are ready, create the DAOs for both `User` and `Task`—`UserDAO` and `TaskDAO`—annotated with `@Repository`. As the best approach and for proper application layering, create the corresponding `@Service` beans too. Since we already covered the JPA `@Repository` and `@Service` classes in the previous chapters, the code for these beans is not listed here. You can find the exact code in the code bundle provided with this book.

Building API endpoints for the Taskify app

The purpose of the API server is to expose API endpoints for the consumption of clients, including the Taskify Ember frontend app. Let’s build these web services in the REST model, with JSON data format support.

In this section, we will list two classes annotated with `@RestController`: `UserController` and `TaskController`. The handler methods support asynchronous, non-blocking IO so that they are more scalable and faster. Handler methods are designed in the REST model. The HTTP methods `GET`, `POST`, `PUT`, and `DELETE` are mapped against the **Create**, **Read**, **Update**, and **Delete** (CRUD) operations.

UserController.java

`UserController` exposes endpoints for CRUD operations on the `User` entity. You can see the endpoints of `UserController` accepting and producing JSON data appropriately in its code, which is as follows:

```
package com.taskify.web.controller;

import java.util.List;
...

/**
 * Handles requests for user related pages.
 */
@RestController
@RequestMapping("/api/v1/user")
@CrossOrigin
public class UserController {

  private static final Logger =
LoggerFactory.getLogger(UserController.class);
  @Autowired
  private UserService;

  @RequestMapping(method = RequestMethod.GET)
  @ResponseBody
  public Callable<List<User>> listAllUsers() {
    return new Callable<List<User>>() {

      @Override
      public List<User> call() throws Exception {
        return userService.findAllUsers();
      }
    };
  }

  @RequestMapping(method = RequestMethod.POST, consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
  @ResponseBody
  public Callable<User> createNewUser( @RequestBody CreateUserRequest
request) {
    logger.info(">>>>>>>> Creating User, request - " + request);
    return new Callable<User>() {
      @Override
      public User call() throws Exception {
        return userService.createNewUser(request.getUser());
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.PUT, consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
  @ResponseBody
  public Callable<User> updateUser(@PathVariable("id") Long id,
@RequestBody UpdateUserRequest request) {
    logger.info(">>>>>>>> updateUser, request - " + request);
    return new Callable<User>() {
      @Override
      public User call() throws Exception {
        User existingUser = userService.findById(id);
        existingUser.setName(request.getUser().getName());
        existingUser.setPassword(request.getUser().getPassword());
        existingUser.setUserName(request.getUser().getUserName());
        userService.updateUser(existingUser);
        return existingUser;
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.GET)
  public Callable<User> getUser(@PathVariable("id") Long id) {
    return new Callable<User>() {
      @Override
      public User call() throws Exception {
        return userService.findById(id);
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.DELETE)
  @ResponseStatus(value = HttpStatus.NO_CONTENT)
  public Callable<Void> deleteUser(@PathVariable("id") Long id) {
    return new Callable<Void>() {
      @Override
      public Void call() throws Exception {
        userService.deleteUser(userService.findById(id));
        return null;
      }
    };
  }
}
```

TaskController.java

`TaskController` maps request endpoints for CRUD operations around the `Task` entity. Its code is as follows:

```
package com.taskify.web.controller;

import java.util.List;
...

@RestController
@RequestMapping("/api/v1/task")
@CrossOrigin
public class TaskController {

  private static final Logger =
LoggerFactory.getLogger(TaskController.class);

  @Autowired
  private UserService;

  @Autowired
  private TaskService;

  private static final int[] priorities = new int[] { 1, 2, 3, 4, 5, 6, 7,
8, 9, 10 };

  @RequestMapping(method = RequestMethod.GET)
  @ResponseBody
  public Callable<List<Task>> listAllTask() {
    return new Callable<List<Task>>() {
      @Override
      public List<Task> call() throws Exception {
        return taskService.findAllTasks();
      }
    };
  }

  @RequestMapping(method = RequestMethod.POST, consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
  @ResponseBody
  public Callable<Task> createNewTask( @RequestBody CreateTaskRequest
request) {
    logger.info(">>>>>>>> Creating Task, request - " + request);
    return new Callable<Task>() {
      @Override
      public Task call() throws Exception {
        return taskService.createTask(request.getTask());
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.PUT, consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
  @ResponseBody
  public Callable<Task> updateTask(@PathVariable("id") Long id,
@RequestBody UpdateTaskRequest request) {
    logger.info(">>>>>>>> updateTask, request - " + request);
    return new Callable<Task>() {
      @Override
      public Task call() throws Exception {
        Task existingTask = taskService.findTaskById(id);
        existingTask.setName(request.getTask().getName());
        existingTask.setPriority(request.getTask().getPriority());
        existingTask.setStatus(request.getTask().getStatus());
        existingTask.setCreatedBy(userService.findById(
request.getTask().getCreatedBy().getId()));

        if(request.getTask().getAssignee() != null &&
           request.getTask().getAssignee().getId() != null) {
             existingTask.setAssignee(userService.findById(
             request.getTask().getAssignee().getId()));
        } else {
          existingTask.setAssignee(null);
        }
        taskService.updateTask(existingTask);
        return existingTask;
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.GET)
  public Callable<Task> getTask(@PathVariable("id") Long id) {
    return new Callable<Task>() {
      @Override
      public Task call() throws Exception {
        return taskService.findTaskById(id);
      }
    };
  }

  @RequestMapping(path = "/{id}", method = RequestMethod.DELETE)
  @ResponseStatus(value = HttpStatus.NO_CONTENT)
  public Callable<Void> deleteTask(@PathVariable("id") Long id) {
    return new Callable<Void>() {
      @Override
      public Void call() throws Exception {
        taskService.deleteTask(id);
        return null;
      }
    };
  }
}
```

We have built all the necessary artifacts for the API server. You can package the application and deploy it. You should be able to access the `UserController` handlers at `http://<app-context-root>/api/v1/user` and the `TaskController` handlers at `http://<app-context-root>/api/v1/task/`. Now let’s go build the frontend.

