

THE EXPERT'S VOICE® IN SPRING

Covers
Spring
Framework 4.0

Spring Recipes

A Problem-Solution Approach

THIRD EDITION

Marten Deinum
with Daniel Rubio, Josh Long, and Gary Mak

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Authors.....	xxxiii
About the Technical Reviewer	xxxv
Acknowledgments	xxxvii
Introduction	xxxix
■ Chapter 1: Spring Development Tools.....	1
■ Chapter 2: Spring Core Tasks	47
■ Chapter 3: Spring Annotation Driven Core Tasks.....	135
■ Chapter 4: Spring @MVC	217
■ Chapter 5: Spring REST	275
■ Chapter 6: Spring Social.....	303
■ Chapter 7: Spring Security	331
■ Chapter 8: Spring Mobile.....	385
■ Chapter 9: Spring with Other Web Frameworks	401
■ Chapter 10: Data Access	419
■ Chapter 11: Spring Transaction Management	475
■ Chapter 12: Spring Batch	511
■ Chapter 13: NoSQL and BigData	549
■ Chapter 14: Spring Java Enterprise Services and Remoting Technologies	591
■ Chapter 15: Spring Messaging	659

■ Chapter 16: Spring Integration	691
■ Chapter 17: Spring Testing	723
■ Chapter 18: Grails.....	757
Index.....	799

Introduction

The Spring framework is growing. It has always been about choice. Java EE focused on a few technologies, largely to the detriment of alternative, better solutions. When the Spring framework debuted, few would have agreed that Java EE represented the best-in-breed architectures of the day. Spring debuted to great fanfare, because it sought to simplify Java EE. Each release since marks the introduction of new features designed to both simplify and enable solutions.

With version 2.0 and later, the Spring framework started targeting multiple platforms. The framework provided services on top of existing platforms, as always, but was decoupled from the underlying platform wherever possible. Java EE is still a major reference point, but it's not the only target. Additionally, the Spring framework runs on different Cloud environments. With the introduction of Java based configuration and more XML schemas, the Spring framework created powerful configuration options. Frameworks built on top of the Spring framework have emerged supporting application integration, batch processing, messaging, and much more.

This is the 3rd edition of the superb Spring Recipes and it contains mostly updated frameworks, describing the new features and explaining the different configuration (Java and/or XML) options. Additionally, new projects have been added to the Spring ecosystem like the Spring Data family of products.

It was impossible to describe each and every project in the Spring ecosystem, so we had to decide what to keep, what to add, and what to update. This was a hard decision but we think we have included the most important and used content.

Who This Book Is For

This book is for Java developers who want to simplify their architecture and solve problems outside the scope of the Java EE platform. If you are already using Spring in your projects, the more advanced chapters present discussions of newer technologies that you might not know about already. If you are new to the framework, this book will get you started in no time.

This book assumes that you have some familiarity with Java and an IDE of some sort. While it is possible, and indeed useful, to use Java exclusively with client applications, Java's largest community lives in the enterprise space and that, too, is where you'll see most of these technologies deliver the most benefit. Thus, some familiarity with basic enterprise programming concepts like the Servlet API is assumed.

How This Book Is Structured

Chapter 1, "Spring Development Tools," gives an overview of tools supporting the Spring framework and how to use them.

Chapter 2, "Spring core tasks," gives a general overview of the Spring framework: how to set it up, what it is, and how it's used.

Chapter 3, "Spring Annotation Driven Core Task," reviews, in addition to Chapter 2 more annotation driven concepts that are still key to fully exploiting the container.

Chapter 4, "Spring @MVC," covers web-based application development using the Spring Web MVC framework.

Chapter 5, "Spring REST," provides an introduction to Spring's support for RESTful web services.

Chapter 6, "Spring Social," provides an introduction of Spring Social, which lets you integrate easily with social networks.

Chapter 7, "Spring Security," provides an overview of the Spring Security project, to help you better secure your application.

Chapter 8, "Spring Mobile," provides an introduction of Spring Mobile, which lets you integrate Mobile device detection and usage in your application.

Chapter 9, “Integrating Spring with Other Web Frameworks,” introduces the core web-tier support that Spring provides. This provides a base for all technologies that Spring provides in the web tier.

Chapter 10, “Data Access,” discusses using Spring to talk to data stores using APIs like JDBC, Hibernate, and JPA.

Chapter 11, “Transaction Management in Spring,” introduces the concepts behind Spring’s robust transaction management facilities.

Chapter 12, “Spring Batch,” introduces the Spring Batch framework, which provides a way to model solutions traditionally considered the domain of mainframes.

Chapter 13, “NoSQL and BigData,” an introduction to multiple Spring Data portfolio projects, covering different NoSQL technologies and BigData with Hadoop.

Chapter 14, “Spring Java Enterprise Services and Remoting Technologies,” introduces you to the JMX support, scheduling, e-mail support, and various facilities for RPC, including the Spring Web Services project.

Chapter 15, “Spring Messaging,” discusses using Spring with message-oriented middleware through JMS and RabbitMQ and the simplifying Spring abstractions.

Chapter 16, “Spring Integration,” discusses using the Spring Integration framework to integrate disparate services and data.

Chapter 17, “Spring Testing,” discusses unit testing with the Spring framework.

Chapter 18, “Grails,” discusses the Grails framework, with which you can increase your productivity by using best-of-breed pieces and gluing them together with Groovy code.

Appendix A, “Deployment to the Cloud,” shows how to deploy a Java (Web) application to the cloud using the Pivotal’s CloudFoundry solution.

Appendix B, “Spring Caching,” introduces the Spring Caching abstraction, including how to configure it and how to transparently add caching to your application.

Conventions

Sometimes, when we want you to pay particular attention to a part within a code example, we will make the font bold. Please note that the bold doesn’t necessarily reflect a code change from the previous version.

In cases when a code line is too long to fit the page’s width, we will break it with a code continuation character.

Please note that when you try to type the code, you have to concatenate the line by yourself without any spaces.

Prerequisites

Because the Java programming language is platform independent, you are free to choose any supported operating system. However, some of the examples in this book use platform-specific paths. Translate them as necessary to your operating system’s format before typing the examples.

To make the most of this book, install JDK version 1.7 or higher. You should have a Java IDE installed to make development easier. For this book, the sample code is Gradle based. If you’re running Eclipse and Install the Gradle plug-in, you can open the same code in Eclipse and the CLASSPATH and dependencies will be filled in by the Gradle metadata.

If you’re using Eclipse, you might prefer SpringSource’s SpringSource Tool Suite (STS), as it comes preloaded with the plug-ins you’ll need to be productive with the Spring framework in Eclipse. If you use IntelliJ IDEA, you need to enable the Gradle (and Groovy) plugins.

Downloading the code

The source code for this book is available from the Apress web site (www.apress.com) in the Source Code / Download section. The source code is organized by chapters, each of which includes one or more independent examples.

Contacting the Authors

We always welcome your questions and feedback regarding the contents of this book. You can contact Marten Deinum at marten@deinum.biz

CHAPTER 1



Spring Development Tools

In this chapter you'll learn how to setup and work with the most popular development tools to create Spring applications. Like many other software frameworks, Spring has a wide array of development tools to choose from, from bare-bones command line tools to sophisticated graphical tools the software industry calls IDEs (Integrated Development Environments).

Whether you already use certain Java development tools or are a first-time developer, the following recipes will guide you on how to set up different toolboxes to do the exercises in the upcoming chapters, as well as develop any Spring application.

Table 1-1 describes a list of toolboxes and the corresponding recipes you need to follow to get the right tools to start a Spring application.

Table 1-1. Toolboxes to develop Spring applications

Toolbox A	Toolbox B	Toolbox C	Toolbox D
Spring Tool Suite	Eclipse IDE	IntelliJ IDE	Text editor
Recipe 1-1	Recipe 1-2	Recipe 1-3 w/Maven CLI Recipe 1-4 w/Gradle CLI Recipe 1-5	w/Maven CLI Recipe 1-4 w/Gradle CLI Recipe 1-5

Bear in mind you don't need to install every toolbox described in Table 1-1 to work with Spring. It can be helpful to try them all out, so you can use the toolbox you feel most comfortable with.

1-1. Build a Spring application with the Spring Tool Suite

Problem

You want to use the Spring Tool Suite (STS) to build a Spring application.

Solution

Install STS on your workstation. Open STS and click the 'Open Dashboard' link. To create a new Spring application, click on the 'Spring project' link on the dashboard window inside the 'Create' table. To open a Spring application that uses Maven, from the top level 'File' menu select the 'Import...' option, click on the 'Maven' icon and select the 'Existing Maven projects'. Next, select the Spring application based on Maven from your workstation.

To install Gradle on STS, click on the ‘Extensions’ tab at the bottom of the dashboard window. Click on the ‘Gradle Support’ checkbox. Proceed with the Gradle extension installation and restart STS once the installation is complete. To open a Spring application that uses Gradle, from the top level ‘File’ menu select the ‘Import...’ option, click on the ‘Gradle’ icon and select the ‘Gradle project’. Next, select the Spring application based on Gradle from your workstation. Click on the ‘Build Model’ button and last click ‘Finish’ to start working on the project.

How It Works

STS is the IDE developed by SpringSource -- a division of Pivotal -- creators of the Spring framework. STS is specifically designed to develop Spring applications, making it one of the most complete tools for this purpose. STS is an Eclipse-powered tool, so it has the same ‘look and feel’ as the Eclipse open source IDE.

STS can be downloaded for free from <http://spring.io/tools/sts>. STS is available for all six major operating system (OS) versions: Windows, Windows (64bit), Mac OS X (Cocoa), Mac OS X (Cocoa, 64bit), Linux (GTK), and Linux (GTK, 64bit). STS is also distributed in two versions of Eclipse, the Eclipse 3.x branch and the Eclipse 4.x branch. In addition, STS is itself versioned, so you have the option to download the latest stable release or a milestone/development version.

Download the version suited to your OS and I recommend you chose the Eclipse 4.x branch because it’s newer. At the time of this writing the latest stable release of the Spring Tool Suite is the 3.5 version. Therefore the download link you chose should have the title ‘SPRING TOOL SUITE 3.5.1.RELEASE - BASED ON ECLIPSE 4.3.2’ or you can choose a newer release if you prefer.

Once you download STS, ensure you have a Java SDK installed on your system since this is an STS installation requirement. Proceed to install STS. Follow the installation wizard and you should have STS setup in 5 to 10 minutes. Upon termination, a folder with the name `STS_<VERSION>` is created under the home folder of the user making the installation or where the user chooses to place the installation-based folder. If you inspect this folder, you’ll see the STS executable which is used to start STS.

Start STS. At startup, STS asks you to define a workspace location. A workspace is where STS places all project information. You can keep the default directory which is under the main STS installation directory or define a different directory to your liking. After startup is complete you’ll see a screen like the one in Figure 1-1.

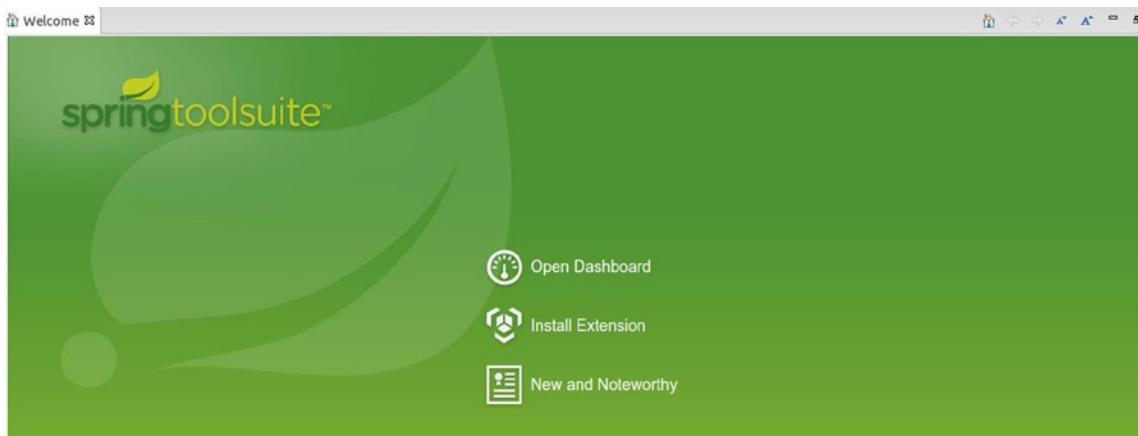


Figure 1-1. STS startup screen

Click on the ‘Open Dashboard’ link. Then you’ll see the STS Dashboard illustrated in Figure 1-2.

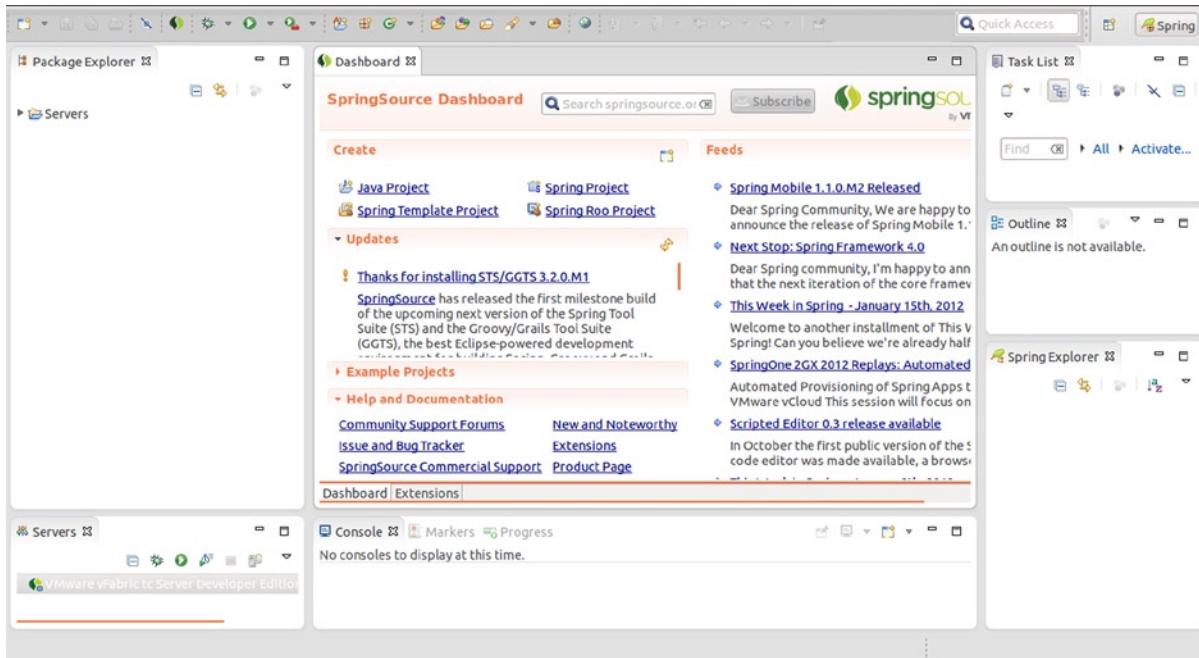


Figure 1-2. STS Dashboard

On the STS Dashboard, in the center column inside the ‘Create’ box there’s a link called ‘Spring project’. You can click on this link to create a new Spring application. You can go ahead and create an empty application if you like. You’ll be asked for a name and to define a series of parameters which you can leave with default values.

A more common case than creating a Spring application from scratch is to continue development on a pre-existing Spring application. Under such circumstances, the owner of an application generally distributes the application’s source code with a build script to facilitate its ongoing development.

The build script of choice for most Java application is a `pom.xml` file designed around the build tool called Maven and more recently a `build.gradle` file designed around the build tool called Gradle. The book’s source code and its applications are provided with Gradle build files, in addition to a single application with a Maven build file.

JAVA BUILD TOOLS, A MEANS TO AN END: ANT, MAVEN, IVY, GRADLE

In a Java application there can be dozens or hundreds of menial tasks required to put together an application (e.g., Copying JARs or configuration files, setting up Java’s classpath to perform compilation, downloading JAR dependencies, etc.). Java build tools emerged to perform such tasks in Java applications.

Java build tools continue to have their place because applications distributed with build files ensure that all menial tasks intended by the creator of an application are replicated exactly by anyone else using the application. If an application is distributed with an Ant `build.xml` file, a Maven `pom.xml` file, an Ivy `ivy.xml` file or a Gradle `build.gradle` file, each of these build files guarantees build consistency across users and different systems.

Some of the newer Java build tools are more powerful and enhance the way their earlier counterparts work and each build file uses its own syntax to define actions, dependencies, and practically any other task required to build an application. However, you should never lose sight of the fact that a Java build tool is just a means to an end. It's a choice made by the creator of an application to streamline the build process.

Don't panic if you see an application distributed with a build file from the oldest Ant version or the newest Gradle version, from an end user perspective all you need to do is download and install the build tool to create the application as its creator intended.

Since many Spring applications continue to use Maven and some of the newer Spring applications use Gradle, I'll describe the import process into STS for both types of projects.

Once you download the book's source and unpack it to a local directory, click on the STS top level 'File' menu and select the 'Import...' option. A pop-up window appears. In the pop-up window, click on the 'Maven' icon and select the 'Existing Maven Projects' option as illustrated in Figure 1-3.

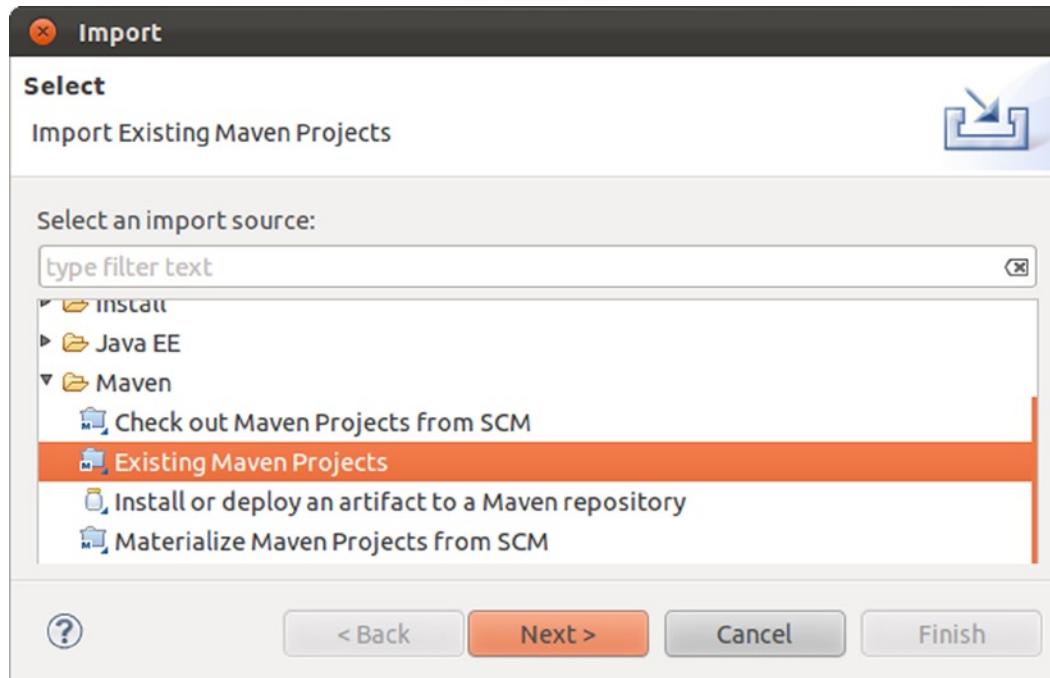


Figure 1-3. STS maven import

Next, click on the ‘Next’ button. In the following screen, on the ‘Select root directory’ line click on the ‘Browse’ button and select the directory of the book’s source code in Ch1 called `springintro_mvn` as illustrated in the Figure 1-4.

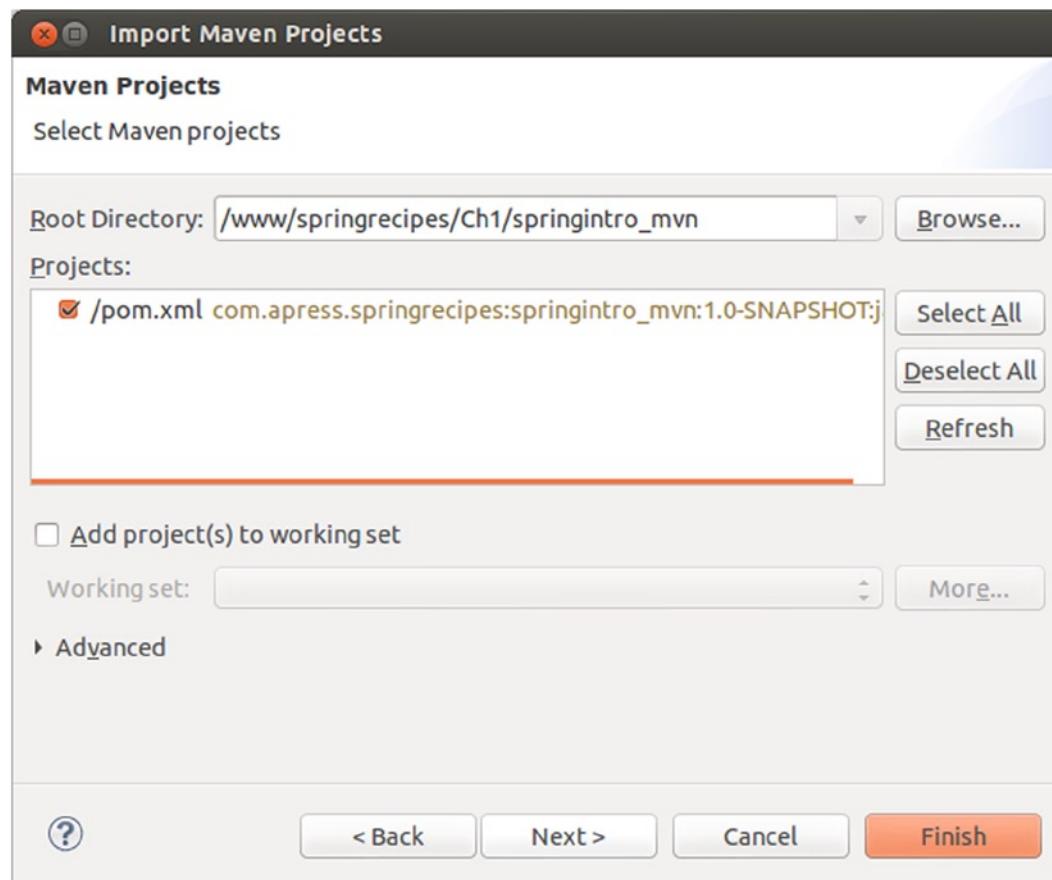


Figure 1-4. STS select maven project

Notice in Figure 1-4 the ‘Projects:’ window is updated to include the line `pom.xml com.apress.springrecipes...` which reflects the Maven project to import. Select the project checkbox and click on the ‘Finish’ button to import the project. All projects in STS are accessible on the left-hand side in the ‘Package Explorer’ window. In this case, the project appears with the name `springintro_mvn`.

If you click on the project icon in the ‘Package Explorer’ window, you’ll be able to see the project structure (i.e., java classes, dependencies, configuration files, etc.). If you double click on any of the project files inside the ‘Package Explorer’, the file is opened in a separate tab in the center window -- alongside the dashboard. Once a file is opened, you can inspect, edit, or delete its contents.

Select the project icon in the ‘Package Explorer’ window and click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Maven build’ option. A pop-up window appears to edit and configure the project build. Just click on the ‘Run’ button in the bottom right. In the bottom center of STS you’ll see the ‘Console’ window appear. In this case, the ‘Console’ window displays a series of build messages produced by maven, as well as any possible errors in case the build process fails.

You've just built the application, congratulations! Now let's run it. Select the project icon from the 'Package Explorer' window once again and press the F5 key to refresh the project directory. Expand the project tree. Toward the bottom you'll see a new directory called target which contains the built application. Expand the target directory by clicking on its icon. Next, select the file `springintro_mvn-1.0-SNAPSHOT.jar` as illustrated in Figure 1-5.

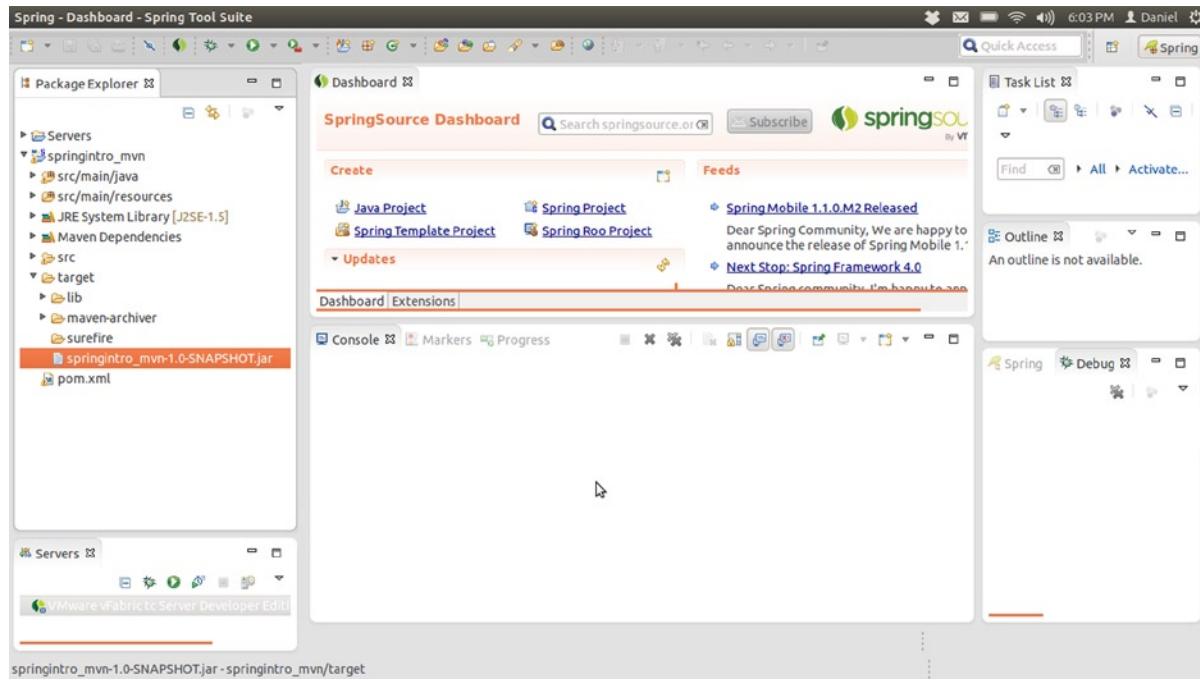


Figure 1-5. Select executable in STS

With the file selected, click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Run configurations...’ option. A pop-up window to edit and configure the run appears. Ensure the ‘Java application’ option is selected on the left-hand side. In the ‘Main class:’ box introduce: `com.apress.springrecipes.hello.Main`. This is the main class for this project, as illustrated in Figure 1-6.

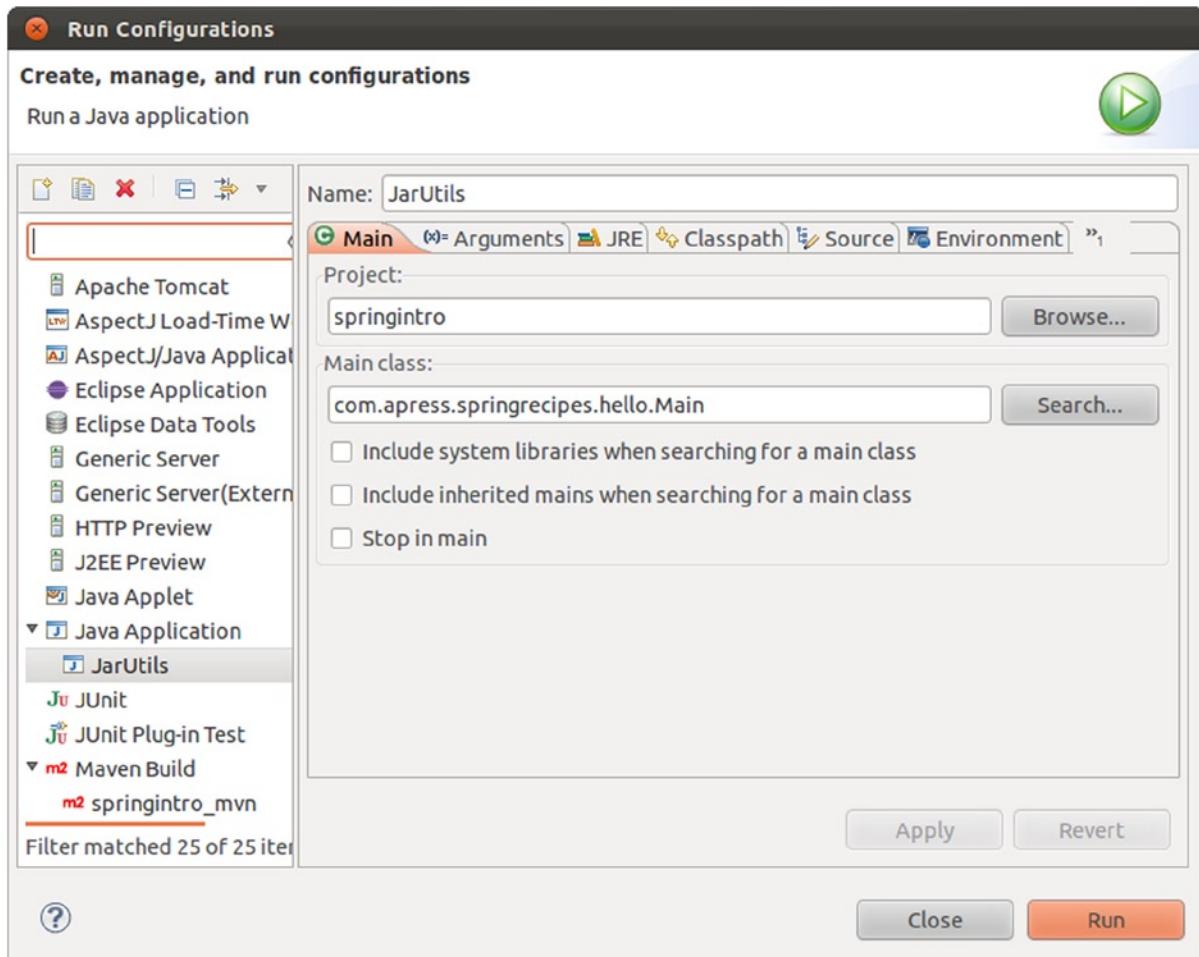


Figure 1-6. Define main executable class in STS

Click on the ‘Run’ button in the bottom right. In the bottom center of STS you’ll see the ‘Console’ window. In this case, the ‘Console’ window displays the application logging messages, as well as a greeting message defined by the application.

Even though you’ve built and run a Spring application with STS, we’re still not done. The process you just completed with STS was mostly done behind the scenes by the build tool called Maven. Next, it’s time to import a Spring application that uses one of the newer build tools call Gradle.

While gradle is still a relatively new tool, there are signs that gradle will supplant Maven in the future. For example, many large Java projects -- such as the Spring framework itself -- now use Gradle instead of maven due to its greater simplicity and power. Given this tendency, it’s a worth describing how to use Gradle with STS.

Tip If you have a Maven project (i.e., pom.xml file) you can use the bootstrap plugin or maven2gradle tool to create a Gradle project (i.e., build.gradle file). The bootstrap plugin is included with Gradle (See documentation at http://gradle.org/docs/current/userguide/bootstrap_plugin.html) and the maven2gradle tool is available at <https://github.com/jbaruch/maven2gradle.git>.

To install Gradle in STS go to the dashboard window. At the bottom of the dashboard window click on the ‘Extensions’ tab. A list of STS extensions is loaded in the window. Click on the ‘Gradle Support’ checkbox from the list as illustrated in Figure 1-7. If you don’t see the ‘Gradle Support’ extension in the list, type the word ‘gradle’ in the ‘Find’ textbox. If you still don’t see the ‘Gradle Support’ extension after this step, click on the ‘Find Updates’ button.

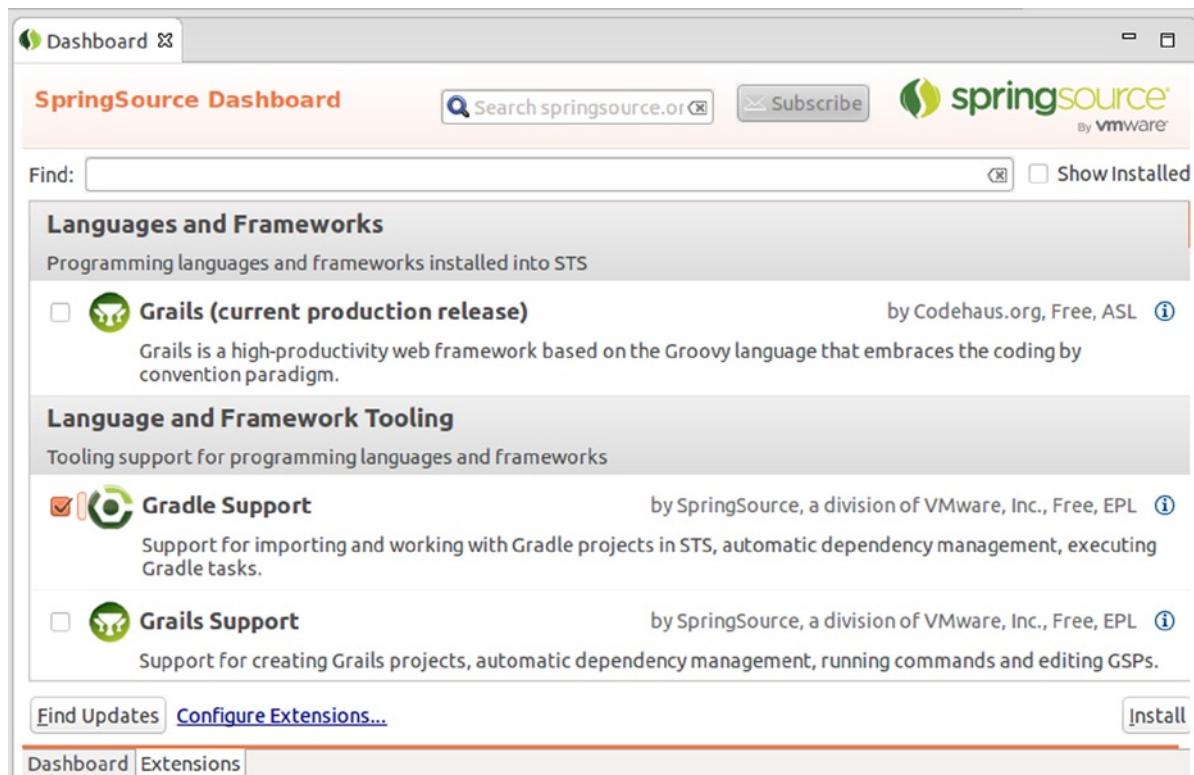


Figure 1-7. Gradle STS installation

Click on the ‘Next’ button in the bottom right to proceed with the Gradle extension installation. A confirmation pop-up window appears as illustrated in Figure 1-8.

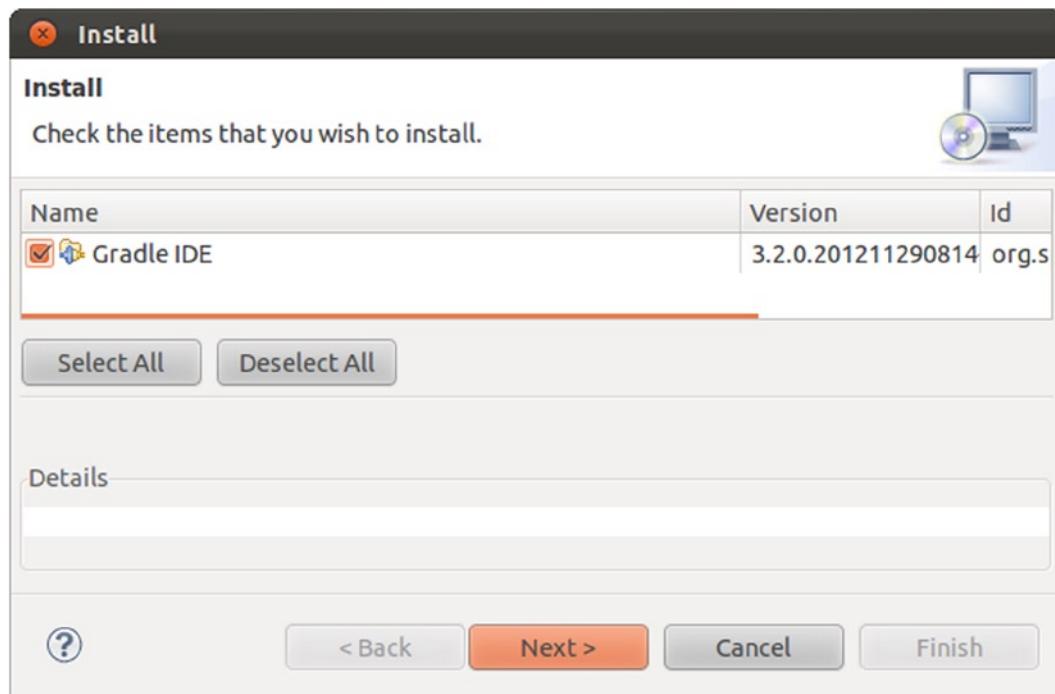


Figure 1-8. Gradle STS installation confirmation

Click on the pop-up window’s ‘Next’ button. Once you read the license and accept the terms click on the pop-up window’s ‘Finish’ button. The Gradle extension installation process starts. Once the installation process finishes, you’ll be prompted to restart STS for the changes to take effect. Confirm the STS restart to finish the Gradle installation.

The book's source contains numerous Spring applications designed to be built with Gradle, so I'll describe how to import these Spring applications into STS. Once you download the book's source and unpack it to a local directory, in STS click on the top level 'File' menu and select the 'Import...' option. A pop-up window appears. In the pop-up window, click on the 'Gradle' icon and select the 'Gradle Project' option as illustrated in Figure 1-9.

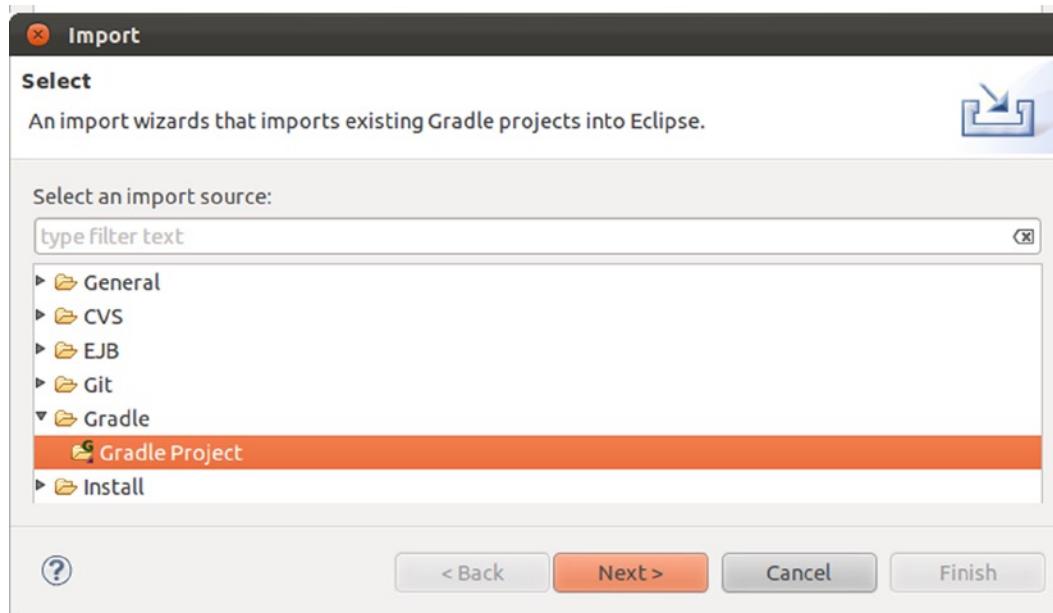


Figure 1-9. STS gradle import

Next, click on the 'Next' button. In the following screen, in the 'Select root directory' line click on the 'Browse' button and select the book's source code top level directory. Next, click on the 'Build model' button beside the 'Browse' button. The build model process retrieves the various Gradle subprojects contained in the book's source code. A pop-up window appears indicating the progress of the build model process. Once the build model process finishes, you'll see a list of Gradle projects, click on the project checkbox called 'springintro' inside 'Ch1' as illustrated in Figure 1-10.

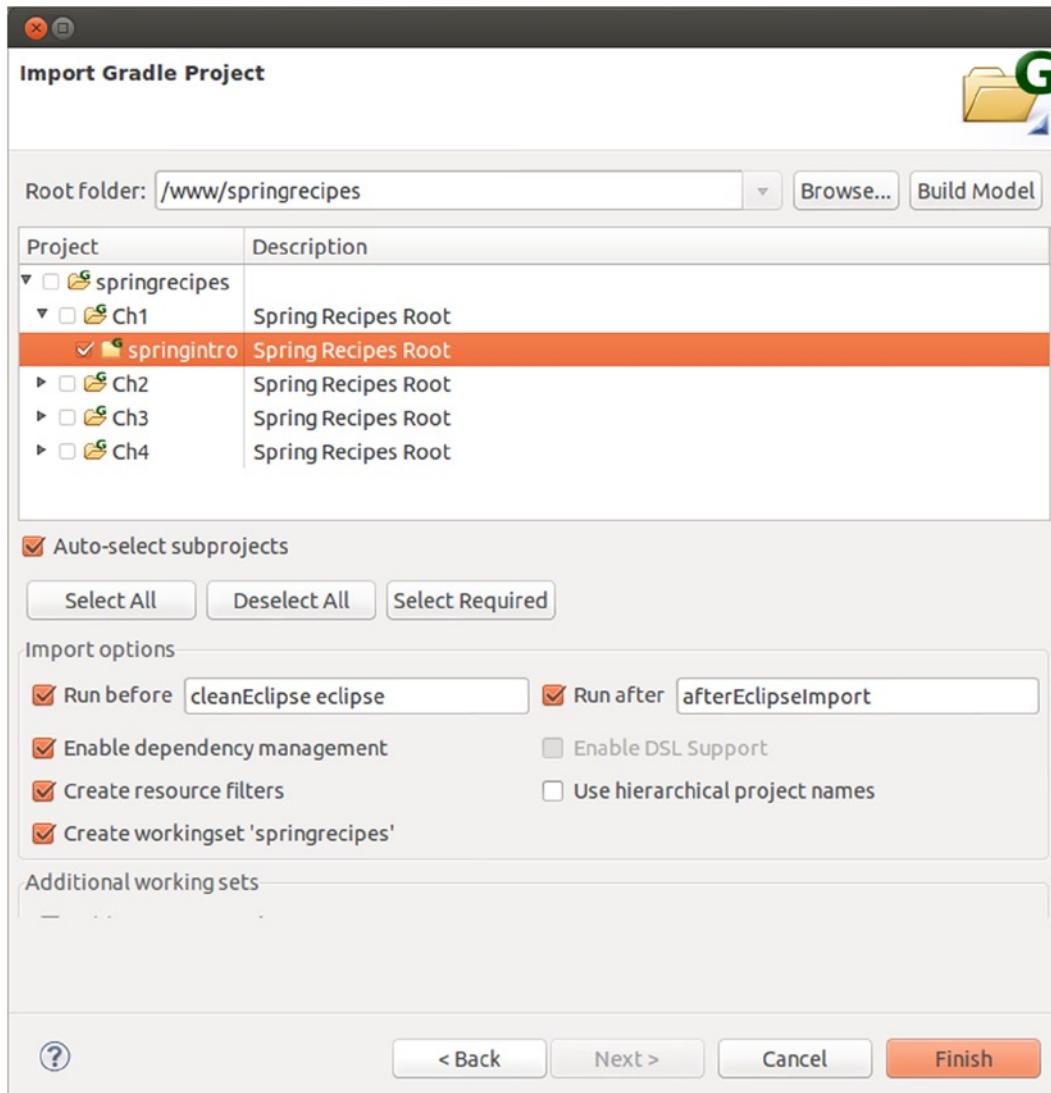


Figure 1-10. STS select gradle subprocess

Click on the ‘Finish’ button to import the project. If you look at the left-hand side of STS in the ‘Package Explorer’ window you’ll see the project is loaded with the name `springintro`. If you click on the project icon, you’ll be able to see the project structure (i.e., java classes, dependencies, configuration files, etc.).

Select the project icon and click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Gradle build’ option. A pop-up window appears to edit and configure the build. Click on the Project/task option ‘build’ and then click on the ‘Run’ button in the bottom right. In the bottom center of STS you’ll see the ‘Console’ window appear. In this case, the ‘Console’ window displays a series of build messages produced by gradle, as well as any possible errors in case the build process fails.

You’ve just built the application, now let’s run it. Select the project icon once again and press the F5 key to refresh the project directory. Expand the project tree. Toward the middle you’ll see a new directory called `libs` which contains the built application. Expand the `libs` directory by clicking on the icon. Next, select the file `springintro-1.0-SNAPSHOT.jar`.

With the file selected, from the top level menu ‘Run’ select the ‘Run configurations...’ option. A pop-up window appears to edit and configure the run. Ensure the ‘Java application’ option is selected in the left-hand side. In the ‘Main class:’ box introduce `com.apress.springrecipes.hello.Main`. This is the main class for this project. Click on the ‘Run’ button in the bottom right. In the bottom center of STS you’ll see the ‘Console’ window. In this case, the ‘Console’ window displays the application logging messages, as well as a greeting message defined by the application.

1-2. Build a Spring application with the Eclipse IDE Problem

You want to use the Eclipse IDE to build Spring applications.

Solution

From Eclipse’s top level ‘Help’ menu select the ‘Eclipse Marketplace...’. Install STS for Eclipse. To create a new Spring application, click on the top level ‘File’ menu select t ‘New’ and then the ‘Project’ . Next, click on the ‘Spring project’ option.

To open a Spring application that uses Maven, you first need to install Maven Integration for Eclipse (a.k.a. m2eclipse). From Eclipse’s top level ‘Help’ menu select the ‘Eclipse Marketplace...’ Install Maven Integration for Eclipse. Click on the Eclipse top level ‘File’ menu and select the ‘Import...’ option. A pop-up window appears. In the pop-up window, click on the ‘Maven’ icon and select the ‘Existing Maven Projects’ option. Next, seledct the Spring application based on Maven from your workstation.

To open a Spring application that uses Gradle, you first need to install Gradle Integration for Eclipse. From Eclipse’s top level ‘Help’ menu select the ‘Eclipse Marketplace...’ Install Maven Integration for Eclipse. Click on the the Eclipse top level ‘File’ menu and select the ‘Import...’ option. A pop-up window appears. In the pop-up window, click on the ‘Gradle’ icon and select the ‘Gradle Project’ option. Next, select the Spring application based on Gradle from your workstation.

How It Works

Eclipse is one of the most popular IDEs to develop Java applications, therefore it can be a natural choice to develop Spring applications. For this recipe I’ll assume you’ve already installed Eclipse and are familiar with its environment.

Tip If you haven’t installed Eclipse, I recommend you try out STS which is described in Recipe 1-1. STS is an Eclipse-powered tool -- meaning it has the same ‘look and feel’ as Eclipse -- but it’s a product made by the creators of the Spring framework -- SpringSource -- so it’s specifically designed to fulfill the needs of Spring application development.

Eclipse supports a wide variety of tools that make it easy to work with certain technologies directly in the Eclipse IDE. Some of these tools include support for version control technologies like svn and git, support for platform technologies like Python and Android and as you probably already guessed support for the Spring framework. Eclipse makes all these tools available in what it calls the Eclipse marketplace.