Building the Taskify Ember app

Let’s get back to Ember development to build our SPA. Follow these steps. We will occasionally refer to previous sections of this chapter, and detail the specifics here.

Setting up Taskify as an Ember CLI project

Let's generate the project and set up all the artifacts. Follow these steps:

1. Create a new Ember project using Ember CLI from the command line:

   ```
   ember new taskify
   ```

2. Install `broccoli-merge-trees` and `broccoli-static-compiler` for a richer Broccoli configuration. Issue the following commands from the command line:

   ```
   npm install --save-dev broccoli-merge-trees
   npm install --save-dev broccoli-static-compiler
   ```

3. Install Bootstrap with Bower from the command line:

   ```
   bower install bootstrap
   ```

4. Configure Broccoli to include bootstrap.js, CSS, and fonts in the `ember-cli-build.js` file:

   ```
     var mergeTrees = require('broccoli-merge-trees');
     var pickFiles = require('broccoli-static-compiler');
     var extraAssets = pickFiles('bower_components/bootstrap/dist/fonts',{
   srcDir: '/', files: ['**/*'], destDir: '/fonts' });

     app.import('bower_components/bootstrap/dist/css/bootstrap.css');
     app.import('bower_components/bootstrap/dist/js/bootstrap.js');

     return mergeTrees([app.toTree(), extraAssets]);
   ```

5. In the application, we will be using a third-party Ember add-on called `ember-bootstrap-datetimepicker`. Let's install it into the project:

   ```
   ember install ember-bootstrap-datetimepicker
   ```

6. Build `npm` and `bower` dependencies:

   ```
   npm install
   bower install
   ```

7. Start the Ember server using the `ember serve` command, and make sure your application is accessible at `http://localhost:4200/`.
8. Set the POD directory inside `/config/environment.js`:

   ```
     var ENV = {
       modulePrefix: 'ember-webapp-forspring',
       podModulePrefix: 'ember-webapp-forspring/modules',
       ...
     }
   ```

Now we can start generating the required Ember artifacts in this POD directory.

Setting up Ember Data

We need two models: `User` and `Task`. Let's generate them first with the following code. For models, we do not use POD:

```
ember generate model user
ember generate model task
```

Find the generated models under the `/app/models/` folder. Open them and set the attributes and relationships:

`User.js`

```
import DS from 'ember-data';

export default DS.Model.extend({
  name: DS.attr('string'),
  userName: DS.attr('string'),
  password: DS.attr('string'),
  dateOfBirth: DS.attr('date')
});
```

`Task.js`

```
import DS from 'ember-data';

export default DS.Model.extend({
  name: DS.attr('string'),
  priority: DS.attr('number'),
  status: DS.attr('string'),
  createdBy: DS.belongsTo('user'),
  createdDate: DS.attr('date'),
  assignee: DS.belongsTo('user'),
  completedDate: DS.attr('date'),
  comments: DS.attr('string'),
});
```

Let's generate an (Ember Data) application adapter that has some global properties common to all adapters:

```
ember generate adapter application
```

Open the generated `/app/adapters/application.js` file, and add two attributes, `host` and `namespace`, with the right values as shown in the following code. After this, adapters for all models will take these attributes unless overridden individually:

```
import Ember from 'ember';
import DS from 'ember-data';

export default DS.RESTAdapter.extend({
  host: 'http://<apiserver-context-root>',
  namespace: 'api/v1'
});
```

We need to override the default serializers, as Ember Data expects the ID of the dependent objects for sideloading, where the API server sends out nested objects embedded within.

So, generate both serializers from the command line and then update the content appropriately:

```
ember generate serializer user
ember generate serializer task
```

Update the generated `/app/serializers/user.js` file with the following content:

```
import DS from 'ember-data';

export default DS.RESTSerializer.extend(DS.EmbeddedRecordsMixin, {
    attrs: {
        profileImage: {embedded: 'always'},
    },
});
```

Update the generated `/app/serializers/task.js` file with the following content:

```
import DS from 'ember-data';

export default DS.RESTSerializer.extend(DS.EmbeddedRecordsMixin, {
    attrs: {
        createdBy: {embedded: 'always'},
        assignee: {embedded: 'always'},
    },
});
```

Configuring application routes

Routes represent application states. They need to be registered with the router of the application in order to enable navigation. Our application has three primary routes: `index`, `user`, and `task`. Let's generate them in the `pod` directory. Do it from the command line:

```
ember generate route index --pod
ember generate route user --pod
ember generate route task --pod
```

Take a look at `router.js` now; you will see these new routes registered there. Also, the `route.js` and `template.hbs` files generated for each of these under the POD directories will be present.