You can access the Eclipse marketplace at its website (<http://marketplace.eclipse.org/>) or directly in the Eclipse IDE from the top level ‘Help’ menu selecting the ‘Eclipse Marketplace...’ option as illustrated in Figure 1-11.

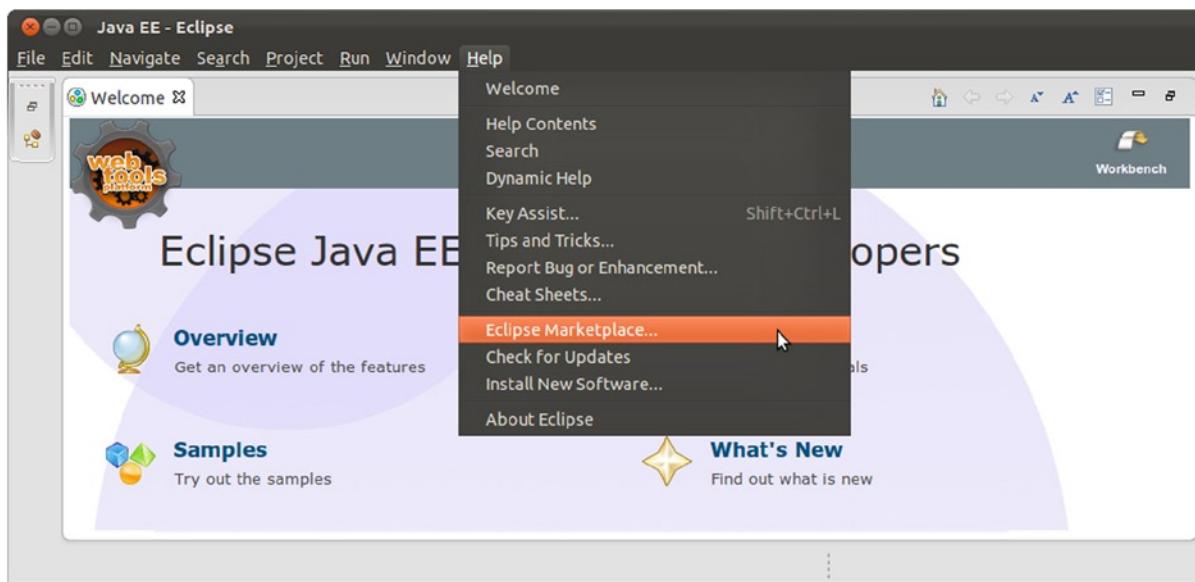


Figure 1-11. Eclipse Marketplace option

■ Warning If you don't see the ‘Eclipse Marketplace...’ option illustrated in Figure 1-11 it means your Eclipse version doesn't support the Eclipse Marketplace. The Eclipse Marketplace is included in all Eclipse packages available from the Eclipse download page, except the Eclipse classic package.

Once you select the ‘Eclipse Marketplace...’ option a pop-up window appears where you can access and search the Eclipse marketplace.

The Eclipse tool for the Spring framework is called 'STS for Eclipse.' In the pop-up window you can scroll through the main list or type 'sts' in the 'Find:' box to limit the main list as illustrated in Figure 1-12.

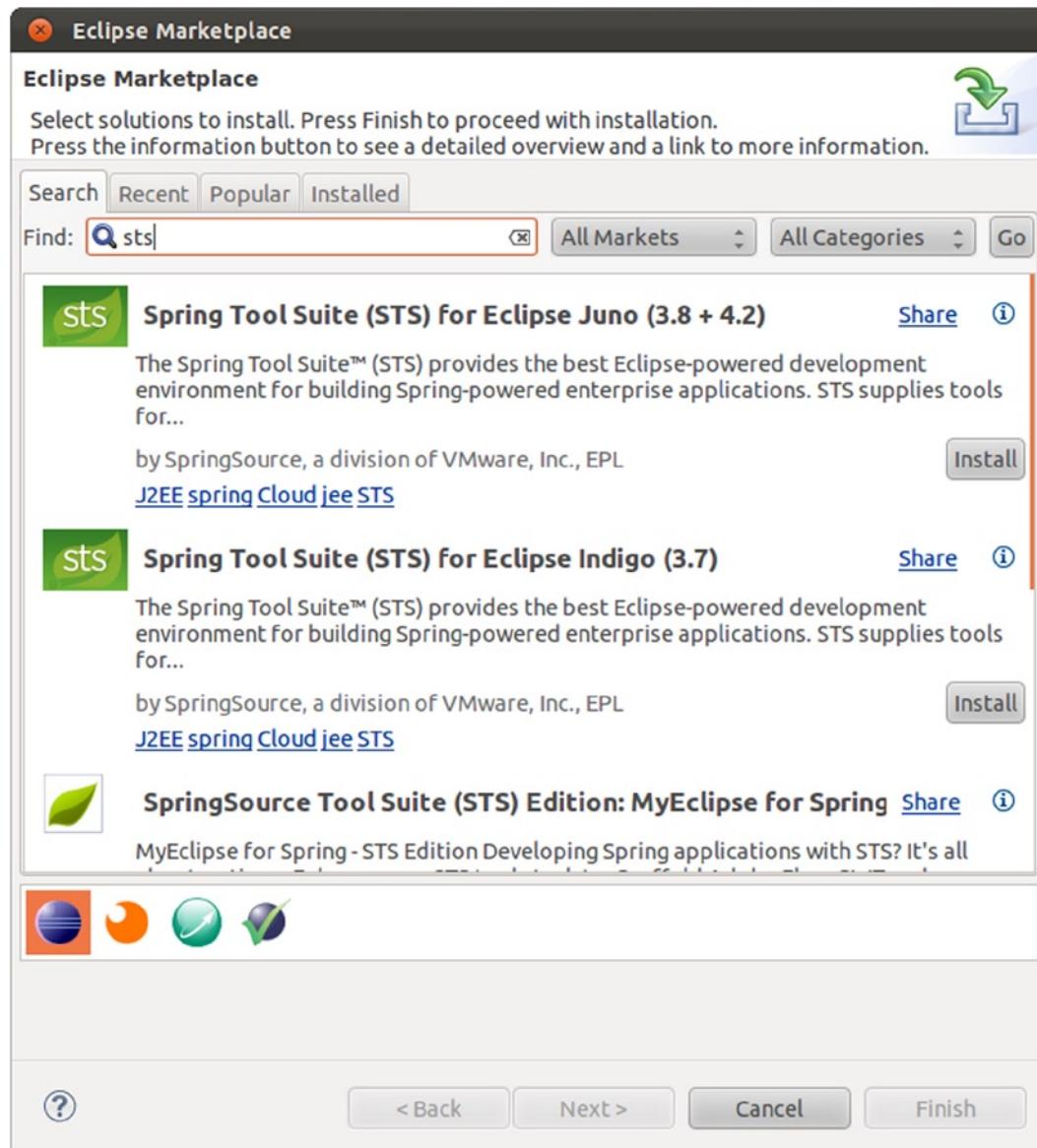


Figure 1-12. STS for Eclipse installation

Click on the 'Install' button alongside the STS for Eclipse tool to begin the process. Follow the installation wizard. Once you read the license and accept the terms click on the pop-up window's 'Finish' button. The STS for Eclipse tool installation process initiates. Once the installation process finishes, you'll be prompted to restart Eclipse for the changes to take effect. Confirm the Eclipse restart to finish the STS for Eclipse installation.

With the STS for Eclipse tool you gain access to several Spring specific features not available in a regular Eclipse installation. For example, to create a new Spring project from the top level 'File' menu select the 'New' and then the 'Project' option as illustrated in Figure 1-13.

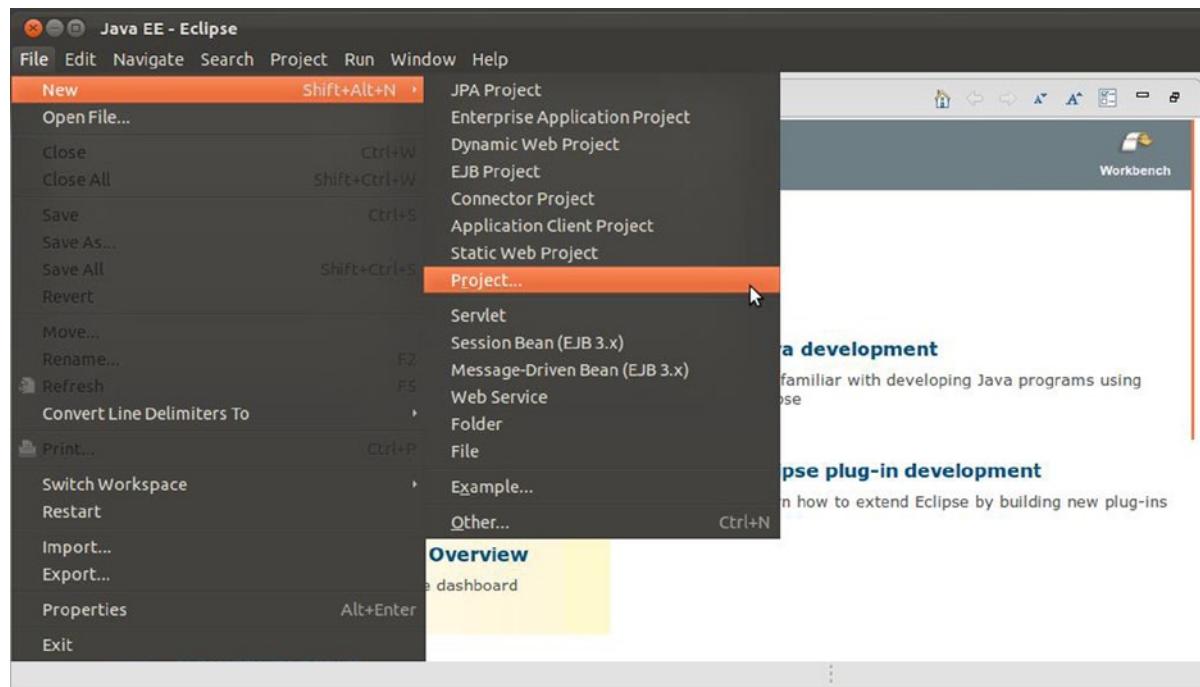


Figure 1-13. Create new Eclipse project

A pop-up window appears to create a new project. If you scroll toward the bottom you'll see the option to create a Spring project as illustrated in Figure 1-14. You can go ahead and create an empty application if you like, you'll be asked for a name and to define a series of parameters which you can leave with default values.

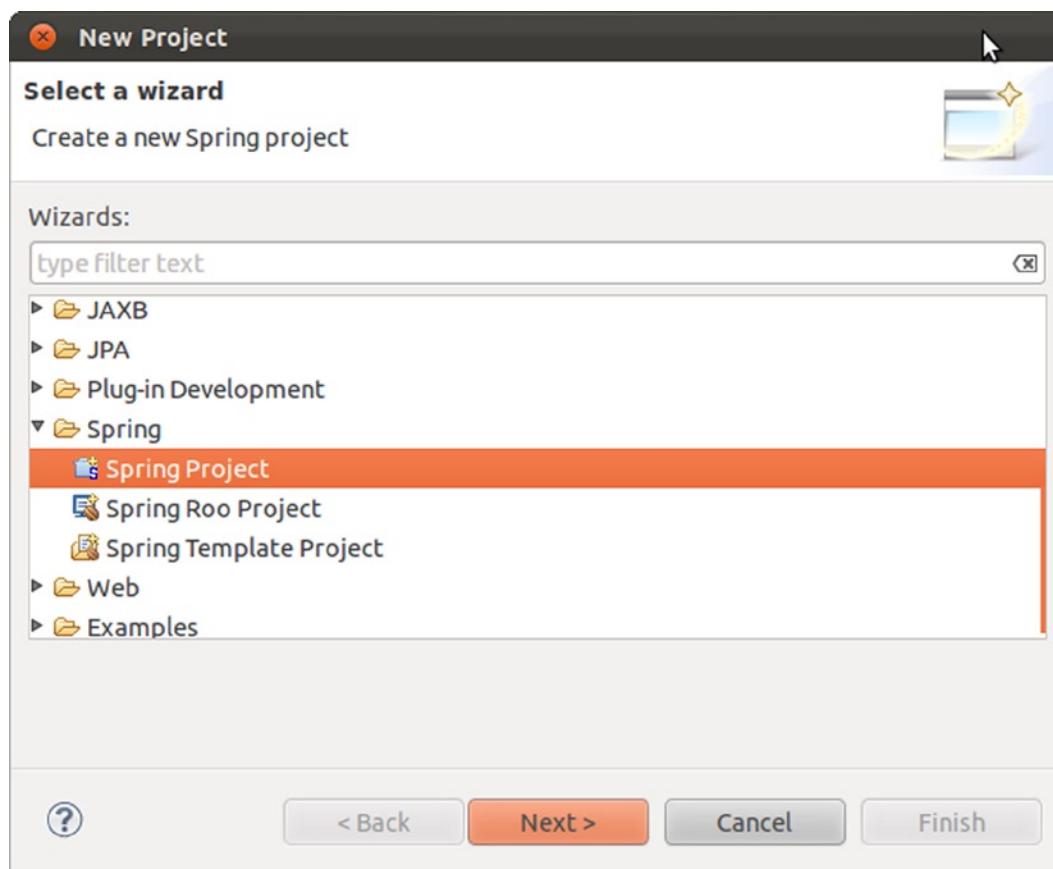


Figure 1-14. Create new Spring project in Eclipse

A more common case than creating a Spring application from scratch is to continue development of a pre-existing Spring application. Under such circumstances, the owner of an application generally distributes the application's source code with a build script to facilitate its ongoing development.

The build script of choice for most Java application is a `pom.xml` file designed around the build tool called Maven and more recently a `build.gradle` file designed around the build tool called Gradle. The book's source code and its applications are provided with Gradle build files, in addition to a single application with a Maven build file.

To access Spring applications distributed with Maven or Gradle build files, it's necessary to install additional Eclipse tools from the Eclipse Marketplace. Once again you'll need to use the Eclipse marketplace. From the top level 'Help' menu select the 'Eclipse Marketplace...' option as illustrated in Figure 1-11. A pop-up window appears where you can access the Eclipse marketplace.

In the ‘Find’ box type ‘maven integration’ and you’ll see the ‘Maven Integration for Eclipse’ in the main list as illustrated in Figure 1-15. Click on the ‘Install’ button and follow the installation wizard. Postpone the Eclipse restart request so you can also install Gradle. Go back to the Eclipse marketplace. In the ‘Find’ box type ‘gradle’ and you’ll see the ‘Gradle Integration for Eclipse’ in the main list as illustrated in Figure 1-16. Click on the ‘Install’ button and follow the installation wizard. You’ll be prompted to restart Eclipse for the changes to take effect. Confirm the Eclipse restart to finish the Maven Integration for Eclipse and Gradle Integration for Eclipse installations.

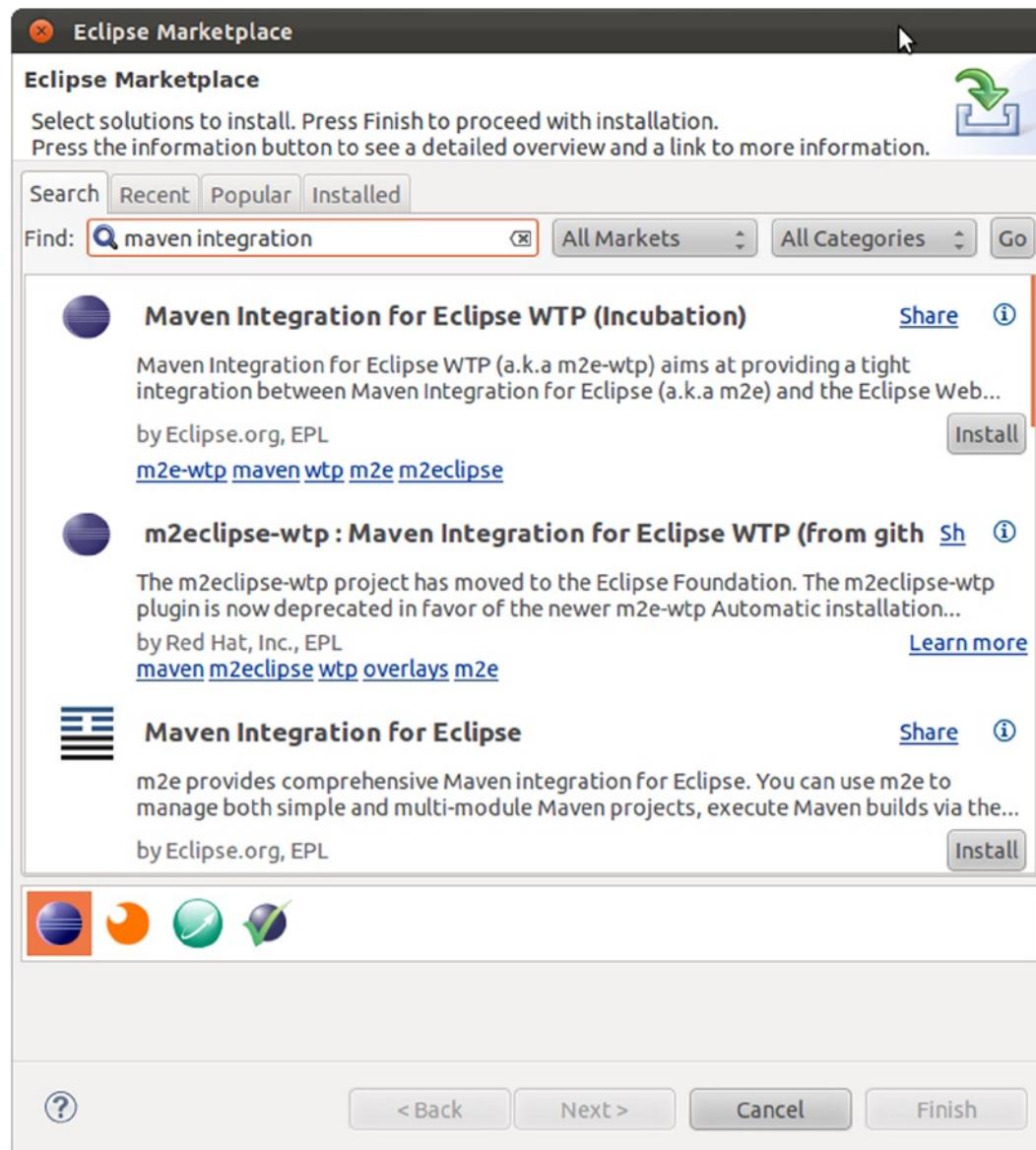


Figure 1-15. Maven Integration for Eclipse installation

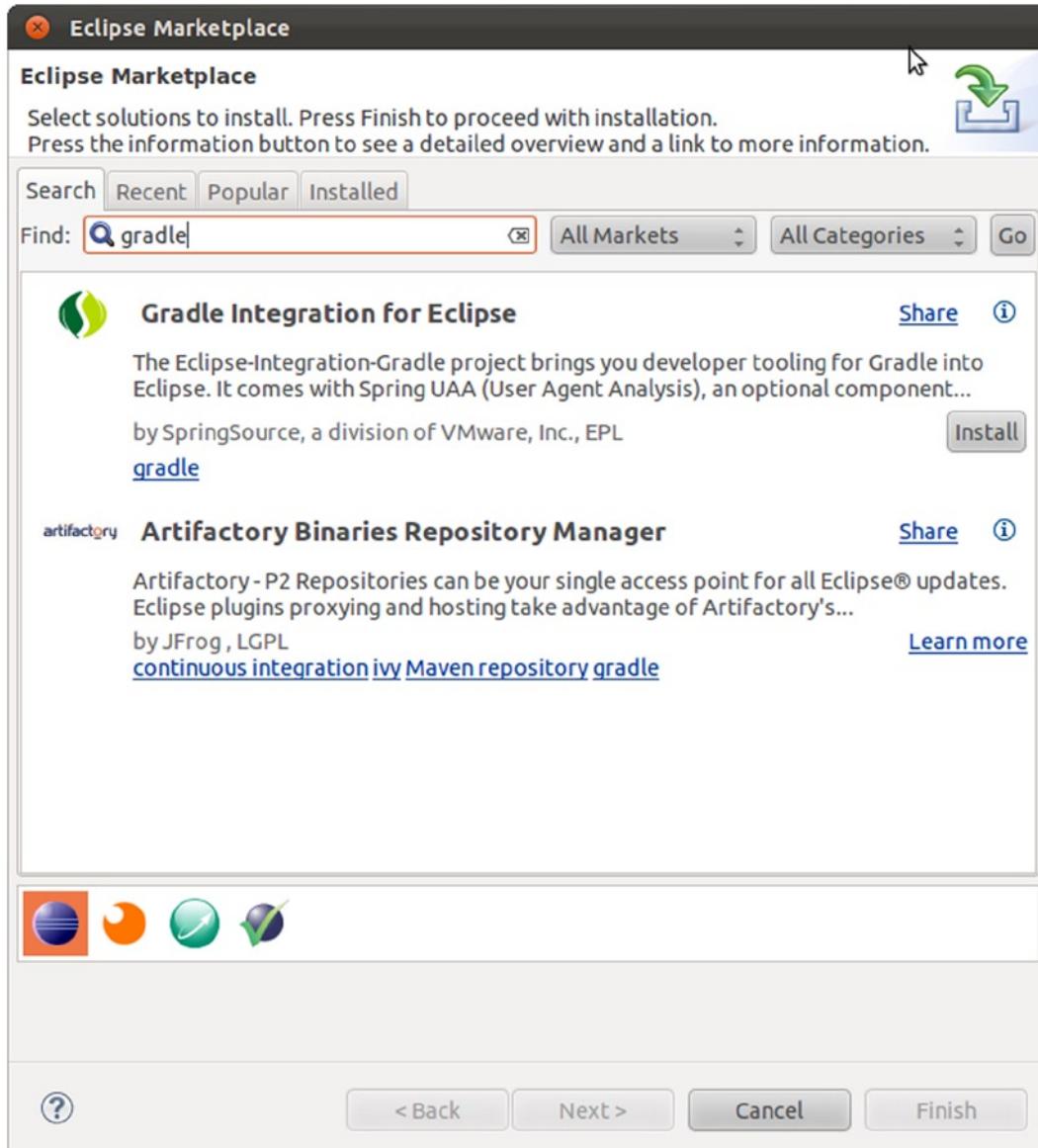


Figure 1-16. Gradle Integration for Eclipse installation

With access to both maven and gradle from Eclipse, let's access some of the book's applications. Once you download the book's source and unpack it to a local directory, click on the Eclipse top level 'File' menu and select the 'Import...' option. A pop-up window appears. In the pop-up window, click on the 'Maven' icon and select the 'Existing Maven Projects' option as illustrated in Figure 1-17.

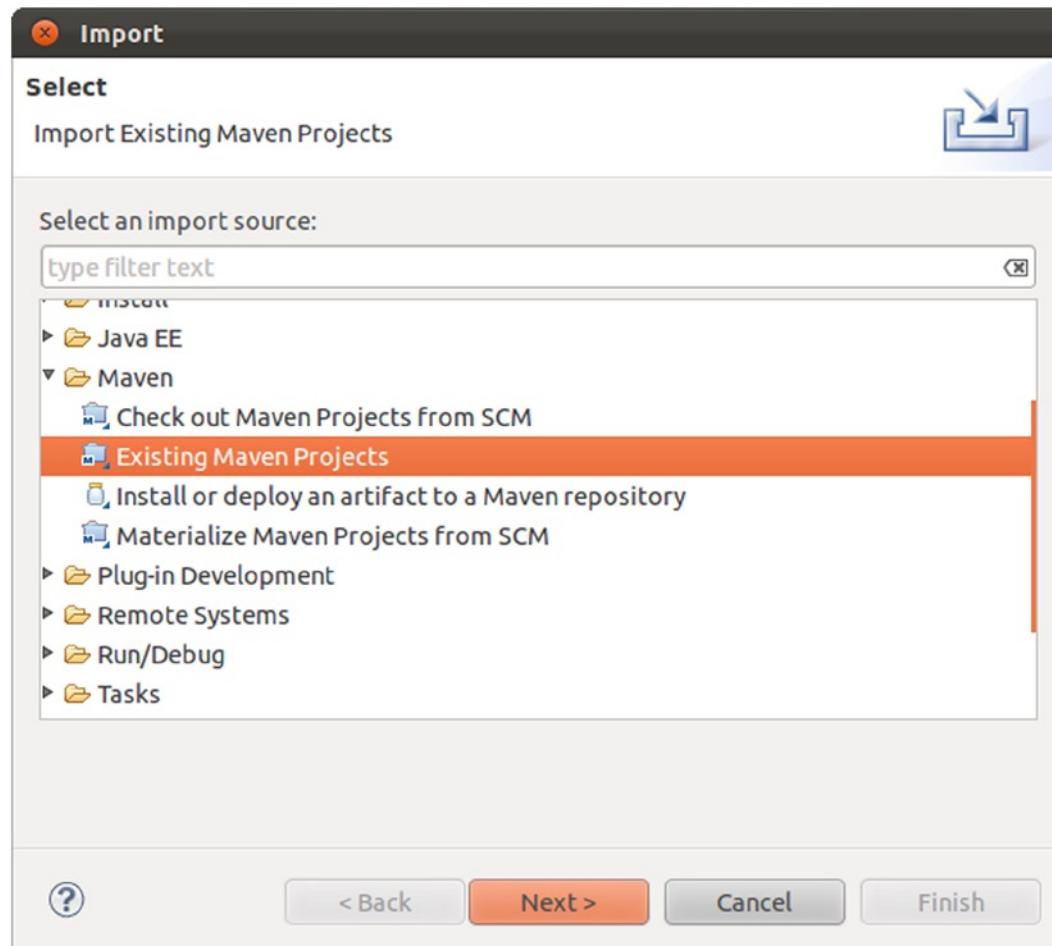


Figure 1-17. Eclipse maven import

Next, click on the ‘Next’ button. In the following screen, on the ‘Select root directory’ line click on the ‘Browse’ button and select the directory Ch1 of the book’s source code and select `sprintro_mvn` as illustrated in the Figure 1-18.

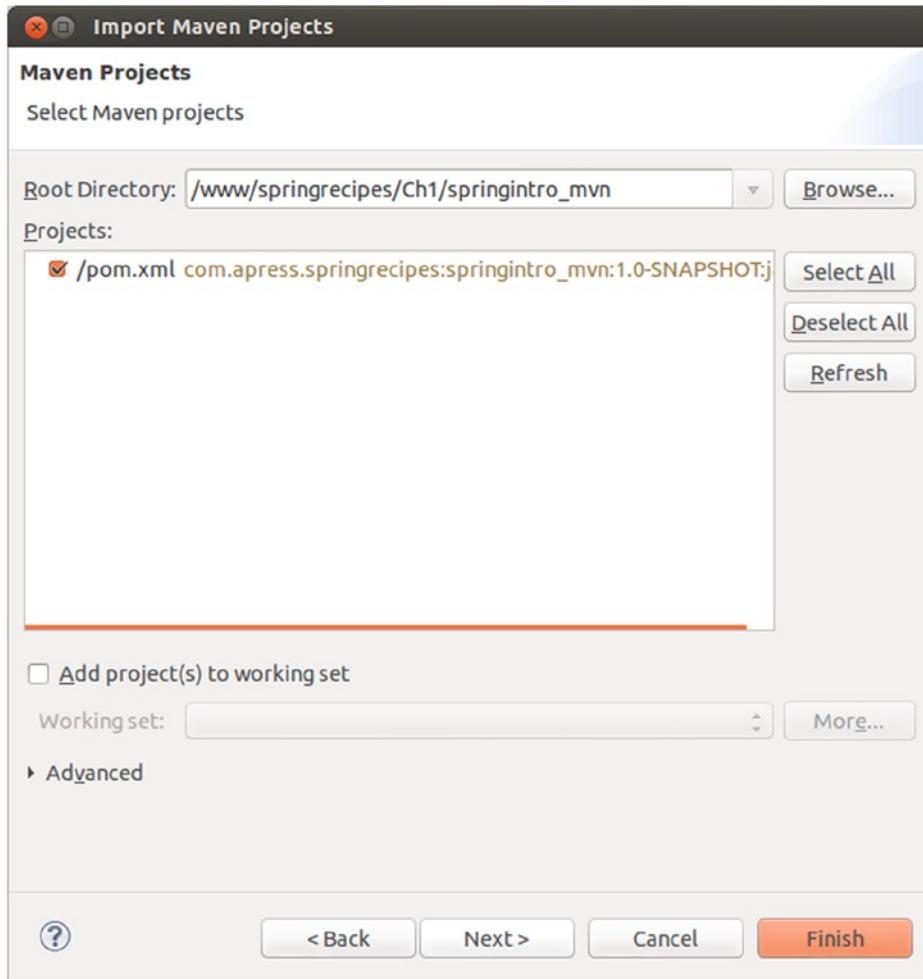


Figure 1-18. Eclipse select maven project

Notice in Figure 1-18 the ‘Projects:’ window is updated to include the line `pom.xml com.apress.springrecipes...` which reflects the Maven project to import. Select the project checkbox and click on the ‘Finish’ button to import the project. All projects in Eclipse are accessible on the left-hand side in the ‘Project Explorer’ window. In this case, the project appears with the name `sprintro_mvN`.

If you click on the project icon in the ‘Project Explorer’ window, you’ll be able to see the project structure (i.e., java classes, dependencies, configuration files, etc.). If you double click on any of the project files inside the ‘Project Explorer’, the file is opened in a separate tab in the center window -- alongside the dashboard. Once a file is opened, you can inspect, edit, or delete its contents.

Select the project icon in the ‘Project Explorer’ window and click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Maven build’ option. A pop-up window appears to edit and configure the project build. Just click on the ‘Run’ button in the bottom right. In the bottom center of Eclipse you’ll see the ‘Console’ window. In this case, the ‘Console’ window displays a series of build messages produced by Maven, as well as any possible errors in case the build process fails.

You've just built the application, congratulations! Now let's run it. Select the project icon from the 'Project Explorer' window once again and press the F5 key to refresh the project directory. Expand the project tree. Toward the bottom you'll see a new directory called target which contains the built application. Expand the target directory by clicking on its icon. Next, select the file `springintro_mvN-1.0-SNAPSHOT.jar` as illustrated in Figure 1-19.

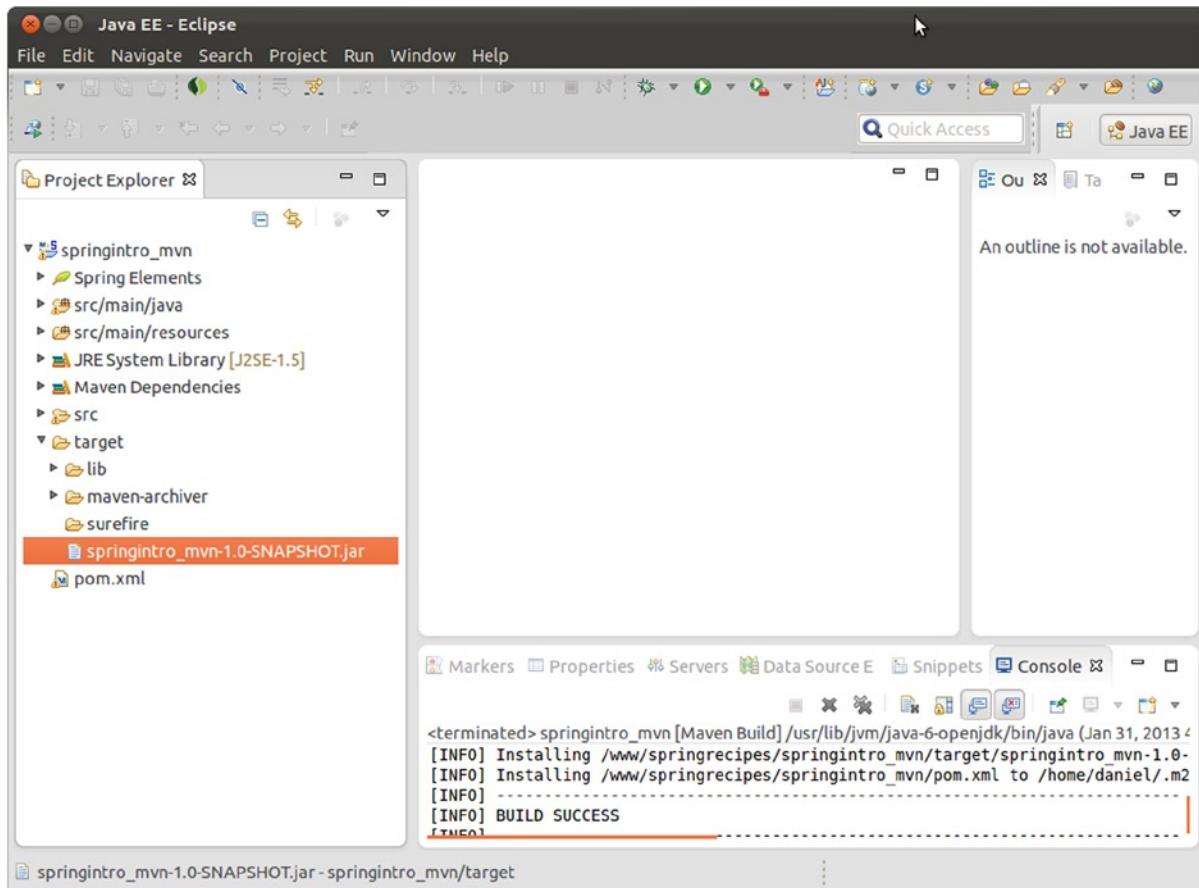


Figure 1-19. Select executable in Eclipse

With the file selected, click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Run configurations...’ option. A pop-up window to edit and configure the run appears. Ensure the ‘Java application’ option is selected on the left-hand side. In the ‘Main class:’ box introduce: `com.apress.springrecipes.hello.Main`. This is the main class for this project, as illustrated in Figure 1-20.

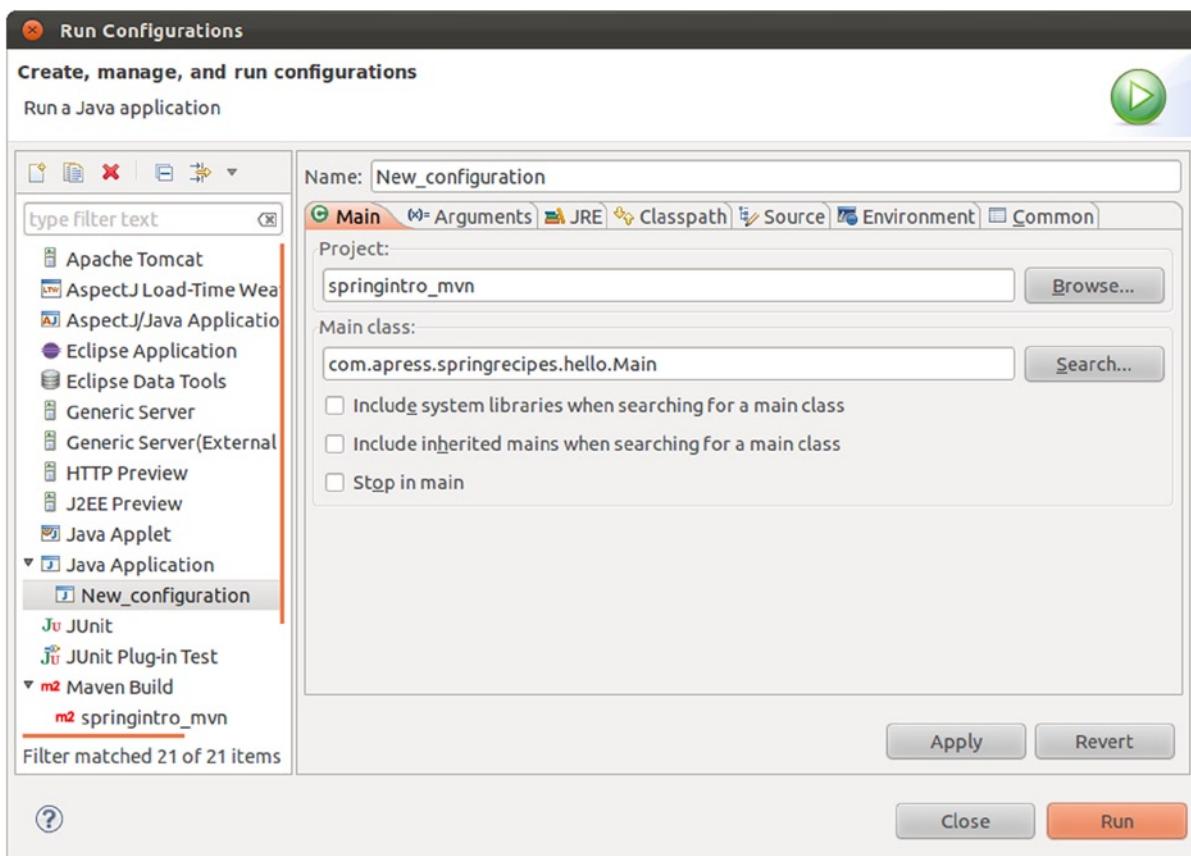


Figure 1-20. Define main executable class in Eclipse

Click on the ‘Run’ button in the bottom right. In the bottom center of Eclipse you’ll see the ‘Console’ window. In this case, the ‘Console’ window displays the application logging messages, as well as a greeting message defined by the application.

Next, let's build a Gradle application with Eclipse. Go to Eclipse's top level 'File' menu and select the 'Import...' option. A pop-up window appears. In the pop-up window, click on the 'Gradle' icon and select the 'Gradle Project' option as illustrated in Figure 1-21.

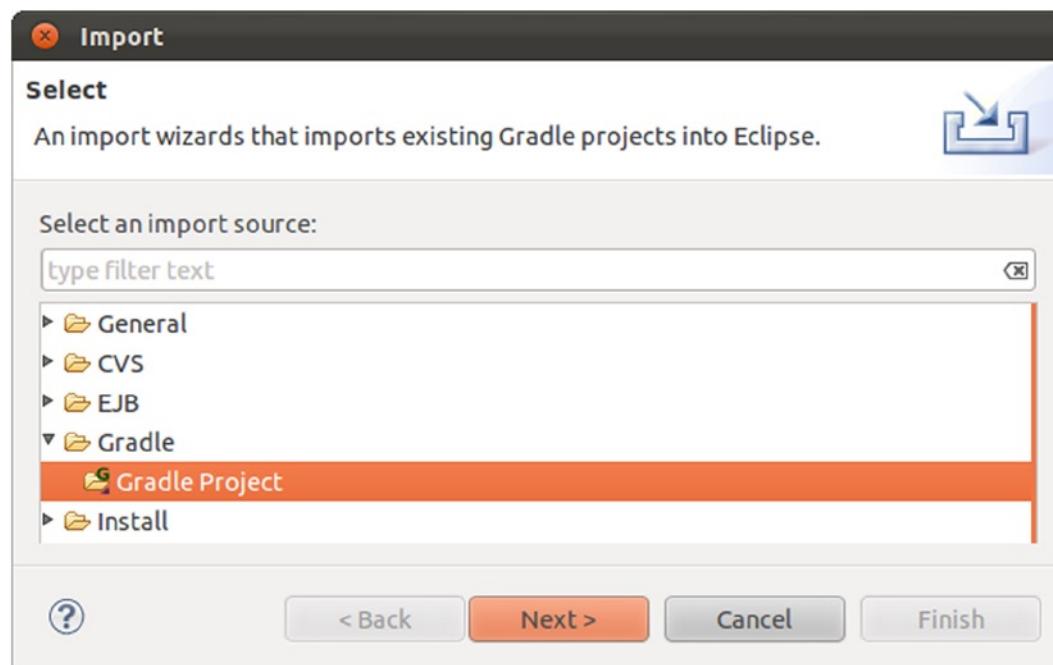


Figure 1-21. Eclipse Gradle import

Next, click on the 'Next' button. In the following screen, in the 'Select root directory' line click on the 'Browse' button and select the top level directory of the book's source code. Next, click on the 'Build model' button beside the 'Browse' button. The build model process retrieves the various Gradle subprojects contained in the book's source code. A pop-up window appears indicating the progress of the build model process. Once the build model process finishes, you'll see a list of Gradle projects, click on the project checkbox called 'springintro' inside 'Ch1' as illustrated in Figure 1-22.

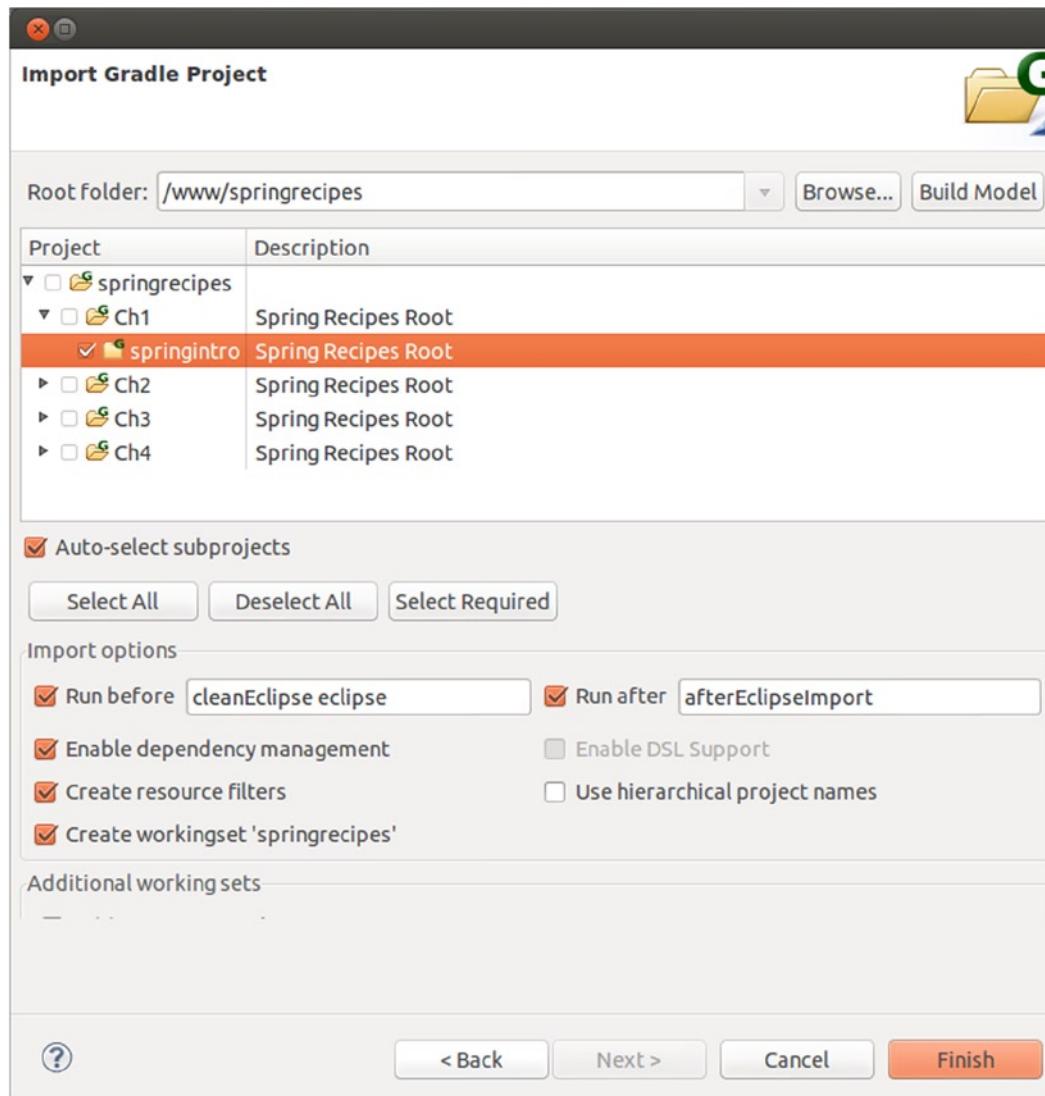


Figure 1-22. Eclipse select Gradle subproject

Click on the 'Finish' button to import the project. If you look at the left hand side of Eclipse in the 'Project Explorer' window you'll see the project is loaded with the name `springintro`. If you click on the project icon, you'll be able to see the project structure (i.e., java classes, dependencies, configuration files, etc.).

Select the project icon and click on the right button of your mouse. A contextual menu appears with various project commands. Select the ‘Run as’ option followed by the ‘Gradle build’ option. A pop-up window appears to edit and configure the build. Click on the Project/task option ‘build’ as illustrated in Figure 1-23.

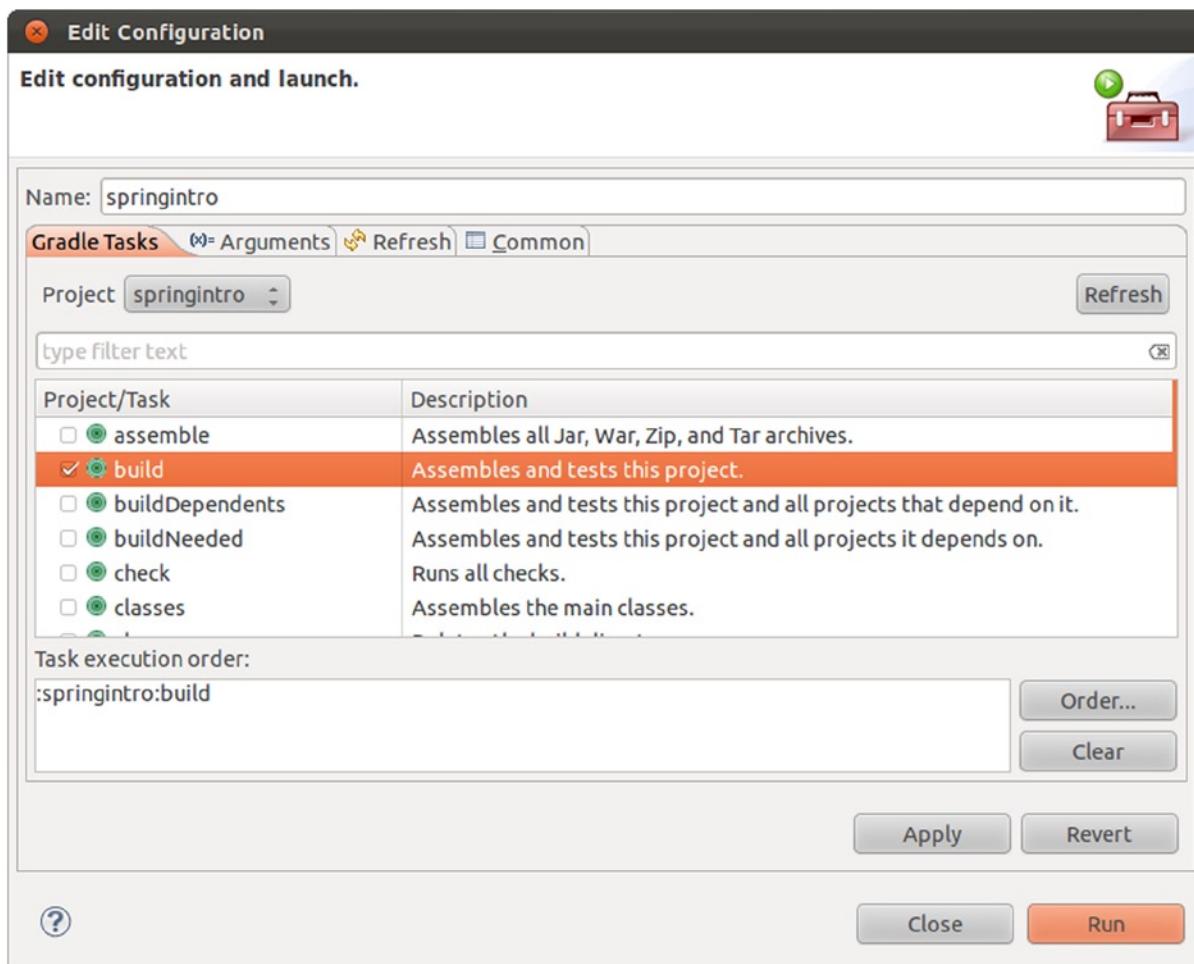


Figure 1-23. Eclipse select Gradle ‘build’ option

Next, click on the ‘Run’ button in the bottom right. In the bottom center of Eclipse you’ll see the ‘Console’ window appear. In this case, the ‘Console’ window displays a series of build messages produced by gradle, as well as any possible errors in case the build process fails.