Building the home screen

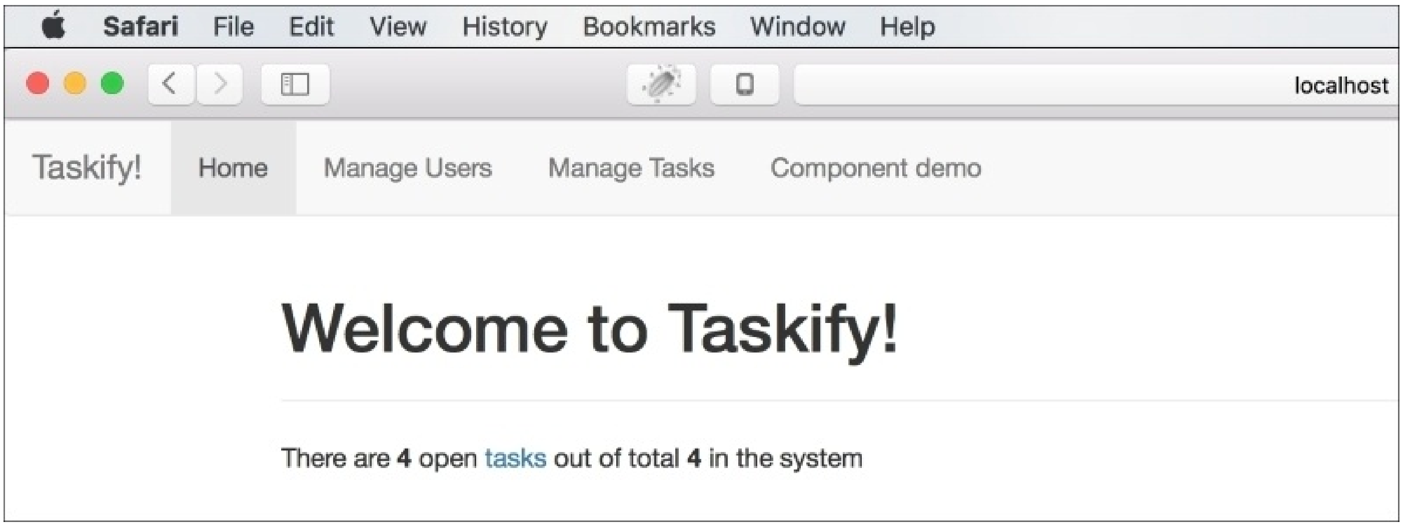
Now, let’s set up the index template to show counts for the total number of tasks and the number of open tasks in the system. Open the `/app/modules/index/template.js` file and update it with this content:

```
<div class="container">
  <h1>Welcome to Taskify!</h1>
  <hr />
  <P>There are <strong>{{model.openTasks.length}}</strong> open
    {{#link-to "task"}}tasks{{/link-to}} out of total
    <strong>{{model.tasks.length}}</strong> in the system</P>
</div>
```

The preceding template binds the model attributes using Handlebars and expects the model to be loaded with proper data. Let’s go build the model in the `route.js` file:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model: function() {
    var _model = Ember.Object.extend({
      tasks: null,
      openTasks: Ember.computed("tasks", function() {
        var _tasks = this.get("tasks");
        return Ember.isEmpty(_tasks) ? Ember.A([]):
_tasks.filterBy("status", "Open");
      }),
    }).create();

    this.store.findAll('task').then(function(_tasks) {
    _model.set("tasks", _tasks);
    return _model;
  });
    return _model;
});
```

In the preceding code, the model hook first loads data from the server using `DS.Store` (Ember Data), constructs the model object with attributes, including computed properties, and then returns. The home screen will look like the following image (ignore the headers for now):



Welcome to Taskify!

There are **4** open tasks out of total **4** in the system

Building the user screen

Now, let’s build the user screen for listing all the users in the system. Let’s build the model inside the route’s model hook first. Add this method inside `/app/modules/user/route.js`:

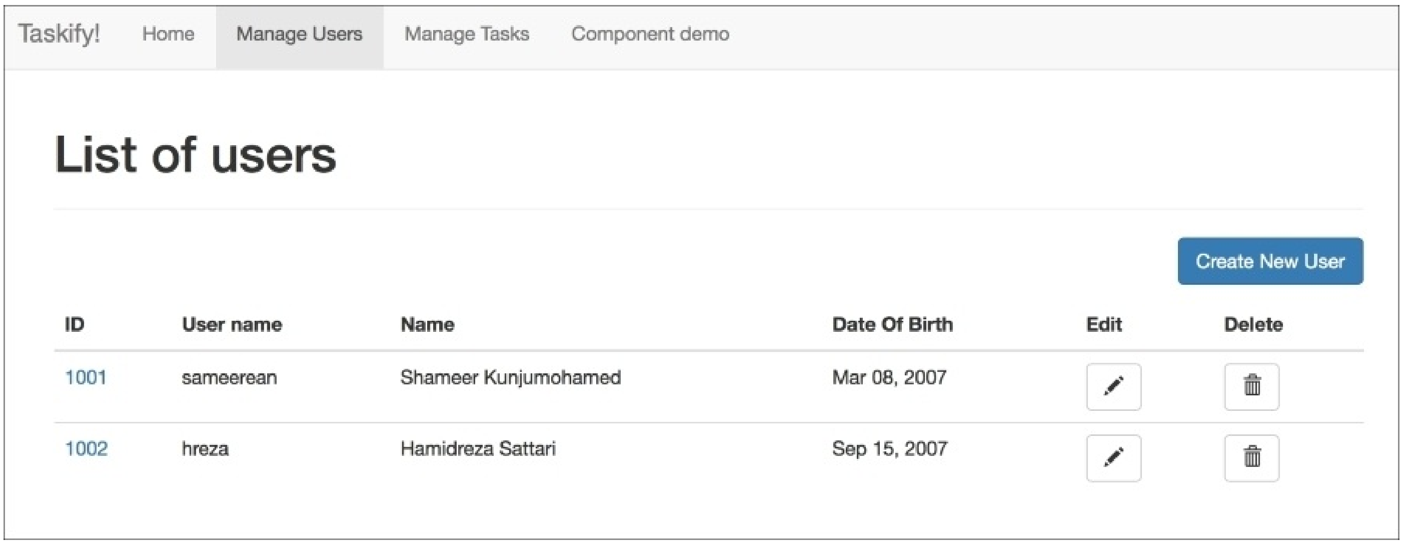
```
model: function() {
  return this.store.findAll('user');
},
```

You can see how beautifully Ember and Ember Data work together to simplify such an otherwise complex task of fetching, transforming, and deserializing data into model instances and finally making it available for the consumption of the template and controller, asynchronously, without blocking the screen.

Now let’s display this data on a screen. Update the `/app/modules/user/template.hbs` file with the following content:

```
<div class="container">
  <h1>List of users</h1><hr />
  <p class="text-right">
    <a {{action 'createNewUser'}} class="btn btn-primary"
role="button">Create New User</a></p>

  <table class="table table-hover">
    <thead><tr>
      <th>ID</th>
      <th>User name</th>
      <th>Name</th>
      <th>Date Of Birth</th>
      <th>Edit</th>
      <th>Delete</th>
    </tr></thead>
  <tbody>
  {{#each model as |user|}}
  <tr>
    <td><a {{action 'showUser' user }}>{{user.id}}</a></td>
    <td>{{user.userName}}</td>
    <td>{{user.name}}</td>
    <td>{{format-date user.dateOfBirth format='MMM DD, YYYY'}}</td>
    <td><button type="button" class="btn btn-default" aria-label="Edit
user" {{action 'editUser' user}}>
        <span class="glyphicon glyphicon-pencil" aria-hidden="true"></span>
</button></td>
    <td><button type="button" class="btn btn-default" aria-label="Delete
user" {{action 'deleteUser' user}}>
        <span class="glyphicon glyphicon-trash" aria-hidden="true"></span>
</button></td>
  </tr>
  {{/each}}
  </tbody>
  </table>
</div>
```

Now you can see the `user` route at `http://localhost:4200/user`, which looks like this:



List of users

ID	User name	Name	Date Of Birth	Edit	Delete
1001	sameerean	Shameer Kunjumohamed	Mar 08, 2007		
1002	hreza	Hamidreza Sattari	Sep 15, 2007		

Building a custom helper

In the `template.hbs` file, you may notice a custom helper:

```
{{format-date user.dateOfBirth format='MMM DD, YYYY'}}
```

Let’s go build it; you should have already got an error since this helper hasn’t been defined yet. From the command line, generate it using the following command:

```
ember generate helper format-date
```

Update the generated `/app/helpers/format-date.js` file with the following script:

```
import Ember from 'ember';

export function formatDate(params, hash) {
  if(!Ember.isEmpty(hash.format)) {
    return moment(new Date(params)).format(hash.format);
  }
  return params;
}

export default Ember.Helper.helper(formatDate);
```

Now look at your browser; the user list should render properly.

Adding action handlers

Inside the `/app/modules/user/template.hbs` file, there are four action invocations: `createNewUser`, `showUser`, `editUser`, and `deleteUser`. All these methods accept a `user` variable as a parameter. Let’s add these actions inside `/app/modules/user/route.js` first:

```
actions: {
  createNewUser: function() {
    this.controller.set("_editingUser", null);
    this.controller.set("editingUser", Ember.Object.create({
      name: null,
      userName: null,
      password: null,
      dateOfBirth: new Date()
    }));

  Ember.$("#userEditModal").modal("show");
  },
  showUser: function(_user) {
    this.controller.set("_editingUser", _user);
    this.controller.set("editingUser", Ember.Object.create(
    _user.getProperties("id", "name", "userName", "password",
"dateOfBirth", "profileImage")));
    Ember.$("#userViewModal").modal("show");
  },
  editUser: function(_user) {
    this.actions.closeViewModal.call(this);
    this.controller.set("_editingUser", _user);
    this.controller.set("editingUser", Ember.Object.create(
    _user.getProperties("id", "name", "userName", "password",
"dateOfBirth", "profileImage")));
    Ember.$("#userEditModal").modal("show");
  },
  deleteUser: function(_user) {
    if(confirm("Delete User, " + _user.get("name") + " ?")) {
      var _this = this.controller;
      _user.destroyRecord().then(function() {
        _this.set("editingUser", null);
        _this.set("_editingUser", null);
        _this.set("model", _this.store.findAll('user'));
      });
    }
  }
}
```

Building a custom component – modal window

In the preceding code listing, both the `createNewUser` and `editUser` methods use `userViewModal` using jQuery. This is a Bootstrap modal window built as a custom Ember component. In fact, there are four components working together in a nested fashion: `{{modal-window}}`, `{{modal-header}}`, `{modal-body}}`, and `{{modal-footer}}`.

Let's generate the artifacts from a commandline first:

```
ember generate component modal-window --pod
ember generate component modal-header --pod
ember generate component modal-body --pod
ember generate component modal-footer --pod
```

The `component.js` and `template.hbs` files should be generated under the `/app/modules/components/<component-name>/` directory. Now let's update the `.js` and `.hbs` files to make it a true modal window:

`modal-window/template.hbs`

```
<div class="modal-dialog" role="document">
<div class="modal-content">{{yield}}</div>
</div>
```

`modal-window/component.js`

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["modal", "fade"],
  attributeBindings: ['label:aria-label', 'tabindex', 'labelId:aria-
labelledby'], ariaRole: "dialog", tabindex: -1, labelId:
Ember.computed('id', function() {
    if(Ember.isEmpty(this.get("id"))) {
      this.set("id", this.get("parentView.elementId") + "_Modal");
    }
  return this.get('id') + "Label";
  })
});
```

`modal-header/template.hbs`

```
{{yield}}
```

`modal-header/component.js`

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["modal-header"],
});
```

`modal-body/template.hbs`

```
{{yield}}
```

`modal-body/component.js`

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["modal-body"],
});
```

modal-footer/template.hbs

```
{{yield}}
```

modal-footer/component.js

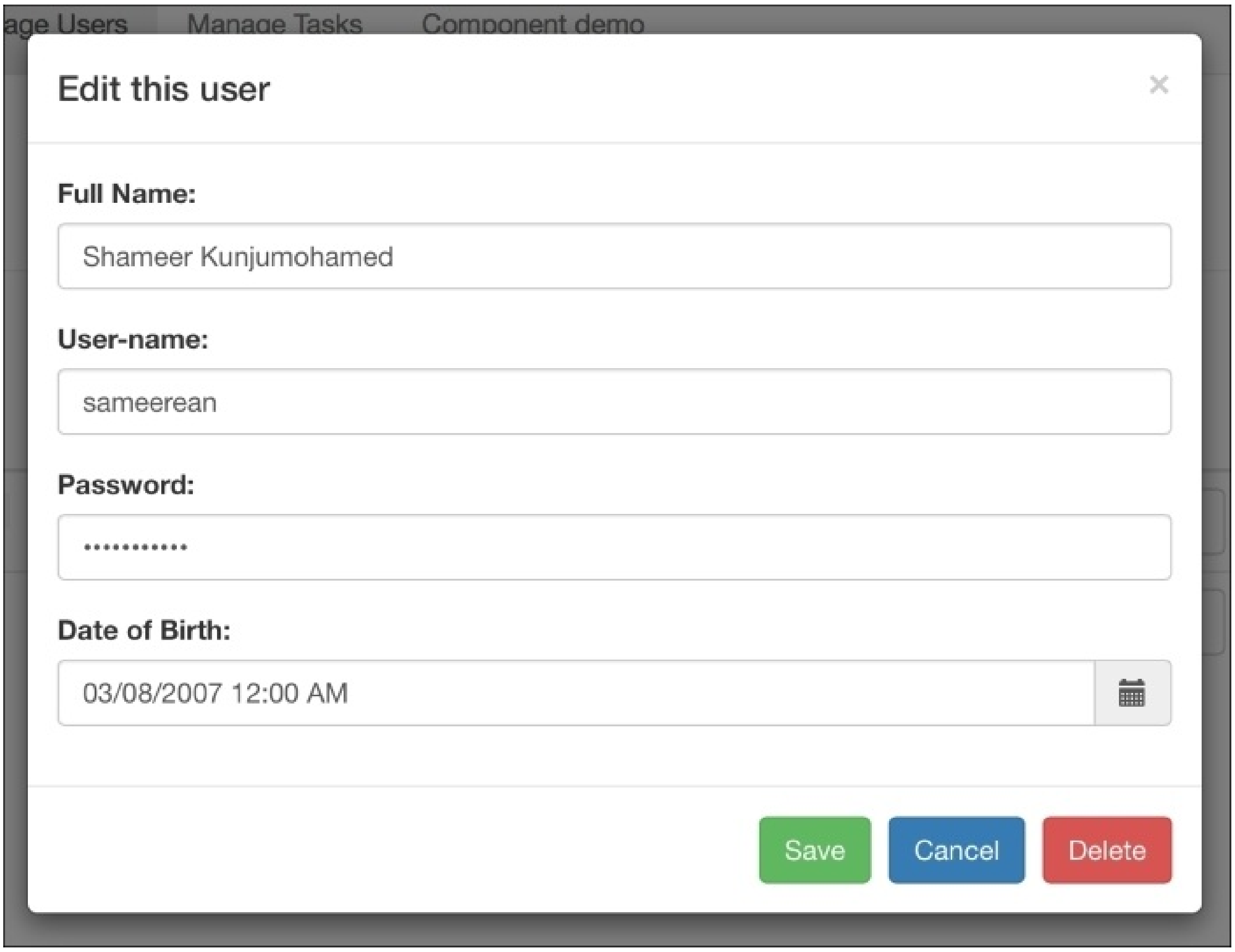
```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["modal-footer"],
});
```

Building userEditModal using {{modal-window}}

The four modal related components have been built; it’s time to add `userEditModal` into the `user/template.js` file. Add the following code or `userEditModal` into the `user/template.js` file:

```
{{#modal-window id="userEditModal"}}

  {{#modal-header}}
  <button type="button" class="close" {{action "closeEditModal"}} aria-
label="Close"><span aria-hidden="true">&times;</span></button>
  <h4 class="modal-title" id=labelId>{{modalTitle}}</h4>
  {{/modal-header}}

  {{#modal-body}}
  <form> <div class="form-group">
  <label for="txtName">Full Name:</label>
  {{input class="form-control" id="txtName" placeholder="Full Name"
value=editingUser.name}} </div>
  <div class="form-group"> <label for="txtUserName">Username:</label>
  {{input class="form-control" id="txtUserName" placeholder="User Name"
value=editingUser.userName}}</div>
  <div class="form-group"> <label for="txtPassword">Password:</label>
  {{input type="password" class="form-control" id="txtPassword"
placeholder="Your secret password" value=editingUser.password}}</div>
  <div class="form-group"><label for="calDob">Date of Birth:</label>
  {{bs-datetimepicker id="calDob" date=editingUser.dateOfBirth
       updateDate=(action (mut editingUser.dateOfBirth))
       forceDateOutput=true}} </div> </form>
  {{/modal-body}}

  {{#modal-footer}}
  <a {{action "saveUser"}} class="btn btn-success">Save</a>
  <a {{action "closeEditModal"}} class="btn btn-primary">Cancel</a>
  <a {{action 'deleteUser' _editingUser}} class="btn btn-danger"> Delete
</a>
  {{/modal-footer}}
{{/modal-window}}
```