You’ve just built the application, now let’s run it. Select the project icon once again and press the F5 key to refresh the project directory. Expand the project tree. Toward the middle inside the build directory you’ll see a new directory called `libs` which contains the built application. Expand the `libs` directory by clicking on the icon. Next, select the file `springintro-1.0-SNAPSHOT.jar`.

With the file selected, from Eclipse’s top level ‘Run’ menu select the ‘Run configurations...’ option. A pop-up window appears to edit and configure the run. Ensure the ‘Java application’ option is selected in the left-hand side. In the ‘Main class.’ box introduce `com.apress.springrecipes.hello.Main`. This is the main class for this project.

Click on the ‘Run’ button in the bottom right. In the bottom center of Eclipse you’ll see the ‘Console’ window. In this case, the ‘Console’ window displays the application logging messages, as well as a greeting message defined by the application.

1-3. Build a Spring application with the IntelliJ IDE

Problem

You want to use the IntelliJ IDE to build Spring applications.

Solution

To start a new Spring application in the IntelliJ ‘Quick Start’ window click on the ‘Create New Project’ link. In the next window, assign a name to the project, a run-time JDK and select the ‘Java Module’ option. In the next window, click on the various Spring checkboxes so IntelliJ download’s the necessary Spring dependencies for the project.

To open a Spring application that uses Maven, you first need to install Maven to work from a command line interface (See Recipes 1-4). From the IntelliJ top level ‘File’ menu select the ‘Import Project’ option. Next, select the Spring application based on Maven from your workstation. In the next screen select the ‘Import project from external model’ option and select a ‘Maven’ type.

To open a Spring application that uses Gradle, you first need to install Gradle to work from a command line interface (See Recipe 1-5). From the IntelliJ top level ‘File’ menu select the ‘Import Project’ option. Next, select the Spring application based on Gradle from your workstation. In the next screen select the ‘Import project from external model’ option and select a ‘Gradle’ type.

How It Works

IntelliJ is one of the most popular commercial IDEs in the market. Unlike other IDEs which are produced by a foundation -- such as Eclipse -- or are made to support the flagship software of a company -- such as STS for the Spring framework -- IntelliJ is produced by a company called JetBrains whose sole business is to commercialize development tools. It’s this focus which makes IntelliJ particularly popular for professional developers in corporate environments.

For this recipe I’ll assume you’ve already installed IntelliJ ultimate edition and just want to get up and running with Spring applications.

■ Warning IntelliJ is available in a free community edition and an ultimate edition with a 30-day free trial. Although the free community edition provides good value for application development, the community edition does not include support for Spring applications. The instructions that follow are based on the assumption that you’re using the IntelliJ ultimate edition.

To start a Spring application, in the IntelliJ ‘Quick Start’ window click on the ‘Create New Project’ link. In the next window, assign a name to the project, a run-time JDK and select the ‘Java Module’ option as illustrated in Figure 1-24.

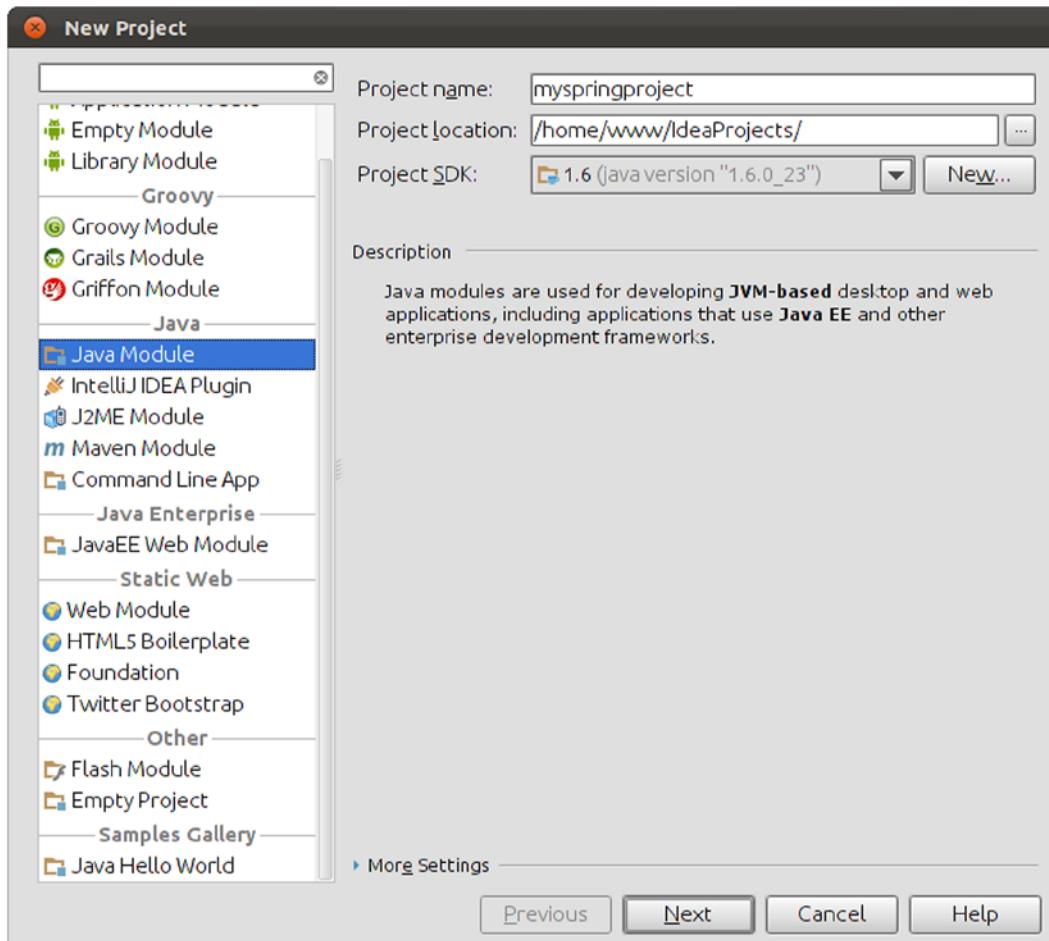


Figure 1-24. IntelliJ create Spring project

Click on the ‘Next’ button. In the next window, click on the various Spring checkboxes -- as illustrated in Figure 1-25 -- for IntelliJ to download the necessary Spring dependencies for the project.

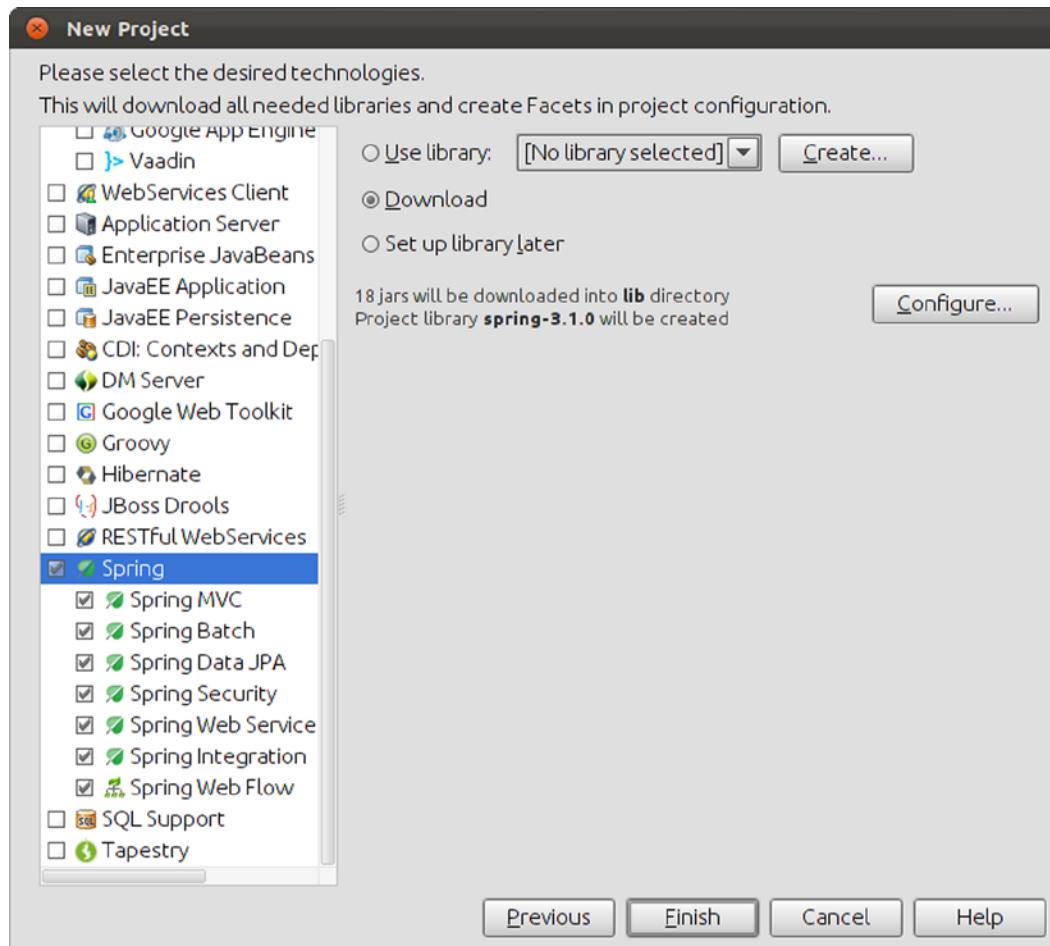


Figure 1-25. IntelliJ define project dependencies

A more common case than creating a Spring application from scratch is to continue development of a pre-existing Spring application. Under such circumstances, the owner of an application generally distributes the application’s source code with a build script to facilitate its ongoing development.

The build script of choice for most Java application is a `pom.xml` file designed around the build tool called Maven and more recently a `build.gradle` file designed around the build tool called Gradle. The book’s source code and its applications are provided with Gradle build files, in addition to a single application with a Maven build file.

Once you download the book's source and unpack it to a local directory, click on the IntelliJ top level 'File' menu and select the 'Import Project' option. A pop-up window appears as illustrated in Figure 1-26.

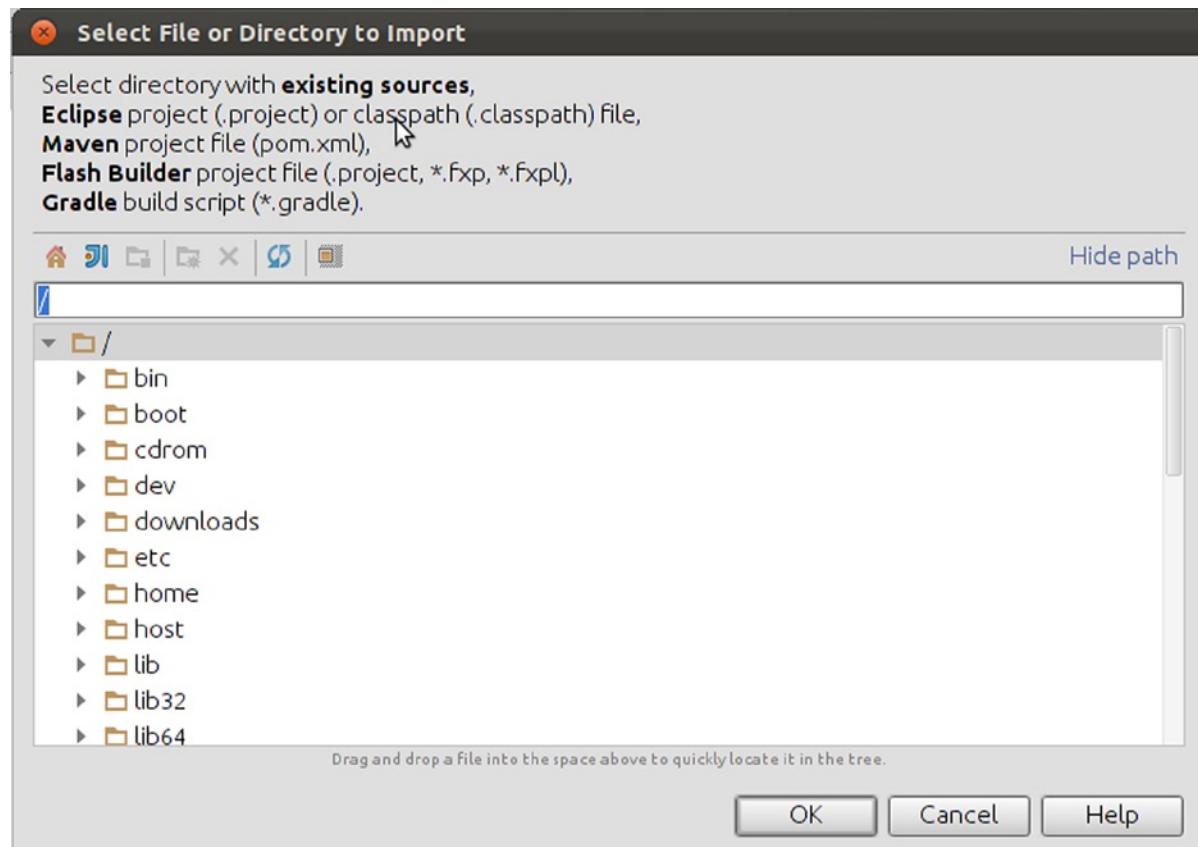


Figure 1-26. IntelliJ select file or directory to import

Click on the directory tree presented in the pop-up window until you select the directory of the book's source code inside Ch1 and then select `springintro_mvn`. Click on the 'Next' button. In the next screen select the 'Import project from external model' option and select a 'Maven' type as illustrated in Figure 1-27.

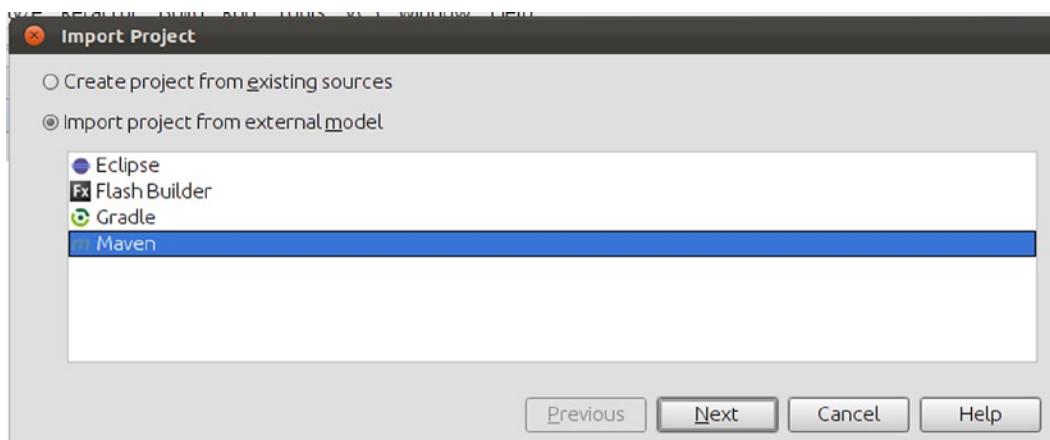


Figure 1-27. IntelliJ import project

In the next window you'll see the line '`com.apress.springrecipes:...:`' as illustrated in Figure 1-28.

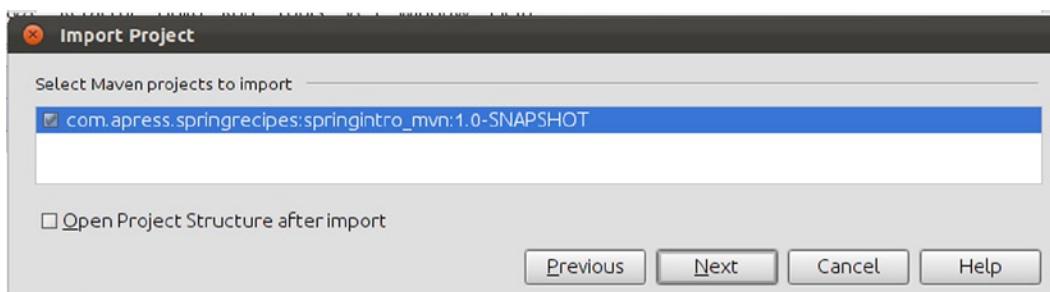


Figure 1-28. IntelliJ import Maven project

Ensure the project checkbox is selected and click on the 'Next' button to import the project. Next, choose the SDK version for the project. Confirm the project name, location and click on the 'Finish' button. All projects in IntelliJ are loaded on the left-hand side in the 'Project' window. In this case, the project appears with the name `springintro_mvn`.

If you click on the project icon, you'll be able to see the project structure (i.e., java classes, dependencies, configuration files, etc.). If you double click on any of the project files inside the 'Project' window, the file is opened in a separate tab in the center window. You can inspect the contents of the file, as well as edit or delete its contents.

Next, we need to setup Maven to work with IntelliJ. Follow the instructions in Recipe 1-4 to install Maven to work from the command line. Once you do this, you can setup IntelliJ to work with Maven.

Click on the IntelliJ top level ‘File’ menu and select the ‘Settings’ option. A pop-up window appears to configure IntelliJ settings. Click on the ‘Maven’ option and in the ‘Maven home directory’ introduce the Maven installation directory based on your system. This is illustrated in Figure 1-29. Click on the ‘Apply’ button, followed by the ‘OK’ button.

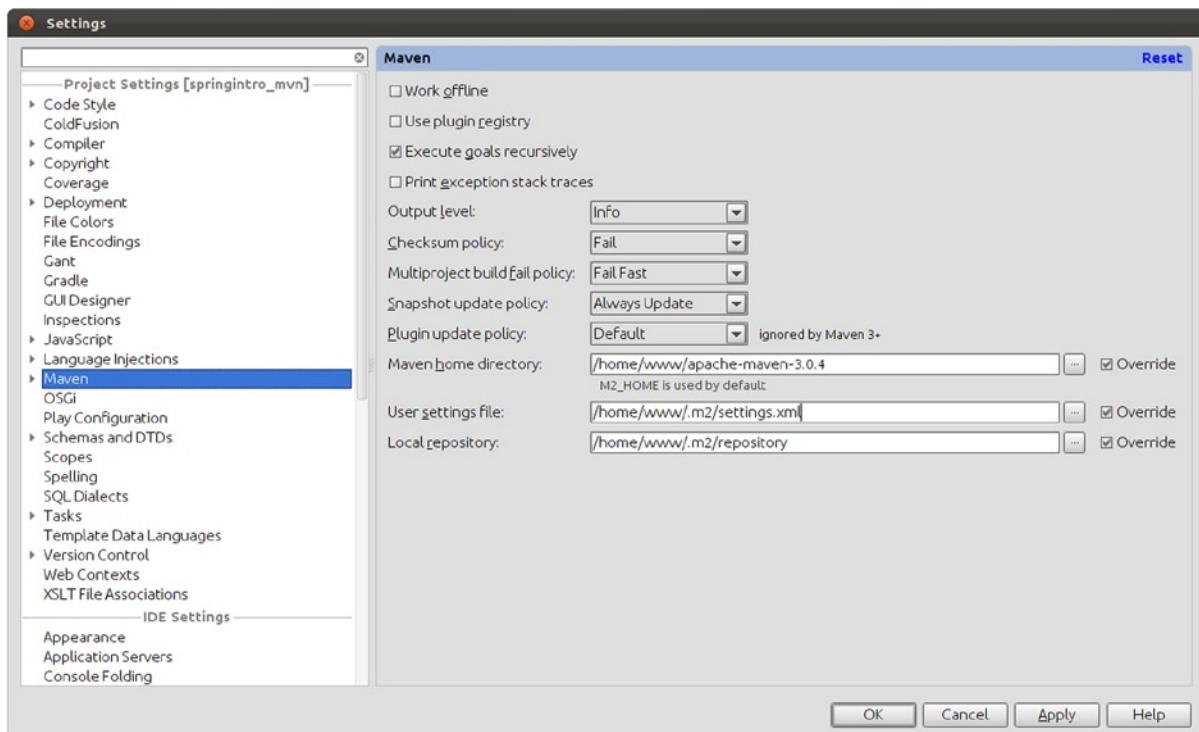


Figure 1-29. IntelliJ Maven configuration

Next, on the right-hand side of IntelliJ click on the vertical tab ‘Maven projects’ to show the Maven project window as illustrated in Figure 1-30.

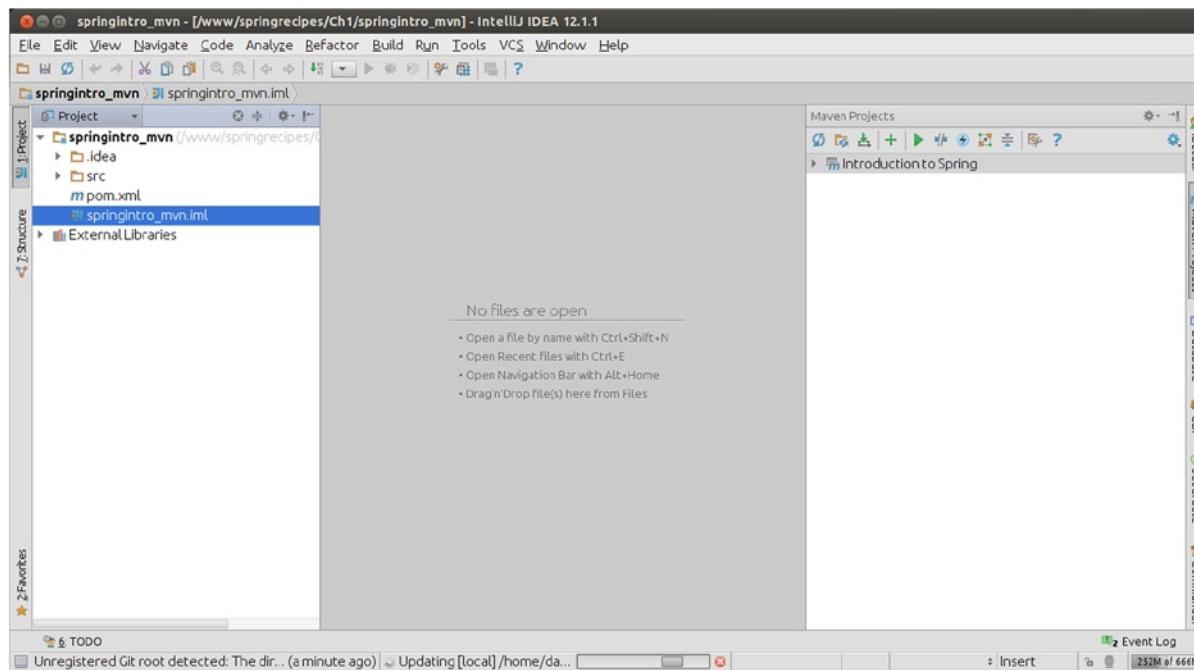


Figure 1-30. IntelliJ Maven projects window

Select the project the ‘Introduction to Spring’ line in the Maven projects window and click on the right button of your mouse. A contextual menu appears with various commands for the project. Select the ‘Run Maven Build’ option. In the bottom center of IntelliJ you’ll see the ‘Run’ window appear. In this case, the ‘Run’ window displays a series of build messages produced by Maven, as well as any possible errors in case the build process fails.

Warning If you see the error message ‘No valid Maven installation found. Either set the home directory in the configuration dialog or set the M2_HOME environment variable on your system’ it means Maven is not being found by IntelliJ. Verify the Maven installation and configuration process.

You've just built the application, congratulations! Now let's run it. By default, IntelliJ hides the target directory created by Maven to place the built application. You'll need to deactivate this behavior in order to run the application directly from IntelliJ. Click on the IntelliJ top level 'File' menu and select the 'Settings' option. A pop-up window appears to configure IntelliJ settings. Double click on the 'Maven' option and click on the 'Importing' sub-option. Uncheck the box 'Exclude build directory' as illustrated in Figure 1-31. Click on the 'Apply' button and then click on the 'OK' button.

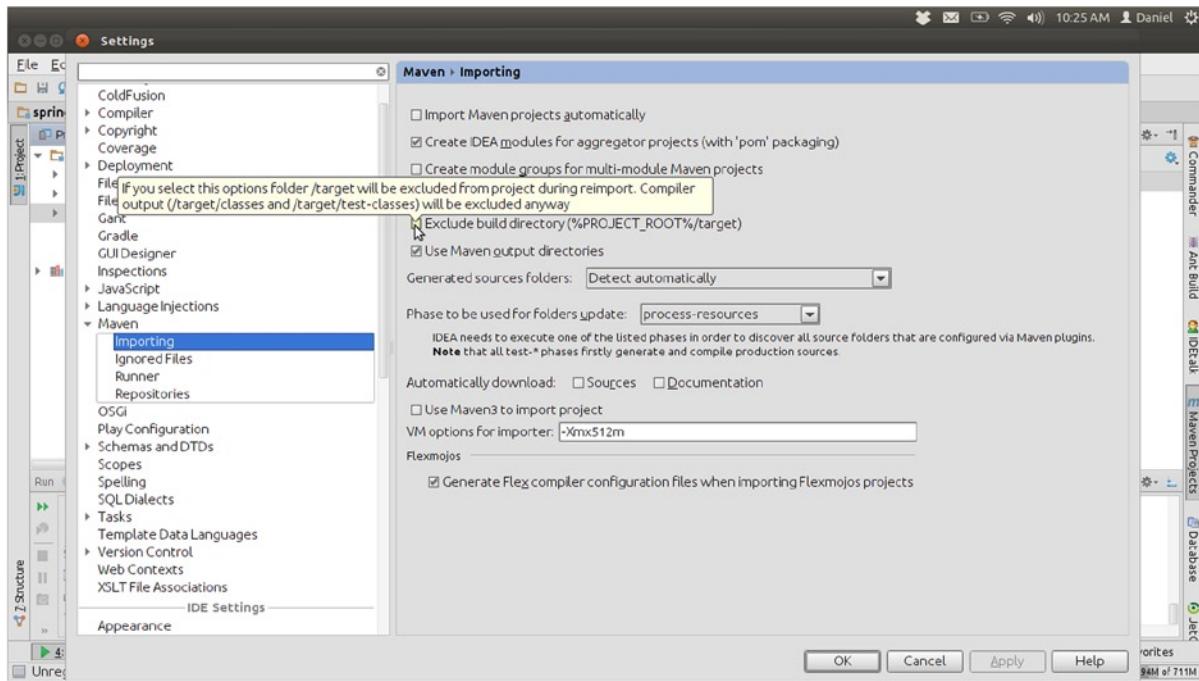


Figure 1-31. IntelliJ Maven show build directory

With this last change you'll see the target directory in the 'Project' window. If you don't see the target directory click the Ctrl+Alt+Y key combination to synchronize the project. Expand the target directory by clicking on its icon. Next, select the file `springintro_mvn-1.0-SNAPSHOT.jar` as illustrated in Figure 1-32.

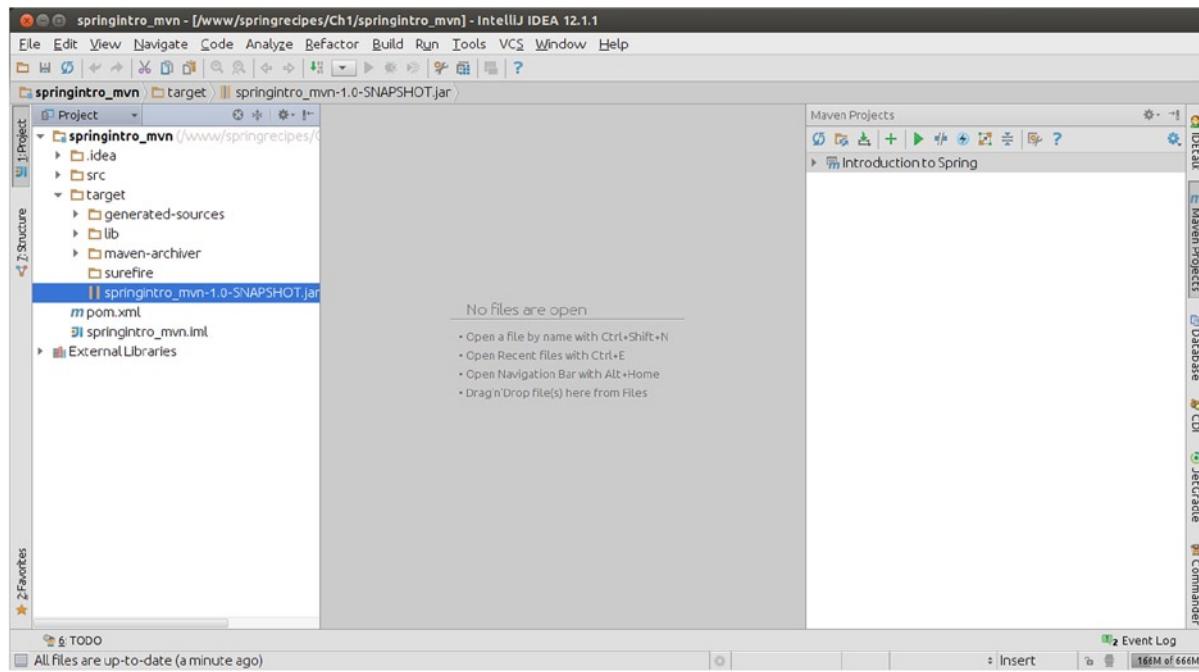


Figure 1-32. IntelliJ select application to run

Click on the IntelliJ top level ‘Run’ menu and select the ‘Run’ option. A small box appears to configure IntelliJ run configurations. Click on the small box and a pop-up window appears to configure the various types of applications supported by IntelliJ. Click on the ‘Application’ option. Add the `com.apress.springrecipes.hello.Main` value to the ‘Main class:’ box and ensure the remaining options are similar to Figure 1-33.

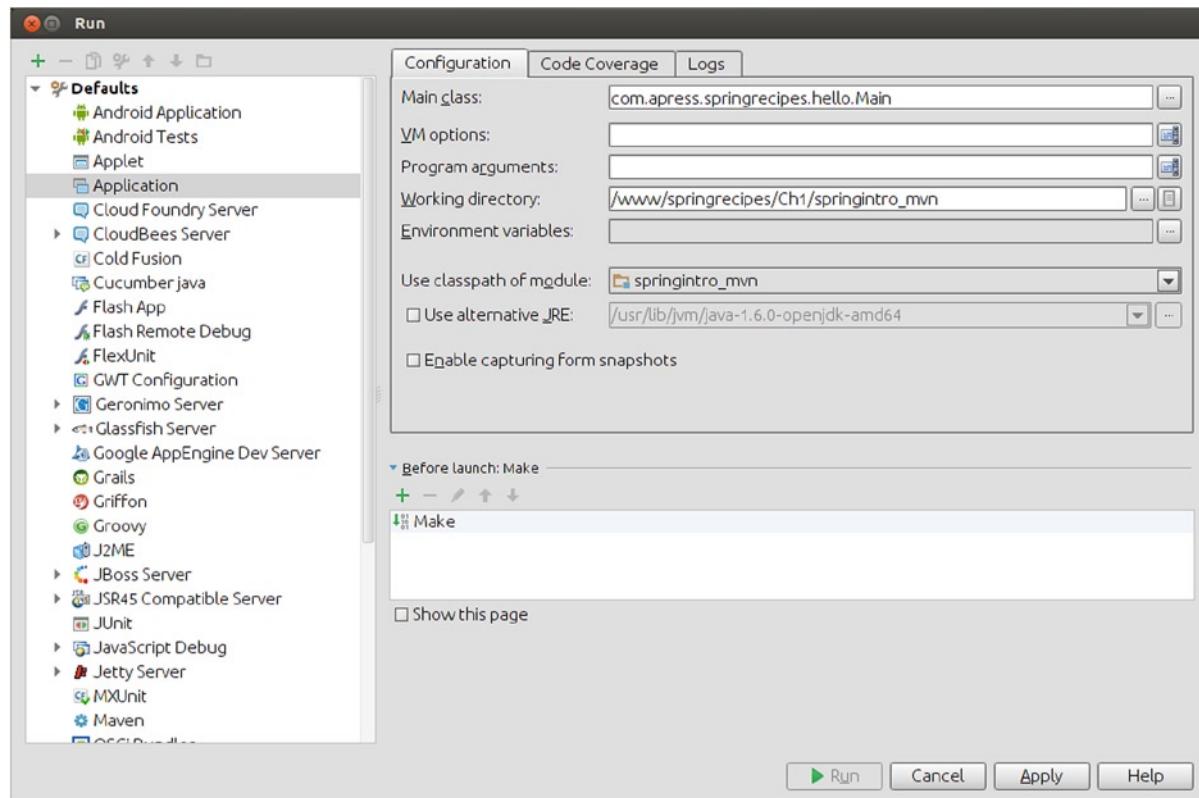


Figure 1-33. Configure IntelliJ application run

Click on the ‘Apply’ button. The ‘Run’ button is disabled because you’re working with the default run configurations. Click on the plus sign in the top-left of the pop-up window. The left column changes to ‘Add new configuration’, click on the ‘Application’ option. A new application configuration is added. Add the application name `springintro_mvnm` to the ‘Name’ box as illustrated in Figure 1-34 and click on the ‘Apply’ button.

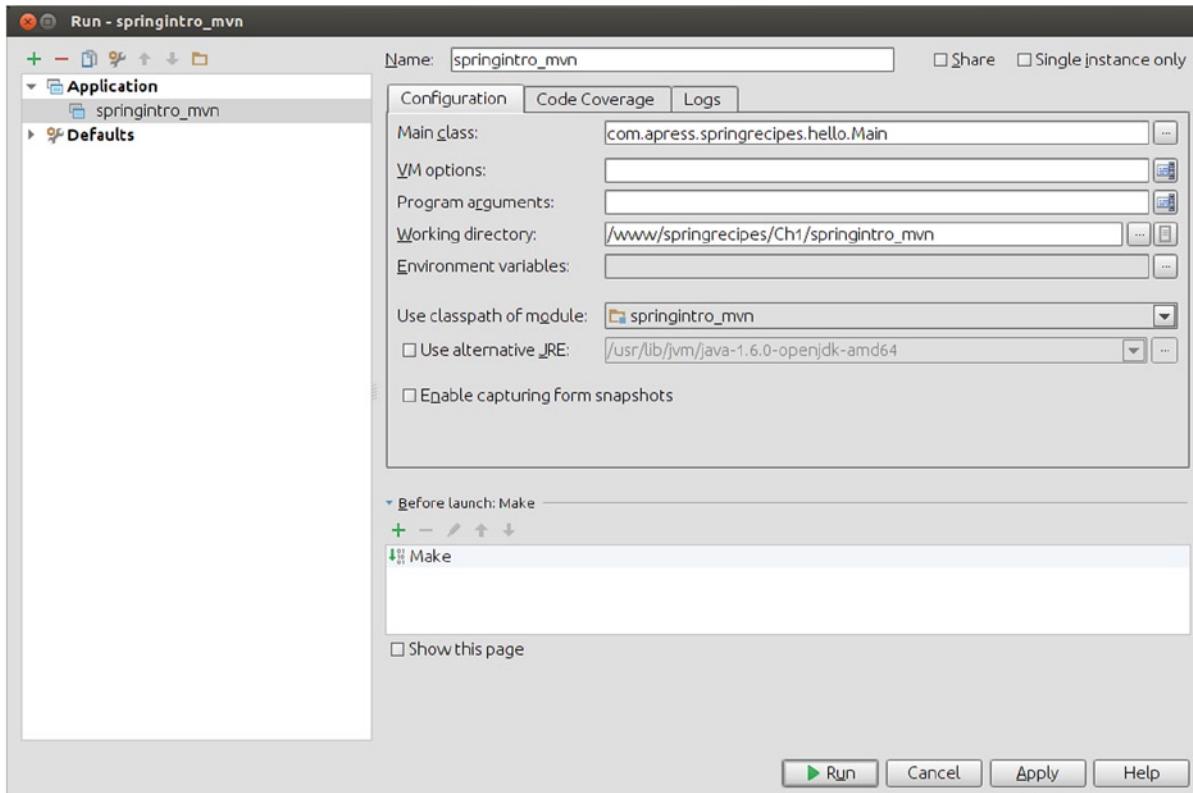


Figure 1-34. Define IntelliJ application run

Click on the ‘Run’ button. In the bottom center of IntelliJ in the ‘Run’ window you’ll see the application logging messages, as well as a greeting message defined by the application.

Now let’s build a Gradle application with IntelliJ. First you need to install Gradle. Follow the instructions in Recipe 1-5 to install Gradle to work from the command line. Once you do this, you can setup IntelliJ to work with Gradle.

Click on the IntelliJ top level ‘File’ menu and select the ‘Import Project’ option. A pop-up window appears as illustrated in Figure 1-26. Click on the directory tree presented in the pop-up window until you select the file `build.gradle` in the top level directory of the book’s source code.

Click on the ‘Next’ button. In the next screen select the ‘Import project from external model’ option and select a ‘Gradle’. In the next screen, introduce the Gradle home directory in the ‘Gradle home’ box, based on the Gradle installation of your system. This is illustrated in Figure 1-35.

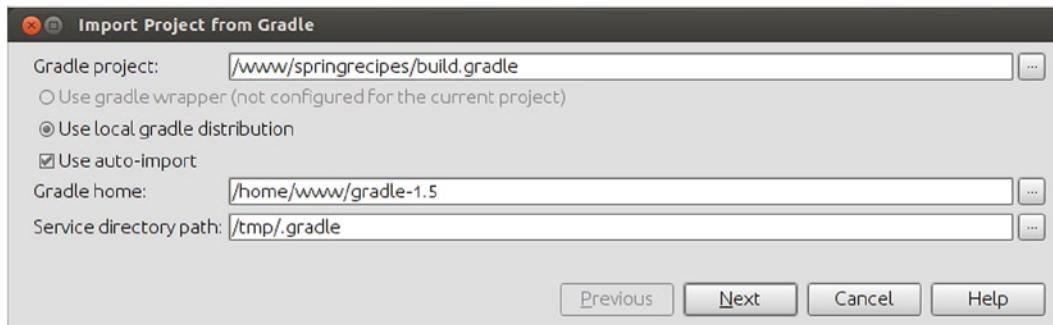


Figure 1-35. Define Gradle home for IntelliJ

Click on the ‘Next’ button to confirm the import process and the click on the ‘Finish’ button to complete the import process. Next, on the right hand side of IntelliJ click on the vertical ‘Jet Gradle’ tab to show Gradle. Select the project the ‘springintro’ inside ‘Ch1’ in the projects window as illustrated in Figure 1-36.

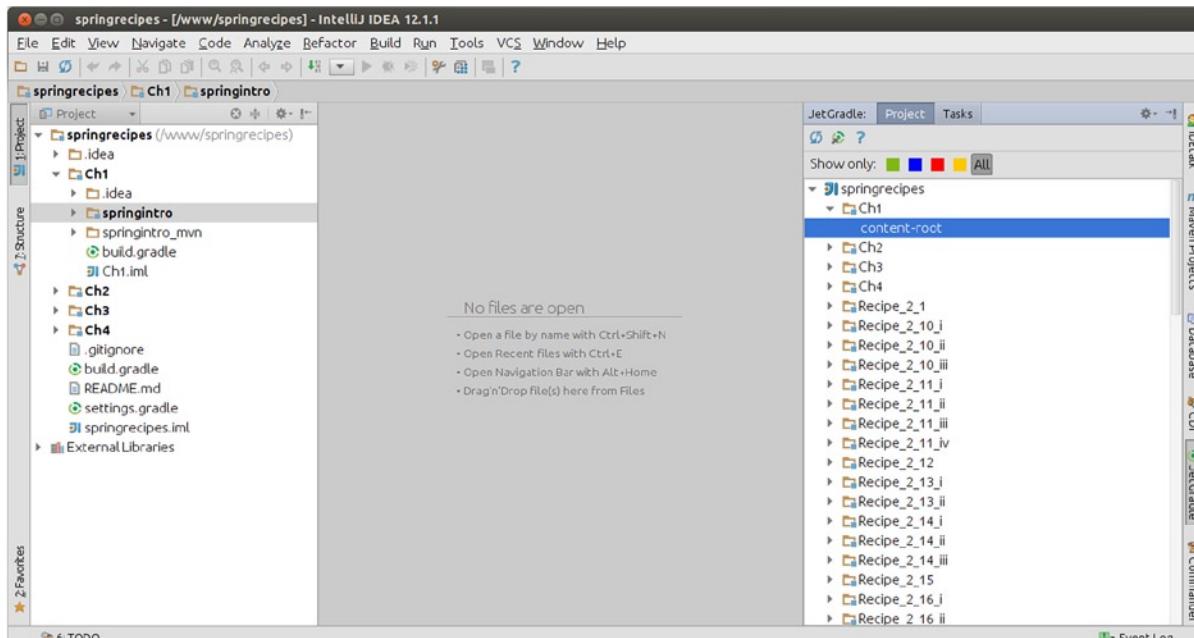


Figure 1-36. IntelliJ JetGradle project window

Click on the IntelliJ top level ‘Run’ menu and select the ‘Run’ option. A small box appears to configure IntelliJ run configurations. Click on the small box and a pop-up window appears to configure the various types of applications supported by IntelliJ. Click on the ‘Groovy’ option -- this is due to Gradle being a Groovy based tool.

In the ‘Script path’ box assign the build.gradle file from the Ch1 directory of the book’s source code. In the ‘Module’ box select the ‘Ch1’ option. In the ‘Script parameters’ box introduce the text build, which is the parameter for the Gradle script. And in the ‘Working directory’ ensure you have the Ch1/springintro directory of the book’s source code.

Click on the ‘Apply’ button. Click on the plus sign in the top-left of the pop-up window. The left column changes to ‘Add new configuration’, click on the ‘Groovy’ option. A new application configuration is added. Add the application name to the ‘springintro’ box as illustrated in Figure 1-37 and click on the ‘Apply’ button once again.

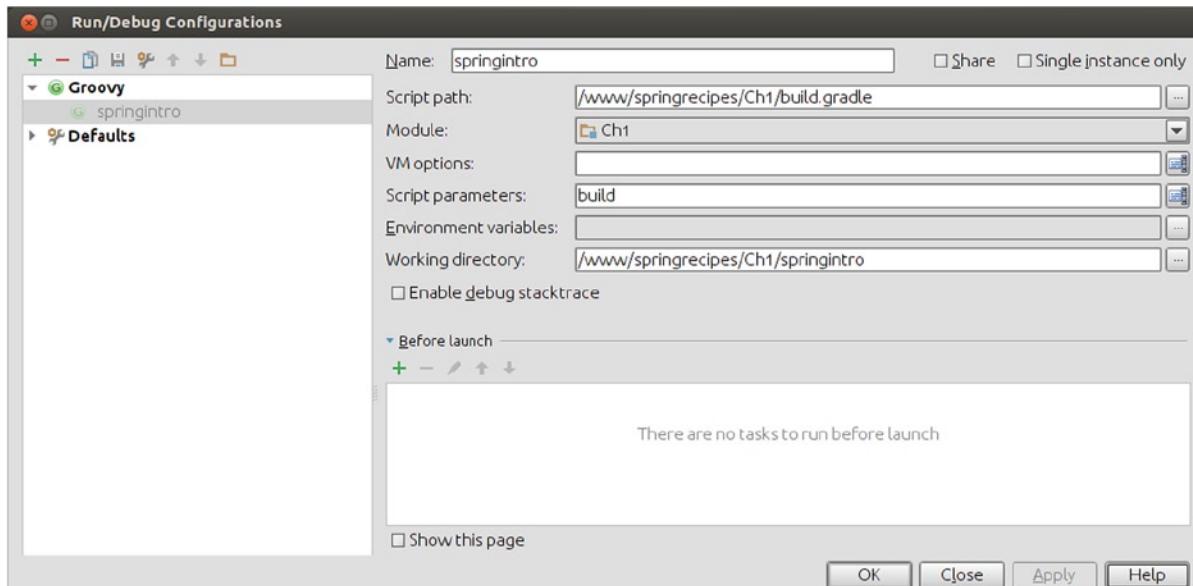


Figure 1-37. IntelliJ Groovy application configuration

Next, click on the IntelliJ top level ‘Run’ menu and select the ‘Run springintro’ option. In the bottom center of IntelliJ you’ll see the ‘Run’ window appear. In this case, the ‘Run’ window displays a series of build messages produced by Gradle, as well as any possible errors in case the build process fails.

You've just built the application. Now let's run it. By default, IntelliJ hides the build directory created by Gradle to place the built application. You'll need to deactivate this behavior in order to run the application directly from IntelliJ. Click on the IntelliJ top level 'File' menu and select the 'Project Structure...' option. A pop-up window appears to configure IntelliJ project structure. Ensure the 'Modules' option is selected on the left-hand side. In the 'Excluded folders' list, click on the cross next to the 'build' folder. This is illustrated in Figure 1-38. Click on the 'Apply' button and then click on the 'OK' button.

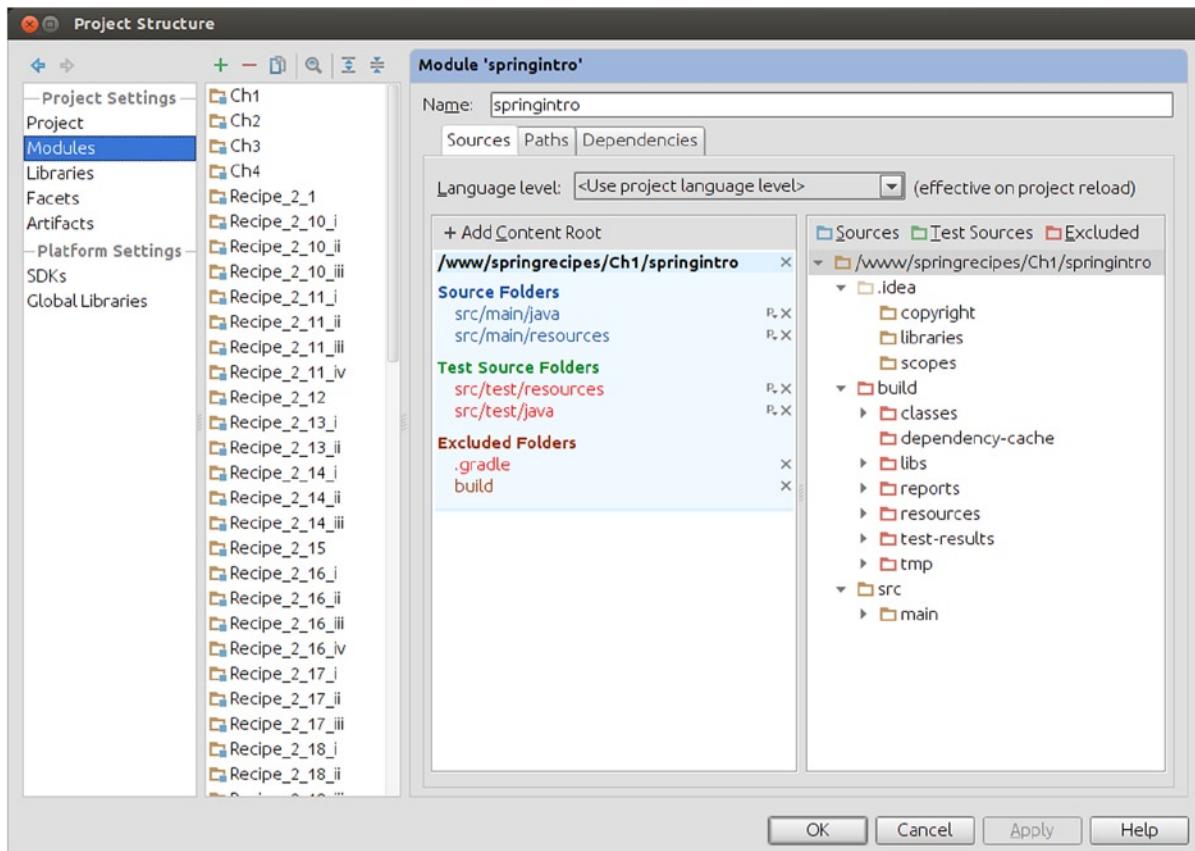


Figure 1-38. IntelliJ project structure show build folder

With this last change you'll see the build directory in the 'Project' window. If you don't see the target directory click the key combination **Ctrl+Alt+Y** to synchronize the project. Expand the target directory by clicking on its icon. Next, select the file `springintro-1.0-SNAPSHOT.jar` inside the `libs` directory of the build directory as illustrated in Figure 1-39.

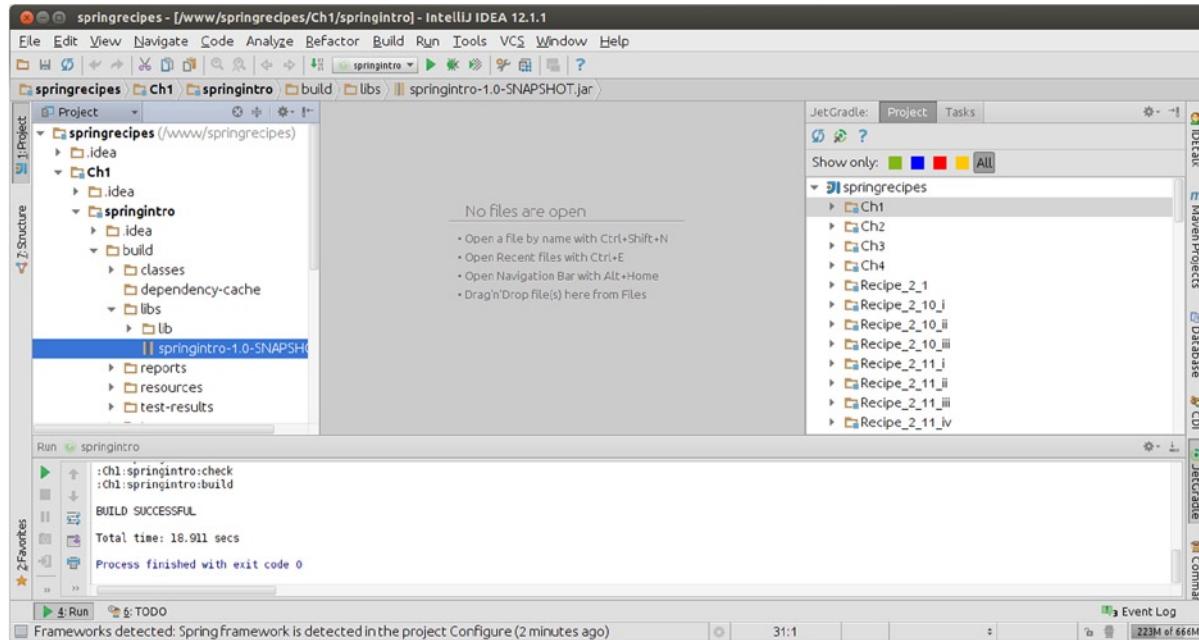


Figure 1-39. IntelliJ select application to run

Click on the IntelliJ top level ‘Run’ menu and select the ‘Run...’ option. A small box appears to configure IntelliJ run configurations. Click on the ‘Edit configurations...’ of the small box, a pop-up window appears to configure the various types of applications supported by IntelliJ. Click on the ‘Application’ option inside the ‘Defaults’ list. Add `com.apress.springrecipes.hello.Main` to the ‘Main class:’ box and ensure the remaining options are similar to Figure 1-40.

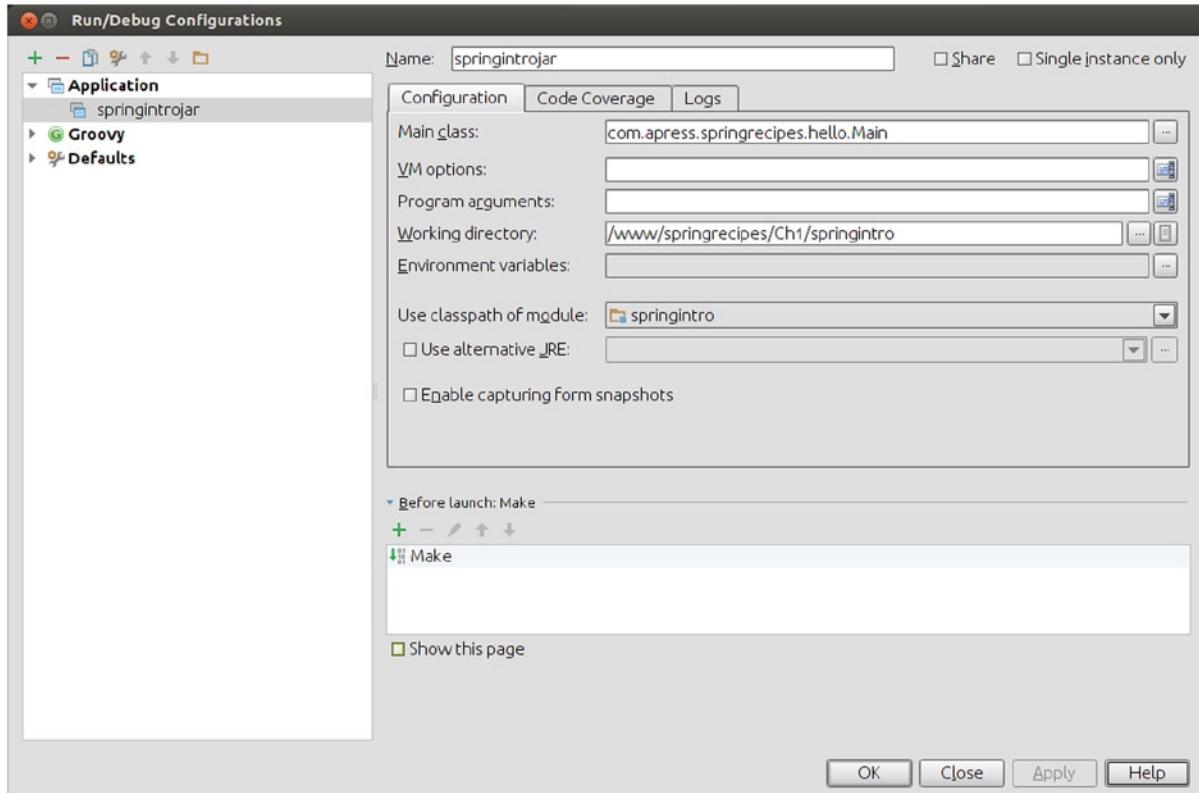


Figure 1-40. Configure IntelliJ application run

Click on the ‘Apply’ button. The ‘Run’ button is disabled because you’re working with the default run configurations. Click the plus sign in the top-left of the pop-up window. The left column changes to ‘Add new configuration’, click on the ‘Application’ option. A new application configuration is added. Add the application name ‘springintrojar’ to the ‘Name’ box and click on the ‘Apply’ button.

Select the file `springintro-1.0-SNAPSHOT.jar` inside the `libs` directory of the `build` directory once more and click on the right button of your mouse. From the contextual menu select the ‘Add as Library...’ option near the bottom. This is necessary due to Gradle with IntelliJ classpath bug. Finally, from the top level ‘Run’ menu select the ‘Run `springintrojar...`’ button. In the bottom center of IntelliJ in the ‘Run’ window you’ll see the application logging messages, as well as a greeting message defined by the application.

1-4. Build a Spring application with the Maven command line interface

Problem

You want to build a Spring application with Maven from the command line.

Solution

Download Maven from <http://maven.apache.org/download.cgi>. Ensure the JAVA_HOME environment variable is set to Java's SDK main directory. Modify the PATH environment variable to include Maven's bin directory.

How It Works

Maven is available as a standalone command line interface tool. This allows Maven to be leveraged from a wide variety of development environments. For example, if you prefer to use a text editor like emacs or vi to edit an application's code, it becomes essential to be able to access a build tool like Maven to automate the grunt work (e.g., copying files, one step compiling) typically associated with the build process for Java applications.

Maven can be downloaded for free from <http://maven.apache.org/download.cgi>. Maven is available in both source code and binary versions. Since Java tools are cross-platform, I recommend you download the binary version to avoid the additional compilation step. At the time of this writing the latest stable release of Maven is the 3.0.5 version.

Once you download Maven, ensure you have a Java SDK installed on your system as Maven requires it at run-time. Proceed to install Maven by unpacking it and defining the JAVA_HOME and PATH environment variables.

```
www@ubuntu:~$ tar -xzvf apache-maven-3.0.5-bin.tar.gz
apache-maven-3.0.5/boot/plexus-classworlds-2.4.jar
apache-maven-3.0.5/lib/maven-embedder-3.0.5.jar
apache-maven-3.0.5/lib/maven-settings-3.0.5.jar
....
....
# Add JAVA_HOME variable
www@ubuntu:~$ export JAVA_HOME=/usr/lib/jvm/java-6-openjdk/
# Add Maven executable to PATH variable
www@ubuntu:~$ export PATH=$PATH:/home/www/apache-maven-3.0.5/bin/
```

Notice the maven binary installation is straightforward. After you unpack the maven binary, you need to define the JAVA_HOME environment so maven has access to Java libraries at run-time. In addition you should also add the maven bin directory to the PATH environment variable so maven is accessible from any location on your system.

Tip If you declare the variables JAVA_HOME and PATH as illustrated previously, you'll need to do this process every time you open a new shell session to use Maven. On Unix/Linux systems you can open the .bashrc file inside a user's home directory and add the same export lines to avoid the need to declare the environment variables each session. On Windows systems you can set environment variables permanently by selecting the 'My Computer' icon, clicking on the right mouse button and then selecting the 'Properties' option. Then in the pop-up window select the 'Advanced' tab and click on the 'Environment variables' button.

The Maven executable is available through the `mvn` command. If you set the environment variables correctly as described previously, typing `mvn` from any directory on your system invokes Maven. Describing any more details about Maven execution would go beyond the scope of this recipe. However, next I'll describe how to use Maven to build a Spring application from the book's source code.

Once you download the book's source code and unpack it to a local directory, go to the directory called `Ch1/springintro_mvnmvn`. Type `mvn` to invoke maven and build the application under `springintro_mvnmvn`. The output should look like the following

```
www@ubuntu:/www/springrecipes/Ch1/springintro_mvnmvn$ mvn
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Introduction to Spring 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ springintro ---
[INFO] ...
[INFO] ...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 8.004s
```

You've just built the application, congratulations! Now let's run it. Descend into the directory called `target` created by maven under the `Ch1/springintro_mvnmvn` directory. You'll see the file `springintro_mvnmvn-1.0-SNAPSHOT.jar` which is the built application. Execute the command `java -jar springintro_mvnmvn-1.0-SNAPSHOT.jar` to run the application. You'll see application logging messages, as well as a greeting message defined by the application.

1-5. Build a Spring application with the Gradle command line interface

Problem

You want to build a Spring application with Gradle from the command line.

Solution

Download Gradle from <http://www.gradle.org/downloads>. Ensure the `JAVA_HOME` environment variable is set to Java's SDK main directory. Modify the `PATH` environment variable to include Gradle's bin directory.

How It Works

Gradle is available as a standalone command line tool. This allows Gradle to be leveraged from a wide variety of development environments. For example, if you prefer to use a text editor like emacs or vi to edit an application's code, it becomes essential to be able to access a build tool like Gradle to automate the grunt work (e.g., copying files, one step compiling) typically associated with the build process for Java applications.

Gradle can be downloaded for free from <http://www.gradle.org/downloads>. Gradle is available in both source code and binary versions. Since Java tools are cross-platform, I recommend you download the binary version to avoid the additional compilation step. At the time of this writing the latest stable release of Gradle is the 1.11 version.

Once you download Gradle, ensure you have a Java SDK installed on your system as Gradle requires it at run-time. Proceed to install Gradle by unpacking it and defining the `JAVA_HOME` and `PATH` environment variables.

```
www@ubuntu:~$ unzip gradle-1.5-bin.zip
Archive: gradle-1.5-bin.zip
  creating: gradle-1.5/
  inflating: gradle-1.5/getting-started.html
  inflating: gradle-1.5/LICENSE
  inflating: gradle-1.5/NOTICE
....
....
# Add JAVA_HOME variable
www@ubuntu:~$ export JAVA_HOME=/usr/lib/jvm/java-6-openjdk/
# Add Maven executable to PATH variable
www@ubuntu:~$ export PATH=$PATH:/home/www/gradle-1.5/bin/
```

Notice the Gradle binary installation is straightforward. After you unpack the Gradle binary, you need to define the `JAVA_HOME` environment so Gradle has access to Java libraries at run-time. In addition you should also add the Gradle bin directory to the `PATH` environment variable so Gradle is accessible from any location on your system.

Tip If you declare the variables `JAVA_HOME` and `PATH` as illustrated previously, you'll need to do this process every time you open a new shell session to use Gradle. On Unix/Linux systems you can open the `.bashrc` file inside a user's home directory and add the same export lines to avoid the need to declare the environment variables each session. On Windows systems you can set environment variables permanently by selecting the 'My Computer' icon, clicking on the right mouse button and then selecting the 'Properties' option. Then in the pop-up window select the 'Advanced' tab and click on the 'Environment variables' button.

The Gradle executable is available through the `gradle` command. If you set the environment variables correctly as described previously, typing `gradle` from any directory on your system invokes Gradle. Describing any more details about Gradle execution would go beyond the scope of this recipe. However, since the book's source code has numerous Spring applications that use Gradle, I'll describe how to use Gradle to build one of these Spring applications.

Once you download the book's source and unpack it to a local directory, go to the directory called `Ch1/springintro`. Type `gradle` to invoke Gradle and build the application under `springintro`. The output should look like the following

```
www@ubuntu:/www/springrecipes/Ch1/springintro$ gradle
:springintro:clean UP-TO-DATE
:springintro:compileJava
:springintro:processResources
:springintro:classes
:springintro:copyDependenciesToLibDir
:springintro:jar
:springintro:assemble
:springintro:compileTestJava UP-TO-DATE
```

```
:springintro:processTestResources UP-TO-DATE
:springintro:testClasses UP-TO-DATE
:springintro:test
:springintro:check
:springintro:build
```

BUILD SUCCESSFUL

Total time: 8.975 secs

You've just built the application, congratulations! Now let's run it. Descend into the directory called `libs` created by gradle under the `Ch1/springintro` directory. You'll see the file `springintro-1-0.SNAPSHOT.jar` which is the built application. Execute the command `java -jar springintro-1-0.SNAPSHOT.jar` to run the application. You'll see application logging messages, as well as a greeting message defined by the application.

1.6 Build a Spring application with the Gradle Wrapper

Problem

You want to build a Spring application utilizing the Gradle Wrapper from the command line.

Solution

Run the `gradlew` script from the command line.

How It Works

Although Gradle (see Recipe 1.5) is available as a standalone command line tool, a lot of (Open Source) projects use the Gradle Wrapper to give you access to Gradle. The advantage of this approach is that the application is completely self-providing. You as a developer don't need to have Gradle installed, as the Gradle Wrapper will download a specific version of Gradle to build the project.

Once you have a project that utilizes the Gradle Wrapper you can simply type `./gradlew build` on the command line to have Gradle automatically download and run the build. The only prerequisite is to have a Java SDK installed as Gradle requires it at run-time and the Gradle Wrapper needs it to run.

Once you download the book's source code and unpack it to a local directory, go to the directory called `Ch1/Recipe_1_6`. Type `./gradlew` to invoke the Gradle Wrapper and automatically build the application under `Recipe_1_6`. The output will something look like Figure 1-41.

```
marten@iMac-van-marten:~/Repositories/springrecipes/Ch1/Recipe_1_6$ ./gradlew
Downloading https://services.gradle.org/distributions/gradle-1.11-bin.zip
Unzipping /Users/marten/.gradle/wrapper/dists/gradle-1.11-bin/3htd48aunap3sf1ofa6341nhkf/gradle-1.11-bin.zip into /Users/marten/.gradle/wrapper/dists/gradle-1.11-bin/3htd48aunap3sf1ofa6341nhkf/gradle-1.11/bin/gradle
Grant executable permissions for: /Users/marten/.gradle/wrapper/dists/gradle-1.11-bin/3htd48aunap3sf1ofa6341nhkf/gradle-1.11/bin/gradle
clean
compileJava
processResources
classes
jar
assemble
completeJava UP-TO-DATE
processResources UP-TO-DATE
testClasses UP-TO-DATE
test UP-TO-DATE
check UP-TO-DATE
build

BUILD SUCCESSFUL

Total time: 10.353 secs
marten@iMac-van-marten:~/Repositories/springrecipes/Ch1/Recipe_1_6$
```

Figure 1-41. Gradle build output

TIP The source code from the book can be built with either plain Gradle or by using the Gradle Wrapper. The latter is preferable as the code will be built using the same Gradle version while developing the samples.

Summary

In this chapter you learned how to set up the most popular development tools to create Spring applications. You explored how to build and run the Spring application with five toolboxes.

Three toolboxes consisted of using IDEs. The Spring Tool Suite distributed by the creators of the Spring Framework. The Eclipse IDE distributed by the Eclipse Software foundation. And the IntelliJ IDE distributed by IntelliJ. Two toolboxes consisted of using command line tools. The Maven build tool and the newer Gradle build tool which is gaining popularity against the Maven build tool.

CHAPTER 2



Spring Core Tasks

In this chapter, you'll learn about the core tasks associated with Spring. At the heart of the Spring framework is the Spring Inversion of Control (IoC) container. The IoC container is used to manage and configure POJOs or Plain Old Java Objects.¹ Because one of the primary appeals of the Spring framework is to build Java applications with POJOs, many of Spring's core tasks involve managing and configuring POJOs in the IoC container.

So whether you plan to use the Spring framework for web applications, enterprise integration, or some other type of project, working with POJOs and the IoC container is one of the first steps you need to take to work with the Spring framework. The majority of the recipes in this chapter cover tasks that you'll use throughout the book and on a daily basis to develop Spring applications.

Note The term ‘bean’ is used interchangeably with a POJO instance both in the book and the Spring documentation. Both refer to an object instance created from a Java class. The term ‘component’ is used interchangeably with a POJO class both in the book and the Spring documentation. Both refer to the actual Java class from which object instances are created.

The source code download is organized to use gradle to build the different Recipe applications. Gradle takes care of loading all the necessary Java classes, dependencies and creating an executable. Chapter 1 describes how to setup the Gradle tool. Furthermore, if a Recipe illustrates more than one approach, the source code is classified with various examples with roman letters (e.g., Recipe_2_1_i, Recipe_2_1_ii, Recipe_2_1_iii, etc.).

Note To build each application, go inside the Recipe directory (e.g., Ch2/Recipe_2_1_i/) and execute the gradle command to compile the source code. Once the source code is compiled, a build/libs sub-directory is created with the application executable. You can then run the application JAR from the command line (e.g., java -jar Recipe_2_1_i-1.0.SNAPSHOT.jar)

¹The term POJO means an ordinary Java object without any specific requirements, such as to implement an interface or to extend a base class. This term is used to distinguish lightweight Java components from heavyweight components in other complex component models (e.g., EJB components prior to version 3.1 of the EJB specification).

2-1. Manage and Configure POJOs with the Spring IoC Container

Problem

You want to manage POJOs with Spring's IoC container.

Solution

Design a POJO class. Next, configure POJO instance values for the Spring IoC container in an XML file. Next, instantiate the Spring IoC container to gain access to the POJO instance values defined in an XML file. The POJO instances or bean instances become accessible to put together as part of an application.

How It Works

Suppose you are going to develop an application to generate sequence numbers. And that you are also going to need many series of sequence numbers for different purposes. Each sequence will have its own prefix, suffix, and initial value. So you have to create and maintain multiple generator instances for the application.

Create the POJO Class

Let's create a `SequenceGenerator` class that has three properties—`prefix`, `suffix`, and `initial`. The class will have a private field `counter` to store the numeric value of each generator. Each time you call the `getSequence()` method on a generator instance, you get the last sequence number with the prefix and suffix joined. You declare this last method as synchronized to make it thread-safe.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    private String prefix;
    private String suffix;
    private int initial;
    private int counter;
    public SequenceGenerator() {}
    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }
    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    public void setInitial(int initial) {
        this.initial = initial;
    }
}
```

```

public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(prefix);
    buffer.append(initial + counter++);
    buffer.append(suffix);
    return buffer.toString();
}
}

```

Note this last SequenceGenerator class can be instantiated by setter methods or by a standard Java constructor. When you use the Spring IoC container to initialize POJOs, if you use the standard Java constructor the mechanism is called constructor injection, where as if you use setter methods the mechanism is called setter injection.

Create a XML Configuration for your POJO

To define instances of a POJO class in the Spring IoC container, you have to create an XML configuration with instantiation values.

Spring XML configuration files can have any name, but by convention we'll call the configuration file beans.xml. You can put this file in the root of the classpath for easier testing within an IDE.

Tip Spring allows you to configure POJO instances or beans in one or more XML configuration files. For a simple application, you can just centralize your beans in a single configuration file (e.g., beans.xml). But for a large application with a lot of beans, you should separate them in multiple configuration files according to their functionalities (e.g., DAOs and controllers). One useful division is by the architectural layer of context services. Recipe 2-3, section ‘Resolve POJO references from multiple locations’ describes this process.

The Spring XML configuration allows you to use custom tags from different schemas (tx, jndi, jee, and so on) to make the bean configuration simpler and clearer. Here's an example of the simplest XML configuration possible.

```

<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-3.2.xsd">
    ...
</beans>

```

Each POJO instance or bean should have a unique name or id and a fully qualified POJO class name so the Spring IoC container to instantiate it. For each bean property of a simple type (e.g., String and other primitive types), you can specify a <value> element for it. Spring will attempt to convert your value into the declaring type of this property. To configure a property via setter injection, you use the <property> element and specify the property name in its name attribute. A <property> requires that the underlying POJO class contain a corresponding setter method.

```

<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
      <value>30</value>
    </property>

```

```
<property name="suffix">
    <value>A</value>
</property>
<property name="initial">
    <value>100000</value>
</property>
</bean>
```

You can also configure bean properties via constructor injection by declaring them in the `<constructor-arg>` elements. Note there's no name attribute in `<constructor-arg>`, because constructor arguments are position based.

```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <value>30</value>
    </constructor-arg>
    <constructor-arg>
        <value>A</value>
    </constructor-arg>
    <constructor-arg>
        <value>100000</value>
    </constructor-arg>
</bean>
```

In the Spring IoC container, each bean's name should be unique for each context. Duplicate names are allowed for overriding bean declaration if more than one context is loaded. The concept of contexts is explained in the next section 'Instantiate the Spring IoC container'.

Although a bean's name can be defined by the name attribute of the `<bean>` element, the preferred way of identifying a bean is through the standard XML id attribute. In this way, if your text editor is XML-aware, it can help to validate each bean's uniqueness at design time.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

However, be aware XML has restrictions on the characters that can appear in the XML id attribute. For this reason alone, the name attribute can be helpful if you use such special characters in a bean name. In addition, Spring allows you to specify multiple bean names for the same bean separated by commas in the name attribute, something you can't do this with the XML id attribute because commas are not allowed.

With respect to bean names it's also worth mentioning that neither the name attribute nor the id attribute is required. A bean that has no name defined is called an anonymous bean. You will usually create these types of beans for administrative purposes, mainly because application beans are referenced by name or id to be used by other application beans.

Spring also supports a shortcut for specifying the value of a simple type property. You can present a value attribute in the `<property>` element instead of enclosing a `<value>` element inside.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="suffix" value="A" />
    <property name="initial" value="100000" />
</bean>
```

This shortcut also works for constructor arguments.

```
<bean name="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <constructor-arg value="100000" />
</bean>
```

Spring also has another convenient shortcut to define properties via setter injection. It consists of using the p schema to define bean properties as attributes of the <bean> element. This can shorten the lines of XML configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        p:prefix="30" p:suffix="A" p:initial="100000" />
</beans>
```

And Spring also has another convenient shortcut to define properties via constructor injection. It consists of using the c schema to define constructor arguments as attributes of the <bean> element. This can shorten the lines of XML configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="http://www.springframework.org/schema/c"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        c:prefix = "30" c:suffix = "A" c:initial="100000" />
</beans>
```

Instantiate the Spring IoC Container

To create bean instances, you have to instantiate the Spring IoC container by reading the XML configuration files with bean instantiation values. Then, you can get the bean instances from the IoC container itself.

Spring provides two types of IoC container implementations. The basic one is called bean factory. The more advanced one is called application context, which is compatible with the bean factory. Note the configuration files for these two types of IoC containers are identical.

The application context provides more advanced features than the bean factory while keeping the basic features compatible. Therefore, we strongly recommend using the application context for every application unless the resources of an application are restricted (e.g., such as when running Spring for an applet or a mobile device).

The interfaces for the bean factory and the application context are BeanFactory and ApplicationContext, respectively. The ApplicationContext interface is a subinterface of BeanFactory for maintaining compatibility.

Since `ApplicationContext` is an interface, you have to instantiate an implementation of it. Spring has several application context implementations, we recommend you use `GenericXmlApplicationContext` which is the newest and most flexible implementation. With this class you can load the XML configuration file from the classpath.

```
ApplicationContext context = new GenericXmlApplicationContext("beans.xml");
```

Once the application context is instantiated, the object reference—in this case `context`—provides an entry point to access the POJO instances or beans.

Get POJO Instances or Beans from the IoC Container

To get a declared bean from a bean factory or an application context, you just make a call to the `getBean()` method and pass in the unique bean name. The return type of the `getBean()` method is `java.lang.Object`, so you have to cast it to its actual type before using it.

```
SequenceGenerator generator = (SequenceGenerator) context.getBean("sequenceGenerator");
```

The `getBean()` method also supports another variation where you can provide the bean class name to avoid making the cast.

```
SequenceGenerator generator = context.getBean("sequenceGenerator", SequenceGenerator.class);
```

Once you reach this step you can use the POJO or bean just like any object created using a constructor outside of Spring.

A Main class for running the sequence generator application would be the following:

```
package com.apress.springrecipes.sequence;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");

        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");

        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

If everything is available in the Java classpath (the `beans.xml` file, `SequenceGenerator` POJO class and the Spring JAR dependencies), you should see the following output, along with some logging messages:

```
30100000A
30100001A
```

2-2. Create POJOs by Invoking a Constructor

Problem

You would like to create a POJO instance or bean in the Spring IoC container by invoking its constructor, which is the most common and direct way of creating beans. It is equivalent to using the new operator to create objects in Java.

Solution

Define a POJO class with a constructor or constructors. Use Spring's `<constructor-arg>` element to define constructor arguments, then for each `<property>` element Spring injects the value through the setter method. If no `<constructor-arg>` is specified, the default constructor with no arguments is invoked. You can specify the type and index attributes or the name attribute for the `<constructor-arg>` element to avoid constructor ambiguity.

How It Works

Suppose you're going to develop a shop application to sell products online. First of all, you create the Product POJO class, which has several properties, such as the product name and price. As there are many types of products in your shop, you make the Product class abstract to extend it for different product subclasses.

```
package com.apress.springrecipes.shop;

public abstract class Product {

    private String name;
    private double price;

    public Product() {}

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Getters and Setters
    ...

    public String toString() {
        return name + " " + price;
    }
}
```

Create the POJO Classes with Constructors

Then you create two product subclasses, `Battery` and `Disc`. Each of them has its own properties.

```
package com.apress.springrecipes.shop;
```

```
public class Battery extends Product {
```

```
    private boolean rechargeable;
```

```
    public Battery() {
        super();
    }
```

```
    public Battery(String name, double price) {
        super(name, price);
    }
```

```
    // Getters and Setters
```

```
    ...
```

```
}
```

```
package com.apress.springrecipes.shop;
```

```
public class Disc extends Product {
```

```
    private int capacity;
```

```
    public Disc() {
        super();
    }
```

```
    public Disc(String name, double price) {
        super(name, price);
    }
```

```
    // Getters and Setters
```

```
    ...
```

```
}
```

Create XML Configuration to Create POJOs via Constructor

To define some products in the Spring IoC container, you create the following bean configuration file:

```
<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-3.2.xsd">
```

```

<bean id="aaa" class="com.apress.springrecipes.shop.Battery">
    <property name="name" value="AAA" />
    <property name="price" value="2.5" />
    <property name="rechargeable" value="true" />
</bean>

<bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="CD-RW" />
    <property name="price" value="1.5" />
    <property name="capacity" value="700" />
</bean>
</beans>

```

If there's no `<constructor-arg>` element specified, the default constructor with no argument is invoked. Then for each `<property>` element, Spring injects the value through the setter method. The preceding bean configuration is equivalent to the following code snippet:

```

Product aaa = new Battery();
aaa.setName("AAA");
aaa.setPrice(2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc();
cdrw.setName("CD-RW");
cdrw.setPrice(1.5);
cdrw.setCapacity(700);

```

Otherwise, if one or more `<constructor-arg>` elements is specified, Spring invokes the most appropriate constructor that matches your arguments.

```

<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <constructor-arg value="AAA" />
        <constructor-arg value="2.5" />
        <property name="rechargeable" value="true" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
        <constructor-arg value="CD-RW" />
        <constructor-arg value="1.5" />
        <property name="capacity" value="700" />
    </bean>
</beans>

```

As there is no constructor ambiguity for the `Product` class and subclasses, the preceding bean configuration is equivalent to the following code snippet:

```

Product aaa = new Battery("AAA", 2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc("CD-RW", 1.5);
cdrw.setCapacity(700);

```

You can write the following Main class to test your products by retrieving them from the Spring IoC container:

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");

        Product aaa = (Product) context.getBean("aaa");
        Product cdrw = (Product) context.getBean("cdrw");
        System.out.println(aaa);
        System.out.println(cdrw);
    }
}
```

Resolve Constructor Ambiguity

When you specify one or more constructor arguments for a POJO instance or bean, Spring attempts to find an appropriate constructor in the POJO class and pass in your arguments for bean instantiation. However, if the arguments can be applied to more than one constructor, it may cause ambiguity in constructor matching. In this case, Spring may not be able to invoke your expected constructor.

Now let's use another example of the SequenceGenerator class—introduced in Recipe 2-1. Let's add a new constructor to the SequenceGenerator class with the prefix and suffix as arguments.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }
}
```

In the bean declaration (i.e., XML file), you can specify one or more constructor arguments through the <constructor-arg> elements. Spring attempts to find an appropriate constructor for that class and pass in your arguments for bean instantiation.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <property name="initial" value="100000" />
</bean>
```

It's easy for Spring to find a constructor for these two arguments, as there is only one constructor that requires two arguments. But suppose you have to add another constructor to `SequenceGenerator` with `prefix` and `initial` as arguments.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
}
```

To invoke this constructor, you make the following bean declaration to pass a prefix and an initial value. The remaining suffix is injected through the setter method.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="100000" />
    <property name="suffix" value="A" />
</bean>
```

However, if you run the application now, you will get the following result:

```
300A
301A
```

The cause of this unexpected result is that the first constructor, with `prefix` and `suffix` as arguments, has been invoked, but not the second. This is because Spring resolves both of your arguments as `String` type by default and considers the first constructor was most suitable, as no type conversion was required. To specify the expected type of your arguments, you have to set it in the `type` attribute in `<constructor-arg>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="java.lang.String" value="30" />
    <constructor-arg type="int" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Now add one more constructor to SequenceGenerator with initial and suffix as arguments.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }

    public SequenceGenerator(int initial, String suffix) {
        this.initial = initial;
        this.suffix = suffix;
    }
}
```

Next, modify the bean declaration accordingly to match this new constructor.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" value="100000" />
    <constructor-arg type="java.lang.String" value="A" />
    <property name="prefix" value="30" />
</bean>
```

If you run the application again, you may get the right result or the following unexpected result:

```
3010000null
3010001null
```

The reason for this uncertainty is that Spring internally scores each constructor for compatibility with your arguments. But during the scoring process, the order in which your arguments appear in the XML is not considered. This means that from the view of Spring, the second and the third constructors get the same score. Which one Spring picks depends on which one is matched first. According to the Java Reflection API, or more accurately the `Class.getDeclaredConstructors()` method, the constructors returned will be in an arbitrary order that may differ from the declaration order. All these factors, acting together, cause ambiguity in constructor matching.

To avoid this problem, you have to indicate the indexes of your arguments explicitly through the `index` attribute of `<constructor-arg>`. With both the `<type>` and `<index>` attributes set, Spring is able to find the expected constructor for a bean accurately.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" index="0" value="100000" />
    <constructor-arg type="java.lang.String" index="1" value="A" />
    <property name="prefix" value="30" />
</bean>
```

And yet another approach to avoid ambiguity in constructors is to specify the constructor parameter names with the name attribute.

```
<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg name="initial" value="100000" />
    <constructor-arg name="suffix" value="A" />
    <property name="prefix" value="30" />
</bean>
```

However, to make the constructor name attribute work out of the box, the code must be compiled with the debug flag enabled. This allows Spring to look up the parameter name from the constructor. If you can't compile your code with debug flag (or don't want to) you can use `@ConstructorProperties` JDK annotation to explicitly name your constructor arguments. In this case the POJO class with the `@ConstructorProperties` JDK annotation would look like:

```
public class SequenceGenerator {
    ...
    @ConstructorProperties({"prefix", "suffix"})
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }
    @ConstructorProperties({"prefix", "initial"})
    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
    @ConstructorProperties({"initial", "suffix"})
    public SequenceGenerator(int initial, String suffix) {
        this.initial = initial;
        this.suffix = suffix;
    }
}
```

2-3. Use POJO References, Auto-Wiring, and Imports to Interact with Other POJOs

Problem

The POJO instances or beans that make up an application often need to collaborate with each other to complete the application's functions. For beans to access each other, you have to specify bean references in the bean configuration file. The Spring container can auto-wire your beans to save you the trouble of configuring the wirings manually.

Solution

In the bean configuration file, you can specify a bean reference for a bean property or a constructor argument by the `<ref>` element. It's as easy as specifying a simple value by the `<value>` element. You can also enclose a bean declaration in a property or a constructor argument directly as an inner bean. To specify auto-wiring mode in the Spring IoC container you only have to specify the `autowire` attribute of `<bean>` tag.

How It Works

It can be common to have POJO instances reference one another. For example, the prefix value of the `SequenceGenerator` class — initially described in Recipe 2-1 — uses a string. However, it's possible to use another POJO instead of using a simple string. This can make it easier to adapt to future requirements, because instead of hardcoded values, values are based on programming logic contained in another POJOs.

Create Multiple POJO Classes that Reference One Another

You can create the `PrefixGenerator` interface to define the prefix generation operation.

```
package com.apress.springrecipes.sequence;

public interface PrefixGenerator {

    public String getPrefix();
}
```

A prefix generation strategy could be to use the current system date. Let's create the `DatePrefixGenerator` class that implements the `PrefixGenerator` interface.

```
package com.apress.springrecipes.sequence;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.util.Date;

public class DatePrefixGenerator implements PrefixGenerator {

    private DateFormat formatter;

    public void setPattern(String pattern) {
        this.formatter = new SimpleDateFormat(pattern);
    }

    public String getPrefix() {
        return formatter.format(new Date());
    }
}
```

The pattern of this generator will be injected through the setter method `setPattern()` and then used to create a `java.text.DateFormat` object to format the date. Note that since the pattern string won't be used once the `DateFormat` object is created, it's not necessary to store it in a private field. Next, you can declare a `DatePrefixGenerator` bean with string pattern for date formatting.

```
<bean id="datePrefixGenerator"
      class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

Specify POJO References for Setter Methods

To apply this prefix generator approach, the `SequenceGenerator` class should accept a `PrefixGenerator` object instead of a simple string. You can choose setter injection to accept this prefix generator. You have to delete the `prefix` property, its setter methods and constructors to avoid compiler errors.

```
package com.apress.springrecipes.sequence;
public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;

    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefixGenerator.getPrefix());
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

With this change a `SequenceGenerator` bean can refer to the `dataPrefixGenerator` bean as its `prefixGenerator` property by enclosing it in the `<ref>` element.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
        <ref bean="datePrefixGenerator" />
    </property>
</bean>
```

The `bean` attribute in the `<ref>` element can be a reference to any bean in the Spring IoC container, even if it's not defined in the same XML configuration file. If you are referring to a bean in the same XML file, you should use the `local` attribute, as it is an XML ID reference. Your XML editor can help to validate whether a bean with that ID exists in the same XML file (i.e., the reference integrity).

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator">
        <ref local="datePrefixGenerator" />
    </property>
</bean>
```

There is also a convenient shortcut to specify a bean reference in the `ref` attribute of the `<property>` element.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Another convenient shortcut to specify bean references is by using the `p` schema to specify bean references as attributes of the `<bean>` element. This can shorten the lines of XML configuration.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>

  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    p:suffix="A" p:initial="1000000"
    p:prefixGenerator-ref="datePrefixGenerator" />
</beans>
```

To distinguish a bean reference from a simple property value, you have to add the `-ref` suffix to the property name.

Specify POJO References for Constructor Arguments

Bean references can also be applied to constructor injection. For example, you can add a constructor that accepts a `PrefixGenerator` object as an argument.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
  ...
  private PrefixGenerator prefixGenerator;

  public SequenceGenerator(PrefixGenerator prefixGenerator) {
    this.prefixGenerator = prefixGenerator;
  }
}
```

In the `<constructor-arg>` element, you can enclose a bean reference by `<ref>` just like in the `<property>` element.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg>
    <ref local="datePrefixGenerator" />
  </constructor-arg>
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

The shortcut for specifying a bean reference also works for `<constructor-arg>`.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg ref="datePrefixGenerator" />
  ...
</bean>
```

Declare Inner or Anonymous POJO References

Whenever a bean instance is used for only one particular property, it can be declared as an inner bean. An inner bean declaration is enclosed in `<property>` or `<constructor-arg>` directly, without any `id` or `name` attribute. In this way, the bean is anonymous and therefore can't be used anywhere else. In fact, even if you define an `id` or a `name` attribute for an inner bean, it's ignored.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />

  <property name="prefixGenerator">
    <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
      <property name="pattern" value="yyyyMMdd" />
    </bean>
  </property>
</bean>
```

An inner bean can also be declared in a constructor argument.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <constructor-arg>
    <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
      <property name="pattern" value="yyyyMMdd" />
    </bean>
  </constructor-arg>
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
```

Auto-Wire POJOs

To specify the auto-wiring mode in the Spring IoC container you only have to specify the `autowire` attribute of `<bean>` element. Table 2-1 lists the auto-wiring modes supported by Spring.

Table 2-1. Auto-Wiring Modes Supported by Spring

Mode	Description
no*	No auto-wiring will be performed. You must wire the dependencies explicitly.
byName	For each bean property, wire a bean with the same name as the property.
byType	For each bean property, wire a bean whose type is compatible with that of the property. If more than one bean is found, an <code>UnsatisfiedDependencyException</code> will be thrown.
constructor	For each argument of each constructor, first find a bean whose type is compatible with the argument. Then, pick the constructor with the most matching arguments. In case of any ambiguity, an <code>UnsatisfiedDependencyException</code> will be thrown.

* The default mode is no, but this can be changed by setting the `default-autowire` attribute of the `<beans>` root element. This default mode can be overridden by a bean's own mode if specified.

Tip Although the auto-wiring feature is very powerful, the cost is that it reduces the readability of bean configurations. Because auto-wiring is performed by Spring at runtime, you cannot derive how beans are wired from the bean configuration file. In practice, we recommend applying auto-wiring only in applications whose component dependencies are not complicated.

For example, to auto wire by type you can set the `autowire` attribute of the `sequenceGenerator` bean to `byType` and leave the `prefixGenerator` property unset. Then, Spring will attempt to wire a bean whose type is compatible with `PrefixGenerator`. In this case, the `dataPrefixGenerator` bean is wired automatically.

```
<beans ...>
  <bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator"
    autowire="byType">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>

  <bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
  </bean>
</beans>
```

Resolve Auto-Wire Ambiguity with the Primary Attribute

The main problem of auto-wiring by type is that sometimes there is more than one bean in the IoC container compatible with the target type. In this case, Spring is not be able to decide which bean is most suitable for the property, and hence cannot perform auto-wiring.

For example, if you have multiple beans that are based on the same interface and try to auto-wire by interface, auto-wiring by type won't work. Spring will throw an `UnsatisfiedDependencyException` if more than one bean is found for auto-wiring.

To help Spring decide which bean to use in case there is more than one bean compatible with the target type to auto-wire, you can use the `primary` attribute on a bean definition. The `primary` attribute gives preference to a bean when multiple candidates are qualified to autowire a single-valued dependency.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator" primary="true"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    </bean>

    <bean id="numberprefixGenerator"
        class="com.apress.springrecipes.sequence.NumberPrefixGenerator">
    </bean>
</beans>
```

Resolve Auto-Wire Ambiguity with the ByName Attribute

Another mode of auto-wiring by name using the `byName` attribute, which can sometimes resolve the problems of auto-wiring by type. It works very similarly to `byType`, but in this case, Spring attempts to wire a bean whose class name is the same as the property name, rather than with the compatible type. As the bean name is unique within a container, auto-wiring by name does not cause ambiguity.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byName">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="prefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

However, auto-wiring by name may not work in all cases. Sometimes, it's not possible to make the name of the target bean the same as the property. So in practice, you often need to specify ambiguous dependencies explicitly while keeping others beans auto-wired. That means you employ a mixture of explicit wiring and auto-wiring.

Another alternative to auto-wiring is to do it by constructor. This process works like auto-wiring by type, but it's more complicated. For a bean with a single constructor, Spring attempts to wire a bean with a compatible type for each constructor argument. But for beans with multiple constructors this process gets more complicated. Spring first attempts to find a bean with a compatible type for each argument of each constructor. Then, it picks the constructor with the most matching arguments.

Suppose that `SequenceGenerator` has one default constructor and one constructor with an argument `PrefixGenerator`.

```
package com.apress.springenterpriserecipes.sequence;

public class SequenceGenerator {

    public SequenceGenerator() {}

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
    ...
}
```

In this case, the second constructor is matched and picked because Spring can find a bean whose type is compatible with `PrefixGenerator`.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="constructor">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

However, multiple constructors in a class can cause ambiguity for constructor argument matching. The situation may be further complicated if you ask Spring to determine a constructor for you.

As you have seen, if Spring finds more than one candidate bean for auto-wiring, it will throw an `UnsatisfiedDependencyException`. In addition, if the auto-wiring mode is set to `byName` or `byType`, and Spring cannot find a matching bean to wire, it leaves the property unset, which may cause a `NullPointerException` or a value that has not been initialized.

So, if you use this auto-wiring mode, take great care to avoid ambiguity and try to use the following guidelines to avoid it:

- Exclusively use constructor injection (instead of setter injection) to ensure the right properties are set.
- Create setters with a dedicated init method.
- Create setters with `@Required` annotation when the property is required.
- Use the `@Autowired` annotation which implies a required property by default.

Note The @Required and @Autowired annotations are described in the next chapter.

And to avoid auto-wiring issues altogether, it's also possible to exclude beans from auto-wiring by adding the autowire-candidate attribute of the <bean> element to false. This technique is useful for beans you never want to be injected into other beans by autowiring. It does not mean that an excluded bean cannot itself be configured using autowiring. Rather, the bean itself is not a candidate for autowiring other beans.

Resolve POJO References from Multiple Locations

As an application grows it can become difficult to manage every POJO in a single configuration file (e.g., beans.xml). A common practice is to separate POJOs into multiple configuration files according to their functionalities. When you create multiple configuration files, obtaining references and auto-wiring POJOs that are defined in different files isn't as straightforward as when everything is in a single configuration file.

One approach is to initialize the application context with the location of each configuration file. In this manner, the POJOs for each configuration file are loaded into the context and references and autowiring between POJOs is possible.

```
ApplicationContext context = new GenericXmlApplicationContext(new String[]
 {"beans.xml", "generators.xml"});
```

Instead of initializing the application context with a String configuration file, you use a String array to specify multiple configuration files.

Another alternative is to use the <import> tag so Spring makes the POJOs from one configuration file available in another.

```
<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-3.2.xsd">

    <import resource="generators.xml"/>

    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          autowire="byType" depends-on="datePrefixGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>
</beans>
```

The sequenceGenerator bean requires the datePrefixGenerator bean due to the depends-on attribute. But notice the datePrefixGenerator bean is not defined in the configuration file. The datePrefixGenerator bean is defined in a separate configuration file generators.xml. With the <import resource="generators.xml"/> statement, Spring brings all the POJOs in generators.xml into the scope of the present configuration file where they can be referenced or auto-wired.

2-4. Configure POJOs with Java Collection Attributes

Problem

You want to configure POJOs with Java collection attributes.

Solution

Use the Spring tags `<list>`, `<set>`, and `<map>`.

How It Works

Most POJOs use simple type properties (e.g., String and other primitive types), but it can also be common to have POJOs that have more elaborate properties like collections. The Spring IoC container can also instantiate POJOs with Java collection attributes.

`List`, `Set`, and `Map` are the core interfaces representing the three main types of collections in the Java SDK, part of a framework called the Java Collections framework. For each collection type, Java provides several implementations with different functions and characteristics from which you can choose. In Spring, these types of collections are configured with a group of built-in XML tags: `<list>`, `<set>`, and `<map>`.

Based on the application presented in Recipe 2-1, suppose you're going to allow more than one suffix for a sequence generator, with the suffixes appended to the sequence numbers with hyphens as the separators. This allows the sequence generator to accept suffixes of arbitrary data types and convert them to strings when appending to the sequence numbers.

Lists, Arrays, and Sets

First, let's use a `java.util.List` collection to contain the suffixes. A list is an ordered and indexed collection whose elements can be accessed either by index or with a for-each loop.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;

    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(suffix);
        }
        return buffer.toString();
    }
}
```

To define a property of the interface `java.util.List` in the bean configuration, you specify a `<list>` tag that contains the elements. The elements allowed inside the `<list>` tag can be a simple constant value specified by `<value>`, a bean reference specified by `<ref>`, an inner bean definition specified by `<bean>`, an ID reference definition specified by `<idref>`, or a null element specified by `<null>`. You can even embed other collections in a collection.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">

    <property name="prefix" value="30" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <list>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </list>
    </property>
</bean>
```

Conceptually, an array is very similar to a list in that it's also an ordered and indexed collection that can be accessed by index. The main difference is that the length of an array is fixed and cannot be extended dynamically. In the Java Collections framework, an array and a list can be converted to each other through the `Arrays.asList()` and `List.toArray()` methods. For the sequence generator, you can use an `Object[]` array to contain the suffixes and access them either by index or with a for-each loop.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Object[] suffixes;

    public void setSuffixes(Object[] suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

The definition of an array in the bean configuration file is identical to a list denoted by the `<list>` tag.

Another common collection type is a set. Both the `java.util.List` interface and the `java.util.Set` interface extend the same interface: `java.util.Collection`. A set differs from a list in that it is neither ordered nor indexed, and it can store unique objects only. That means no duplicate element can be contained in a set. When the same element is added to a set for the second time, it will replace the old one. The equality of elements is determined by the `equals()` method.