The preceding code listing integrates the user edit form with `{{modal-body}}`, with the form title inside `{{modal-header}}`, action buttons inside `{{modal-footer}}`, and all of this inside `{{modal-window}}` with the ID `userEditModal`. Just click the **Edit** button of a user row; you will see this nice modal window pop up in front of you:



Edit this user

Full Name: Shameer Kunjumohamed

User-name: sameerean

Password: •••••••••••

Date of Birth: 03/08/2007 12:00 AM

The **Save** button of `userEditModal` invokes the `saveUser` action method, the **Cancel** button invokes the `closeEditModal` action, and the **Delete** button invokes `deleteUser`. Let's add them inside the `actions` hash of `user/route.js`, next to `deleteUser`:

```
...
closeEditModal: function() {
  Ember.$("#userEditModal").modal("hide");
  this.controller.set("editingUser", null);
  this.controller.set("_editingUser", null);
},
closeViewModal: function() {
  Ember.$("#userViewModal").modal("hide");
  this.controller.set("editingUser", null);
  this.controller.set("_editingUser", null);
},

saveUser: function() {
  if(this.controller.get("_editingUser") === null) {
```

```
  this.controller.set("_editingUser",this.store.createRecord("user",
    this.controller.get("editingUser").getProperties("id", "name",
"userName", "password", "dateOfBirth")));
  } else {
    this.controller.get("_editingUser").setProperties(
         this.controller.get("editingUser").getProperties("name",
"userName", "password", "dateOfBirth"));
  }
  this.controller.get("_editingUser").save();
  this.actions.closeEditModal.call(this);
}
```

Similarly, `user/template.js` has `userViewModal`, which just displays the user data in read-only format. Now, you can easily derive it from `userEditModal`; hence, we’re not listing it here.

Building the task screen

The task screen follows the same pattern as the user screen. This section describes only the portions logically different from the user screen and assumes that you will start developing the task screen from the user screen and incorporate the changes described here. Also, you can see the complete code from the project files attached to this chapter of the book.

The task screen has some extra state-specific data besides the model data (the list of tasks). For maintaining that data while the task screen is active, we will create a controller:

```
ember generate controller task --pod
```

The relationship between `Task` and `User` is that a task is created by a user and assigned to another user. So, on the edit task (or create new task) screen, a list of users should be shown in a selection box so that one can be selected from the list. For that, we need to load the list of users from `DS.store` to a variable inside the controller. Here is the controller method that loads the `user` list:

```
loadUsers: function() {
  this.set("allUsers", this.store.findAll('user'));
}.on("init"),
```

This method will get fired on initialization of the controller, courtesy of the `.on("init")` construct. The template code extract that renders the user list in an HTML selection is here:

```
<div class="form-group">
  <label for="calDob">Created By:</label>
  <select onchange={{action "changeCreatedBy" value="target.value"}}
class="form-control">
  {{#each allUsers as |user|}}
    <option value={{user.id}} selected={{eq editingTask.createdBy.id
user.id}}>{{user.name}}</option>
  {{/each}}
  </select>
</div>
```

The action method, `changeCreatedBy`, is listed here:

```
changeCreatedBy: function(_userId) {
  this.get("editingTask").set("createdBy",
this.get("allUsers").findBy("id", _userId));
},
```

Similarly, task priorities are also a list of integers from 1 to 10. The code to load them is here (this goes inside the controller):

```
taskPriorities: [],
  loadTaskPriorities: function() {
  for(var _idx=1; _idx<11; _idx++) {
    this.taskPriorities.pushObject(_idx);
  }
}.on("init"),
```

Code for the priority selection box is as follows:

```
<div class="form-group">
  <label for="selectPriority">Priority:</label>
  <select onchange={{action (mut editingTask.priority)
value="target.value"}} class="form-control">
  {{#each taskPriorities as |priority|}}
   <option value={{priority}} selected={{eq editingTask.priority
priority}}>{{priority}}</option>
  {{/each}}
  </select>
</div>
```

As a further step, you may add security to both ends of the application. You may personalize tasks for the logged-in user. Ember also supports WebSockets. Tasks can be pushed to the client as they are assigned to the logged-in user by another user somewhere else. For simplicity, those advanced features are not covered in this chapter. However, with the knowledge you have gained in this and the previous chapters, you are already at a comfortable stage to implement end-to-end security and real-time updates using WebSockets inside Taskify.

Summary

This chapter introduced the concept of single-page applications and implemented a Taskify frontend as an SPA, connecting to the Spring-based API server on the backend. We got a fair understanding of Ember.js and its tools as we built our frontend. Spring and Ember have together simplified the building of an otherwise complex rich web application of this type. The use of Ember is just an illustration of how Spring can power the backend of modern SPAs. Spring powers SPAs built on other frameworks, such as Angular, React, and Backbone, created by teams across the globe.

So far, we have successfully covered the most important features of Spring Framework. This foundation enables you to venture into more advanced features of Spring, packaged as Spring portfolio projects. Projects such as Spring Integration, Spring AMQP, Spring Cloud, and Spring Web Services solve the more complex problems of enterprise computing. With the knowledge you have gained from this book, you can now design powerful solutions using Spring Framework and its subprojects.