```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Set<Object> suffixes;
    public void setSuffixes(Set<Object> suffixes) {
        this.suffixes = suffixes;
    }
    ...
}

```

To define a property of `java.util.Set` type, use the `<set>` tag to define the elements in the same way as a list.

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <set>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
            <!-- The following is ignored because the collection is a set -->
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </set>
    </property>
</bean>

```

Although there's no order concept in the original set semantics, Spring preserves the order of your elements by using `java.util.LinkedHashSet`, an implementation of the `java.util.Set` interface that does preserve element order.

Maps and Properties

A map interface is a table that stores its entries in key/value pairs. You can get a particular value from a map by its key, and also iterate the map entries with a for-each loop. Both the keys and values of a map can be of arbitrary type. Equality between keys is also determined by the `equals()` method. For example, you can modify your sequence generator to accept a `java.util.Map` collection that contains suffixes with keys.

```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Map<Object, Object> suffixes;

    public void setSuffixes(Map<Object, Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Map.Entry entry : suffixes.entrySet()) {
            buffer.append("-");
            buffer.append(entry.getKey());
            buffer.append("@");
            buffer.append(entry.getValue());
        }
        return buffer.toString();
    }
}

```

In Spring, a map is defined by the `<map>` tag, with multiple `<entry>` tags as children. Each entry contains a key and a value. The key must be defined inside the `<key>` tag. There is no restriction on the type of the key and value, so you are free to specify a `<value>`, `<ref>`, `<bean>`, `<idref>`, or `<null>` element for them. Spring will also preserve the order of the map entries by using `java.util.LinkedHashMap`.

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <map>
            <entry>
                <key>
                    <value>type</value>
                </key>
                <value>A</value>
            </entry>
            <entry>
                <key>
                    <value>url</value>
                </key>
                <bean class="java.net.URL">
                    <constructor-arg value="http" />
                    <constructor-arg value="www.apress.com" />
                    <constructor-arg value="/" />
                </bean>
            </entry>
        </map>
    </property>
</bean>

```

There are shortcuts to defining map keys and values as attributes of the `<entry>` tag. If they are simple constant values, you can define them by key and value. If they are bean references, you can define them by `key-ref` and `value-ref`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <map>
        <entry key="type" value="A" />
        <entry key="url">
          <bean class="java.net.URL">
            <constructor-arg value="http" />
            <constructor-arg value="www.apress.com" />
            <constructor-arg value="/" />
          </bean>
        </entry>
      </map>
    </property>
  </bean>
```

In all the collection classes seen thus far you've used values. Sometimes, the desired goal is to configure a null value using a Map instance. Spring's XML configuration schema includes explicit support for this. Here is a map with null values for the value of an entry:

```
<property name="nulledMapValue">
  <map>
    <entry>
      <key> <value>null</value> </key>
    </entry>
  </map>
</property>
```

A `java.util.Properties` collection is very similar to a map. It also implements the `java.util.Map` interface and stores entries in key/value pairs. The only difference is that the keys and values of a Properties collection are always strings.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Properties suffixes;

  public void setSuffixes(Properties suffixes) {
    this.suffixes = suffixes;
  }
  ...
}
```

To define a `java.util.Properties` collection in Spring, use the `<props>` tag with multiple `<prop>` tags as children. Each `<prop>` tag must have a key attribute defined and the corresponding value enclosed.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <props>
        <prop key="type">A</prop>
        <prop key="url">http://www.apress.com</prop>
        <prop key="null">null</prop>
      </props>
    </property>
  </bean>
```

Specify the Data Type for Collection Elements

By default, Spring treats every element in a collection as a string. You have to specify the data type for your collection elements if you are not going to use them as strings. You can either specify the data type for each collection element by the `type` attribute of the `<value>` tag, or specify the data type for all elements by the `value-type` attribute of the `collection` tag. Another approach available is to define a type-safe collection so that Spring reads a collection's type information.

Suppose you want to accept a list of integer numbers as the suffixes of your sequence generator. Each number will be formatted into four digits by an instance of `java.text.DecimalFormat`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private List<Object> suffixes;

  public void setSuffixes(List<Object> suffixes) {
    this.suffixes = suffixes;
  }

  public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    ...
    DecimalFormat formatter = new DecimalFormat("0000");
    for (Object suffix : suffixes) {
      buffer.append("-");
      buffer.append(formatter.format((Integer) suffix));
    }
    return buffer.toString();
  }
}
```

Then define several suffixes for your sequence generator in the bean configuration file as usual.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  <property name="prefixGenerator" ref="datePrefixGenerator" />
  <property name="initial" value="100000" />
  <property name="suffixes">
    <list>
      <value>5</value>
      <value>10</value>
      <value>20</value>
    </list>
  </property>
</bean>
```

If you run the application this way, you will encounter a `ClassCastException`, indicating the suffixes can't be cast into integers because their type is a `String`. Spring treats every element in a collection as a string by default. You have to set the `type` attribute of the `<value>` tag to specify the element type.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list>
      <value type="int">5</value>
      <value type="int">10</value>
      <value type="int">20</value>
    </list>
  </property>
</bean>
```

Or you may set the `value-type` attribute of the `collection` tag to specify the type for all elements in this collection.

```
<bean id="sequenceGenerator"
  class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list value-type="int">
      <value>5</value>
      <value>10</value>
      <value>20</value>
    </list>
  </property>
</bean>
```

And another alternative is to define the suffixes list with a type-safe collection that stores integers.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private List<Integer> suffixes;
```

```

public void setSuffixes(List<Integer> suffixes) {
    this.suffixes = suffixes;
}

public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    ...
    DecimalFormat formatter = new DecimalFormat("0000");
    for (int suffix : suffixes) {
        buffer.append("-");
        buffer.append(formatter.format(suffix));
    }
    return buffer.toString();
}
}

```

Once you define your collections in a type-safe way, Spring is able to read the collection's type information through reflection. In this way, you no longer need to specify the value-type attribute of <list>.

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>

```

Specify Concrete Classes for Collections

When using the basic collection tags to define collections, you can't specify the concrete class of a collection, such as `LinkedList`, `TreeSet`, or `TreeMap`. Moreover, you cannot share a collection among different beans by defining it as a stand-alone bean for other beans to refer to.

Spring provides a couple of options to overcome the shortcomings of the basic collection tags. One option is to use a collection factory beans like `ListFactoryBean`, `SetFactoryBean`, and `MapFactoryBean`. A factory bean is a special kind of Spring bean that is used to create another bean. The second option is to use collection tags such as `<util:list>`, `<util:set>`, and `<util:map>` in the `util` schema.

You can use a collection factory bean to define a collection and specify its target class. For example, you can specify the `targetSetClass` property for `SetFactoryBean`. Then Spring instantiates the specified class for the collection.

```

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <bean class="org.springframework.beans.factory.config.SetFactoryBean">

```

```

<property name="targetSetClass">
    <value>java.util.TreeSet</value>
</property>
<property name="sourceSet">
    <set>
        <value>5</value>
        <value>10</value>
        <value>20</value>
    </set>
</property>
</bean>
</property>
</bean>

```

Or you can use a collection tag in the util schema to define a collection and set its target class (e.g., by the `set-class` attribute of `<util:set>`). But you must remember to add the util schema definition to your `<beans>` root element.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<bean id="sequenceGenerator"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="initial" value="100000" />
    <property name="suffixes">
        <util:set set-class="java.util.TreeSet">
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </util:set>
    </property>
</bean>
</beans>

```

An advantage of collection factory beans is that you can define a collection as a stand-alone bean for other beans to refer to. For example, you can define a stand-alone set by using `SetFactoryBean`.

```

<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes" />
        </property>
    </bean>

```

```

<bean id="suffixes"
    class="org.springframework.beans.factory.config.SetFactoryBean">
    <property name="sourceSet">
        <set>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </set>
    </property>
</bean>
...
</beans>
```

Or you can define a stand-alone set by using the `<util:set>` tag in the util schema.

```

<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes"/>
        </property>
    </bean>

    <util:set id="suffixes">
        <value>5</value>
        <value>10</value>
        <value>20</value>
    </util:set>
    ...
</beans>
```

2-5. Set a POJOs Scope

Problem

When you declare a POJO instance or bean in the configuration file, you are actually defining a template for bean creation, not an actual bean instance. When a bean is requested by the `getBean()` method or a reference from other beans, Spring decides which bean instance should be returned according to the bean scope. Sometimes, you have to set an appropriate scope for a bean other than the default scope.

Solution

A bean's scope is set in the `scope` attribute of the `<bean>` element. By default, Spring creates exactly one instance for each bean declared in the IoC container, and this instance is shared in the scope of the entire IoC container. This unique bean instance is returned for all subsequent `getBean()` calls and bean references. This scope is called singleton, which is the default scope of all beans. Table 2-2 lists all valid bean scopes in Spring.

Table 2-2. Valid Bean Scopes in Spring

Scope	Description
singleton	Creates a single bean instance per Spring IoC container
prototype	Creates a new bean instance each time when requested
request	Creates a single bean instance per HTTP request; only valid in the context of a web application
session	Creates a single bean instance per HTTP session; only valid in the context of a web application
globalSession	Creates a single bean instance per global HTTP session; only valid in the context of a portal application

How It Works

To demonstrate the concept of bean scope, let's consider a shopping cart example in a shopping application. First, you create the `ShoppingCart` class as follows:

```
package com.apress.springrecipes.shop;
...
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();

    public void addItem(Product item) {
        items.add(item);
    }

    public List<Product> getItems() {
        return items;
    }
}
```

Then you declare some product beans and a shopping cart bean in the IoC container as usual:

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <property name="name" value="AAA" />
        <property name="price" value="2.5" />
        <property name="rechargeable" value="true" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
        <property name="capacity" value="700" />
    </bean>
```

```

<bean id="dvdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="DVD-RW" />
    <property name="price" value="3.0" />
    <property name="capacity" value="700" />
</bean>

<bean id="shoppingCart" class="com.apress.springrecipes.shop.ShoppingCart" />
</beans>

```

In the following Main class, you can test your shopping cart by adding some products to it. Suppose that there are two customers navigating in your shop at the same time. The first one gets a shopping cart by the getBean() method and adds two products to it. Then, the second customer also gets a shopping cart by the getBean() method and adds another product to it.

```

package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");

        Product aaa = (Product) context.getBean("aaa");
        Product cdrw = (Product) context.getBean("cdrw");
        Product dvdrw = (Product) context.getBean("dvdrw");

        ShoppingCart cart1 = (ShoppingCart) context.getBean("shoppingCart");
        cart1.addItem(aaa);
        cart1.addItem(cdrw);
        System.out.println("Shopping cart 1 contains " + cart1.getItems());

        ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
        cart2.addItem(dvdrw);
        System.out.println("Shopping cart 2 contains " + cart2.getItems());
    }
}

```

As a result of the preceding bean declaration, you can see that the two customers get the same shopping cart instance.

```

Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
Shopping cart 2 contains [AAA 2.5, CD-RW 1.5, DVD-RW 3.0]

```

This is because Spring's default bean scope is singleton, which means Spring creates exactly one shopping cart instance per IoC container.

```

<bean id="shoppingCart"
    class="com.apress.springrecipes.shop.ShoppingCart"
    scope="singleton" />

```

In your shop application, you expect each customer to get a different shopping cart instance when the `getBean()` method is called. To ensure this behavior, the scope of the `shoppingCart` bean needs to be set to `prototype`. Then Spring creates a new bean instance for each `getBean()` method call.

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="prototype" />
```

Now if you run the `Main` class again, you can see the two customers get a different shopping cart instance.

```
Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
Shopping cart 2 contains [DVD-RW 3.0]
```

2-6. Use Data from External Resources (Text Files, XML Files, Properties Files, or Image Files)

Problem

Sometimes, applications need to read external resources (e.g., text files, XML files, properties file, or image files) from different locations (e.g., a file system, classpath, or URL). Usually, you have to deal with different APIs for loading resources from different locations.

Solution

Spring offers the `PropertySourcesPlaceholderConfigurer` class as a facility to load the contents of `.properties` file (i.e. key-value pairs) to set up bean properties.

Spring also has a resource loader mechanism which provides a unified `Resource` interface to retrieve any type of external resource by a resource path. You can specify different prefixes for this path to load resources from different locations. To load a resource from a file system, you use the `file` prefix. To load a resource from the classpath, you use the `classpath` prefix. You can also specify a URL in the resource path.

How It Works

To read the contents of a `properties` file (i.e., key-value pairs) to setup bean properties you can use Spring's `PropertySourcesPlaceholderConfigurer` class. If you want to read the contents of any file you can use Spring's `Resource` mechanism.

Use Properties File data to Setup POJO Instantiation Values

Let's assume you have a series of values in a `properties` file you want to access to setup bean properties. Typically this can be the configuration properties of a database or some other application values composed of key-values. For example, take the following key-values stored in a file called `discounts.properties`.

```
specialcustomer.discount=0.1
summer.discount=0.15
endofyear.discount=0.2
```

Note To read properties files for the purpose of internationalization (i18n) see the next recipe

To make the contents of the `discounts.properties` file accessible to set up other beans, you can use the `PropertySourcesPlaceholderConfigurer` class to convert the key-values into a bean.

```
<bean id="discountPropertyConfigurer"
      class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="location">
      <value>classpath:discounts.properties</value>
    </property>
    <property name="ignoreResourceNotFound" value="true"/>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
</bean>
```

You define a `location` property for the bean with a value of `classpath:discounts.properties`. The `classpath:` prefix tells Spring to look for the `discounts.properties` file in the Java classpath.

The properties `ignoreResourceNotFound` and `ignoreUnresolvablePlaceholders` are defined explicitly to ignore potential errors in the properties file. The property `ignoreResourceNotFound` set to true, tells Spring to ignore errors if the properties file is not found. The property `ignoreUnresolvablePlaceholders` set to true, tells Spring to ignore attempts to access non-existent keys in the properties file. If these last properties are not explicitly set to true, they default to false and generate exceptions if either the properties file doesn't exist or an attempt is made to access non-existent keys.

Once the `PropertySourcesPlaceholderConfigurer` bean is declared, you can access the values of the `discounts.properties` file to setup other bean properties.

If you add a `discount` property to the `Product`, `Battery` and `Disc` classes of the shopping application, you can set a bean property using values from a properties file.

```
<bean id="aaa" class="com.apress.springrecipes.shop.Battery">
  <property name="name" value="AAA" />
  <property name="price" value="2.5" />
  <property name="discount" value="${specialcustomer.discount:0}" />
  <property name="rechargeable" value="true" />
</bean>

<bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
  <property name="name" value="CD-RW" />
  <property name="price" value="1.5" />
  <property name="discount" value="${summer.discount:0}" />
  <property name="capacity" value="700" />
</bean>
```

The syntax is `${key:default_value}`. A search is done for the key value in all the loaded application properties. If a matching key=value is found in the properties file, the corresponding value is assigned to the bean property. If no matching key=value is found in the loaded application properties, the `default_value` (i.e., after `${key:}`) is assigned to the bean property.

If you want to use properties file data for a different purpose than setting up bean properties, you should use Spring's Resource mechanism which is described next.

Use Data from any External Resource File for use in a POJO

Suppose you want to display a banner at the startup of an application. The banner is made up of the following characters and stored in a text file called `banner.txt`. This file can be put in the classpath of your application.

```
*****
* Welcome to My Shop! *
*****
```

Next, let's write a `BannerLoader` POJO class to load the banner and output it to the console.

```
package com.apress.springrecipes.shop;

import org.springframework.core.io.Resource;
...
public class BannerLoader {

    private Resource banner;

    public void setBanner(Resource banner) {
        this.banner = banner;
    }

    public void showBanner() throws IOException {
        InputStream in = banner.getInputStream();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        while (true) {
            String line = reader.readLine();
            if (line == null)
                break;
            System.out.println(line);
        }
        reader.close();
    }
}
```

Notice the POJO `banner` field is a Spring `Resource` type. The field value will be populated through setter injection when the bean instance is created (to be explained shortly). The `showBanner` method makes a call to the `getInputStream()` method to retrieve the input stream from the `Resource` field. Once you have an `InputStream`, you're able to use standard Java file manipulation class. In this case, the file contents are read line by line with `BufferedReader` and outputted to the console.

Finally, you declare a `BannerLoader` instance in the bean configuration file to display the banner.

```
<bean id="bannerLoader"
      class="com.apress.springrecipes.shop.BannerLoader"
      init-method="showBanner">
    <property name="banner">
        <value>classpath:banner.txt</value>
    </property>
</bean>
```

Because you want to show the banner at startup, you specify the `showBanner()` method as the initialization method so it's invoked automatically upon creation. Also notice the banner property with the `classpath:banner.txt` value. This tells Spring to search for the `banner.txt` file in the classpath and inject it as the bean's banner property. Spring uses the preregistered property editor `ResourceEditor` to convert the file definition into a `Resource` object before injecting it into the bean.

Because the banner file is located in the Java classpath, the resource path starts with the `classpath:` prefix. The previous resource path specifies a resource in the relative path of the file system. You can specify an absolute path as well.

```
file:c:/shop/banner.txt
```

When a resource is located in Java's classpath, you have to use the `classpath` prefix. If there's no path information presented, it will be loaded from the root of the classpath.

```
classpath:banner.txt
```

If the resource is located in a particular package, you can specify the absolute path from the classpath root.

```
classpath:com/apress/springrecipes/shop/banner.txt
```

Besides support to load from a file system path or the classpath, a resource can also be loaded by specifying a URL.

```
http://springrecipes.apress.com/shop/banner.txt
```

Since the bean instance uses the `init-method` to automatically invoke the `showBanner()` method, the banner is sent to output when the IoC container is setup. Because of this, there's no need to tinker with an application's context or explicitly call the bean to output the banner. However, sometimes it can be necessary to access an external resource to interact with an application's context.

Now suppose you want to display a legend at the end of an application. The legend is made up of the discounts previously described in the `discounts.properties` file. To access the contents of the properties file you can also leverage Spring's `Resource` mechanism.

Next, let's use Spring's Resource mechanism, but this time directly inside an application's Main class to output a legend when the application finishes.

```
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.support.PropertiesLoaderUtils;
...
...
public class Main {

    public static void main(String[] args) throws Exception {
    ...
    ...
        ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
        cart2.addItem(dvd);
        System.out.println("Shopping cart 2 contains " + cart2.getItems());

        Resource resource = new ClassPathResource("discounts.properties");
        Properties props = PropertiesLoaderUtils.loadProperties(resource);
        System.out.println("And don't forget our discounts!");
        System.out.println(props);
    }
}
```

Spring's `ClassPathResource` class is used to access the `discounts.properties` file, which casts the file's contents into a `Resource` object. Next, the `Resource` object is processed into a `Properties` object with Spring's `PropertiesLoaderUtils` class. Finally, the contents of the `Properties` object are sent to the console as the final output of the application.

Because the legend file (i.e., `discounts.properties`) is located in the Java classpath, the resource is accessed with Spring's `ClassPathResource` class. If the external resource were in a file system path the resource would be loaded with Spring's `FileSystemResource`.

```
Resource resource = new FileSystemResource("c:/shop/banner.txt")
```

If the external resource were at a URL the resource would be loaded with Spring's `UrlResource`.

```
Resource resource = new UrlResource("http://www.apress.com/context/banner.txt")
```

2-7. Resolve I18N Text Messages for Different Locales in Properties Files

Problem

For an application to support internationalization (I18N for short, as there are 18 characters between the first character, "i," and the last character, "n"), it requires the capability to resolve text messages for different locales.

Solution

Spring's application context is able to resolve text messages for target locales by keys. Typically, the messages for one locale are stored in separate properties file. These properties files are called resource bundles and follow the naming convention `messages_<language_code>_<country_code>.properties`.

`MessageSource` is an interface that defines several methods for resolving messages in resource bundles. The `ApplicationContext` interface extends this interface so that all application contexts are able to resolve text messages. An application context delegates the message resolution to a bean with the name `messageSource`. `ResourceBundleMessageSource` is the most common `MessageSource` implementation that resolves messages from resource bundles for different locales.

How It Works

As an example, create the following resource bundle, `messages_en_US.properties`, for the English language in the United States. Resource bundles are loaded from the root of the classpath, so ensure it's available on the Java classpath. Place the following key-value in the file:

```
alert.checkout=A shopping cart has been checked out.
```

To resolve messages from resource bundles, we'll use the `ReloadableResourceBundleMessageSource` as the `MessageSource` implementation. This bean's name must be set to `messageSource` for the application context to detect it. You have to specify a list of basenames to locate bundles for `ResourceBundleMessageSource`. For this case, we just specify the default convention to lookup files located in Java classpath that start with `messages`. In addition, the `cacheSeconds` property set to 1 to avoid reading stale message. Note that a refresh attempt first checks the last-modified timestamp of the properties file before actually reloading it; so if files don't change, the `cacheSeconds` interval can be set rather low, as refresh attempts aren't actually reloaded.

```
<beans ...>
  ...
  <bean id="messageSource"
    class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
      <list>
        <value>classpath:messages</value>
      </list>
    </property>
    <property name="cacheSeconds" value="1"/>
  </bean>
</beans>
```

For this `MessageSource` definition, if you look up a text message for the United States locale, whose preferred language is English, the resource bundle `message_en_US.properties` is considered first. If there's no such resource bundle or the message can't be found, then a `message_en.properties` file that matches the language is considered. If a resource bundle still can't be found, the default `message.properties` for all locales is chosen. For more information on resource bundle loading, you can refer to the Javadoc of the `java.util.ResourceBundle` class.

Next, you can configure the application context to resolve messages with the `getMessage()` method. The first argument is the key corresponding to the message, and the third is the target locale.

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");
        ...
        String alert = context.getMessage("alert.checkout", null, Locale.US);
        System.out.println(alert);
    }
}
```

The second argument of the `getMessage()` method is an array of message parameters. In the text message, you can define multiple parameters by index:

`alert.inventory.checkout=A shopping cart with {0} has been checked out at {1}.`

You have to pass in an object array to fill in the message parameters. The elements in this array are converted into strings before filling in the parameters.

```
package com.apress.springrecipes.shop;
...
public class Main {

    public static void main(String[] args) throws Exception {
        ...
        String alert = context.getMessage("alert.checkout",
            new Object[] { 4, new Date(), Locale.US });
        System.out.println(alert);
    }
}
```

In the Main class, you can resolve text messages because you can access the application context directly. But for a bean to resolve text messages, you have to inject a MessageSource implementation into the bean that needs to resolve test message. Let's implement a Cashier class for the shopping application that illustrates how to resolve messages.

```
package com.apress.springrecipes.shop;
...

public class Cashier {

    private MessageSource messageSource;

    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public void checkout(ShoppingCart cart) throws IOException {
        String alert = messageSource.getMessage("alert.inventory.checkout",
            new Object[] { cart.getItems(), new Date(),
                Locale.US });
        System.out.println(alert);
    }
}
```

Notice the POJO messageSource field is a Spring MessageSource type. The field value will be populated through setter injection when the bean instance is created (to be explained shortly). Then the checkout method can access the messageSource field, which gives the bean access to the getMessage method to gain access to text messages based on I18N criteria.

Finally, you need to setup the bean definitions to setup the proper references.

```
<bean id="messageSource"
    class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>classpath:messages</value>
        </list>
    </property>
    <property name="cacheSeconds" value="1"/>
</bean>
```

```
<bean id="cashier" class="com.apress.springrecipes.shop.Cashier">
    <property name="messageSource">
        <ref bean="messageSource"/>
    </property>
</bean>
```

2-8. Customize POJO Initialization and Destruction

Problem

Some POJOs have to perform certain types of initialization tasks before they're used. These tasks can include opening a file, opening a network/database connection, allocating memory, and so on. In addition, these same POJO also have to perform the corresponding destruction tasks at the end of their life cycle. Therefore, sometimes it's necessary to customize bean initialization and destruction in the Spring IoC container.

Solution

In addition to bean registration, the Spring IoC container also manages a POJOs life cycle. The Spring IoC container supports the addition of custom tasks at particular points in a POJOs life cycle. Custom life cycle tasks need to be encapsulated in callback methods for the Spring IoC container to call at a suitable time.

The following list shows the steps through which the Spring IoC container manages the life cycle of a bean. This list will be expanded as more features of the IoC container are introduced.

1. Create the bean instance either by a constructor or by a factory method.
2. Set the values and bean references to the bean properties.
3. Call the initialization callback methods.
4. The bean is ready to be used.
5. When the container is shut down, call the destruction callback methods.

Spring can recognize initialization and destruction callback methods by setting up the `init-method` and `destroy-method` attributes in the bean declaration and specify the callback method names.

How It Works

There are various ways to customize POJO the initialization and destruction. Next, we'll take a look at three techniques. One approach consists of defining a set of methods to run before initialization and destruction. The other technique called lazy initialization consists of delaying the POJO initialization process until a bean is required. And the last approach consists of initializing POJOs before other POJOs using the `depends-on` attribute.

Define Methods to Run Before POJO Initialization and Destruction

To illustrate how the Spring IoC container can call methods before a POJO's initialization and destruction, let's consider an example involving a checkout function for the shopping application. Let's modify the `Cashier` class from the previous recipe to record a shopping cart's products and the checkout time to a text file.

```
package com.apress.springrecipes.shop;
...
public class Cashier {

    private String fileName;
    private String path;
    private BufferedWriter writer;

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public void setPath(String path) {
        this.path = path;
    }

    public void openFile() throws IOException {
        File targetDir = new File(path);
        if (!targetDir.exists()) {
            targetDir.mkdir();
        }
        File checkoutFile = new File(path, fileName + ".txt");
        if (!checkoutFile.exists()) {
            checkoutFile.createNewFile();
        }
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(checkoutFile, true)));
    }

    public void checkout(ShoppingCart cart) throws IOException {
        writer.write(new Date() + "\t" + cart.getItems() + "\r\n");
        writer.flush();
    }

    public void closeFile() throws IOException {
        writer.close();
    }
}
```

In the `Cashier` class, the `openFile()` method first verifies if the target directory and the file to write the data exists. It then opens the text file in the specified system path and assigns it to the `writer` field. Then each time the `checkout()` method is called, the date and cart items are appended to the text file. Finally, the `closeFile()` method closes the file to release its system resources.

Next, it's necessary to declare the bean in the IoC container. We'll specify two properties to inject into the bean, a path directory and a file name.

```
<beans ...>
  ...
  <bean id="cashier" class="com.apress.springrecipes.shop.Cashier"
    init-method="openFile" destroy-method="closeFile">
    <property name="fileName" value="checkout" />
    <property name="path" value="c:/Windows/Temp/cashier" />
  </bean>
</beans>
```

Note The path value is set to c:/Windows/Temp/ because it's a Windows world-writeable directory. If you use another path, ensure it's accessible by the user that executes the application.

Spring allows a bean to perform initialization and destruction tasks by assigning POJO methods to the `init-method` and `destroy-method` attributes in a bean declaration. Note how the `init-method` attribute has the `openFile` value (i.e., the `openFile` method) and the attribute `destroy-method` attribute has the `closeFile` value (i.e., the `closeFile` method).

If these two attributes are set in the bean declaration, it indicates that when a `Cashier` bean instance is created, that it first trigger the `openFile` method—verifying if the target directory and the file to write the data exist, as well as opening the file to append records—and when the bean is destroyed that it trigger the `closeFile` method—ensuring the file reference is closed to release system resources.

Define Lazy Initialization for POJOs

By default, Spring performs eager initialization on all POJOs. This means POJOs are initialized at startup. In certain circumstances though, it can be convenient to delay the POJO initialization process until a bean is required. Delaying the initialization is called 'lazy initialization'.

Lazy initialization helps limit resource consumption peaks at startup and save overall system resources. Lazy initialization can be particularly relevant for POJOs that perform heavyweight operations (e.g., network connections, file operations). To mark a bean with lazy initialization you set the `lazy-init` attribute to true in a `<bean>` element.

```
<bean id="shoppingCart" lazy-init="true" class="com.apress.springrecipes.shop.ShoppingCart"
  scope="prototype" />
```

In the previous declaration because `lazy-init="true"`, if the POJO is never required by the application or referenced by another POJO, it's never instantiated.

Define Initialization of POJOs Before Other POJOs

As an application's POJOs grow, so does the number of POJO initializations. This can create race conditions if POJOs reference one another and are spread out in different configuration files. What happens if bean 'C' requires the logic in bean 'B' and bean 'F'? If bean 'C' is detected first and Spring hasn't initialized bean 'B' and bean 'F', you'll get an error which can be hard to detect.

To ensure that certain POJOs are initialized before other POJOs and to get a more descriptive error in case of a failed initialization process, Spring offers the depends-on attribute. The depends-on attribute ensures a given bean is initialized before another bean.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      autowire="byType" depends-on="datePrefixGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

In the previous snippet, the declaration depends-on="datePrefixGenerator" ensures the datePrefixGenerator bean is created before the sequenceGenerator bean. The depends-on attribute also supports defining multiple dependency beans with a CSV list (e.g., depends-on="datePrefixGenerator,numberPrefixGenerator,randomPrefixGenerator").

2-9. Create Post Processors to Validate and Modify POJOs

Problem

You want to apply tasks to all bean instances or specific types of instances during construction to validate or modify bean properties according to particular criteria.

Solution

A bean post processor allows bean processing before and after the initialization callback method (i.e., the one assigned to the init-method attribute). The main characteristic of a bean post processor is that it processes all the bean instances in the IoC container, not just a single bean instance. Typically, bean post processors are used to check the validity of bean properties, alter bean properties according to particular criteria or apply certain tasks to all bean instances.

The basic requirement of a bean post processor is to implement the BeanPostProcessor interface. You can process every bean before and after the initialization callback method by implementing the postProcessBeforeInitialization() and postProcessAfterInitialization() methods. Then Spring passes each bean instance to these two methods before and after calling the initialization callback method, as illustrated in the following list:

1. Create the bean instance either by a constructor or by a factory method.
2. Set the values and bean references to the bean properties.
3. Pass the bean instance to the postProcessBeforeInitialization() method of each bean post processor.
4. Call the initialization callback methods.
5. Pass the bean instance to the postProcessAfterInitialization() method of each bean post processor.
6. The bean is ready to be used.
7. When the container is shut down, call the destruction callback methods.

When using a bean factory as your IoC container, bean post processors can only be registered programmatically, or more accurately, via the `addBeanPostProcessor()` method. However, if you are using an application context, the registration is as simple as declaring an instance of the processor in the bean configuration file, and then it gets registered automatically.

How It Works

Suppose you want to audit the creation of every bean. You may want to do this to debug an application, verify the properties of every bean or some other scenario. A bean post processor is an ideal choice to implement this feature, because you don't have to modify any pre-existing POJO code.

Create POJO to Process Every Bean Instance

To write a bean post processor a class has to implement `BeanPostProcessor`. When Spring detects a bean that implements this class, it applies the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods to all bean instances managed by Spring. You can implement any logic you wish in these methods, to either inspect, modify or verify the status of a bean.

```
package com.apress.springrecipes.shop;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class AuditCheckBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("In AuditCheckBeanPostProcessor.postProcessBeforeInitialization,
processing bean type: " + bean.getClass());
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
}
```

Notice the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods must return the original bean instance even if you don't do anything in the method.

To register a bean post processor in an application context, just declare an instance of it in the bean configuration file. The application context is able to detect which bean implements the `BeanPostProcessor` interface and register it to process all other bean instances in the container.

```
<bean class="com.apress.springrecipes.shop.ProductCheckBeanPostProcessor"/>
```

Create POJO to Process Selected Bean Instances

During bean construction, the Spring IoC container passes all the bean instances to the bean post processor one by one. This means if you only want to apply a bean post processor to certain types of beans, you must filter the beans by checking their instance type. This allows you to apply logic more selectively across beans.

Suppose you want to apply a bean post processor but just to Product bean instances. The following example is another bean post processor that does just this.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class ProductCheckBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof Product) {
            String productName = ((Product) bean).getName();
            System.out.println("In ProductCheckBeanPostProcessor.postProcessBeforeInitialization,
processing Product: " + productName);
        }
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof Product) {
            String productName = ((Product) bean).getName();
            System.out.println("In ProductCheckBeanPostProcessor.postProcessAfterInitialization,
processing Product: " + productName);
        }
        return bean;
    }
}
```

Both the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods must return an instance of the bean being processed. However, this also means you can even replace the original bean instance with a brand-new instance in your bean post processor.

2-10. Create POJOs with a Factory (Static Method, Instance Method, Spring's FactoryBean)

Problem

You want to create a POJO instance in the Spring IoC container by invoking a static factory method or instance factory method. The purpose of this approach is to encapsulate the object-creation process in either a static method or in a method of another object instance, respectively. The client who requests an object can simply make a call to this method without knowing about the creation details.

You want to create a POJO instance in the Spring IoC container using Spring's factory bean. A factory bean is a bean that serves as a factory for creating other beans within the IoC container. Conceptually, a factory bean is very similar to a factory method, but it's a Spring-specific bean that can be identified by the Spring IoC container during bean construction.

Solution

To support the creation of a POJO by invoking a static factory method you use the `factory-method` attribute of the `<bean>` element. To support the creation of a POJO by invoking an instance factory method, the POJO instance is specified in the `factory-bean` attribute of the `<bean>` element.

As a convenience, Spring provides an abstract template class called `AbstractFactoryBean` to extend Spring's `FactoryBean` interface. However, Spring Factory beans are mostly used to implement framework facilities, so you'll seldom have to write custom factory beans. In addition, Spring already includes the most common factory implementations required by applications (e.g., `JndiObjectFactoryBean` to lookup an object from JNDI, `ProxyFactoryBean` to create an AOP proxy for a bean and `LocalSessionFactoryBean` to create a Hibernate session factory).

How It Works

Create POJOs by Invoking a Static Factory Method

For example, you can write the following `createProduct` static factory method to create a product from a predefined product ID. According to the product ID, this method decides which concrete product class to instantiate. If there is no product matching this ID, Spring throws an `IllegalArgumentException`.

```
package com.apress.springrecipes.shop;

public class ProductCreator {

    public static Product createProduct(String productId) {
        if ("aaa".equals(productId)) {
            return new Battery("AAA", 2.5);
        } else if ("cdrw".equals(productId)) {
            return new Disc("CD-RW", 1.5);
        } else if ("dvdrw".equals(productId)) {
            return new Disc("DVD-RW", 3.0);
        }
        throw new IllegalArgumentException("Unknown product");
    }
}
```

To declare a bean created by a static factory method, you specify the class hosting the factory method in the `class` attribute and the factory method's name in the `factory-method` attribute. Finally, you pass the method arguments by using the `<constructor-arg>` elements.

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.ProductCreator"
          factory-method="createProduct">
        <constructor-arg value="aaa" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.ProductCreator"
          factory-method="createProduct">
        <constructor-arg value="cdrw" />
    </bean>
```

```
<bean id="dvdrw" class="com.apress.springrecipes.shop.ProductCreator"
    factory-method="createProduct">
    <constructor-arg value="dvdrw" />
</bean>
</beans>
```

In case of any exception thrown by the factory method, Spring wraps it with a `BeanCreationException`. The equivalent code snippet for the preceding bean configuration is :

```
Product aaa = ProductCreator.createProduct("aaa");
Product cdrw = ProductCreator.createProduct("cdrw");
Product dvdrw = ProductCreator.createProduct("dvdrw");
```

Create POJOs by Invoking an Instance Factory Method

For example, you can write the following `ProductCreator` class by using a configurable map to store predefined products. The `createProduct()` instance factory method finds a product by looking up the supplied `productId` in the map. If there is no product matching this ID, it will throw an `IllegalArgumentException`.

```
package com.apress.springrecipes.shop;
...
public class ProductCreator {

    private Map<String, Product> products;

    public void setProducts(Map<String, Product> products) {
        this.products = products;
    }

    public Product createProduct(String productId) {
        Product product = products.get(productId);
        if (product != null) {
            return product;
        }
        throw new IllegalArgumentException("Unknown product");
    }
}
```

To create products from this `ProductCreator`, you first declare an instance of it in the IoC container and configure its product map. You can declare the products in the map as inner beans. To declare a bean created by an instance factory method, you specify the bean hosting the factory method in the `factory-bean` attribute, and the factory method's name in the `factory-method` attribute. Finally, you pass the method arguments by using the `<constructor-arg>` elements.

```
<beans ...>
    <bean id="productCreator"
        class="com.apress.springrecipes.shop.ProductCreator">
        <property name="products">
            <map>
                <entry key="aaa">
                    <bean class="com.apress.springrecipes.shop.Battery">
                        <property name="name" value="AAA" />

```

```

        <property name="price" value="2.5" />
    </bean>
</entry>
<entry key="cdrw">
    <bean class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
    </bean>
</entry>
<entry key="dvdrw">
    <bean class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="DVD-RW" />
        <property name="price" value="3.0" />
    </bean>
</entry>

</map>
</property>
</bean>

<bean id="aaa" factory-bean="productCreator"
      factory-method="createProduct">
    <constructor-arg value="aaa" />
</bean>

<bean id="cdrw" factory-bean="productCreator"
      factory-method="createProduct">
    <constructor-arg value="cdrw" />
</bean>
<bean id="dvdrw" factory-bean="productCreator"
      factory-method="createProduct">
    <constructor-arg value="dvdrw" />
</bean>

</beans>
```

If any exception is thrown by the factory method, Spring wraps it with a `BeanCreateException`. The equivalent code snippet for the preceding bean configuration is

```

ProductCreator productCreator = new ProductCreator();
productCreator.setProducts(...);

Product aaa = productCreator.createProduct("aaa");
Product cdrw = productCreator.createProduct("cdrw");
Product dvdrw = productCreator.createProduct("dvdrw");
```

Create POJOs using Spring's Factory bean

Although you'll seldom have to write custom factory beans, you may find it helpful to understand their internal mechanisms through an example. For example, you can write a factory bean for creating a product with a discount applied to the price. It accepts a product property and a discount property to apply the discount to the product and return it as a new bean.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.factory.config.AbstractFactoryBean;

public class DiscountFactoryBean extends AbstractFactoryBean {

    private Product product;
    private double discount;

    public void setProduct(Product product) {
        this.product = product;
    }

    public void setDiscount(double discount) {
        this.discount = discount;
    }

    public Class getObjectType() {
        return product.getClass();
    }

    protected Object createInstance() throws Exception {
        product.setPrice(product.getPrice() * (1 - discount));
        return product;
    }
}
```

By extending the `AbstractFactoryBean` class, the factory bean can simply override the `createInstance()` method to create the target bean instance. In addition, you have to return the target bean's type in the `getObjectType()` method for the auto-wiring feature to work properly.

Next, you can declare product instances with `DiscountFactoryBean`. Each time you request a bean that implements the `FactoryBean` interface, the Spring IoC container uses this factory bean to create the target bean. If you are sure that you want to get the factory bean instance itself, you can use the bean name preceded by &.

```
<beans ...>
    <bean id="aaa"
          class="com.apress.springrecipes.shop.DiscountFactoryBean">
        <property name="product">
            <bean class="com.apress.springrecipes.shop.Battery">
                <constructor-arg value="AAA" />
                <constructor-arg value="2.5" />
            </bean>
        </property>
        <property name="discount" value="0.2" />
    </bean>
```

```

<bean id="cdrw"
      class="com.apress.springrecipes.shop.DiscountFactoryBean">
    <property name="product">
      <bean class="com.apress.springrecipes.shop.Disc">
        <constructor-arg value="CD-RW" />
        <constructor-arg value="1.5" />
      </bean>
    </property>
    <property name="discount" value="0.1" />
  </bean>
</beans>

```

The preceding factory bean configuration works in a similar way to the following code snippet:

```

DiscountFactoryBean aaa = new DiscountFactoryBean();
aaa.setProduct(new Battery("AAA", 2.5));
aaa.setDiscount(0.2);
Product aaa = (Product) aaa.createInstance();

DiscountFactoryBean cdrw = new DiscountFactoryBean();
cdrw.setProduct(new Disc("CD-RW", 1.5));
cdrw.setDiscount(0.1);
Product cdrw = (Product) cdrw.createInstance();
DiscountFactoryBean dvdrw = new DiscountFactoryBean();
dvdrw.setProduct(new Disc("DVD-RW", 1.5));
dvdrw.setDiscount(0.1);
Product dvdrw = (Product) dvdrw.createInstance();

```

2-11. Use Spring Environments and Profiles to Load Different Sets of POJOs

Problem

You want to use the same set of POJO instances or beans but with different instantiation values for different application scenarios (e.g., ‘production,’ ‘development’ & ‘testing’). In addition, you want to use a single XML configuration file to define all beans.

Solution

Create a single XML configuration file and group POJOs instances or beans into multiple `<beans>` sections. Assign a profile name to each `<beans>` section based on the purpose of the group. Get the environment for an application’s context and set the profile to load a specific group of POJOs.

How It Works

POJO instantiation values can vary depending on different application scenarios. For example, a common scenario can occur when an application goes from development, to testing and on to production. In each of these scenarios, the properties for certain beans can vary slightly to accommodate environment changes (e.g., database username/password, file paths, etc.).

Prior to Spring Environments and Profiles, a common approach was to create multiple XML files with separate beans for each scenario. For example, `beans-dev.xml`, `beans-test.xml` and `beans-prod.xml` and in the application context only load a given configuration file based on the scenario. Even though this last approach works, it's error prone and can become unmanageable if you start to declare dozens of beans that can change often.

Assign POJOs to Profile

Let's create multiple profiles for the shopping application presented in previous recipes. To illustrate the flexibility of Spring environments and profiles, we'll classify the shopping application's products into five profiles: spring, summer, autumn, winter, and global. Profiles are set on groups of beans and can't be set on individual beans. You use the `profile` attribute inside the `<beans>` tag to define profiles.

```
<beans profile='spring'>
  <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
    <property name="name" value="AAA" />
    <property name="price" value="2.5" />
  </bean>

  <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="CD-RW" />
    <property name="price" value="1.5" />
  </beans>

<beans profile='summer,winter'>
  <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
    <property name="name" value="AAA" />
    <property name="price" value="2.0" />
  </bean>

  <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="CD-RW" />
    <property name="price" value="1.0" />
</beans>
```

Notice how the `<beans>` tags are nested to delimit the different beans that belong to each profile. In addition, a group of beans can also belong to several profiles, where each profile name is defined as a csv list value for the `profile` attribute.

Load Profile into Environment

To load the beans from a certain profile into an application, you need to activate a profile. You can load multiple profiles at a time and it's also possible to load profiles programmatically, through a Java runtime flag or even as an initialization parameter of a WAR file.

To load profiles programmatically (i.e., via the application context) you get the context environment from where you can load profiles via the `setActiveProfiles()` method.

```
GenericXmlApplicationContext ApplicationContext context = new GenericXmlApplicationContext();
context.getEnvironment(). setActiveProfiles("global","summer");
context.load("beans.xml");
context.refresh();
/**At this point beans declared in beans.xml with the global and summer profiles are accessible*/
```

It's also possible to indicate which Spring profile to load via a Java runtime flag. In this manner, you can pass the following runtime flag to load all beans that belong to the global and winter profiles: `-Dspring.profiles.active=global,winter`

Finally, a Spring profile can also be setup directly inside a WAR's `web.xml` configuration file. This scenario is relevant for Spring web applications.

```
<servlet>
<servlet-name>dispatcher</servlet-name>
<servlet-class>
  org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<init-param>
  <param-name> spring.profiles.active</param-name>
  <param-value>winter</param-value>
</init-param>
</servlet>
```

Set Default Profile

To avoid the possibility of errors because no profiles are loaded into an application, you can define default profiles. Default profiles are only used when Spring can't detect any active profiles—defined using any of the previous methods: programmatically, via a Java runtime flag or web application initialization parameter.

To setup default profiles, you can also use any of the three methods to setup active profiles. Programmatically you use the method `setDefaultProfiles()` instead of `setActiveProfiles()`, and via a Java runtime flag or web application initialization parameter you can use the `spring.profiles.default` parameter instead of `spring.profiles.active`.

2-12. Aspect Orientated Programming

Problem

You want to use aspect orientated programming with Spring.

Solution

Define POJOs to be aspects and define them inside the `<aop:aspect>` XML element, surrounded by the `<aop:config>` element. Next, define pointcuts to apply the aspect to certain objects and types with the `<aop:pointcut>` element. Finally, to indicate at which point to apply aspects, define any of the five advices supported by Spring AOP: `<aop:before>`, `<aop:after>`, `<aop:after-returning>`, `<aop:after-throwing>`, or `<aop:around>`.

How It Works

To illustrate how to apply aspect orientated programming in Spring, we'll use the following calculator interfaces to define a set of sample POJOs:

```
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
```

```

    public double mul(double a, double b);
    public double div(double a, double b);
}

package com.apress.springrecipes.calculator;

public interface UnitCalculator {

    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}

```

Next, let's create POJO classes for each interface with `println` statements to know when each method is executed:

```

package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }

    public double mul(double a, double b) {
        double result = a * b;
        System.out.println(a + " * " + b + " = " + result);
        return result;
    }

    public double div(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero");
        }
        double result = a / b;
        System.out.println(a + " / " + b + " = " + result);
        return result;
    }
}

package com.apress.springrecipes.calculator;

public class UnitCalculatorImpl implements UnitCalculator {

    public double kilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogram = " + pound + " pound");
        return pound;
    }
}

```

```

public double kilometerToMile(double kilometer) {
    double mile = kilometer * 0.62;
    System.out.println(kilometer + " kilometer = " + mile + " mile");
    return mile;
}
}

```

Declaring Aspects

An aspect is a Java class that modularizes a set of concerns (e.g., logging or transaction management) that cuts across multiple types and objects. In AOP terminology, aspects are also complemented by advices, which in themselves have pointcuts. An advice is a simple Java method—contained in an aspect class—that contains logic to execute at different lifecycles of the aspect. Whereas a pointcut is an expression that looks for types and objects on which to apply the aspect's advices.

The following is a POJO class to be used as an aspect with several methods to act as advices.

```

package com.apress.springrecipes.calculator;

import java.util.Arrays;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;

public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    public void logBefore(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }

    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }

    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends with " + result);
    }

    public void logAfterThrowing(JoinPoint joinPoint,
        IllegalArgumentException e) {
        log.error("Illegal argument " + Arrays.toString(joinPoint.getArgs())
            + " in " + joinPoint.getSignature().getName() + "()");
    }
}

```

```

public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    log.info("The method " + joinPoint.getSignature().getName()
        + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    try {
        Object result = joinPoint.proceed();
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends with " + result);
        return result;
    } catch (IllegalArgumentException e) {
        log.error("Illegal argument "
            + Arrays.toString(joinPoint.getArgs()) + " in "
            + joinPoint.getSignature().getName() + "()");
        throw e;
    }
}
}

```

Note For the previous aspect to work (i.e., output its message) you need to setup logging. Specifically create a `log4j.properties` file with configuration properties like the following:

```

log4j.rootLogger=INFO, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

```

Once you have the POJO aspect class, you declare aspects in Spring simply by defining them as bean instances in the IoC container. The Spring AOP schema must be present in the `<beans>` root element, because all the Spring AOP configurations must be defined inside the `<aop:config>` element.

For each aspect, you create a standard bean for the aspect POJO, as well as an `<aop:aspect>` element to refer to a backing bean instance for the concrete aspect implementation. So, your aspect beans must have an identifier for the `<aop:aspect>` elements to refer to.

```

<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-3.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">

    <aop:config>
        <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
        </aop:aspect>
    </aop:config>

    <bean id="calculatorLoggingAspect"
        class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />

</beans>

```

Declaring Pointcuts

A pointcut may be defined either under the `<aop:aspect>` element or directly under the `<aop:config>` element. In the former case, the pointcut becomes visible to the declaring aspect only. In the latter case, it becomes a global pointcut definition visible to all the aspects.

Unlike AspectJ annotations, XML-based AOP configurations don't allow you to refer to other pointcuts by name within a pointcut expression. That means you must copy the referred pointcut expression and embed it directly.

```
<aop:config>
    <aop:pointcut id="loggingOperation" expression=
        "within(com.apress.springrecipes.calculator.ArithmetricCalculator+) ||
         within(com.apress.springrecipes.calculator.UnitCalculator+)" />

</aop:config>
```

This last snippet applies the `loggingOperation` aspect to classes that either use the `ArtihmeticCalculator` or `UnitCalculator`. The `||` operator stands for or. Note that when using AspectJ annotations, you can join two pointcut expressions with the operator `&&` (and). However, the character `&` stands for “entity reference” in XML, so the pointcut operator `&&` isn't valid in an XML document. You have to use the keyword `and` instead.

Declaring Advices

In the AOP schema, there are five different XML elements to declare advices: `<aop:before>`, `<aop:after>`, `<aop:after-returning>`, `<aop:after-throwing>`, or `<aop:around>`.

An advice element requires either a `pointcut-ref` attribute to refer to a pointcut or a `pointcut` attribute to embed a pointcut expression directly. The `method` attribute specifies the name of the advice method in the aspect class.

```
<aop:config>
    ...
    <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
        <aop:before pointcut-ref="loggingOperation"
            method="logBefore" />

        <aop:after pointcut-ref="loggingOperation"
            method="logAfter" />

        <aop:after-returning pointcut-ref="loggingOperation"
            returning="result" method="logAfterReturning" />

        <aop:after-throwing pointcut-ref="loggingOperation"
            throwing="e" method="logAfterThrowing" />

        <aop:around pointcut-ref="loggingOperation"
            method="logAround" />
    </aop:aspect>

</aop:config>
```

Define a Concrete Class to Test AOP

Once you've defined the Aspect class, as well as declared the aspects, pointcuts and advices in a XML configuration file, you can test the aspect with the following Main class:

```
package com.apress.springrecipes.calculator;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}
```

2-13. Declare POJOs from Static Fields or Object Properties

Problem

You want to declare a bean in the Spring IoC container from a static field, from an object property or a nested property (i.e., a property path).

Solution

To declare a bean from a static field, you can make use of either the built-in factory bean `FieldRetrievingFactoryBean` or the `<util:constant>` tag.

To declare a bean from an object property or a property path, you can make use of either the built-in factory bean `PropertyPathFactoryBean` or the `<util:property-path>` tag.

How It Works

Declare POJOs from Static Fields

First, let's define two product constants in the Product class.

```
package com.apress.springrecipes.shop;
public abstract class Product {

    public static final Product AAA = new Battery("AAA", 2.5);
    public static final Product CDRW = new Disc("CD-RW", 1.5);
    ...
}
```

To declare a bean from a static field, you can make use of the built-in factory bean `FieldRetrievingFactoryBean` and specify the fully qualified field name in the `staticField` property.

```
<beans ...>
    <bean id="aaa" class="org.springframework.beans.factory.config.
        FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.AAA</value>
        </property>
    </bean>

    <bean id="cdrw" class="org.springframework.beans.factory.config.
        FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.CDRW</value>
        </property>
    </bean>
</beans>
```

The preceding bean configuration is equivalent to the following code snippet:

```
Product aaa = com.apress.springrecipes.shop.Product.AAA;
Product cdrw = com.apress.springrecipes.shop.Product.CDRW;
```

Another alternative is to declare a bean from a static field by using the `<util:constant>` tag. Compared to using `FieldRetrievingFactoryBean`, it is a simpler way of declaring beans from static fields. But before this tag can work, you must add the `util` schema definition to your `<beans>` root element.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.2.xsd">

    <util:constant id="aaa"
        static-field="com.apress.springrecipes.shop.Product.AAA" />

    <util:constant id="cdrw"
        static-field="com.apress.springrecipes.shop.Product.CDRW" />
</beans>
```

Declare POJOs from Object Properties

As an example, let's create a `ProductRanking` class with a `bestSeller` property whose type is `Product`.

```
package com.apress.springrecipes.shop;
public class ProductRanking {

    private Product bestSeller;

    public Product getBestSeller() {
        return bestSeller;
    }

    public void setBestSeller(Product bestSeller) {
        this.bestSeller = bestSeller;
    }
}
```

In the following bean declaration, the `bestSeller` property is declared as an inner bean. By definition, you cannot retrieve an inner bean by its name. However, you can retrieve it as a property of the `productRanking` bean. The factory bean `PropertyPathFactoryBean` can be used to declare a bean from an object property or a property path.

```
<beans ...>
    <bean id="productRanking"
        class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <bean class="com.apress.springrecipes.shop.Disc">
                <property name="name" value="CD-RW" />
                <property name="price" value="1.5" />
            </bean>
        </property>
    </bean>

    <bean id="bestSeller"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
        <property name="targetObject" ref="productRanking" />
        <property name="propertyPath" value="bestSeller" />
    </bean>
</beans>
```

Note that the `propertyPath` property of `PropertyPathFactoryBean` can accept not only a single property name but also a property path with dots as the separators. The preceding bean configuration is equivalent to the following code snippet:

```
Product bestSeller = productRanking.getBestSeller();
```

Another alternative is to declare a bean from an object property or a property path by using the `<util:property-path>` tag. Compared to using `PropertyPathFactoryBean`, it is a simpler way of declaring beans from properties. But before this tag can work, you must add the util schema definition to your `<beans>` root element.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/util
        http://www.springframework.org/schema/util/spring-util-3.2.xsd">
    ...
    <util:property-path id="bestSeller" path="productRanking.bestSeller" />
</beans>
```

You can test this property path by retrieving it from the IoC container and printing it to the console.

```
package com.apress.springrecipes.shop;
...
public class Main {
    public static void main(String[] args) throws Exception {
        ...
        Product bestSeller = (Product) context.getBean("bestSeller");
        System.out.println(bestSeller);
    }
}
```

2-14. Making POJOs Aware of Spring's IoC Container Resources

Problem

Even though a well-designed component should not have direct dependencies on Spring's IoC container, sometimes it's necessary for beans to be aware of the container's resources.

Solution

Your beans can be aware of the Spring IoC container's resources by implementing certain "aware" interfaces, shown in Table 2-3. Spring injects the corresponding resources to beans that implement these interfaces via the setter methods defined in these interfaces.

Table 2-3. Common Aware Interfaces in Spring

Aware Interface	Target Resource Type
BeanNameAware	The bean name of its instances configured in the IoC container.
BeanFactoryAware	The current bean factory, through which you can invoke the container's services
ApplicationContextAware*	The current application context, through which you can invoke the container's services
MessageSourceAware	A message source, through which you can resolve text messages
ApplicationEventPublisherAware	An application event publisher, through which you can publish application events
ResourceLoaderAware	A resource loader, through which you can load external resources

* The `ApplicationContext` interface in fact extends the `MessageSource`, `ApplicationEventPublisher`, and `ResourceLoader` interfaces, so you only need to be aware of the application context to access all these services. However, the best practice is to choose an aware interface with minimum scope that can satisfy your requirement.

The setter methods in the aware interfaces are called by Spring after the bean properties have been set, but before the initialization callback methods are called, as illustrated in the following list:

1. Create the bean instance either by a constructor or by a factory method.
2. Set the values and bean references to the bean properties.
3. Call the setter methods defined in the aware interfaces.
4. Pass the bean instance to the `postProcessBeforeInitialization()` method of each bean post processor.
5. Call the initialization callback methods.
6. Pass the bean instance to the `postProcessAfterInitialization()` method of each bean post processor.
7. The bean is ready to be used.
8. When the container is shut down, call the destruction callback methods.

Keep in mind that once a bean implements an aware interface, they are bound to Spring and won't work properly outside the Spring IoC container. So consider carefully whether it's necessary to implement such proprietary interfaces.

How It Works

For example, you can make the shopping application's POJO instances of the `Cashier` class aware of their corresponding bean name by implementing the `BeanNameAware` interface. By implementing the interface, Spring automatically injects the bean name into the POJO instance. In addition to implementing the interface, you also need to add the necessary setter method to handle the bean name.

```

package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.BeanNameAware;

public class Cashier implements BeanNameAware {
    ...
    private String beanName;

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

```

When the bean name is injected, you can use the value to do a related POJO task that requires the bean name. For example, you could use the value to set the filename to record a cashier's checkout data. In this way, you can erase the configuration of the name property, as well as the POJO name field and setName() method.

```

<bean id="cashier" class="com.apress.springrecipes.shop.Cashier" init-method="openFile"
    destroy-method="closefile">
    <property name="path" value="c:/Windows/Temp/cashier" />
</bean>

```

2-15. Communicate Application Events Between POJOs

Problem

In a typical communication between POJOs, the sender has to locate the receiver to call a method on it. In this case, the sender POJO must be aware of the receiver component. This kind of communication is direct and simple, but the sender and receiver POJOs are tightly coupled.

When using an IoC container, POJOs can communicate by interface rather than by implementation. This communication model helps reduce coupling. However, it is only efficient when a sender component has to communicate with one receiver. When a sender needs to communicate with multiple receivers, it has to call the receivers one by one.

Solution

Spring's application context supports event-based communication between its beans. In the event-based communication model, the sender POJO just publishes an event without knowing who the receiver is. Since there can actually be more than one receiver. Also, the receiver doesn't necessarily know who is publishing the event. It can listen to multiple events from different senders at the same time. In this way, the sender and receiver components are loosely coupled.

In Spring, all event classes must extend the `ApplicationEvent` class. For such cases, any bean can publish an event calling an application event publisher's `publishEvent()` method. For a bean to listen to certain events, it must implement the `ApplicationListener` interface and handle the events in the `onApplicationEvent()` method. Spring notifies a listener of all events, so you must filter the events yourself. If you use exploit generics, however, Spring only delivers messages that match the generic type parameter.

How It Works

Define Events

The first step of enabling event-based communication is to define the event. Suppose you would like a cashier bean to publish a `CheckoutEvent` after the shopping cart is checked out. This event includes a checkout time property. In Spring, all events must extend the abstract class `ApplicationEvent` and pass the event source as a constructor argument.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;

public class CheckoutEvent extends ApplicationEvent {

    private Date time;

    public CheckoutEvent(Object source, Date time) {
        super(source);
        this.time = time;
    }

    public Date getTime() {
        return time;
    }
}
```

Publish Events

To publish an event, you just create an event instance and make a call to the `publishEvent()` method of an application event publisher, which becomes accessible by implementing the `ApplicationEventPublisherAware` interface.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class Cashier implements ApplicationEventPublisherAware {
    ...
    private ApplicationEventPublisher applicationEventPublisher;

    public void setApplicationEventPublisher(
        ApplicationEventPublisher applicationEventPublisher) {
        this.applicationEventPublisher = applicationEventPublisher;
    }

    public void checkout(ShoppingCart cart) throws IOException {
        ...
        CheckoutEvent event = new CheckoutEvent(this, new Date());
        applicationEventPublisher.publishEvent(event);
    }
}
```

Listen to Events

Any bean defined in the application context that implements the `ApplicationListener` interface is notified of all events. So in the `onApplicationEvent()` method, you can filter the events that a listener wants to handle—similar to how it's done with bean post processors (Recipe 2-9). In the following listener, we use an instance of `check` to filter on the nongeneric `ApplicationEvent` parameter.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class CheckoutListener implements ApplicationListener {
    public void onApplicationEvent(ApplicationEvent event) {
        if (event instanceof CheckoutEvent) {
            Date time = ((CheckoutEvent) event).getTime();

            // Do anything you like with the checkout amount and time
            System.out.println("Checkout event [" + time + "]");
        }
    }
}
```

Rewritten to take advantage of the generics functionality, it's a bit briefer:

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;
import org.springframework.context.ApplicationListener;

public class CheckoutListener implements ApplicationListener<CheckoutEvent> {
    public void onApplicationEvent(CheckoutEvent event) {
        Date time = ((CheckoutEvent) event).getTime();

        // Do anything you like with the checkout amount and time
        System.out.println("Checkout event [" + time + "]");
    }
}
```

Next, you have to register the listener in the application context to listen for all events. The registration is as simple as declaring a bean instance of this listener. The application context recognizes the beans that implement the `ApplicationListener` interface and notify them of each event.

```
<beans ...>
    ...
    <bean class="com.apress.springrecipes.shop.CheckoutListener" />
</beans>
```

Finally, remember the application context itself also publishes container events such as `ContextClosedEvent`, `ContextRefreshedEvent`, and `RequestHandledEvent`. If any beans want to be notified of these events, they can implement the `ApplicationListener` interface.

2-16. Use Property Editors in Spring

Problem

A property editor is a feature of the JavaBeans API for converting property values to and from text values. Each property editor is designed for a certain type of property only. You may wish to employ property editors to simplify bean configurations.

In addition to registering the built-in property editors, it's also possible to write custom property editors for converting custom data types.

Solution

The Spring IoC container supports property editors to help with bean configurations. For example, with a property editor for the `java.util.Date` type, Spring would automatically convert the data string into a `java.util.Date` object, instead of automatically creating a `String` type. Spring comes with several property editors for converting common type bean properties.

Before you can use a custom property editor in the Spring IoC container you need to register it. The `CustomEditorConfigurer` is implemented as a bean factory post processor for you to register custom property editors before any of the beans get instantiated.

You can write custom property editors by implementing the `java.beans.PropertyEditor` interface or extending the convenient support class `java.beans.PropertyEditorSupport`.

How It Works

As an example, suppose you define product ranking class to keep track of sales for a particular period.

```
package com.apress.springrecipes.shop;
...
public class ProductRanking {

    private Product bestSeller;
    private Date fromDate;
    private Date toDate;

    // Getters and Setters
    ...
}
```

To specify the value for a `java.util.Date` property in a Java program, you can convert it from a date string of particular pattern with the help of `DateFormat.parse()` method.

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
productRanking.setFromDate(dateFormat.parse("2007-09-01"));
productRanking.setToDate(dateFormat.parse("2007-09-30"));
```

To write the equivalent bean configuration without the help of property editors, you could declare a `dateFormat` bean with the pattern. As the `parse()` method is called for converting the date strings into date objects, you can consider it as an instance factory method to create the date beans.

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

<bean id="productRanking"
    class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
        <bean class="com.apress.springrecipes.shop.Disc">
            <property name="name" value="CD-RW" />
            <property name="price" value="1.5" />
        </bean>
    </property>
    <property name="fromDate">
        <bean factory-bean="dateFormat" factory-method="parse">
            <constructor-arg value="2013-09-01" />
        </bean>
    </property>
    <property name="toDate">
        <bean factory-bean="dateFormat" factory-method="parse">
            <constructor-arg value="2013-09-30" />
        </bean>
    </property>
    </bean>
</beans>
```

As you can see, the preceding configuration is too complicated for setting date properties. The Spring IoC container is able to convert the text values for properties using property editors. The `CustomDateEditor` class that comes with Spring is used to convert date strings into `java.util.Date` properties. First, you have to declare an instance of it in the bean configuration file.

```
<beans ...>
...
<bean id="dateEditor"
    class="org.springframework.beans.propertyeditors.CustomDateEditor">
    <constructor-arg>
        <bean class="java.text.SimpleDateFormat">
            <constructor-arg value="yyyy-MM-dd" />
        </bean>
    </constructor-arg>
    <constructor-arg value="true" />
</bean>
</beans>
```

This editor requires a `DateFormat` object as the first constructor argument. The second argument indicates whether the editor allows empty values.

Next, you have to register the property editor in a `CustomEditorConfigurer` instance so Spring can convert properties whose type is `java.util.Date`. Now, you can specify a date value in text format for any `java.util.Date` property:

```
<beans ...>
  ...
  <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
      <map>
        <entry key="java.util.Date">
          <ref local="dateEditor" />
        </entry>
      </map>
    </property>
  </bean>

  <bean id="productRanking"
    class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
      <bean class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
      </bean>
    </property>
    <property name="fromDate" value="2007-09-01" />
    <property name="toDate" value="2007-09-30" />
  </bean>
</beans>
```

You can test whether the `CustomDateEditor` configuration works with the following `Main` class:

```
package com.apress.springrecipes.shop;

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

public class Main {

  public static void main(String[] args) throws Exception {
    ApplicationContext context =
      new ClassPathXmlApplicationContext("beans.xml");
    ...
    ProductRanking productRanking =
      (ProductRanking) context.getBean("productRanking");
    System.out.println(
      "Product ranking from" + productRanking.getFromDate() +
      "to" + productRanking.getToDate());
  }
}
```

In addition to `CustomDateEditor`, Spring comes with several property editors for converting common data types, such as `CustomNumberEditor`, `ClassEditor`, `FileEditor`, `LocaleEditor`, `StringArrayPropertyEditor`, and `URLEditor`. Among them, `ClassEditor`, `FileEditor`, `LocaleEditor`, and `URLEditor` are preregistered by Spring, so you

don't need to register them again. For more information on using these editors, you can consult the Javadoc of these classes in the `org.springframework.beans.propertyeditors` package.

Next, let's write a property editor for the `Product` class. You can design the string representation of a product as three parts, which are the concrete class name, the product name, and the price. Each part is separated by a comma. Then, you can write the following `ProductEditor` class for converting them:

```
package com.apress.springrecipes.shop;

import java.beans.PropertyEditorSupport;

public class ProductEditor extends PropertyEditorSupport {

    public String getAsText() {
        Product product = (Product) getValue();
        return product.getClass().getName() + "," + product.getName() + ","
            + product.getPrice();
    }

    public void setAsText(String text) throws IllegalArgumentException {
        String[] parts = text.split(",");
        try {
            Product product = (Product) Class.forName(parts[0]).newInstance();
            product.setName(parts[1]);
            product.setPrice(Double.parseDouble(parts[2]));
            setValue(product);
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

The `getAsText()` method converts a property into a string value, while the `setAsText()` method converts a string back into a property. The property value is retrieved and set by calling the `getValue()` and `setValue()` methods.

Next, you have to register the custom editor in a `CustomEditorConfigurer` instance before it can be used. Registration is the same as for the built-in editors. Now, you can specify a product in text format for any property whose type is `Product`.

```
<beans ...>
    ...
    <bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                ...
                <entry key="com.apress.springrecipes.shop.Product">
                    <bean class="com.apress.springrecipes.shop.ProductEditor" />
                </entry>
            </map>
        </property>
    </bean>
```

```
<bean id="productRanking"
    class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
        <value>com.apress.springrecipes.shop.Disc,CD-RW,1.5</value>
    </property>
    ...
</bean>
</beans>
```

Tip The JavaBeans API automatically searches for a property editor in a given package. For a property editor to be searched automatically, it must be located in the same package as the target class, and the name must be the target class name with Editor as its suffix. If the property editor is provided with this convention, such as in the preceding ProductEditor, there's no need to register it again in the Spring IoC container.

2-17. Inherit POJO Configurations

Problem

When configuring beans in the Spring IoC container, you can have more than one bean sharing common configurations (e.g., bean properties and attributes in the `<bean>` element). You often have to repeat these configurations for multiple beans.

Solution

Spring allows you to extract common bean configurations to form a parent bean. The beans that inherit from this parent bean are called child beans. The child beans inherit the bean configurations, including bean properties and attributes in the `<bean>` element, from the parent bean to avoid duplicate configurations. The child beans can also override the inherited configurations when necessary.

The parent bean can act as a configuration template and also as a bean instance at the same time. However, if you want the parent bean to act only as a template that cannot be retrieved, you must set the `abstract` attribute to `true`, asking Spring not to instantiate this bean.

Be advised that not all attributes defined in the parent `<bean>` element are inherited. For example, the `autowire` attribute is not inherited from the parent. To find out more about which attributes are inherited from the parent and which aren't, please refer to the Spring documentation about bean inheritance.

How It Works

Suppose you need to add a new sequence generator instance whose initial value and suffix are the same as the existing ones.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>
```

```

<bean id="sequenceGenerator1"
    class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>

<bean id="datePrefixGenerator"
    class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
</beans>

```

Reuse Common Properties Across POJOs

To avoid duplicating the same properties, you can declare a base sequence generator bean with a base set of properties. Then the two sequence generators can inherit this base generator so that they also have the base set of properties set automatically. You needn't specify the class attributes of the child beans if they are the same as the parent's.

```

<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="prefixGenerator" />
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator" />

    <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
    ...
</beans>

```

The inherited properties can be overridden by the child beans. For example, you can add a child sequence generator with a different initial value.

```

<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="sequenceGenerator2" parent="baseSequenceGenerator">
        <property name="initial" value="200000" />
    </bean>
    ...
</beans>

```

The base sequence generator bean can now be retrieved as a bean instance. If you want it to act as a template only, you have to set the abstract attribute to true. Then Spring will not instantiate the bean.

```
<bean id="baseSequenceGenerator" abstract="true"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

You can also omit the class of the parent bean and let the child beans specify their own, especially when the parent bean and child beans are not in the same class hierarchy, but share some properties of the same name. In this case, the parent bean's abstract attribute must be set to true, as the parent bean can't be instantiated.

Inherit Properties from Parent POJOs with Different Classes

For example, let's add another ReverseGenerator class that also has an initial property.

```
package com.apress.springrecipes.sequence;

public class ReverseGenerator {

    private int initial;

    public void setInitial(int initial) {
        this.initial = initial;
    }
}
```

Now SequenceGenerator and ReverseGenerator don't extend the same base class—that is, they're not in the same class hierarchy, but they have a property of the same name: initial. To extract this common initial property, you need a baseGenerator parent bean with no class attribute.

```
<beans ...>
  <bean id="baseGenerator" abstract="true">
    <property name="initial" value="100000" />
  </bean>

  <bean id="baseSequenceGenerator" abstract="true" parent="baseGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="suffix" value="A" />
    <property name="prefixGenerator" ref="prefixGenerator" />
  </bean>

  <bean id="reverseGenerator" parent="baseGenerator"
        class="com.apress.springrecipes.sequence.ReverseGenerator" />

  <bean id="sequenceGenerator" parent="baseSequenceGenerator" />
  <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
  <bean id="sequenceGenerator2" parent="baseSequenceGenerator"/>
  ...
</beans>
```

Merge a Collection of a Parent POJO

If you define your beans with inheritance, a child bean's collection can be merged with that of its parent by setting the `merge` attribute to true. For a `<list>` collection, the child elements will be appended after the parent's to preserve the order. So, the following sequence generator will have four suffixes: A, B, A, and C.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefix" value="30" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <list>
                <value>A</value>
                <value>B</value>
            </list>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <list merge="true">
                <value>A</value>
                <value>C</value>
            </list>
        </property>
    </bean>
    ...
</beans>
```

For a `<set>` or `<map>` collection, the child elements will overwrite the parent's collection if they have the same value. So, the following sequence generator will have three suffixes: A, B, and C.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefix" value="30" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <set>
                <value>A</value>
                <value>B</value>
            </set>
        </property>
    </bean>
```

```

<bean id="sequenceGenerator" parent="baseSequenceGenerator">
    <property name="suffixes">
        <set merge="true">
            <value>A</value>
            <value>C</value>
        </set>
    </property>
</bean>
...
</beans>

```

2-18. Implement POJOs with Scripting Languages

Problem

Sometimes, applications have certain modules that require frequent and dynamic changes. If you implement such modules in Java, you have to recompile, repackage, and redeploy an application each time after the change. You may not be allowed to perform these actions at any time you want or need, especially for a 24/7 application.

Solution

You can consider implementing modules that require frequent and dynamic changes with scripting languages. The advantage of scripting languages is that they don't need to be recompiled after changes, so you simply deploy the new script for the change to take effect.

Spring allows you to implement beans with one of its supported scripting languages. You can configure a scripted bean in the IoC container just like a normal bean implemented in Java.

How It Works

Suppose you are going to develop an application that requires interest calculation. First of all, you define the following `InterestCalculator` interface:

```

package com.apress.springrecipes.interest;

public interface InterestCalculator {
    public void setRate(double rate);
    public double calculate(double amount, double year);
}

```

Implementing this interface is not difficult at all. However, as there are many interest calculation strategies, users may need to change the implementation very frequently and dynamically. You don't want to recompile, repackage, and redeploy an application every time this happens. So, you consider implementing this interface with one of the supported scripting languages in Spring.

In Spring's bean configuration file, you have to include the `lang` schema definition in the `<beans>` root element to make use of the scripting language support.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/spring-lang-3.2.xsd">
    ...
</beans>

```

Spring supports three scripting languages: JRuby, Groovy, and BeanShell. Next, you will implement the `InterestCalculator` interface with these three languages. For simplicity's sake, let's consider the following simple interest formula for interest calculation:

`Interest = Amount x Rate x Year`

Scripting Beans with JRuby

First, let's implement the `InterestCalculator` interface with JRuby by creating the JRuby script, `SimpleInterestCalculator.rb`, in the `com.apress.springrecipes.interest` package of your classpath.

```

class SimpleInterestCalculator

    def setRate(rate)
        @rate = rate
    end

    def calculate(amount, year)
        amount * year * @rate
    end
end

SimpleInterestCalculator.new

```

The preceding JRuby script declares a `SimpleInterestCalculator` class with a setter method for the `rate` property and a `calculate()` method. In Ruby, an instance variable begins with the `@` sign. Note that, in the last line, you return a new instance of the target JRuby class. Failure to return this instance can result in Spring performing a lookup for an appropriate Ruby class to instantiate. As there can be multiple classes defined in a single JRuby script file, Spring will throw an exception if it cannot find an appropriate one that implements the methods declared in the interface.

Note To use JRuby in a Spring application, you have to include the appropriate dependencies on your classpath (i.e., the `org.jruby` JAR).

In the bean configuration file, you declare a bean implemented with JRuby using the `<lang:jruby>` element and specifying the script's location in the `script-source` attribute. You can specify any resource path with a resource prefix supported by Spring, such as `file` or `classpath`.

You also have to specify one or more interfaces in the `script-interfaces` attribute for a JRuby bean. It's up to Spring to create a dynamic proxy for this bean and convert the Java method calls into JRuby method calls.

```
<lang:jruby id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
        SimpleInterestCalculator.rb"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:property name="rate" value="0.05" />
</lang:jruby>
```

Finally, you can specify the property values for a scripting bean in the `<lang:property>` elements. Now, you can get the `interestCalculator` bean from the IoC container to use and inject it into other bean properties as well. The following Main class helps you verify whether the scripted bean works properly:

```
package com.apress.springrecipes.interest;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new GenericXmlApplicationContext("beans.xml");

        InterestCalculator calculator =
            (InterestCalculator) context.getBean("interestCalculator");
        System.out.println(calculator.calculate(100000, 1));
    }
}
```

Scripting Beans with Groovy

Next, let's implement the `InterestCalculator` interface with Groovy by creating the Groovy script, `SimpleInterestCalculator.groovy`, in the `com.apress.springrecipes.interest` package of your classpath.

```
import com.apress.springrecipes.interest.InterestCalculator;

class SimpleInterestCalculator implements InterestCalculator {

    double rate

    double calculate(double amount, double year) {
        return amount * year * rate
    }
}
```

The preceding Groovy script declares a `SimpleInterestCalculator` class that implements the `InterestCalculator` interface. In Groovy, you can simply declare a property with no access modifier, and then it generates a private field with a public getter and setter automatically.

Note To use Groovy in a Spring application, you have to include the appropriate dependencies on your classpath (i.e., the org.codehaus.groovy JAR).

In the bean configuration file, you can declare a bean implemented with Groovy by using the `<lang:groovy>` element and specifying the script's location in the `script-source` attribute. You can specify the property values for a scripting bean in the `<lang:property>` elements.

```
<lang:groovy id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.groovy">
    <lang:property name="rate" value="0.05" />
</lang:groovy>
```

Notice that it's unnecessary to specify the `script-interfaces` attribute for a Groovy bean, as the Groovy class has declared which interfaces it implements.

Scripting Beans with BeanShell

Finally, let's implement the `InterestCalculator` interface with BeanShell by creating the BeanShell script, `SimpleInterestCalculator.bsh`, in the `com.apress.springrecipes.interest` package of your classpath.

```
double rate;

void setRate(double aRate) {
    rate = aRate;
}

double calculate(double amount, double year) {
    return amount * year * rate;
}
```

In BeanShell, you cannot declare classes explicitly, but you can declare variables and methods. So, you implement your `InterestCalculator` interface by providing all the methods required by the interface.

Note To use BeanShell in a Spring application, you have to include the appropriate dependencies on your classpath (i.e., the org.beanshell JAR).

In the bean configuration file, you can declare a bean implemented with BeanShell by using the `<lang:bsh>` element and specifying the script's location in the `script-source` attribute. You can specify the property values for a scripting bean in the `<lang:property>` elements.

You also have to specify one or more interfaces in the `script-interfaces` attribute for a bean implemented with BeanShell. It's up to Spring to create a dynamic proxy for this bean and convert the Java method calls into BeanShell calls.

```
<lang:bsh id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.bsh"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:property name="rate" value="0.05" />
</lang:bsh>
```

2-19. Inject Spring POJOs into Scripts

Problem

Sometimes, scripts may need the help of certain Java objects to complete their tasks. In Spring, you can allow scripts to access beans declared in the IoC container.

Solution

You can inject beans declared in the Spring IoC container into scripts in the same way as properties of simple data types.

How It Works

Suppose you want the application from the previous recipe to make interest calculations dynamically. First, you define the following interface to allow implementations to return the annual, monthly, and daily interest rates:

```
package com.apress.springrecipes.interest;

public interface RateCalculator {

    public double getAnnualRate();
    public double getMonthlyRate();
    public double getDailyRate();
}
```

Next, you can implement this interface to calculate rates from a fixed annual interest rate, which is then injected through a setter method.

```
package com.apress.springrecipes.interest;

public class FixedRateCalculator implements RateCalculator {

    private double rate;

    public void setRate(double rate) {
        this.rate = rate;
    }

    public double getAnnualRate() {
        return rate;
    }

    public double getMonthlyRate() {
        return rate / 12;
    }

    public double getDailyRate() {
        return rate / 365;
    }
}
```

Then you declare this rate calculator in the IoC container by supplying an annual interest rate.

```
<bean id="rateCalculator"
    class="com.apress.springrecipes.interest.FixedRateCalculator">
    <property name="rate" value="0.05" />
</bean>
```

Last, the interest calculator should use a RateCalculator object rather than a fixed rate value.

```
package com.apress.springrecipes.interest;

public interface InterestCalculator {

    public void setRateCalculator(RateCalculator rateCalculator);
    public double calculate(double amount, double year);
}
```

Inject Spring Beans into JRuby

In a JRuby script, you can store the injected RateCalculator object in an instance variable and use it for rate calculation.

```
class SimpleInterestCalculator

    def setRateCalculator(rateCalculator)
        @rateCalculator = rateCalculator
    end

    def calculate(amount, year)
        amount * year * @rateCalculator.getAnnualRate
    end
end

SimpleInterestCalculator.new
```

In the bean declaration, you can inject another bean into a scripted bean's property by specifying the bean name in the `ref` attribute.

```
<lang:jruby id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
        SimpleInterestCalculator.rb"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:jruby>
```

Inject Spring Beans into Groovy

In a Groovy script, you just declare a property of type RateCalculator, and it automatically generates a public getter and setter method.

```
import com.apress.springrecipes.interest.InterestCalculator;
import com.apress.springrecipes.interest.RateCalculator;

class SimpleInterestCalculator implements InterestCalculator {

    RateCalculator rateCalculator

    double calculate(double amount, double year) {
        return amount * year * rateCalculator.getAnnualRate()
    }
}
```

And you can inject another bean into a scripted bean's property by specifying the bean name in the `ref` attribute.

```
<lang:groovy id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.groovy">
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:groovy>
```

Inject Spring Beans into BeanShell

In a BeanShell script, you need a global variable of type RateCalculator and a setter method for it.

```
import com.apress.springrecipes.interest.RateCalculator;
RateCalculator rateCalculator;

void setRateCalculator(RateCalculator aRateCalculator) {
    rateCalculator = aRateCalculator;
}

double calculate(double amount, double year) {
    return amount * year * rateCalculator.getAnnualRate();
}
```

You can also inject another bean into a scripted bean's property by specifying the bean name in the `ref` attribute.

```
<lang:bsh id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/SimpleInterestCalculator.bsh"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:bsh>
```

2-20. Refresh POJOs from Scripts

Problem

As the modules implemented with scripting languages may change frequently and dynamically, you want the Spring IoC container to be able to detect and refresh changes automatically from the script sources.

Solution

Spring is able to refresh a scripted bean definition from its source once you have specified the checking interval in the `refresh-check-delay` attribute. When a method is called on that bean, Spring checks the script source to verify if the specified checking interval has elapsed. Then, Spring refreshes the bean definition from the script source if it has changed.

How It Works

By default, the `refresh-check-delay` attribute is negative, so the refresh checking feature is disabled. You can assign the milliseconds for a refresh check in this attribute to enable this feature. For example, you can specify 5 seconds for the refresh check interval for a JRuby bean.

```
<lang:jruby id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
        SimpleInterestCalculator.rb"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator"
    refresh-check-delay="5000">
    ...
</lang:jruby>
```

Of course, the `refresh-check-delay` attribute also works for beans implemented in Groovy or BeanShell.

```
<lang:groovy id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
        SimpleInterestCalculator.groovy"
    refresh-check-delay="5000">
    ...
</lang:groovy>

<lang:bsh id="interestCalculator"
    script-source="classpath:com/apress/springrecipes/interest/
        SimpleInterestCalculator.bsh"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator"
    refresh-check-delay="5000">
    ...
</lang:bsh>
```

2-21. Define Script Sources as Inline Code and not in External Files

Problem

You want to define script sources, which are not likely to change often directly in the bean configuration file, rather than in external script source files.

Solution

You can define inline script sources in the `<lang:inline-script>` element of a scripted bean to replace references to external script source files that use the `script-source` attribute. Note that the refresh check feature is not applicable for inline script sources, because the Spring IoC container only loads the bean configuration once, at startup.

How It Works

For example, you can define a JRuby script inline using the `<lang:inline-script>` element. To prevent the characters in the script from conflicting with reserved XML characters, you should surround the script source with the `<![CDATA[...]]>` tag. In this scenario, you no longer need to specify the reference to the external script source file in the `script-source` attribute.

```
<lang:jruby id="interestCalculator"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:inline-script>
        <![CDATA[
            class SimpleInterestCalculator

                def setRateCalculator(rateCalculator)
                    @rateCalculator = rateCalculator
                end

                def calculate(amount, year)
                    amount * year * @rateCalculator.getAnnualRate
                end
            end

            SimpleInterestCalculator.new
        ]]>
    </lang:inline-script>
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:jruby>
```

Of course, you can also define the Groovy or BeanShell script sources inline with the `<lang:inline-script>` element.

```
<lang:groovy id="interestCalculator">
    <lang:inline-script>
        <![CDATA[
            import com.apress.springrecipes.interest.InterestCalculator;
            import com.apress.springrecipes.interest.RateCalculator;
```

```

class SimpleInterestCalculator implements InterestCalculator {

    RateCalculator rateCalculator

    double calculate(double amount, double year) {
        return amount * year * rateCalculator.getAnnualRate()
    }
}

]]>
</lang:inline-script>
<lang:property name="rateCalculator" ref="rateCalculator" />
</lang:groovy>

<lang:bsh id="interestCalculator"
    script-interfaces="com.apress.springrecipes.interest.InterestCalculator">
    <lang:inline-script>
        <![CDATA[
            import com.apress.springrecipes.interest.RateCalculator;

            RateCalculator rateCalculator;

            void setRateCalculator(RateCalculator aRateCalculator) {
                rateCalculator = aRateCalculator;
            }

            double calculate(double amount, double year) {
                return amount * year * rateCalculator.getAnnualRate();
            }
        ]]>
    </lang:inline-script>
    <lang:property name="rateCalculator" ref="rateCalculator" />
</lang:bsh>

```

2-22. Use the Spring Expression Language to Configure POJOs

Problem

You want to dynamically evaluate a condition or property and use it as a configuration value in the IoC container. Or perhaps want to defer evaluation of something—not at design time but at runtime, as might be the case in a custom scope. Or you just want a way to use an expression language for your applications.

Solution

The Spring Expression Language (SpEL) provides functionality similar to the Unified EL from JSF and JSP, or Object Graph Navigation Language (OGNL). SpEL provides easy-to-use infrastructure that can be leveraged outside of Spring. Within Spring, it can be used to make configuration much easier in a lot of cases.

How It Works

The SpEL provides the means to evaluate certain expressions at arbitrary points in POJO's life cycle, such as during a scoped beans initialization. It can be used in constructs for both XML configurations, as well as annotations which are described in the chapter.

Features of the SpEL Syntax

The expression language supports a long list of features. Table 2-4 briefly runs through the various constructs and demonstrates their usage.

Table 2-4. Expression Language Features

Type	Use	Example
Literal expression	The simplest thing you can do in the expression language, essentially the same as if you were writing Java code. The language supports String literals as well as all sorts of numbers.	2342 'Hello Spring Enterprise Recipes'
Boolean and relational operator	The expression language provides the ability to evaluate conditionals using standard idioms from Java.	T(java.lang.Math).random()>.5
Standard expression	You can iterate and return the properties on beans in the same way you might with Unified EL, separating each dereferenced property with a period and using JavaBean-style naming conventions. In the example to the right, the expression would be equivalent to getCat().getMate().getName().	cat.mate.name
Class expression	T() tells the expression language to act on the type of the class, not an instance. In the examples on the right, the first would yield the Class instance for java.lang.Math—equivalent to calling java.lang.Math.class. The second example calls a static method on a given type. Thus, T(java.lang.Math).random() is equivalent to calling java.lang.Math.random().	T(java.lang.Math) T(java.lang.Math).random()
Accessing arrays, lists, maps	You can index lists, arrays, and maps using brackets and the key—which for arrays or lists is the index number, and for maps is an object. In the examples, you see a java.util.List with four chars being indexed at index 1, which returns 'b'. The second example demonstrates accessing a map by the index 'OR', yielding the value associated with that key.	T(java.util.Arrays.asList('a','b','c','d')[1]) T(SpelExamplesDemo).MapOfStatesAndCapitals['OR']
Method invocation	Methods may be invoked in instances just as you would in Java. This is a marked improvement over the basic JSF or JSP expression languages.	'Hello, World'.toLowerCase()
Relational operators	You can compare or equate values, and the returned value will be a Boolean.	23 == person.age 'fala' < 'fido'

(continued)

Table 2-4. (continued)

Type	Use	Example
Calling constructor	You can create objects and invoke their constructors. Here, you create simple String and Cat objects.	new String('Hello Spring Recipes, again!') new Cat('Felix')
Ternary operator	Ternary expressions work as you'd expect, yielding the value in the true case.	T(java.lang.Math).random() > .5 ? 'She loves me' : 'She loves me not'
Variable	The SpEL lets you set and evaluate variables. The variables can be installed by the context of the expression parser, and there are some implicit variables, such as #this, which always refer to the root object of the context.	#this.firstName #customer.email
Collection projection	A very powerful feature inside of SpEL is the capability to perform very sophisticated manipulations of maps and collections. Here, you create a projection for the list cats. In this example, the returned value is a collection of as many elements being iterated that has the value for the name property on each cat in the collection. In this case, cats is a collection of Cat objects. The returned value is a collection of String objects.	cats.!name]
Collection selection	Selection lets you dynamically filter objects from a collection or map by evaluating a predicate on each item in the collection and keeping only those elements for which the predicate is true. In this case, you evaluate the java.util.Map.Entry.value property for each Entry in the Map and if the value (in this case a String), lowercased, starts with "s", then it is kept. Everything else is discarded.	mapOfStatesAndCapitals. [value.toLowerCase(). startsWith('s')]
Templated expression	You can use the expression language to evaluate expressions inside of string expressions. The result is returned. In this case, the result is dynamically created by evaluating the ternary expression and including 'good' or 'bad' based on the result.	Your fortune is \${T(java.lang. Math).random()} > .5 ? 'good' : 'bad'}

Use SpEL in your XML Configurations

The expression language can be used in XML configuration files. The expressions are evaluated at creation time for the bean, not at the initialization of the context. This has the effect that beans created in a custom scope are not configured until the bean is in the appropriate scope.

The first example is the injection of a named expression language variable, systemProperties, which is just a special variable for the `java.util.Properties` instance that's available from `System.getProperties()`. The next example shows a POJO definition, followed by the bean definition which injects the system properties:

```
package com.apress.springrecipes.spel;
...
public class CommonData {
    private Properties commonProperties;
    private String userOS;
```

```

private String userRegion;
private double randomNumber;
private String email;

//Setter & getter methods ommited

}

<bean id="commonData"
      class="com.apress.springrecipes.spel.CommonData"
      p:commonProperties="#{@systemProperties}"
      p:userOS="#{@systemProperties['os.name']}"
      p:userRegion="#{@systemProperties['user.region']?:'unknown region'}"/>

```

The first property injects the full `Properties` object using the syntax `#{@systemProperties}`. The second property injects the `os.name` value from the `Properties` object using brackets to qualify a specific property. And the third property uses syntax to specify a default value in case the given property is empty.

With SpEL you can also inject computations or method invocations, as illustrated next:

```

<bean id="commonData"
      ...
      p:randomNumber="#{ T(java.lang.Math).random() * 100.0 }"/>

```

In addition, you can also reference properties from other beans. To reference your own bean properties, you don't use the `@` symbol at the start like you did to reference system properties:

```

<bean id="emailUtilities"
      class="com.apress.springrecipes.spel.EmailUtilities"
      p:email="webmaster@acme.org"
      p:password="springrecipes"
      p:host="localhost:25"/>

<bean id="commonData"
      ...
      p:email="#{emailUtilities.email}"/>

```

Use SpEL in your Annotation Configurations

The expression language can also be used with annotations. The next example shows a POJO that injects system properties directly into a bean using the `@Value` annotation:

```

package com.apress.springrecipes.spel;
...
@Component
public class CommonData {

    @Value("#{systemProperties}")
    private Properties commonProperties;
    @Value("#{systemProperties['os.name']?:'unknown OS'}")
    private String userOS;
}

```

```

@Value("${systemProperties['user.region']?:'unknown region'}")
private String userRegion;

//Setter & getter methods omitted

```

Note The overall setup process to use Spring annotations is different than using Spring XML configurations. The next chapter is dedicated to the use of Spring annotations.

The SpEL syntax used to inject values with the @Value annotation, is identical to that of the XML configuration version.

With annotations you can also use SpEL to inject computations or method invocations, as well as references to other bean properties.

```

@Value("#{ T(java.lang.Math).random() * 100.0 }")
private double randomNumber;
@Value("#{emailUtilities.email}")
private String email;

```

Summary

In this chapter, you learned the core tasks associated with Spring's IoC container. Spring supports both setter injection and constructor injection to define POJO properties, which can be simple values, collections, or bean references. You also learned about referencing POJOs from other POJOs, as well as auto-wiring which can automatically associate POJO's by either type or name.

As collections are essential Java programming elements, you explored how Spring provides support to configure collections. You can use the collection tags in the utility schema to specify more details for a collection, and also define collections as stand-alone beans to share between multiple beans.

You then learned about Spring POJO scopes, which allows POJO instances to be managed differently than the default singleton scope. You also learned how Spring can read external resources (e.g., text files, XML files, properties file, or image files) from different locations (e.g., a file system, classpath, or URL) and use this data in the context of POJO configuration and creation. In addition, you learned how Spring supports different languages in POJOs through the use of i18n resource bundles.

Next, you learned how to customize the initialization and destruction of POJOs, as well as how to use Spring post processors to validate and modify POJOs after creations. Then you explored how to create POJOs using different factory techniques, via a static method, an instance method and Spring's factory bean.

You then learned how to work with Spring environments and profiles to load different sets of POJOs. Next, you explored aspect oriented programming in the context of Spring and learned how to create aspects, pointcuts, and advices. Then you explored how to create POJOs from static fields and object properties, as well as how to make POJOs aware of Spring's IoC container resources.

Next, you learned how to communicate application events between POJOs and use property editors in Spring to automatically handle complex types when creating POJO instances. And then you learned about inheriting POJO configuration which provides support for bean inheritance by extracting common bean configurations to form a parent bean. The parent bean can act as a configuration template, a bean instance, or both at the same time.

In this chapter, you also learned how to use scripting languages supported by Spring to implement POJOs and how to declare them in the Spring IoC container. Spring supports three scripting languages: JRuby, Groovy, and BeanShell. Finally, you learned about Spring's Expression Language or SpEL which provides a means to evaluate certain expressions for configuration purposes or at arbitrary points in POJO's life cycle.

CHAPTER 3



Spring Annotation Driven Core Tasks

In this chapter, you'll learn about Spring's annotation driven tasks. Both annotation driven tasks and regular tasks — like the ones in the previous chapter — can achieve the same end results. The only difference between these types of tasks is how you setup an application and its POJOs (Plain Old Java Objects).

Annotations are declarations added to Java source code to associate configurations with a class or method. The general syntax for Java annotations is the @ symbol, followed by the annotation name and optional parameters inside parenthesis (e.g., @Component("sequenceGenerator")).

This means that if you use an annotation driven task, setting up and configuring a POJO in Spring is as simple as decorating a POJO class or method with an annotation. Whereas if you use a regular task — like the ones described in chapter 2 — setting up and configuring a POJO in Spring requires a separate XML configuration file, that's also more verbose versus an annotation.

Spring annotation driven tasks have gained popularity because they're simpler to use. However, this doesn't mean the Spring regular tasks presented in the previous chapter are deprecated or irrelevant. Both types of tasks are supported in the latest Spring 3.2 version.

The first twelve recipes of this chapter represent equivalent solutions to the first twelve recipes in Chapter 2. This way you can compare both approaches and use the one you're most comfortable with, since what can be a 'simple' and 'less verbose' approach for some can be 'complex' and 'cryptic' approach for others.

For the remaining recipes, such as when an annotation driven approach is an order of magnitude simpler to use than a regular approach (e.g., AOP) only a recipe in this chapter is presented. Similarly, when an annotation driven approach is not available or requires a custom annotation (e.g., Scripting beans), only Chapter 2 contains the relevant recipe.

Therefore the recipes in Chapter 2 and this chapter are complementary to one another.

Tip The source code download is organized to use gradle to build the different Recipe applications. Gradle takes care of loading all the necessary Java classes, dependencies and creating an executable. Chapter 1 describes how to setup the Gradle tool. Furthermore, if a Recipe illustrates more than one approach, the source code is classified with various examples with roman letters (e.g., Recipe_3_1_i, Recipe_3_1_ii, Recipe_3_1_iii,etc).

To build each application, go inside the Recipe directory (e.g., Ch3/Recipe_3_1_i/) and execute the ./gradlew command to compile the source code. Once the source code is compiled, a build/libs sub-directory is created with the application executable. You can then run the application JAR from the command line (e.g., java -jar Recipe_3_1_i-1.0.SNAPSHOT.jar).

3-1. Using Java Config to configure POJOs

Problem

You want to manage POJOs with annotations with Spring's IoC container.

Solution

Design a POJO class. Next, create a Java Config class with a `@Configuration` and `@Bean` annotations to configure POJO instance values or setup Java components with `@Component`, `@Repository`, `@Service` or `@Controller` annotations to later create POJO instance values. Next, instantiate the Spring IoC container to scan for Java classes with annotations. The POJO instances or bean instances then become accessible to put together as part of an application.