Chapter 7. Integrating with Other Web Frameworks

The flexibility offered by Spring Framework to pick third-party products is one of the core value propositions of Spring and Spring supports integration with third-party presentation frameworks. While Spring's presentation layer framework—Spring MVC, brings the maximum extent of flexibility and efficiency to the development of web applications, Spring lets you integrate most popular presentation frameworks.

Spring can be integrated with far too many of Java's web frameworks to be included in this chapter, and only the most popular ones, JSF and Struts, will be explained.

Spring’s JSF integration

A JSF web application can be easily integrated with Spring by loading a Spring context file within `web.xml` (through a context loader listener). Since JSF 1.2, Spring’s `SpringBeanFacesELResolver` object reads Spring beans as JSF managed beans. JSF only deals with the presentation tier and has a controller named `FacesServlet`. All we need to do is register `FacesServlet` in the application deployment descriptor or `web.xml` (in this section, we use JavaConfig to register it) and map any request with the desired extension (`.xhtml` here) to go through `FacesServlet`.

First, we should include the JSF API and its implementation in the project dependencies:

```
<properties>
  <spring-framework-version>4.1.6.RELEASE</spring-framework-version>
  <mojarra-version>2.2.12</mojarra-version>
</properties>
  ...
<dependency>
  <groupId>com.sun.faces</groupId>
  <artifactId>jsf-api</artifactId>
  <version>${mojarra-version}</version>
</dependency>
<dependency>
  <groupId>com.sun.faces</groupId>
  <artifactId>jsf-impl</artifactId>
  <version>${mojarra-version}</version>
</dependency>
...
```

The dispatcher Servlet initializer is the location to register `FacesServlet`. Notice that we set a mapping request to `FacesServlet` here. Since we use JavaConfig to register settings, we register `FacesServlet` in the `AnnotationConfigDispchServletInit` class, as follows:

```
@Configuration
@Order(2)
public class AnnotationConfigDispchServletInit extends
AbstractAnnotationConfigDispatcherServletInitializer {
  @Override
  protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[] { AppConfig.class };
  }
  @Override
  protected Class<?>[] getServletConfigClasses() {
    return null;
  }
  @Override
  protected String[] getServletMappings() {
    return new String[] { "*.xhtml" };
  }
  @Override
  protected Filter[] getServletFilters() {
    return new Filter[] { new CharacterEncodingFilter() };
  }
  @Override
  public void onStartup(ServletContext servletContext) throws
ServletException {
    // Use JSF view templates saved as *.xhtml, for use with // Facelets
    servletContext.setInitParameter("javax.faces.DEFAULT_SUFFIX",
".xhtml");
    // Enable special Facelets debug output during development
    servletContext.setInitParameter("javax.faces.PROJECT_STAGE",
"Development");
    // Causes Facelets to refresh templates during development
    servletContext.setInitParameter("javax.faces.FACELETS_REFRESH_PERIOD",
"1");
    servletContext.setInitParameter("facelets.DEVELOPMENT", "true");
    servletContext.setInitParameter("javax.faces.STATE_SAVING_METHOD",
"server");
    servletContext.setInitParameter(
      "javax.faces.PARTIAL_STATE_SAVING_METHOD", "true");

servletContext.addListener(com.sun.faces.config.ConfigureListener.class);
    ServletRegistration.Dynamic facesServlet =
servletContext.addServlet("Faces Servlet", FacesServlet.class);
    facesServlet.setLoadOnStartup(1);
    facesServlet.addMapping("*.xhtml");
    // Let the DispatcherServlet be registered
    super.onStartup(servletContext);
  }
}
```

Note

We must set `FacesServlet` to start up on load prior to the others (notice `facesServlet.setLoadOnStartup`).

Another important setting is configuring the listener to read the `faces-config` XML file. By default, it looks for `faces-config.xml` under the `WEB-INF` folder. By setting `org.springframework.web.jsf.el.SpringBeanFacesELResolver` as `ELResolver`, we access Spring POJOs as JSF beans. By registering `DelegatingPhaseListenerMulticaster`, any Spring's bean that implements the `PhaseListener` interface, JSF's phase events will be broadcasted to corresponding implemented methods of `PhaseListener` in the Spring's bean.

Here is the `faces-config.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee/web-
facesconfig_2_2.xsd"
version="2.2">
  <application>
    <el-
resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-
resolver>
  </application>
  <lifecycle>
```

```
    <phase-
listener>org.springframework.web.jsf.DelegatingPhaseListenerMulticaster</ph
ase-listener>
  </lifecycle>
</faces-config>
```

In JSF, we can define beans with a session, request, or application scope and the bean values retained within the specific scope. Setting the `eager` flag to `false` implies lazy initialization, which creates beans when the first request arrives, whereas `true` implies creating the beans on startup. The code for the `OrderBean` class is:

```
@ManagedBean(name = "orderBean", eager = true)
@RequestScoped
@Component
public class OrderBean {
  private String orderName;
  private Integer orderId;

  @Autowired
  public OrderServiceorder Service;
  public String placeAnOrder(){
    orderName=orderService.placeAnOrder(orderId);
    return "confirmation";
  }

  public String getOrderName() {
    return orderName;
  }
  public void setOrderName(String orderName) {
    this.orderName = orderName;
  }
  public Integer getOrderId() {
    return orderId;
  }
  public void setOrderId(Integer orderId) {
    this.orderId = orderId;
  }

}
```

Also, these beans are available in the presentation layer to interact with the backend. On the first screen (`order.xhtml`), we call the bean's method (`placeAnOrder`):