How It Works

Suppose you're going to develop an application to generate sequence numbers. And that you are also going to need many series of sequence numbers for different purposes. Each sequence will have its own prefix, suffix, and initial value. So you have to create and maintain multiple generator instances for the application.

Create POJO Class to create beans with Java Config

In accordance with the requirements, you create a `SequenceGenerator` class that has three properties — `prefix`, `suffix`, and `initial`. You also create a private field `counter` to store the numeric value of each generator. Each time you call the `getSequence()` method on a generator instance, you get the last sequence number with the prefix and suffix joined. You declare this last method as synchronized to make it thread-safe.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    private String prefix;
    private String suffix;
    private int initial;

    // Constructors, Getters, and Setters
    // See Recipe 2-1 or book's source code
}
```

Note this last `SequenceGenerator` class can be instantiated by setter methods or by a standard Java constructor.

Create Java Config with `@Configuration` and `@Bean` to creates POJOs

To define instances of a POJO class in the Spring IoC container, you can create a Java Config class with instantiation values. A Java Config class with a POJO or bean definition would look like this:

```
package com.apress.springrecipes.sequence.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.apress.springrecipes.sequence.SequenceGenerator;
```

```

@Configuration
public class SequenceGeneratorConfiguration {
    @Bean
    public SequenceGenerator sequenceGenerator() {
        SequenceGenerator seqgen = new SequenceGenerator();
        seqgen.setPrefix("30");
        seqgen.setSuffix("A");
        seqgen.setInitial("100000");
        return seqgen;
    }
}

```

This is equivalent to an XML configuration with the following definition:

```

<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
      <value>30</value>
    </property>
    <property name="suffix">
      <value>A</value>
    </property>
    <property name="initial">
      <value>100000</value>
    </property>
</bean>

```

Notice the Java Config class is decorated with the `@Configuration` annotation, this tells Spring it's a configuration class. When Spring encounters a class with the `@Configuration` annotation it looks for bean instance definitions in the class, which are Java methods decorated with the `@Bean` annotation. The Java methods create and return a bean instance.

Any method definitions decorated with the `@Bean` annotation generate a bean name based on the method name. Alternatively, you can explicitly specify the bean name in the `@Bean` annotation with the `name` attribute (e.g., `@Bean(name="mys1")`, makes the bean available as `mys1`; NOTE: if you explicitly specify the bean name, the method name is ignored for purposes of bean creation).

Instantiate the Spring IoC container to scan for annotations

You have to instantiate the Spring IoC container to scan for Java classes that contain annotations. In doing so, Spring detects `@Configuration` and `@Bean` annotations so you can later get bean instances from the IoC container itself.

Spring provides two types of IoC container implementations. The basic one is called bean factory. The more advanced one is called application context, which is compatible with the bean factory. Note the configuration files for these two types of IoC containers are identical.

The application context provides more advanced features than the bean factory while keeping the basic features compatible. Therefore, we strongly recommend using the application context for every application unless the resources of an application are restricted (e.g., such as when running Spring for an applet or a mobile device).

The interfaces for the bean factory and the application context are `BeanFactory` and `ApplicationContext`, respectively. The `ApplicationContext` interface is a subinterface of `BeanFactory` for maintaining compatibility.

Since `ApplicationContext` is an interface, you have to instantiate an implementation of it. Spring has several application context implementations, we recommend you use `GenericXmlApplicationContext` which is the newest and most flexible implementation. With this class you can load the XML configuration file from the classpath.

```
ApplicationContext context = new GenericXmlApplicationContext ("appContext.xml");
```

In this case, because a Java Config class is used to define bean instances, the XML configuration file `appContext.xml` is different from the XML configuration files to create application contexts in the previous chapter — which contained bean instance definitions. The `appContext.xml` file just needs to specify which Java packages to scan for annotations:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <context:component-scan base-package="com.apress.springrecipes.sequence.config" />
</beans>
```

Tip The use of `<context:component-scan>` implicitly enables the functionality of `<context:annotation-config>`. There is no need to include the `<context:annotation-config>` element when using `<context:component-scan>`.

Once the application context is instantiated, the object reference — in this case `context` — provides an entry point to access the POJO instances or beans.

Get POJO instances or beans from the IoC Container

You can access beans from a Spring application context just like you did in the previous chapter:

```
ApplicationContext context = ... ;
SequenceGenerator generator = context.getBean("sequenceGenerator", SequenceGenerator.class);
```

A Main class to run the sequence generator application would be the following:

```
package com.apress.springrecipes.sequence;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");

        SequenceGenerator generator = context.getBean("sequenceGenerator", SequenceGenerator.class);

        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

If everything is available in the Java classpath (the `appContext.xml` file, `SequenceGenerator` POJO class and the Spring JAR dependencies), you should see the following output, along with some logging messages:

```
30100000A
30100001A
```

Create POJO Class with `@Component` annotation to create beans with DAO

Up to this point, the Spring bean instantiations have been done hardcoding the values in either an XML configuration file (e.g., examples in Chapter 2) or a Java Config class (e.g., the previous example in this recipe). This was the preferred approach to simplify the Spring examples.

However, the POJO instantiation process for most applications is done from either a database or user input. So now it's time to move forward and use a more real-world scenario. For this section, we'll use a Domain class and a Data Access Object (DAO) class to create POJOs. You still won't need to setup a database — we'll actually hardcode values in the DAO class — but familiarizing yourself with this type of application structure is important since it's the basis for most real-world applications and future recipes.

Suppose you are asked to develop a sequence generator application like the one you did in the last section. We'll need to modify the class structure slightly to accommodate a Domain class and DAO pattern. First, create a domain class `Sequence` containing the `id`, `prefix`, and `suffix` properties.

```
package com.apress.springrecipes.sequence;

public class Sequence {

    private String id;
    private String prefix;
    private String suffix;

    // Constructors, Getters, and Setters

}
```

Then, you create an interface for the Data Access Object (DAO), which is responsible for accessing data from the database. The `getSequence()` method loads a POJO or `Sequence` object from a database table by its ID, while the `getNextValue()` method retrieves the next value of a particular database sequence.

```
package com.apress.springrecipes.sequence;

public interface SequenceDao {

    public Sequence getSequence(String sequenceId);
    public int getNextValue(String sequenceId);
}
```

In a production application, you would implement this DAO interface to use a data-access technology. But to simplify this example, we'll implement a DAO with hardcoded values in a Map to store the sequence instances and values.

```
package com.apress.springrecipes.sequence;
import org.springframework.stereotype.Component;
import java.util.Map;
```

```

@Component("sequenceDao")
public class SequenceDaoImpl implements SequenceDao {

    private Map<String, Sequence> sequences;
    private Map<String, Integer> values;

    public SequenceDaoImpl() {
        sequences = new HashMap<String, Sequence>();
        sequences.put("IT", new Sequence("IT", "30", "A"));
        values = new HashMap<String, Integer>();
        values.put("IT", 100000);
    }

    public Sequence getSequence(String sequenceId) {
        return sequences.get(sequenceId);
    }

    public synchronized int getNextValue(String sequenceId) {
        int value = values.get(sequenceId);
        values.put(sequenceId, value + 1);
        return value;
    }
}

```

Observe how the SequenceDaoImpl class is decorated with the @Component("sequenceDao") annotation. This marks the class so Spring can create POJOs from it. The value inside the @Component annotation defines the bean instance id, in this case sequenceDao. If no bean value name is provided in the @Component annotation, by default bean name is assigned as the uncapitalized non-qualified class name. (e.g., For the SequenceDaoImpl class the default bean name would be sequenceDaoImpl).

A call to the getSequence method returns the value of the given sequenceID. And a call to the getNextValue method creates a new value based on the value of the given sequenceID and returns the new value.

@COMPONENT, @REPOSITORY, @SERVICE OR @CONTROLLER ?

POJOs are classified in application layers. In Spring these layers are three: persistence, service and presentation. @Component is a general purpose annotation to decorate POJOs for Spring detection. Whereas @Repository, @Service, and @Controller are specializations of @Component for more specific cases of POJOs associated with the persistence, service, and presentation layers.

If you're unsure about a POJOs purpose you can decorate it with the @Component annotation. However, it's better to use the specialization annotations where possible, because these provide extra facilities based on a POJOs purpose (e.g., @Repository causes exceptions to be wrapped up as DataAccessExceptions which makes debugging easier).

Instantiate the Spring IoC container with filters to scan for annotations

By default, Spring detects all classes decorated with @Configuration, @Bean, @Component, @Repository, @Service, and @Controller annotations, among others. You can customize the scan process to include one or more include/exclude filters. This is helpful when a Java package has dozens or hundreds of classes. For certain Spring application contexts,

it can be necessary to exclude or include POJOs with certain annotations. In addition, scanning every package class can slow down the process unnecessarily.

Spring supports four types of filter expressions. The annotation and assignable types are to specify an annotation type and a class/interface for filtering. The regex and aspectj types allow you to specify a regular expression and an AspectJ pointcut expression for matching the classes. You can also disable the default filters with the use-default-filters attribute.

For example, the following component scan includes all classes in the `com.apress.springrecipes.sequence` whose name contains the word Dao or Service, and excludes the classes with the `@Controller` annotation:

```
<beans ...>
    <context:component-scan base-package="com.apress.springrecipes.sequence">
        <context:include-filter type="regex"
            expression="com\apress\springrecipes\sequence\..*Dao.*" />
        <context:include-filter type="regex"
            expression="com\apress\springrecipes\sequence\..*Service.*" />
        <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Controller" />
    </context:component-scan>
</beans>
```

When applying include filters to detect all classes whose name contains the word Dao or Service, even classes that don't have annotations are auto-detected.

Get POJO instances or beans from the IoC Container

Then, you can test the preceding components with the following Main class:

```
package com.apress.springrecipes.sequence;

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

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");

        SequenceDao sequenceDao =
            (SequenceDao) context.getBean("sequenceDao");

        System.out.println(sequenceDao.getNextValue("IT"));
        System.out.println(sequenceDao.getNextValue("IT"));
    }
}
```

Remove all XML type configuration files and use annotations all around

Up to this point, you still have an XML configuration file that defines which packages to scan so Spring can detect annotations and create the necessary POJOs. But Spring can actually forgo the use XML configuration entirely, by specifying which packages to scan in the application context.

In addition to the `GenericXmlApplicationContext` class used up to this point, Spring also has the `AnnotationConfigApplicationContext` class which can be configured to scan packages for annotations and create the declared POJOs. The following Main class illustrated the use of `AnnotationConfigApplicationContext`.

```
package com.apress.springrecipes.sequence;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
        context.scan("com.apress.springrecipes.sequence");
        SequenceDao sequenceDao =
            (SequenceDao) context.getBean("sequenceDao");

        System.out.println(sequenceDao.getNextValue("IT"));
        System.out.println(sequenceDao.getNextValue("IT"));
    }
}
```

Notice that after an instance of the `AnnotationConfigApplicationContext` class is created, a call is made to the `scan("com.apress.springrecipes.sequence")` method. This tells Spring to scan classes in the `com.apress.springrecipes.sequence` package for annotations and create the declared POJOs. In addition to the `scan()` method which inspects complete packages, you can also use the `register()` method to inspect individual classes for annotations.

```
context.register(SequenceDaoImpl.class, SequenceConfiguration.class)
```

Tip You can use a mix of annotations and XML configuration files to configure Spring applications. You don't necessarily need to use annotations or XML exclusively. Depending on personal preferences and the type of application, it can be easier to setup certain tasks with annotations and others with XML configuration files. Go with the approach that feels easiest and most readable to you. The source code for future chapters uses a mixed configuration approach to illustrate this concept.

3-2. Create POJOs by Invoking a Constructor

Problem

You would like to create a POJO instance or bean in the Spring IoC container by invoking its constructor, which is the most common and direct way of creating beans. It is equivalent to using the `new` operator to create objects in Java.

Solution

Define a POJO class with a constructor or constructors. Next, create a Java Config class to configure POJO instance values with constructors for the Spring IoC container. Next, instantiate the Spring IoC container to scan for Java classes with annotations. The POJO instances or bean instances become accessible to put together as part of an application.

How It Works

Suppose you're going to develop a shop application to sell products online. First of all, you create the Product POJO class, which has several properties, such as the product name and price. As there are many types of products in your shop, you make the Product class abstract to extend it for different product subclasses.

```
package com.apress.springrecipes.shop;

public abstract class Product {

    private String name;
    private double price;

    public Product() {}

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Getters and Setters
    ...

    public String toString() {
        return name + " " + price;
    }
}
```

Create the POJO Classes with Constructors

Then you create two product subclasses, Battery and Disc. Each of them has its own properties.

```
package com.apress.springrecipes.shop;

public class Battery extends Product {

    private boolean rechargeable;

    public Battery() {
        super();
    }

    public Battery(String name, double price) {
        super(name, price);
    }

    // Getters and Setters
    ...
}
```

```

package com.apress.springrecipes.shop;

public class Disc extends Product {

    private int capacity;

    public Disc() {
        super();
    }

    public Disc(String name, double price) {
        super(name, price);
    }

    // Getters and Setters
    ...
}

```

Create Java Config for your POJO

To define instances of a POJO class in the Spring IoC container, you have to create a Java Config class with instantiation values. A Java Config class with POJO or bean definition made by invoking constructors would look like this:

```

package com.apress.springrecipes.shop.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.apress.springrecipes.shop.Product;
import com.apress.springrecipes.shop.Disc;
import com.apress.springrecipes.shop.Battery;

@Configuration
public class ShopConfiguration {
    @Bean
    public Product aaa() {
        Battery p1 = new Battery();
        p1.setName("AAA");
        p1.setPrice(2.5);
        p1.setRechargeable(true);
        return p1;
    }
    @Bean
    public Product cdrw() {
        Disc p2 = new Disc("CD-RW",1.5);
        p2.setCapacity(700);
        return p2;
    }
}

```

To create the beans you'll have to create an XML file to tell Spring to scan the `com.apress.springrecipes.shop.config` package.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <context:component-scan base-package="com.apress.springrecipes.shop.config" />
</beans>
```

Next, you can write the following Main class to test your products by retrieving them from the Spring IoC container:

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");

        Product aaa = context.getBean("aaa",Product.class);
        Product cdrw = context.getBean("cdrw",Product.class);
        System.out.println(aaa);
        System.out.println(cdrw);
    }
}
```

3-3. Use POJO References and Auto-Wiring to Interact with other POJOs

Problem

The POJO instances or beans that make up an application often need to collaborate with each other to complete the application's functions. You want to use annotations to use POJO references and auto-wiring.

Solution

For POJO instances defined in a Java config class you can use standard Java code to create references between beans. To auto-wire POJO references you can mark a field, a setter method, a constructor, or even an arbitrary method with the `@Autowired` annotation.

How It Works

Reference POJOs in a Java config class

When POJO instances are defined in a Java config class — as illustrated in Recipe 3-1 and Recipe 3-2 — POJO references are straightforward to use because everything is Java code. Take for example, when a bean property references another bean.

```
@Configuration
public class SequenceConfiguration {
    @Bean
    public DatePrefixGenerator datePrefixGenerator() {
        DatePrefixGenerator dpg = new DatePrefixGenerator();
        dpg.setPattern("yyyyMMdd");
        return dpg;
    }
    @Bean
    public SequenceGenerator sequenceGenerator() {
        SequenceGenerator sequence= new SequenceGenerator();
        sequence.setInitial(100000);
        sequence.setSuffix("A");
        sequence.setPrefixGenerator(datePrefixGenerator());
        return sequence;
    }
}
```

The prefixGenerator property of the SequenceGenerator class is an instance of a DatePrefixGenerator bean. The first bean declaration creates a DatePrefixGenerator POJO. By convention, the bean becomes accessible with the bean name datePrefixGenerator (i.e., the method name). But since the bean instantiation logic is also a standard Java method, the bean is also accessible making a standard Java call. When the prefixGenerator property is set — in the second bean, via a setter — a standard Java call is made to the method datePrefixGenerator() to reference the bean.

Auto-wire POJO fields with the @Autowired annotation

Next, let's use auto-wiring on the SequenceDao field of the DAO SequenceDaoImpl class introduced in the second part of Recipe 3-1. We'll add a service class to the application to illustrate auto-wiring with the DAO class.

A service class to generate service objects is another real world application best practice, which acts as a façade to access DAO objects — instead of accessing DAO objects directly. Internally, the service object interacts with the DAO to handle the sequence generation requests.

```
package com.apress.springrecipes.sequence;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
```

```

public void setSequenceDao(SequenceDao sequenceDao) {
    this.sequenceDao = sequenceDao;
}

public String generate(String sequenceId) {
    Sequence sequence = sequenceDao.getSequence(sequenceId);
    int value = sequenceDao.getNextValue(sequenceId);
    return sequence.getPrefix() + value + sequence.getSuffix();
}
}

```

The SequenceService class is decorated with the @Component annotation. This allows Spring to detect the POJO. Because the @Component annotation has no name, the default bean name is sequenceService which is based on the class name.

The sequenceDao property of the SequenceService class is decorated with the @Autowired annotation. This allows Spring to auto-wire the property with the sequenceDao bean (i.e., the SequenceDaoImpl class).

Once the POJO classes are annotated with the previous annotations, an XML configuration file has to be setup to tell Spring which packages to scan for annotations — this process is identical to the previous recipes. Then, you can test the components with a Main class like the one used in previous recipes.

The @Autowired annotation can also be applied to a property of an array type to have Spring auto-wire all the matching beans. For example, you can annotate a PrefixGenerator[] property with @Autowired. Then, Spring will auto-wire all the beans whose type is compatible with PrefixGenerator at one time.

```

package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {

    @Autowired
    private PrefixGenerator[] prefixGenerators;
    ...
}

```

If you have multiple beans whose type is compatible with the PrefixGenerator defined in the IoC container, they will be added to the prefixGenerators array automatically.

In a similar way, you can apply the @Autowired annotation to a type-safe collection. Spring can read the type information of this collection and auto-wire all the beans whose type is compatible.

```

package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {

    @Autowired
    private List<PrefixGenerator> prefixGenerators;
    ...
}

```

If Spring notices that the `@Autowired` annotation is applied to a type-safe `java.util.Map` with strings as the keys, it will add all the beans of the compatible type, with the bean names as the keys, to this map.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {

    @Autowired
    private Map<String, PrefixGenerator> prefixGenerators;
    ...
}
```

Auto-wire POJO methods and constructors with the `@Autowired` annotation and make auto-wiring optional

The `@Autowired` annotation can also be applied directly to the setter method of a POJO. As an example, you can annotate the setter method of the `prefixGenerator` property with `@Autowired`. Then, Spring attempts to wire a bean whose type is compatible with `prefixGenerator`.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {
    ...
    @Autowired
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

By default, all the properties with `@Autowired` are required. When Spring can't find a matching bean to wire, it will throw an exception. If you want a certain property to be optional, set the required attribute of `@Autowired` to `false`. Then, when Spring can't find a matching bean, it will leave this property unset.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {
    ...
    @Autowired(required=false)
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

You may also apply the `@Autowired` annotation to a method with an arbitrary name and an arbitrary number of arguments, and, in that case, Spring attempts to wire a bean with the compatible type for each of the method arguments.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {
    ...
    @Autowired
    public void myOwnCustomInjectionName(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Resolve auto-wire ambiguity with the `@Qualifier` annotation

By default, auto-wiring by type will not work when there is more than one bean with the compatible type in the IoC container and the property isn't a group type (e.g., array, list, map), as illustrated previously. However, there are two workarounds to auto-wiring by type if there's more than one bean of the same type, the `@Primary` annotation and the `@Qualifier` annotation.

Resolve auto-wire ambiguity with the `@Primary` annotation

Spring allows you to specify a candidate bean by type decorating the candidate with the `@Primary` annotation. The `@Primary` annotation gives preference to a bean when multiple candidates are qualified to autowire a single-valued dependency.

```
package com.apress.springrecipes.sequence;
...
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Primary;

@Component
@Primary
public class DatePrefixGenerator implements PrefixGenerator {

    public String getPrefix() {
        DateFormat formatter = new SimpleDateFormat("yyyyMMdd");
        return formatter.format(new Date());
    }
}
```

Notice the previous POJO implements the `PrefixGenerator` interface and is decorated with the `@Primary` annotation. If you attempted to autowire a bean with a `PrefixGenerator` type, even if Spring had more than one bean instance with the same `PrefixGenerator` type, Spring would autowire the `DatePrefixGenerator` because it's marked with the `@Primary` annotation.

Resolve auto-wire ambiguity with the @Qualifier annotation

Spring also allows you to specify a candidate bean by type providing its name in the @Qualifier annotation.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {

    @Autowired
    @Qualifier("datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

Once you've done this, Spring attempts to find a bean with that name in the IoC container and wire it into the property.

The @Qualifier annotation can also be applied to a method argument for auto-wiring.

```
package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {
    ...
    @Autowired
    public void myOwnCustomInjectionName(
        @Qualifier("datePrefixGenerator") PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

If you want to auto-wire bean properties by name, you can annotate a setter method, a constructor, or a field with the JSR-250 @Resource annotation described in the next recipe.

Resolve POJO References from multiple locations

As an application grows it can become difficult to manage every POJO in a single Java Configuration class. A common practice is to separate POJOs into multiple Java Configuration classes according to their functionalities. When you create multiple Java Configuration classes, obtaining references and auto-wiring POJOs that are defined in different classes isn't as straightforward as when everything is in a single Java Configuration class.

One approach is to initialize the application context with the location of each Java Configuration class. In this manner, the POJOs for each Java configuration class are loaded into the context and references and autowiring between POJOs is possible.

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();
context.register({"SequenceConfiguration.class", "PrefixConfiguration.class"});
context.refresh();
```

Another alternative is to use the `@Import` annotation so Spring makes the POJOs from one configuration file available in another.

```
package com.apress.springrecipes.sequence.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Import;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import com.apress.springrecipes.sequence.SequenceGenerator;
import com.apress.springrecipes.sequence.PrefixGenerator;

@Configuration
@Import(PrefixConfiguration.class)
public class SequenceConfiguration {
    @Value("#{datePrefixGenerator}")
    private PrefixGenerator prefixGenerator;

    @Bean
    public SequenceGenerator sequenceGenerator() {
        SequenceGenerator sequence= new SequenceGenerator();
        sequence.setInitial(100000);
        sequence.setSuffix("A");
        sequence.setPrefixGenerator(prefixGenerator);
        return sequence;
    }
}
```

The `sequenceGenerator` bean requires to set a `prefixGenerator` bean. But notice no `prefixGenerator` bean is defined in the Java Configuration class. The `prefixGenerator` bean is defined in a separate Java Configuration class `PrefixConfiguration`.

With the `@Import(PrefixConfiguration.class)` annotation, Spring brings all the POJOs in the Java Configuration class into the scope of the present Configuration class. With the POJOs from `PrefixConfiguration` in scope, we use the `@Value` annotation and SpEL to inject the bean named `datePrefixGenerator` into the `prefixGenerator` field. Once the bean is injected it can be used to set a `prefixGenerator` bean for the `sequenceGenerator` bean.

3-4. Auto-wire POJOs the `@Resource` and `@Inject` annotation

POJOS WITH ANNOTATIONS AND JAVA COLLECTION ATTRIBUTES

Recipe 3-4 won't describe how to configure POJOs with Java collection attributes, like Recipe 2-4 described this process with XML configuration files.

POJOs with annotations that use Java collection attributes don't require any specific technique. Java config classes use standard Java syntax and can therefore use standard Java collection code. And to reference Java collection attributes you can create multiple bean instances of the same type and use auto-wiring to automatically group the different beans into a group or collection — this last technique is illustrated in Recipe 3-3.

Problem

You want to use the Java standard `@Resource` and `@Inject` annotations to reference POJOs via autowiring, instead of the Spring specific `@Autowired` annotation.

Solution

JSR-250 or Common Annotations for the Java Platform defines the `@Resource` annotation to autowire POJO references by name. The JSR-330 or Standard Annotations for injection defines the `@Inject` annotations to autowire POJO references by type.

How It Works

The `@Autowired` annotation described in the previous recipe belongs to the Spring framework, specifically to the `org.springframework.beans.factory.annotation` package. This means it can only be used in the context of the Spring framework.

Soon after Spring added support for the `@Autowired` annotation, the Java language itself standardized various annotations to fulfill the same purpose of the `@Autowired` annotation. These annotations are `@Resource` which belongs to the `javax.annotation` package and `@Inject` which belongs to the `javax.inject` package.

Auto-wire POJOs with the `@Resource` annotation

By default, the `@Resource` annotation works like Spring's `@Autowired` annotation and attempts to autowire by type. For example, the following POJO attribute is decorated with the `@Resource` annotation and so Spring attempts to locate a POJO that matches the `PrefixGenerator` type.

```
package com.apress.springrecipes.sequence;

import javax.annotation.Resource;

public class SequenceGenerator {

    @Resource
    private PrefixGenerator prefixGenerator;
    ...
}
```

However, unlike the `@Autowired` annotation which requires the `@Qualifier` annotation to autowire a POJO by name, the `@Resource` ambiguity is eliminated if more than one POJO type of the same kind exists. Essentially, the `@Resource` annotation provides the same functionality as putting together the `@Autowired` annotation and `@Qualifier` annotation.

Auto-wire POJOs with the `@Inject` annotation

By, the `@Inject` annotation also attempts to autowire by type, like the `@Resource` and `@Autowired` annotations. For example, the following POJO attribute is decorated with the `@Inject` annotation and so Spring attempts to locate a POJO that matches the `PrefixGenerator` type.

```

package com.apress.springrecipes.sequence;

import javax.inject.Inject;

public class SequenceGenerator {

    @Inject
    private PrefixGenerator prefixGenerator;
    ...
}

```

But just like the `@Resource` and `@Autowired` annotations, a different approach has to be used to match POJOs by name or avoid ambiguity if more than one POJO type of the same kind exists.

The first step to do autowiring by name with the `@Inject` annotation is to create a custom annotation to identify both the POJO injection class and POJO injection point.

```

package com.apress.springrecipes.sequence;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.PARAMETER})
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface DatePrefixAnnotation {
}

```

Notice the custom annotation makes use of the `@Qualifier` annotation. This annotation is different from the one used with Spring's `@Qualifier` annotation, as this last class belongs to the same Java package as the `@Inject` annotation (i.e., `javax.inject`).

Once the custom annotation is done, it's necessary to decorate the POJO injection class that generates the bean instance, which in this case is the `DatePrefixGenerator` class.

```

package com.apress.springrecipes.sequence;
...
@DatePrefixAnnotation
public class DatePrefixGenerator implements PrefixGenerator {
...
}

```

Finally, the POJO attribute or injection point is decorated with the same custom annotation to qualify the POJO and eliminate any ambiguity.

```
package com.apress.springrecipes.sequence;

import javax.inject.Inject;

public class SequenceGenerator {

    @Inject @DataPrefixAnnotation
    private PrefixGenerator prefixGenerator;
    ...
}
```

@AUTOWIRED, @RESOURCE OR @INJECT ?

As you've seen in Recipes 3-3 and 3-4, the three annotations `@Autowired`, `@Resource` and `@Inject` can achieve the same result. The `@Autowired` annotation is a Spring based solution, whereas the `@Resource` and `@Inject` annotations are Java standard (i.e., JSR) based solutions.

If you're going to do name based autowiring, the `@Resource` annotation offers the simplest syntax. For autowiring by class type, all three annotations are as straightforward to use because all three require a single annotation.

3-5. Set a POJOs Scope with the `@Scope` annotation

Problem

When you declare a POJO instance with an annotation like `@Component`, you are actually defining a template for bean creation, not an actual bean instance. When a bean is requested by the `getBean()` method or a reference from other beans, Spring decides which bean instance should be returned according to the bean scope. Sometimes, you have to set an appropriate scope for a bean other than the default scope.

Solution

A bean's scope is set in the scope is set with the `@Scope` annotation. By default, Spring creates exactly one instance for each bean declared in the IoC container, and this instance is shared in the scope of the entire IoC container. This unique bean instance is returned for all subsequent `getBean()` calls and bean references. This scope is called singleton, which is the default scope of all beans. Table 3-1 lists all valid bean scopes in Spring.

Table 3-1. Valid Bean Scopes in Spring

Scope	Description
singleton	Creates a single bean instance per Spring IoC container
prototype	Creates a new bean instance each time when requested
request	Creates a single bean instance per HTTP request; only valid in the context of a web application
session	Creates a single bean instance per HTTP session; only valid in the context of a web application
globalSession	Creates a single bean instance per global HTTP session; only valid in the context of a portal application

How It Works

To demonstrate the concept of bean scope, let's consider a shopping cart example in a shopping application. First, you create the `ShoppingCart` class as follows:

```
package com.apress.springrecipes.shop;
...
@Component
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();

    public void addItem(Product item) {
        items.add(item);
    }

    public List<Product> getItems() {
        return items;
    }
}
```

Then you declare some product beans in a Java config file so they can later be added to the shopping cart:

```
package com.apress.springrecipes.shop.config;

import org.springframework.stereotype.Component;

@Configuration
public class ShopConfiguration {
    @Bean
    public Product aaa() {
        Battery p1 = new Battery();
        p1.setName("AAA");
        p1.setPrice(2.5);
        p1.setRechargeable(true);
        return p1;
    }
    @Bean
    public Product cdrw() {
        Disc p2 = new Disc("CD-RW",1.5);
        p2.setCapacity(700);
        return p2;
    }
}
```

Next, you declare the necessary statements to scan packages that contain annotations inside a Spring XML configuration file.

```
<context:component-scan base-package="com.apress.springrecipes.shop.config" />
<context:component-scan base-package="com.apress.springrecipes.shop">
    <context:include-filter type="regex" expression="com\.\apress\.
        springrecipes\.\shop\..*\ ShoppingCart.*" />
</context:component-scan>
```

Once you do this, you can define a `Main` class to test the shopping cart by adding some products to it. Suppose there are two customers navigating in your shop at the same time. The first one gets a shopping cart by the `getBean()` method and adds two products to it. Then, the second customer also gets a shopping cart by the `getBean()` method and adds another product to it.

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");

        Product aaa = (Product) context.getBean("aaa");
        Product cdrw = (Product) context.getBean("cdrw");
        Product dvdrw = (Product) context.getBean("dvdrw");

        ShoppingCart cart1 = (ShoppingCart) context.getBean("shoppingCart");
        cart1.addItem(aaa);
        cart1.addItem(cdrw);
        System.out.println("Shopping cart 1 contains " + cart1.getItems());

        ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
        cart2.addItem(dvdrw);
        System.out.println("Shopping cart 2 contains " + cart2.getItems());
    }
}
```

As a result of the preceding bean declaration, you can see that the two customers get the same shopping cart instance.

```
Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
Shopping cart 2 contains [AAA 2.5, CD-RW 1.5, DVD-RW 3.0]
```

This is because Spring's default bean scope is `singleton`, which means Spring creates exactly one shopping cart instance per IoC container.

In your shop application, you expect each customer to get a different shopping cart instance when the `getBean()` method is called. To ensure this behavior, the scope of the `shoppingCart` bean needs to be set to `prototype`. Then Spring creates a new bean instance for each `getBean()` method call.

```
package com.apress.springrecipes.shop;
...
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;

@Component
@Scope("prototype")
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();
    ...
}
```

Now if you run the Main class again, you can see the two customers get a different shopping cart instance.

```
Shopping cart 1 contains [AAA 2.5, CD-RW 1.5]
Shopping cart 2 contains [DVD-RW 3.0]
```

3-6. Use data from External Resources (Text files, XML files, properties files, or image files)

Problem

Sometimes, applications need to read external resources (e.g., text files, XML files, properties file, or image files) from different locations (e.g., a file system, classpath, or URL). Usually, you have to deal with different APIs for loading resources from different locations.

Solution

Spring offers the `@PropertySource` annotation as a facility to load the contents of `.properties` file (i.e., key-value pairs) to set up bean properties.

Spring also has a resource loader mechanism which provides a unified `Resource` interface to retrieve any type of external resource by a resource path. You can specify different prefixes for this path to load resources from different locations with the `@Value` annotation. To load a resource from a file system, you use the `file` prefix. To load a resource from the classpath, you use the `classpath` prefix. You can also specify a URL in the resource path.

How It Works

To read the contents of a `properties` file (i.e., key-value pairs) to setup bean properties you can use Spring's `@PropertySource` annotation with `PropertySourcesPlaceholderConfigurer`. If you want to read the contents of any file you can use Spring's Resource mechanism decorated with the `@Value` annotation.

Use properties file data to setup POJO instantiation values

Let's assume you have a series of values in a `properties` file you want to access to setup bean properties. Typically this can be the configuration properties of a database or some other application values composed of key-values. For example, take the following key-values stored in a file called `discounts.properties`.

```
specialcustomer.discount=0.1
summer.discount=0.15
endofyear.discount=0.2
```

Note To read properties files for the purpose of internationalization (i18n) see the next recipe.

To make the contents of the `discounts.properties` file accessible to set up other beans, you can use the `@PropertySource` annotation to convert the key-values into a bean inside a Java config class.

```
package com.apress.springrecipes.shop.config;
...
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration
@PropertySource("classpath:discounts.properties")
public class ShopConfiguration {
    private @Value("${specialcustomer.discount:0}") double specialCustomerDiscountField;
    ...

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

    @Bean
    public Product aaa() {
        Battery p1 = new Battery();
        p1.setName("AAA");
        p1.setPrice(2.5);
        p1.setRechargeable(true);
        p1.setDiscount(specialCustomerDiscountField);
        return p1;
    }
}
...

```

You define a `@PropertySource` annotation with a value of `classpath:discounts.properties` to decorate the Java config class. The `classpath:` prefix tells Spring to look for the `discounts.properties` file in the Java classpath.

Note The parameters `ignoreResourceNotFound` and `ignoreUnresolvablePlaceholders` to increase fault tolerance in the use properties — as used in the XML equivalent process in Recipe 2-6 — is still not supported for the `@PropertySource` annotation at the time of this writing. See <https://jira.springsource.org/browse/SPR-8371> for updates.

Once you define the `@PropertySource` annotation to load the properties file, you also need to define a `PropertySourcePlaceholderConfigurer` bean with the `@Bean` annotation. Spring automatically wires the `@PropertySource` `discounts.properties` file so its properties become accessible as bean properties.

Next, you need to define Java variables to take values from the discount `discounts.properties` file. To define the Java variable values with these values you make use of the `@Value` annotation with SpeL statements.

The syntax is `@Value("${key:default_value}")`. A search is done for the key value in all the loaded application properties. If a matching `key=value` is found in the properties file, the corresponding value is assigned to the bean property. If no matching `key=value` is found in the loaded application properties, the `default_value` (i.e. after `$(key:)`) is assigned to the bean property.

Once a Java variable is set with a discount value, you can use it to setup bean instances for a bean's discount property.

If you want to use properties file data for a different purpose than setting up bean properties, you should use Spring's Resource mechanism which is described next.

Use data from any external resource file for use in a POJO

Suppose you want to display a banner at the startup of an application. The banner is made up of the following characters and stored in a text file called `banner.txt`. This file can be put in the classpath of your application.

```
*****
* Welcome to My Shop! *
*****
```

Next, let's write a `BannerLoader` POJO class to load the banner and output it to the console.

```
package com.apress.springrecipes.shop;

import org.springframework.core.io.Resource;
...
import javax.annotation.PostConstruct;
public class BannerLoader {

    private Resource banner;

    public void setBanner(Resource banner) {
        this.banner = banner;
    }

    @PostConstruct
    public void showBanner() throws IOException {
        InputStream in = banner.getInputStream();

        BufferedReader reader = new BufferedReader(new InputStreamReader(in));
        while (true) {
            String line = reader.readLine();
            if (line == null)
                break;
            System.out.println(line);
        }
        reader.close();
    }
}
```

Notice the POJO `banner` field is a Spring Resource type. The field value will be populated through setter injection when the bean instance is created — to be explained shortly. The `showBanner` method makes a call to the `getInputStream()` method to retrieve the input stream from the `Resource` field. Once you have an `InputStream`, you're able to use standard Java file manipulation class. In this case, the file contents are read line by line with `BufferedReader` and outputted to the console.

Also notice the `showBanner()` method is decorated with the `@PostConstruct` annotation. Because you want to show the banner at startup, you use this annotation to tell Spring to invoke the method automatically after creation. This guarantees the `showBanner()` method is one of the first methods to be run by the application and therefore ensures the banner appears at the outset.

Next, the POJO `BannerLoader` needs to be initialized as an instance. In addition, the `banner` field of the `BannerLoader` also needs to be injected. So let's create a Java config class for these tasks.

```
@Configuration
@PropertySource("classpath:discounts.properties")
public class ShopConfiguration {
    ...
    @Value("classpath:banner.txt")
    private Resource banner;

    ...
    @Bean
    public BannerLoader bannerLoader() {
        BannerLoader bl = new BannerLoader();
        bl.setBanner(banner);
        return bl;
    }
}
```

See how the `banner` property is decorated with the `@Value("classpath:banner.txt")` annotation. This tells Spring to search for the `banner.txt` file in the classpath and inject it. Spring uses the preregistered property editor `ResourceEditor` to convert the file definition into a `Resource` object before injecting it into the bean.

Once the `banner` property is injected, it's assigned to the `BannerLoader` bean instance via setter injection.

Because the banner file is located in the Java classpath, the resource path starts with the `classpath:` prefix. The previous resource path specifies a resource in the relative path of the file system. You can specify an absolute path as well.

`file:c:/shop/banner.txt`

When a resource is located in Java's classpath, you have to use the `classpath` prefix. If there's no path information presented, it will be loaded from the root of the classpath.

`classpath:banner.txt`

If the resource is located in a particular package, you can specify the absolute path from the classpath root.

`classpath:com/apress/springrecipes/shop/banner.txt`

Besides support to load from a file system path or the classpath, a resource can also be loaded by specifying a URL.

<http://springrecipes.apress.com/shop/banner.txt>

Since the bean class uses the `@PostConstruct` annotation on the `showBanner()` method, the banner is sent to output when the IoC container is setup. Because of this, there's no need to tinker with an application's context or explicitly call the bean to output the banner. However, sometimes it can be necessary to access an external resource to interact with an application's context.

Now suppose you want to display a legend at the end of an application. The legend is made up of the discounts previously described in the `discounts.properties` file. To access the contents of the properties file you can also leverage Spring's Resource mechanism.

Next, let's use Spring's Resource mechanism, but this time directly inside an application's Main class to output a legend when the application finishes.

```
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.support.PropertiesLoaderUtils;
...
...
public class Main {
    public static void main(String[] args) throws Exception {
        ...
        ...
        ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
        cart2.addItem(dvdrw);
        System.out.println("Shopping cart 2 contains " + cart2.getItems());

        Resource resource = new ClassPathResource("discounts.properties");
        Properties props = PropertiesLoaderUtils.loadProperties(resource);
        System.out.println("And don't forget our discounts!");
        System.out.println(props);
    }
}
```

Spring's `ClassPathResource` class is used to access the `discounts.properties` file, which casts the file's contents into a `Resource` object. Next, the `Resource` object is processed into a `Properties` object with Spring's `PropertiesLoaderUtils` class. Finally, the contents of the `Properties` object are sent to the console as the final output of the application.

Because the legend file (i.e., `discounts.properties`) is located in the Java classpath, the resource is accessed with Spring's `ClassPathResource` class. If the external resource were in a file system path the resource would be loaded with Spring's `FileSystemResource`.

```
Resource resource = new FileSystemResource("c:/shop/banner.txt")
```

If the external resource were at a URL the resource would be loaded with Spring's `UrlResource`.

```
Resource resource = new UrlResource("http://www.apress.com/")
```

3-7. Resolve I18N Text Messages for different locales in properties files

Problem

You want an application to support internationalization (I18N) via annotations.

Solution

`MessageSource` is an interface that defines several methods for resolving messages in resource bundles.

`ResourceBundleMessageSource` is the most common `MessageSource` implementation that resolves messages from resource bundles for different locales. After you implement a `ResourceBundleMessageSource` POJO, you can use the `@Bean` annotation in a Java config file to make the I18N data available in an application.

How It Works

As an example, create the following resource bundle, `messages_en_US.properties`, for the English language in the United States. Resource bundles are loaded from the root of the classpath, so ensure it's available on the Java classpath. Place the following key-value in the file:

```
alert.checkout=A shopping cart has been checked out.  
alert.inventory.checkout=A shopping cart with {0} has been checked out at {1}.
```

To resolve messages from resource bundles, let's create a Java config file with an instance of a `ReloadableResourceBundleMessageSource` bean.

```
import org.springframework.context.support.  
ReloadableResourceBundleMessageSource;  
  
@Configuration  
public class ShopConfiguration {  
  
    @Bean  
    public static ReloadableResourceBundleMessageSource messageSource() {  
        ReloadableResourceBundleMessageSource messageSource= new  
        ReloadableResourceBundleMessageSource();  
        String[] resources = {"classpath:messages"};  
        messageSource.setBasename(resources);  
        messageSource.setCacheSeconds(1);  
        return messageSource;  
    }  
}
```

The bean instance must have the name `messageSource` for the application context to detect it. Also notice the `@Bean` definition is declared as `static`, unlike other beans you've defined in Java config files up to this point. The `static` qualifier is necessary because `ReloadableResourceBundleMessageSource` is bean post processor. Bean post processors are discussed in detail in Recipes 3-9 and 2-9.

Inside the bean definition you declare a String list via the `setBasename` method to locate bundles for the `ResourceBundleMessageSource`. In this case, we just specify the default convention to lookup files located in Java classpath that start with `messages`. In addition, the `setCacheSeconds` methods sets a value to 1 to avoid reading stale messages. Note that a refresh attempt first checks the last-modified timestamp of the properties file before actually reloading it; so if files don't change, the `setCacheSeconds` interval can be set rather low, as refresh attempts aren't actually reloaded.

For this `MessageSource` definition, if you look up a text message for the United States locale, whose preferred language is English, the resource bundle `message_en_US.properties` is considered first. If there's no such resource bundle or the message can't be found, then a `message_en.properties` file that matches the language is considered. If a resource bundle still can't be found, the default `message.properties` for all locales is chosen. For more information on resource bundle loading, you can refer to the Javadoc of the `java.util.ResourceBundle` class.

Next, you can configure the application context to resolve messages with the `getMessage()` method. The first argument is the key corresponding to the message, and the third is the target locale.

```
package com.apress.springrecipes.shop;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");
        ...
        String alert = context.getMessage("alert.checkout", null, Locale.US);
        String alert_inventory = context.getMessage("alert.inventory.checkout", new Object[] {
            "[DVD-RW 3.0]", new Date() }, Locale.US);
        System.out.println(alert);
        System.out.println(alert_inventory);
    }
}
```

The second argument of the `getMessage()` method is an array of message parameters. In the first `String` statement the value is `null`, in the second `String` statement an object array to fill in the message parameters is used.

In the `Main` class, you can resolve text messages because you can access the application context directly. But for a bean to resolve text messages, you have to inject a `MessageSource` implementation into the bean that needs to resolve text messages. Let's implement a `Cashier` class for the shopping application that illustrates how to resolve messages.

```
package com.apress.springrecipes.shop;
...
@Component
public class Cashier {

    @Autowired
    private MessageSource messageSource;

    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public void checkout(ShoppingCart cart) throws IOException {
        String alert = messageSource.getMessage("alert.inventory.checkout",
            new Object[] { cart.getItems(), new Date() },
            Locale.US);
        System.out.println(alert);
    }
}
```

Notice the POJO messageSource field is a Spring MessageSource type. The field value is decorated with the @Autowired annotation so it's populated through setter injection when the bean instance is created. Then the checkout method can access the messageSource field, which gives the bean access to the getMessage method to gain access to text messages based on I18N criteria.

Finally, for both of the previous scenarios, remember you need to setup the necessary XML configuration file so Spring is able to detect POJOs with annotations.

3-8. Customize POJO Initialization and Destruction with annotations

Problem

Some POJOs have to perform certain types of initialization tasks before they're used. These tasks can include opening a file, opening a network/database connection, allocating memory, and so on. In addition, these same POJO also have to perform the corresponding destruction tasks at the end of their life cycle. Therefore, sometimes it's necessary to customize bean initialization and destruction in the Spring IoC container.

Solution

Spring can recognize initialization and destruction callback methods by setting the initMethod and destroyMethod attributes of a @Bean definition in a Java config class. Or Spring can also recognize initialization and destruction callback methods if POJO methods are decorated with the @PostConstruct and @PreDestroy annotations, respectively.

Spring can also delay the creation of a bean up until the point it's required — a process called lazy initialization — with the @Lazy annotation. Spring can also ensure the initialization of certain beans before others with the @DependsOn annotation.

How It Works

Define methods to run before POJO initialization and destruction with @Bean

Let's take the case of the shopping application and consider an example involving a checkout function. Let's modify the Cashier class to record a shopping cart's products and the checkout time to a text file.

```
package com.apress.springrecipes.shop;
...
public class Cashier {

    private String fileName;
    private String path;
    private BufferedWriter writer;

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public void setPath(String path) {
        this.path = path;
    }
}
```

```

public void openFile() throws IOException {
    File targetDir = new File(path);
    if (!targetDir.exists()) {
        targetDir.mkdir();
    }
    File checkoutFile = new File(path, fileName + ".txt");
    if(!checkoutFile.exists()) {
        checkoutFile.createNewFile();
    }
    writer = new BufferedWriter(new OutputStreamWriter(
        new FileOutputStream(checkoutFile, true)));
}

public void checkout(ShoppingCart cart) throws IOException {
    writer.write(new Date() + "\t" +cart.getItems() + "\r\n");
    writer.flush();
}

public void closeFile() throws IOException {
    writer.close();
}
}

```

In the `Cashier` class, the `openFile()` method first verifies if the target directory and the file to write the data exists. It then opens the text file in the specified system path and assigns it to the `writer` field. Then each time the `checkout()` method is called, the date and cart items are appended to the text file. Finally, the `closeFile()` method closes the file to release its system resources.

Next, let's explore how this bean definition has to be set up in a Java config class, in order to execute the `openFile()` method just before the bean is created and the `closeFile()` method just before it's destroyed.

```

@Configuration
public class ShopConfiguration {
    ...
    @Bean(initMethod="openFile",destroyMethod="closeFile")
    public Cashier cashier() {
        Cashier c1 = new Cashier();
        c1.setFileName("checkout");
        c1.setPath("c:/Windows/Temp/cashier");
        return c1;
    }
}

```

Note The path value is set to `c:/Windows/Temp/` because it's a Windows world-writeable directory. If you use another path, ensure it's accessible by the user that executes the application.

Notice the POJO's initialization and destruction tasks are defined with the `initMethod` and `destroyMethod` attributes of a `@Bean` annotation. With these two attributes set in the bean declaration, when the `Cashier` class is created it first triggers the `openFile()` method — verifying if the target directory and the file to write the data exist, as well as opening the file to append records — and when the bean is destroyed it triggers the `closeFile()` method —ensuring the file reference is closed to release system resources.

Define methods to run before POJO initialization and destruction with @PostConstruct and @PreDestroy

Another alternative if you'll define POJO instances outside a Java config class (e.g., with the @Component annotation) is to use the @PostConstruct and @PreDestroy annotations directly in the POJO class.