```
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
  <h3>input: JSF 2 and Spring Integration</h3>
    <h:form id="orderForm">
      <h:outputLabel value="Enter order id:" />
      <h:inputText value="#{orderBean.orderId}" /> <br/>
      <h:commandButton value="Submit" action="#{orderBean.placeAnOrder}"/>
    </h:form>
  </h:body>
</html>
```

The method returns a confirmation as a string and specify navigation in the `action` attribute means the next page is `confirmation.xhtml`, which looks like this:

```
<html lang="en"
xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
  <h:body>
  <h3>Confirmation of an order</h3>
  Product Name: #{orderBean.orderName}
  </h:body>
</html>
```

Spring's Struts integration

Spring MVC relies on `DispatcherServlet`, which sends requests to controllers that are configurable mapping handlers with view and theme resolution. In Struts, the controller's name is `Action`. While `Action` instances will be instantiated for every request in Struts 2 to tackle the thread safety issue, Spring MVC creates controllers once, and each controller's instance serves all requests.

To enable Spring integration with Struts 2, Struts provides `struts2-spring-plugin`. In Struts 2.1, Struts introduced the convention plugin (`struts2-convention-plugin`), which simplified the creation of `Action` classes (by annotation) without any configuration file (`struts.xml`). The plugin expects a set of naming conventions for the `Action` class, package, and view naming that will be explained in this section.

To integrate Struts 2 with Spring, you need to add these dependencies:

```
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-core</artifactId>
  <version>2.3.20</version>
</dependency>
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-spring-plugin</artifactId>
  <version>2.3.20</version>
</dependency>
<dependency>
  <groupId>org.apache.struts</groupId>
  <artifactId>struts2-convention-plugin</artifactId>
  <version>2.3.20</version>
</dependency>
```

The `struts2-convention-plugin` plugin searches for packages with the strings "struts", "struts2", "action", or "actions", and detects `Action` classes either whose names end with `Action` (`*Action`) or who implement the interface `com.opensymphony.xwork2.Action` (or extend its subclass `com.opensymphony.xwork2.ActionSupport`). The code for the `ViewOrderAction` class is as follows:

```
package com.springessentialsbook.chapter7.struts;
...
@Action("/order")
@ResultPath("/WEB-INF/pages")
@Result(name = "success", location = "orderEntryForm.jsp")
public class ViewOrderAction extends ActionSupport {
  @Override
  public String execute() throws Exception {
    return super.execute();
  }
}
```

`@Action` maps `/order` (in the request URL) to this action class and `@ResultPath` specifies where views (JSP files) exist. `@Result` specifies navigation to the next page up to the string value of the `execute()` method. We created `ViewOrderAction` to be able to navigate to a new page and to perform an action (business logic) when submitting a form within a view (`orderEntryForm.jsp`):

```
package com.springessentialsbook.chapter7.struts;
…...
@Action("/doOrder")
@ResultPath("/WEB-INF/pages")
@Results({
  @Result(name = "success", location = "orderProceed.jsp"),
  @Result(name = "error", location = "failedOrder.jsp")
})
public class DoOrderAction extends ActionSupport {
  @Autowired
  private OrderService orderService;
  private OrderVO order;

  public void setOrder(OrderVO order) {
    this.order = order;
  }

  public OrderVO getOrder() {
    return order;
  }

  @Override
  public String execute( ) throws Exception {
    if ( orderService.isValidOrder(order.getOrderId())) {
      order.setOrderName(orderService.placeAnOrder(order.getOrderId()));
      return SUCCESS;
    }
    return ERROR;
  }
```

Also, here is the JSP code that calls the `Action` class. Notice the form's `doOrder` action, which calls the `DoOrderAction` class (using `@Action("doOrder")`).

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <div align="center">
      <h1>Spring and Struts Integration</h1>
      <h2>Place an order</h2>
      <s:form action="doOrder" method="post">
        <s:textfield label="OrderId" name="order.orderId" />
        <s:submit value="Order" />
      </s:form>
    </div>
  </body>
</html>
```

As you can see, we used `OrderVO`, whose code is as follows, as the data model in the view. Any changes to this object in the JSP code or action class will be carried forward to the next page:

```
public class OrderVO {
  private String orderName;
  private String orderId;

  public String getOrderName() {
    return orderName;
  }
  public void setOrderName(String orderName) {
    this.orderName = orderName;
  }
  public String getOrderId() {
    return orderId;
  }
  public void setOrderId(String orderId) {
    this.orderId = orderId;
  }
```

In the `DoOrderAction` action class, in the method execution, we implement the business logic and return the string value of the method specified in the navigation logic in the presentation layer. Here, the action class either goes to `orderProceed.jsp` (if it is a valid order) or `failedOrder.jsp` (in the case of a failure). Here is the orderProceed.jsp page, to which a success order will be forwarded:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <div align="center">
      <h1>Order confirmation</h1>
      <s:label label="OrderId" name="order.orderId" />, <s:label
label="OrderName" name="order.orderName" /> <br/>
      has been successfully placed.
    </div>
  </body>
</html>
```

Summary

In this chapter, we explained how to integrate Spring with two famous presentation technologies: JSF and Struts.

You can get more info about Spring's integration with web frameworks here:

http://docs.spring.io/spring/docs/current/spring-framework-reference/html/web-integration.html

To know more about Spring's Struts plugin, visit this link:

http://struts.apache.org/docs/spring-plugin.html

You can get more details about naming conventions in the Struts convention plugin here:

https://struts.apache.org/docs/convention-plugin.html

Nowadays, big companies are shifting toward single-page applications in the presentation layer. To learn about this topic, read Chapter 6, *Building a Single-Page Spring Application*.

Index