```

@Component
public class Cashier {

    @Value("checkout")
    private String fileName;
    @Value("c:/Windows/Temp/cashier")
    private String path;
    private BufferedWriter writer;

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }

    public void setPath(String path) {
        this.path = path;
    }

    @PostConstruct
    public void openFile() throws IOException {
        File targetDir = new File(path);
        if (!targetDir.exists()) {
            targetDir.mkdir();
        }
        File checkoutFile = new File(path, fileName + ".txt");
        if(!checkoutFile.exists()) {
            checkoutFile.createNewFile();
        }
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(checkoutFile, true)));
    }

    public void checkout(ShoppingCart cart) throws IOException {
        writer.write(new Date() + "\t" + cart.getItems() + "\r\n");
        writer.flush();
    }

    @PreDestroy
    public void closeFile() throws IOException {
        writer.close();
    }
}

```

The `@Component` annotation tells Spring to manage the POJO, just like it's been used in previous recipes. Two of the POJO fields' values are set with the `@Value` annotation, a concept that was also explored in a previous recipe.

The `openFile()` method is decorated with the `@PostConstruct` annotation, which tells Spring to execute the method right after a bean is constructed. The `closeFile()` method is decorated with the `@PreDestroy` annotation, which tells Spring to execute the method right before a bean is destroyed.

Define lazy initialization for POJOs with `@Lazy`

By default, Spring performs eager initialization on all POJOs. This means POJOs are initialized at startup. In certain circumstances though, it can be convenient to delay the POJO initialization process until a bean is required. Delaying the initialization is called 'lazy initialization'.

Lazy initialization helps limit resource consumption peaks at startup and save overall system resources. Lazy initialization can be particularly relevant for POJOs that perform heavyweight operations (e.g., network connections, file operations). To mark a bean with lazy initialization you decorate a bean with the `@Lazy` annotation.

```
package com.apress.springrecipes.shop;
...
import org.springframework.stereotype.Component;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.Lazy;

@Component
@Scope("prototype")
@Lazy
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();

    public void addItem(Product item) {
        items.add(item);
    }

    public List<Product> getItems() {
        return items;
    }
}
```

In the previous declaration because the POJO is decorated with the `@Lazy` annotation, if the POJO is never required by the application or referenced by another POJO, it's never instantiated.

Define initialization of POJOs before other POJOs with `@DependsOn`

As an application's POJOs grow, so does the number of POJO initializations. This can create race conditions if POJOs reference one another and are spread out in different Java Configuration classes. What happens if bean 'C' requires the logic in bean 'B' and bean 'F'? If bean 'C' is detected first and Spring hasn't initialized bean 'B' and bean 'F', you'll get an error which can be hard to detect.

To ensure that certain POJOs are initialized before other POJOs and to get a more descriptive error in case of a failed initialization process, Spring offers the `@DependsOn` annotation. The `@DependsOn` annotation ensures a given bean is initialized before another bean.

```

package com.apress.springrecipes.sequence.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.DependsOn;
import org.springframework.context.annotation.Configuration;
import com.apress.springrecipes.sequence.DatePrefixGenerator;
import com.apress.springrecipes.sequence.NumberPrefixGenerator;
import com.apress.springrecipes.sequence.SequenceGenerator;

@Configuration
public class SequenceConfiguration {
    @Bean
    @DependsOn("datePrefixGenerator")
    public SequenceGenerator sequenceGenerator() {
        SequenceGenerator sequence= new SequenceGenerator();
        sequence.setInitial(100000);
        sequence.setSuffix("A");
        return sequence;
    }
}

```

In the previous snippet, the declaration `@DependsOn("datePrefixGenerator")` ensures the `datePrefixGenerator` bean is created before the `sequenceGenerator` bean. The `@DependsOn` attribute also supports defining multiple dependency beans with a CSV list surrounded by {} (e.g., `@DependsOn({"datePrefixGenerator", numberPrefixGenerator, randomPrefixGenerator"})`)

3-9. Create Post Processors to validate and modify POJOs

Problem

You want to apply tasks to all bean instances or specific types of instances during construction to validate or modify bean properties according to particular criteria.

Solution

A bean post processor allows bean processing before and after the initialization callback method (i.e., the one assigned to the `initmethod` attribute of the `@Bean` annotation or the method decorated with the `@PostConstruct` annotation). The main characteristic of a bean post processor is that it processes all the bean instances in the IoC container, not just a single bean instance. Typically, bean post processors are used to check the validity of bean properties, alter bean properties according to particular criteria, or apply certain tasks to all bean instances.

Spring also supports the `@Required` annotation which is backed by the built-in Spring post-processor `RequiredAnnotationBeanPostProcessor`. The `RequiredAnnotationBeanPostProcessor` bean post processor checks if all the bean properties with the `@Required` annotation have been set.

How It Works

Suppose you want to audit the creation of every bean. You may want to do this to debug an application, verify the properties of every bean or some other scenario. A bean post processor is an ideal choice to implement this feature, because you don't have to modify any pre-existing POJO code.

Create POJO to process every bean instance

To write a bean post processor a class has to implement `BeanPostProcessor`. When Spring detects a bean that implements this class, it applies the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods to all bean instances managed by Spring. You can implement any logic you wish in these methods, to either inspect, modify, or verify the status of a bean.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.stereotype.Component;

@Component
public class AuditCheckBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        System.out.println("In AuditCheckBeanPostProcessor.postProcessBeforeInitialization,
processing bean type: " + bean.getClass());
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        return bean;
    }
}
```

Notice the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods must return the original bean instance even if you don't do anything in the method.

To register a bean post processor in an application context, just annotate the class with the `@Component` annotation. The application context is able to detect which bean implements the `BeanPostProcessor` interface and register it to process all other bean instances in the container.

Create POJO to process selected bean instances

During bean construction, the Spring IoC container passes all the bean instances to the bean post processor one by one. This means if you only want to apply a bean post processor to certain types of beans, you must filter the beans by checking their instance type. This allows you to apply logic more selectively across beans.

Suppose you now want to apply a bean post processor but just to `Product` bean instances. The following example is another bean post processor that does just this.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

import org.springframework.stereotype.Component;
```

```

@Component
public class ProductCheckBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof Product) {
            String productName = ((Product) bean).getName();
            System.out.println("In ProductCheckBeanPostProcessor.postProcessBeforeInitialization,
processing Product: " + productName);
        }
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof Product) {
            String productName = ((Product) bean).getName();
            System.out.println("In ProductCheckBeanPostProcessor.postProcessAfterInitialization,
processing Product: " + productName);
        }
        return bean;
    }
}

```

Both the `postProcessBeforeInitialization()` and `postProcessAfterInitialization()` methods must return an instance of the bean being processed. However, this also means you can even replace the original bean instance with a brand-new instance in your bean post processor.

Verify POJO Properties with the `@Required` Annotation

In certain cases, it may be necessary to check if particular properties have been set. Instead of creating of custom post-constructor to verify the particular properties of a bean, it's possible to decorate a property with the `@Required` annotation. The `@Required` annotation provides access to the `@RequiredAnnotationBeanPostProcessor` class — a Spring bean post processor that can check if certain bean properties have been set. Note that this processor can only check if the properties have been set, but can't check if their value is null or something else.

Suppose that both the `prefixGenerator` and `suffix` properties are required for a sequence generator. You can annotate their setter methods with `@Required`.

```

package com.apress.springrecipes.sequence;

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

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Required
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}

```

```

@Required
public void setSuffix(String suffix) {
    this.suffix = suffix;
}
...
}

```

To ask Spring to check if these properties have been set you just need to enable scanning so Spring can detect and enforce the @Required annotation . If any properties with @Required have not been set, a BeanInitializationException error is thrown.

3-10. Create POJOs with a factory (Static method, Instance method, Spring's FactoryBean)

Problem

You want to create a POJO instance in the Spring IoC container by invoking a static factory method or instance factory method. The purpose of this approach is to encapsulate the object-creation process in either a static method or in a method of another object instance, respectively. The client who requests an object can simply make a call to this method without knowing about the creation details.

You want to create a POJO instance in the Spring IoC container using Spring's factory bean. A factory bean is a bean that serves as a factory for creating other beans within the IoC container. Conceptually, a factory bean is very similar to a factory method, but it's a Spring-specific bean that can be identified by the Spring IoC container during bean construction.

Solution

To create a POJO by invoking a static factory inside a @Bean definition of a Java configuration class, you use standard Java syntax to call the static factory method. To create a POJO by invoking an instance factory method inside a @Bean definition of a Java configuration class, you create a POJO to instantiate the factory values and another POJO to act as a façade to access the factory.

As a convenience, Spring provides an abstract template class called `AbstractFactoryBean` to extend Spring's `FactoryBean` interface.

How It Works

Create POJOs by invoking a static factory method

For example, you can write the following `createProduct` static factory method to create a product from a predefined product ID. According to the product ID, this method decides which concrete product class to instantiate. If there is no product matching this ID, Spring throws an `IllegalArgumentException`.

```

package com.apress.springrecipes.shop;

public class ProductCreator {

    public static Product createProduct(String productId) {
        if ("aaa".equals(productId)) {
            return new Battery("AAA", 2.5);
        } else if ("cdrw".equals(productId)) {

```

```

        return new Disc("CD-RW", 1.5);
    } else if ("dvdrw".equals(productId)) {
        return new Disc("DVD-RW", 3.0);
    }
    throw new IllegalArgumentException("Unknown product");
}
}

```

To create a POJO with a static factory method inside a @Bean definition of a Java Configuration class you use regular Java syntax to call the factory method.

```

@Configuration
public class ShopConfiguration {
    @Bean
    public Product aaa() {
        return ProductCreator.createProduct("aaa");
    }
    @Bean
    public Product cdrw() {
        return ProductCreator.createProduct("cdrw");
    }
    @Bean
    public Product dvdrw() {
        return ProductCreator.createProduct("dvdrw");
    }
}

```

Create POJOs by invoking an instance factory method

For example, you can write the following ProductCreator class by using a configurable map to store predefined products. The createProduct() instance factory method finds a product by looking up the supplied productId in the map. If there is no product matching this ID, it will throw an IllegalArgumentException.

```

package com.apress.springrecipes.shop;
...
public class ProductCreator {

    private Map<String, Product> products;

    public void setProducts(Map<String, Product> products) {
        this.products = products;
    }

    public Product createProduct(String productId) {
        Product product = products.get(productId);
        if (product != null) {
            return product;
        }
        throw new IllegalArgumentException("Unknown product");
    }
}

```

To create products from this `ProductCreator`, you first declare a `@Bean` to instantiate the factory values. Next, you declare a second bean to act as a façade to access the factory. Finally, you can call the factory and execute the `createProduct()` method in to instantiate other beans.

```
@Configuration
public class ShopConfiguration {
    @Bean
    public ProductCreator productCreatorFactoryBean() {
        ProductCreator factory = new ProductCreator();
        Map<String, Product> products = new HashMap();
        products.put("aaa", new Battery("AAA", 2.5));
        products.put("cdrw", new Disc("CD-RW", 1.5));
        products.put("dvdrw", new Disc("DVD-RW", 3.0));
        factory.setProducts(products);
        return factory;
    }

    @Bean
    public ProductCreator productCreatorFactory() {
        return (ProductCreator) productCreatorFactoryBean();
    }

    @Bean
    public Product aaa() {
        return productCreatorFactory().createProduct("aaa");
    }
    @Bean
    public Product cdrw() {
        return productCreatorFactory().createProduct("cdrw");
    }
    @Bean
    public Product dvdrw() {
        return productCreatorFactory().createProduct("dvdrw");
    }
}
```

Create POJOs using Spring's factory bean

Although you'll seldom have to write custom factory beans, you may find it helpful to understand their internal mechanisms through an example. For example, you can write a factory bean for creating a product with a discount applied to the price. It accepts a `product` property and a `discount` property to apply the discount to the product and return it as a new bean.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.factory.config.AbstractFactoryBean;

public class DiscountFactoryBean extends AbstractFactoryBean {

    private Product product;
    private double discount;
```

```

public void setProduct(Product product) {
    this.product = product;
}

public void setDiscount(double discount) {
    this.discount = discount;
}

public Class<?> getObjectType() {
    return product.getClass();
}

protected Object createInstance() throws Exception {
    product.setPrice(product.getPrice() * (1 - discount));
    return product;
}
}

```

By extending the `AbstractFactoryBean` class, the factory bean can simply override the `createInstance()` method to create the target bean instance. In addition, you have to return the target bean's type in the `getObjectType()` method for the auto-wiring feature to work properly.

Next, you can declare product instances using a regular `@Bean` annotation to apply `DiscountFactoryBean`.

```

@Configuration
public class ShopConfiguration {
    @Bean
    public Battery aaa() {
        Battery aaa = new Battery("AAA",2.5);
        return aaa;
    }

    @Bean
    public Disc cdrw() {
        Disc aaa = new Disc("CD-RW",1.5);
        return aaa;
    }

    @Bean
    public Disc dvdrw() {
        Disc aaa = new Disc("DVD-RW",3.0);
        return aaa;
    }

    @Bean
    public DiscountFactoryBean discountFactoryBeanAAA() {
        DiscountFactoryBean factory = new DiscountFactoryBean();
        factory.setProduct(aaa());
        factory.setDiscount(0.2);
        return factory;
    }
}

```

```

@Bean
public DiscountFactoryBean discountFactoryBeanCDRW() {
    DiscountFactoryBean factory = new DiscountFactoryBean();
    factory.setProduct(cdrw());
    factory.setDiscount(0.1);
    return factory;
}

@Bean
public DiscountFactoryBean discountFactoryBeanDVDRW() {
    DiscountFactoryBean factory = new DiscountFactoryBean();
    factory.setProduct(dvdrw());
    factory.setDiscount(0.1);
    return factory;
}
}

```

3-11. Use Spring Environments and Profiles to load different sets of POJOs

Problem

You want to use the same set of POJO instances or beans but with different instantiation values for different application scenarios (e.g., ‘production’, ‘development’ & ‘testing’).

Solution

Create multiple Java Configuration classes and group POJOs instances or beans into each of these classes. Assign a profile name to the Java Configuration class with the @Profile annotation based on the purpose of the group. Get the environment for an application’s context and set the profile to load a specific group of POJOs.

How it works

POJO instantiation values can vary depending on different application scenarios. For example, a common scenario can occur when an application goes from development, to testing and on to production. In each of these scenarios, the properties for certain beans can vary slightly to accommodate environment changes (e.g., database username/password, file paths, etc.).

You can create multiple Java Configuration classes each with different POJO (e.g., ShopConfigurationGlobal, ShopConfigurationStr and ShopConfigurationSumWin). And in the application context only load a given Configuration class file based on the scenario.

Create Java Configuration class with @Profile annotation

Let's create a multiple Java Configuration class with a @Profile annotation for the shopping application presented in previous recipes.

```
package com.apress.springrecipes.shop.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("global")
public class ShopConfigurationGlobal {

    @Bean(initMethod="openFile",destroyMethod="closeFile")
    public Cashier cashier() {
        Cashier c1 = new Cashier();
        c1.setFileName("checkout");
        c1.setPath("c:/Windows/Temp/cashier");
        return c1;
    }
}

package com.apress.springrecipes.shop.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
...
@Configuration
@Profile({"summer","winter"})
public class ShopConfigurationSumWin {

    @Bean
    public Product aaa() {
        Battery p1 = new Battery();
        p1.setName("AAA");
        p1.setPrice(2.0);
        p1.setRechargeable(true);
        return p1;
    }
    @Bean
    public Product cdrw() {
        Disc p2 = new Disc("CD-RW",1.0);
        p2.setCapacity(700);
        return p2;
    }
    @Bean
}
```

```

public Product dvdrw() {
    Disc p2 = new Disc("DVD-RW",2.5);
    p2.setCapacity(700);
    return p2;
}

}

```

The `@Profile` annotation decorates the entire Java configuration class, so all the `@Bean` instances belong to the same profile. To assign a `@Profile` name you just place the name inside `""`. Notice it's also possible to assign multiple `@Profile` names using a CSV syntax surrounded by `{}` (e.g., `{"summer", "winter"}`).

Load Profile into Environment

To load the beans from a certain profile into an application, you need to activate a profile. You can load multiple profiles at a time and it's also possible to load profiles programmatically, through a Java runtime flag or even as an initialization parameter of a WAR file.

To load profiles programmatically (i.e., via the application context) you get the context environment from where you can load profiles via the `setActiveProfiles()` method.

```

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext();

context.getEnvironment().setActiveProfiles("global","winter");
context.scan("com.apress.springrecipes.shop");
context.refresh();
/**At this point beans declared inside the package com.apress.springrecipes.shop.config with the
global and summer profiles are accessible*/

```

It's also possible to indicate which Spring profile to load via a Java runtime flag. In this manner, you can pass the following runtime flag to load all beans that belong to the global and winter profiles: `Dspring.profiles.active="global, winter"`

Finally, a Spring profile can also be setup directly inside a WAR's `web.xml` configuration file. This scenario is relevant for Spring web applications.

```

<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>winter</param-value>
  </init-param>
</servlet>

```

Set default Profile

To avoid the possibility of errors because no profiles are loaded into an application, you can define default profiles. Default profiles are only used when Spring can't detect any active profiles — defined using any of the previous methods: programmatically, via a Java runtime flag or web application initialization parameter.

To setup default profiles, you can also use any of the three methods to setup active profiles. Programmatically you use the method `setDefaultProfiles()` instead of `setActiveProfiles()`, and via a Java runtime flag or web application initialization parameter you can use the `spring.profiles.default` parameter instead of `spring.profiles.active`.

3-12. Aspect Orientated programming with Annotations

Problem

You want to use aspect orientated programming with Spring and annotations.

Solution

You define an aspect by decorating a Java class with the `@Aspect` annotation. Each of the methods in a class can become an advice with another annotation. You can use five types of advice annotations: `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, and `@Around`.

To enable annotation support in the Spring IoC container, you also have to define an empty XML element to activate AspectJ in your bean configuration file: `<aop:aspectj-autoproxy>`. For cases in which interfaces are not available or not used in an application's design, it's possible to create proxies by relying on CGLIB. To enable CGLIB, you need to set the attribute `<proxy-target-class=true>` in `<aop:aspectj-autoproxy>`.

How It Works

To support aspect orientated programming with annotations, Spring uses the same annotations as AspectJ, using a library supplied by AspectJ for pointcut parsing and matching. Although the AOP runtime is still pure Spring AOP and there is no dependency on the AspectJ compiler or weaver.

To illustrate the enablement of aspect orientated programming with annotations, we'll use the following calculator interfaces to define a set of sample POJOs:

```
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
    public double mul(double a, double b);
    public double div(double a, double b);
}

package com.apress.springrecipes.calculator;

public interface UnitCalculator {

    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}
```

Next, let's create POJO classes for each interface with `println` statements to know when each method is executed:

```
package com.apress.springrecipes.calculator;

import org.springframework.stereotype.Component;

@Component("arithmeticCalculator")
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }

    public double mul(double a, double b) {
        double result = a * b;
        System.out.println(a + " * " + b + " = " + result);
        return result;
    }

    public double div(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division by zero");
        }
        double result = a / b;
        System.out.println(a + " / " + b + " = " + result);
        return result;
    }
}

package com.apress.springrecipes.calculator;

import org.springframework.stereotype.Component;

@Component("unitCalculator")
public class UnitCalculatorImpl implements UnitCalculator {

    public double kilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogram = " + pound + " pound");
        return pound;
    }
}
```

```

public double kilometerToMile(double kilometer) {
    double mile = kilometer * 0.62;
    System.out.println(kilometer + " kilometer = " + mile + " mile");
    return mile;
}
}

```

Note that each POJO implementation is decorated with the `@Component` annotation to create bean instances.

Declare Aspects, Advices and Pointcuts

An aspect is a Java class that modularizes a set of concerns (e.g., logging or transaction management) that cuts across multiple types and objects. Java classes that modularize such concerns are decorated with the `@Aspect` annotation.

In AOP terminology, aspects are also complemented by advices, which in themselves have pointcuts. An advice is a simple Java method with one of the advice annotations. AspectJ supports five types of advice annotations: `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, and `@Around`. Whereas a pointcut is an expression that looks for types and objects on which to apply the aspect's advices.

Aspect with `@Before` advice

To create a before advice to handle crosscutting concerns before particular program execution points, you use the `@Before` annotation and include the pointcut expression as the annotation value.

```

package com.apress.springrecipes.calculator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());

    @Before("execution(* ArithmeticCalculator.add(..))")
    public void logBefore() {
        log.info("The method add() begins");
    }
}

```

This pointcut expression matches the `add()` method execution of the `ArithmeticCalculator` interface. The preceding wildcard in this expression matches any modifier (public, protected, and private) and any return type. The two dots in the argument list match any number of arguments.

Note For the previous aspect to work (i.e., Output its message) you need to setup logging. Specifically create a log4j.properties file with configuration properties like the following.

```
log4j.rootLogger=INFO, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

The @Aspect annotation is not sufficient for autodetection in the classpath. Therefore you need to add a separate @Component annotation for the POJO to be detected.

Next, you create an XML Spring configuration to scan all POJOs, including the POJO calculator implementation and aspect, as well as include the <aop:aspectj-autoproxy> tag.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.2.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.2.xsd">

    <aop:aspectj-autoproxy/>
    <context:component-scan base-package="com.apress.springrecipes.calculator"/>
</beans>
```

And as the last step you can test the aspect with the following Main class:

```
package com.apress.springrecipes.calculator;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);
    }
}
```

```

        UnitCalculator unitCalculator =
        (UnitCalculator) context.getBean("unitCalculator");
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}

```

The execution points matched by a pointcut are called join points. In this term, a pointcut is an expression to match a set of join points, while an advice is the action to take at a particular join point.

For your advice to access the detail of the current join point, you can declare an argument of type JoinPoint in your advice method. Then, you can get access to join point details such as the method name and argument values. Now, you can expand your pointcut to match all methods by changing the class name and method name to wildcards.

```

package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
@Component
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
    }
}

```

Aspect with @After advice

An after advice is executed after a join point finishes, whenever it returns a result or throws an exception abnormally. The following after advice logs the calculator method ending.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}

```

Aspect with @AfterReturning advice

An after advice is executed regardless of whether a join point returns normally or throws an exception. If you would like to perform logging only when a join point returns, you should replace the after advice with an after returning advice.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends");
    }
}
```

In an after returning advice, you can get access to the return value of a join point by adding a returning attribute to the @AfterReturning annotation. The value of this attribute should be the argument name of this advice method for the return value to pass in. Then, you have to add an argument to the advice method signature with this name. At runtime, Spring AOP will pass in the return value through this argument. Also note that the original pointcut expression needs to be presented in the pointcut attribute instead.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning(
        pointcut = "execution(* *.*(..))",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() ends with " + result);
    }
}
```

Aspect with @AfterThrowing advice

An after throwing advice is executed only when an exception is thrown by a join point.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing("execution(* *.*(..))")
    public void logAfterThrowing(JoinPoint joinPoint) {
        log.error("An exception has been thrown in "
            + joinPoint.getSignature().getName() + "()");
    }
}

```

Similarly, the exception thrown by the join point can be accessed by adding a throwing attribute to the @AfterThrowing annotation. The type Throwable is the superclass of all errors and exceptions in the Java language. So, the following advice will catch any of the errors and exceptions thrown by the join points:

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.error("An exception " + e + " has been thrown in "
            + joinPoint.getSignature().getName() + "()");
    }
}

```

However, if you are interested in one particular type of exception only, you can declare it as the argument type of the exception. Then your advice will be executed only when exceptions of compatible type (i.e., this type and its subtypes) are thrown.

```

package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

```

```

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint,
        IllegalArgumentException e) {
        log.error("Illegal argument " + Arrays.toString(joinPoint.getArgs())
            + " in " + joinPoint.getSignature().getName() + "()");
    }
}

```

Aspect with @Around advice

The last type of advice is an around advice. It is the most powerful of all the advice types. It gains full control of a join point, so you can combine all the actions of the preceding advices into one single advice. You can even control when, and whether, to proceed with the original join point execution.

The following around advice is the combination of the before, after returning, and after throwing advices you created before. Note that for an around advice, the argument type of the join point must be ProceedingJoinPoint. It's a subinterface of JoinPoint that allows you to control when to proceed with the original join point.

```

package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Around("execution(* *.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("The method " + joinPoint.getSignature().getName()
            + "() begins with " + Arrays.toString(joinPoint.getArgs()));
        try {
            Object result = joinPoint.proceed();
            log.info("The method " + joinPoint.getSignature().getName()
                + "() ends with " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Illegal argument "
                + Arrays.toString(joinPoint.getArgs()) + " in "
                + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
}

```

The around advice type is very powerful and flexible in that you can even alter the original argument values and change the final return value. You must use this type of advice with great care, as the call to proceed with the original join point may easily be forgotten.

Tip A common rule for choosing an advice type is to use the least powerful one that can satisfy your requirements.

3-13. Accessing the Join Point Information

Problem

In AOP, an advice is applied to different program execution points, which are called join points. For an advice to take the correct action, it often requires detailed information about join points.

Solution

An advice can access the current join point information by declaring an argument of type `org.aspectj.lang.JoinPoint` in the advice method signature.

How It Works

For example, you can access the join point information through the following advice. The information includes the join point kind (only method-execution in Spring AOP), the method signature (declaring type and method name), and the argument values, as well as the target object and proxy object.

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logJoinPoint(JoinPoint joinPoint) {
        log.info("Join point kind : "
            + joinPoint.getKind());
        log.info("Signature declaring type : "
            + joinPoint.getSignature().getDeclaringTypeName());
        log.info("Signature name : "
            + joinPoint.getSignature().getName());
        log.info("Arguments : "
            + Arrays.toString(joinPoint.getArgs()));
    }
}
```

```

        log.info("Target class : "
            + joinPoint.getTarget().getClass().getName());
        log.info("This class : "
            + joinPoint.getThis().getClass().getName());
    }
}

```

The original bean that was wrapped by a proxy is called the target object, while the proxy object is called the this object. They can be accessed by the join point's `getTarget()` and `getThis()` methods. From the following outputs, you can see that the classes of these two objects are not the same:

```

Join point kind : method-execution
Signature declaring type : com.apress.springrecipes.calculator.ArithmetricCalculator
Signature name : add
Arguments : [1.0, 2.0]
Target class : com.apress.springrecipes.calculator.ArithmetricCalculatorImpl
This class : com.sun.proxy.$Proxy6

```

3-14. Specifying Aspect Precedence with the @Order annotation

Problem

When there's more than one aspect applied to the same join point, the precedence of the aspects is undefined unless you have explicitly specified it.

Solution

The precedence of aspects can be specified either by implementing the `Ordered` interface or by using the `@Order` annotation.

How It Works

Suppose you have written another aspect to validate the calculator arguments. There's only one before advice in this aspect.

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

import org.springframework.stereotype.Component;

@Aspect
@Component
public class CalculatorValidationAspect {

```

```

@Before("execution(* *.*(double, double))")
public void validateBefore(JoinPoint joinPoint) {
    for (Object arg : joinPoint.getArgs()) {
        validate((Double) arg);
    }
}

private void validate(double a) {
    if (a < 0) {
        throw new IllegalArgumentException("Positive numbers only");
    }
}
}

```

If you apply this aspect and the previous you can't guarantee which one is applied first. To guarantee that one aspect is applied before another you need to specify precedence. To specify precedence, you have to make both aspects implement the `Ordered` interface or use the `@Order` annotation.

If you decide to implement the `Ordered` interface, the lower value returned by the `getOrder` method represents higher priority. So, if you prefer the validation aspect to be applied first, it should return a value lower than the logging aspect.

```

package com.apress.springrecipes.calculator;
...
import org.springframework.core.Ordered;

@Aspect
@Component
public class CalculatorValidationAspect implements Ordered {
    ...
    public int getOrder() {
        return 0;
    }
}

package com.apress.springrecipes.calculator;
...
import org.springframework.core.Ordered;

@Aspect
@Component
public class CalculatorLoggingAspect implements Ordered {
    ...
    public int getOrder() {
        return 1;
    }
}

```

Another way to specify precedence is through the `@Order` annotation. The order number should be presented in the annotation value.

```

package com.apress.springrecipes.calculator;
...
import org.springframework.core.annotation.Order;

@Aspect
@Component
@Order(0)
public class CalculatorValidationAspect {
    ...
}

package com.apress.springrecipes.calculator;
...
import org.springframework.core.annotation.Order;

@Aspect
@Component
@Order(1)
public class CalculatorLoggingAspect {
    ...
}

```

3-15. Reuse Aspect Pointcut Definitions

Problem

When writing aspects, you can directly embed a pointcut expression in an advice annotation. You want to use the same pointcut expression in multiple advices without embedding it multiple times.

Solution

You can use the `@Pointcut` annotation to define a pointcut independently to be reused in multiple advices.

How It Works

In an aspect, a pointcut can be declared as a simple method with the `@Pointcut` annotation. The method body of a pointcut is usually empty, as it is unreasonable to mix a pointcut definition with application logic. The access modifier of a pointcut method controls the visibility of this pointcut as well. Other advices can refer to this pointcut by the method name.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Pointcut;

@Aspect
@Component
public class CalculatorLoggingAspect {
    ...
    @Pointcut("execution(* *.*(..))")
    private void loggingOperation() {}
}

```

```

@Before("loggingOperation()")
public void logBefore(JoinPoint joinPoint) {
    ...
}

@AfterReturning(
    pointcut = "loggingOperation()", 
    returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    ...
}

@AfterThrowing(
    pointcut = "loggingOperation()", 
    throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
    ...
}

@Around("loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
}
}

```

Usually, if your pointcuts are shared between multiple aspects, it is better to centralize them in a common class. In this case, they must be declared as public.

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {

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

```

When you refer to this pointcut, you have to include the class name as well. If the class is not located in the same package as the aspect, you have to include the package name also.

```

package com.apress.springrecipes.calculator;
...
@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("CalculatorPointcuts.loggingOperation()")
    public void logBefore(JoinPoint joinPoint) {
        ...
    }

    @AfterReturning(
        pointcut = "CalculatorPointcuts.loggingOperation()",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        ...
    }

    @AfterThrowing(
        pointcut = "CalculatorPointcuts.loggingOperation()",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, IllegalArgumentException e) {
        ...
    }

    @Around("CalculatorPointcuts.loggingOperation()")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        ...
    }
}

```

3-16. Writing AspectJ Pointcut Expressions

Problem

Crosscutting concerns can happen at different program execution points, which are called join points. Because of the variety of join points, you need a powerful expression language to help match them.

Solution

The AspectJ pointcut language is a powerful expression language that can match various kinds of join points. However, Spring AOP only supports method execution join points for beans declared in its IoC container. For this reason, only those pointcut expressions supported by Spring AOP are presented in this recipe. For a full description of the AspectJ pointcut language, please refer to the AspectJ programming guide available on AspectJ's web site (<http://www.eclipse.org/aspectj/>).

Spring AOP makes use of the AspectJ pointcut language for its pointcut definition and interprets the pointcut expressions at runtime by using a library provided by AspectJ.

When writing AspectJ pointcut expressions for Spring AOP, you must keep in mind that Spring AOP only supports method execution join points for the beans in its IoC container. If you use a pointcut expression out of this scope, an `IllegalArgumentException` is thrown.

How It Works

Method Signature Patterns

The most typical pointcut expressions are used to match a number of methods by their signatures. For example, the following pointcut expression matches all of the methods declared in the `ArithmeticCalculator` interface. The initial wildcard matches methods with any modifier (public, protected, and private) and any return type. The two dots in the argument list match any number of arguments.

```
execution(* com.apress.springrecipes.calculator.ArithmeticCalculator.*(..))
```

You can omit the package name if the target class or interface is located in the same package as the aspect.

```
execution(* ArithmeticCalculator.*(..))
```

The following pointcut expression matches all the public methods declared in the `ArithmeticCalculator` interface:

```
execution(public * ArithmeticCalculator.*(..))
```

You can also restrict the method return type. For example, the following pointcut matches the methods that return a double number:

```
execution(public double ArithmeticCalculator.*(..))
```

The argument list of the methods can also be restricted. For example, the following pointcut matches the methods whose first argument is of primitive double type. The two dots then match any number of followed arguments.

```
execution(public double ArithmeticCalculator.*(double, ..))
```

Or, you can specify all the argument types in the method signature for the pointcut to match.

```
execution(public double ArithmeticCalculator.*(double, double))
```

Although the AspectJ pointcut language is powerful in matching various join points, sometimes, you may not be able to find any common characteristics (e.g., modifiers, return types, method name patterns, or arguments) for the methods you would like to match. In such cases, you can consider providing a custom annotation for them. For instance, you can define the following marker annotation. This annotation can be applied to both method level and type level.

```
package com.apress.springrecipes.calculator;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target( { ElementType.METHOD, ElementType.TYPE } )
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LoggingRequired {
}
```

Next, you can annotate all methods that require logging with this annotation or the class itself to apply the behavior to all methods. Note that the annotations must be added to the implementation class but not the interface, as they will not be inherited.

```
package com.apress.springrecipes.calculator;

@LoggingRequired
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        ...
    }

    public double sub(double a, double b) {
        ...
    }

    ...
}
```

Then you can write a pointcut expression to match a class or methods with the `@LoggingRequired` annotation using the `annotation` keyword on `@Pointcut` annotation.

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {

    @Pointcut("annotation(com.apress.springrecipes.calculator.LoggingRequired)")
    public void loggingOperation() {}

}
```

Type Signature Patterns

Another kind of pointcut expressions matches all join points within certain types. When applied to Spring AOP, the scope of these pointcuts will be narrowed to matching all method executions within the types. For example, the following pointcut matches all the method execution join points within the `com.apress.springrecipes.calculator` package:

```
within(com.apress.springrecipes.calculator.*)
```

To match the join points within a package and its subpackage, you have to add one more dot before the wildcard.

```
within(com.apress.springrecipes.calculator..*)
```

The following pointcut expression matches the method execution join points within a particular class:

```
within(com.apress.springrecipes.calculator.ArithmeticCalculatorImpl)
```

Again, if the target class is located in the same package as this aspect, the package name can be omitted.

```
within(ArithmeticCalculatorImpl)
```

You can match the method execution join points within all classes that implement the `ArithmeticCalculator` interface by adding a plus symbol.

```
within(ArithmeticCalculator+)
```

The custom annotation `@LoggingRequired` can be applied to the class or method level as illustrated previously.

```
package com.apress.springrecipes.calculator;

@LoggingRequired
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
    ...
}
```

And then you can match the join points within the classes or methods that have been annotated with `@LoggingRequired` using the `within` keyword on `@Pointcut` annotation.

```
@Pointcut("within(com.apress.springrecipes.calculator.LoggingRequired)")
public void loggingOperation() {}
```

Combining Pointcut Expressions

In AspectJ, pointcut expressions can be combined with the operators `&&` (and), `||` (or), and `!` (not). For example, the following pointcut matches the join points within classes that implement either the `ArithmeticCalculator` or `UnitCalculator` interface:

```
within(ArithmeticCalculator+) || within(UnitCalculator+)
```

The operands of these operators can be any pointcut expressions or references to other pointcuts.

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {

    @Pointcut("within(ArithmeticCalculator+)")
    public void arithmeticOperation() {}

    @Pointcut("within(UnitCalculator+)")
    public void unitOperation() {}

    @Pointcut("arithmeticOperation() || unitOperation()")
    public void loggingOperation()
}
```

Declaring Pointcut Parameters

One way to access join point information is by reflection (i.e., via an argument of type `org.aspectj.lang.JoinPoint` in the advice method). Besides, you can access join point information in a declarative way by using some kinds of special pointcut expressions. For example, the expressions `target()` and `args()` capture the target object and argument values of the current join point and expose them as pointcut parameters. These parameters are passed to your advice method via arguments of the same name.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..)) && target(target) && args(a,b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Target class : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}
```

When declaring an independent pointcut that exposes parameters, you have to include them in the argument list of the pointcut method as well.

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {
    ...
    @Pointcut("execution(* *.*(..)) && target(target) && args(a,b)")
    public void parameterPointcut(Object target, double a, double b) {}
}
```

Any advice that refers to this parameterized pointcut can access the pointcut parameters via method arguments of the same name.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
```

```

@Aspect
public class CalculatorLoggingAspect {

    ...
    @Before("CalculatorPointcuts.parameterPointcut(target, a, b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Target class : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}

```

3-17. AOP introductions for POJOs

Problem

Sometimes, you may have a group of classes that share a common behavior. In OOP, they must extend the same base class or implement the same interface. This issue is actually a crosscutting concern that can be modularized with AOP.

In addition, the single inheritance mechanism of Java only allows a class to extend one base class at most. So, you cannot inherit behaviors from multiple implementation classes at the same time.

Solution

An Introduction is a special type of advice in AOP. It allows objects to implement an interface dynamically by providing an implementation class for that interface. It seems as if objects extend an implementation class at runtime.

Moreover, you are able to introduce multiple interfaces with multiple implementation classes to your objects at the same time. This can achieve the same effect as multiple inheritance.

How It Works

Suppose you have two interfaces, MaxCalculator and MinCalculator, to define the `max()` and `min()` operations.

```

package com.apress.springrecipes.calculator;

public interface MaxCalculator {

    public double max(double a, double b);
}

package com.apress.springrecipes.calculator;

public interface MinCalculator {

    public double min(double a, double b);
}

```

Then you have an implementation for each interface with `println` statements to let you know when the methods are executed.

```

package com.apress.springrecipes.calculator;

public class MaxCalculatorImpl implements MaxCalculator {

```

```

public double max(double a, double b) {
    double result = (a >= b) ? a : b;
    System.out.println("max(" + a + ", " + b + ") = " + result);
    return result;
}
package com.apress.springrecipes.calculator;

public class MinCalculatorImpl implements MinCalculator {

    public double min(double a, double b) {
        double result = (a <= b) ? a : b;
        System.out.println("min(" + a + ", " + b + ") = " + result);
        return result;
    }
}

```

Now, suppose you would like `ArithmeticCalculatorImpl` to perform the `max()` and `min()` calculation also. As the Java language supports single inheritance only, it is not possible for the `ArithmeticCalculatorImpl` class to extend both the `MaxCalculatorImpl` and `MinCalculatorImpl` classes at the same time. The only possible way is to extend either class (e.g., `MaxCalculatorImpl`) and implement another interface (e.g., `MinCalculator`), either by copying the implementation code or delegating the handling to the actual implementation class. In either case, you have to repeat the method declarations.

With an introduction, you can make `ArithmeticCalculatorImpl` dynamically implement both the `MaxCalculator` and `MinCalculator` interfaces by using the implementation classes `MaxCalculatorImpl` and `MinCalculatorImpl`. It has the same effect as multiple inheritance from `MaxCalculatorImpl` and `MinCalculatorImpl`. The idea behind an introduction is that you needn't modify the `ArithmeticCalculatorImpl` class to introduce new methods. That means you can introduce methods to your existing classes even without source code available.

Tip You may wonder how an introduction can do that in Spring AOP. The answer is a dynamic proxy. As you may recall, you can specify a group of interfaces for a dynamic proxy to implement. Introduction works by adding an interface (e.g., `MaxCalculator`) to the dynamic proxy. When the methods declared in this interface are called on the proxy object, the proxy will delegate the calls to the backend implementation class (e.g., `MaxCalculatorImpl`).

Introductions, like advices, must be declared within an aspect. You may create a new aspect or reuse an existing aspect for this purpose. In this aspect, you can declare an introduction by annotating an arbitrary field with the `@DeclareParents` annotation.

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

import org.springframework.stereotype.Component;

@Aspect
@Component
public class CalculatorIntroduction {

```

```

@DeclareParents(
    value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
    defaultImpl = MaxCalculatorImpl.class)
public MaxCalculator maxCalculator;

@DeclareParents(
    value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
    defaultImpl = MinCalculatorImpl.class)
public MinCalculator minCalculator;
}

```

The value attribute of the @DeclareParents annotation type indicates which classes are the targets for this introduction. The interface to introduce is determined by the type of the annotated field. Finally, the implementation class used for this new interface is specified in the defaultImpl attribute.

Through these two introductions, you can dynamically introduce a couple of interfaces to the ArithmeticCalculatorImpl class. Actually, you can specify an AspectJ type-matching expression in the value attribute of the @DeclareParents annotation to introduce an interface to multiple classes.

As you have introduced both the MaxCalculator and MinCalculator interfaces to your arithmetic calculator, you can cast it to the corresponding interface to perform the `max()` and `min()` calculations.

```

package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...
        MaxCalculator maxCalculator = (MaxCalculator) arithmeticCalculator;
        maxCalculator.max(1, 2);

        MinCalculator minCalculator = (MinCalculator) arithmeticCalculator;
        minCalculator.min(1, 2);
    }
}

```

3-18. Introduce states to your POJOs with AOP

Problem

Sometimes, you may want to add new states to a group of existing objects to keep track of their usage, such as the calling count, the last modified date, and so on. It should not be a problem if all the objects have the same base class. However, it's difficult to add such states to different classes if they are not in the same class hierarchy.

Solution

You can introduce a new interface to your objects with an implementation class that holds the state field. Then, you can write another advice to change the state according to a particular condition.

How It Works

Suppose you would like to keep track of the calling count of each calculator object. Since there is no field for storing the counter value in the original calculator classes, you need to introduce one with Spring AOP. First, let's create an interface for the operations of a counter.

```
package com.apress.springrecipes.calculator;

public interface Counter {
    public void increase();
    public int getCount();
}
```

Next, just write a simple implementation class for this interface. This class has a count field for storing the counter value.

```
package com.apress.springrecipes.calculator;

public class CounterImpl implements Counter {
    private int count;

    public void increase() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

To introduce the Counter interface to all your calculator objects with CounterImpl as the implementation, you can write the following introduction with a type-matching expression that matches all the calculator implementations:

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
@Component
public class CalculatorIntroduction {
    ...
    @DeclareParents(
        value = "com.apress.springrecipes.calculator.*CalculatorImpl",
        defaultImpl = CounterImpl.class)
    public Counter counter;
}
```

This introduction introduces CounterImpl to each of your calculator objects. However, it's still not enough to keep track of the calling count. You have to increase the counter value each time a calculator method is called. You can write an after advice for this purpose. Note that you must get this object but not the target object, as only the proxy object implements the Counter interface.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
@Component
public class CalculatorIntroduction {
    ...
    @After("execution(* com.apress.springrecipes.calculator.*Calculator.*(..))"
        + " && this(counter)")
    public void increaseCount(Counter counter) {
        counter.increase();
    }
}
```

In the Main class, you can output the counter value for each of the calculator objects by casting them into the Counter type.

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        ...

        Counter arithmeticCounter = (Counter) arithmeticCalculator;
        System.out.println(arithmeticCounter.getCount());

        Counter unitCounter = (Counter) unitCalculator;
        System.out.println(unitCounter.getCount());
    }
}
```

3-19. Load-Time Weaving AspectJ Aspects in Spring

Problem

The Spring AOP framework supports only limited types of AspectJ pointcuts and allows aspects to apply to beans declared in the IoC container. If you want to use additional pointcut types or apply your aspects to objects created outside the Spring IoC container, you have to use the AspectJ framework in your Spring application.

Solution

Weaving is the process of applying aspects to your target objects. With Spring AOP, weaving happens at runtime through dynamic proxies. In contrast, the AspectJ framework supports both compile-time and load-time weaving.

AspectJ compile-time weaving is done through a special AspectJ compiler called ajc. It can weave aspects into your Java source files and output woven binary class files. It can also weave aspects into your compiled class files or JAR files. This process is known as post-compile-time weaving. You can perform compile-time and post-compile-time weaving for your classes before declaring them in the Spring IoC container. Spring is not involved in the weaving process at all. For more information on compile-time and post-compile-time weaving, please refer to the AspectJ documentation.

AspectJ load-time weaving (also known as LTW) happens when the target classes are loaded into JVM by a class loader. For a class to be woven, a special class loader is required to enhance the bytecode of the target class. Both AspectJ and Spring provide load-time weavers to add load-time weaving capability to the class loader. You need only simple configurations to enable these load-time weavers.

How It Works

To understand the AspectJ load-time weaving process in a Spring application, let's consider a calculator for complex numbers. First, you create the `Complex` class to represent complex numbers. You define the `toString()` method for this class to convert a complex number into the string representation ($a + bi$).

```
package com.apress.springrecipes.calculator;

public class Complex {

    private int real;
    private int imaginary;

    public Complex(int real, int imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Getters and Setters
    ...

    public String toString() {
        return "(" + real + " + " + imaginary + "i)";
    }
}
```

Next, you define an interface for the operations on complex numbers. For simplicity's sake, only `add()` and `sub()` are supported.

```
package com.apress.springrecipes.calculator;

public interface ComplexCalculator {
    public Complex add(Complex a, Complex b);
    public Complex sub(Complex a, Complex b);
}
```

The implementation code for this interface is as follows. Each time, you return a new complex object as the result.

```
package com.apress.springrecipes.calculator;

import org.springframework.stereotype.Component;

@Component("complexCalculator")
public class ComplexCalculatorImpl implements ComplexCalculator {

    public Complex add(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() + b.getReal(),
            a.getImaginary() + b.getImaginary());
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public Complex sub(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() - b.getReal(),
            a.getImaginary() - b.getImaginary());
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }
}
```

Now, you can test this complex number calculator with the following code in the Main class:

```
package com.apress.springrecipes.calculator;
...
public class Main {

    public static void main(String[] args) {
        ...
        ComplexCalculator complexCalculator =
            (ComplexCalculator) context.getBean("complexCalculator");
        complexCalculator.add(new Complex(1, 2), new Complex(2, 3));
        complexCalculator.sub(new Complex(5, 8), new Complex(2, 3));
    }
}
```

So far, the complex calculator is working fine. However, you may want to improve the performance of the calculator by caching complex number objects. As caching is a well-known crosscutting concern, you can modularize it with an aspect.

```
package com.apress.springrecipes.calculator;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public ComplexCachingAspect() {
        cache = Collections.synchronizedMap(new HashMap<String, Complex>());
    }

    @Around("call(public Complex.new(int, int)) && args(a,b)")
    public Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b)
            throws Throwable {
        String key = a + "," + b;
        Complex complex = cache.get(key);
        if (complex == null) {
            System.out.println("Cache MISS for (" + key + ")");
            complex = (Complex) joinPoint.proceed();
            cache.put(key, complex);
        }
        else {
            System.out.println("Cache HIT for (" + key + ")");
        }
        return complex;
    }
}
```

In this aspect, you cache the complex objects in a map with their real and imaginary values as keys. For this map to be thread-safe, you should wrap it with a synchronized map. Then, the most suitable time to look up the cache is when a complex object is created by invoking the constructor. You use the AspectJ pointcut expression call to capture the join points of calling the `Complex(int, int)` constructor.

Next, you need an around advice to alter the return value. If a complex object of the same value is found in the cache, you return it to the caller directly. Otherwise, you proceed with the original constructor invocation to create a new complex object. Before you return it to the caller, you cache it in the map for subsequent usages.

The call pointcut is not supported by Spring AOP, so if you attempt to let Spring scan the pointcut annotation you'll get the error 'unsupported pointcut primitive call'.

Because this type of pointcut is not supported by Spring AOP, you have to use the AspectJ framework to apply this aspect. The configuration of the AspectJ framework is done through a file named `aop.xml` in the `META-INF` directory in the classpath root.

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
 "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
  <weaver>
    <include within="com.apress.springrecipes.calculator.*" />
  </weaver>

  <aspects>
    <aspect
      name="com.apress.springrecipes.calculator.ComplexCachingAspect" />
  </aspects>
</aspectj>
```

In this AspectJ configuration file, you have to specify the aspects and which classes you want your aspects to weave in. Here, you specify weaving `ComplexCachingAspect` into all the classes in the `com.apress.springrecipes.calculator` package.

Finally, to make this load-time weaving you need to run the application in one of two ways as described in the next sections.

Load-Time Weaving by the AspectJ Weaver

AspectJ provides a load-time weaving agent to enable load-time weaving. You need only to add a VM argument to the command that runs your application. Then your classes will get woven when they are loaded into the JVM.

```
java -javaagent:lib/aspectjweaver-1.7.2.jar -jar Recipe_3_19_i1-1.0.jar
```

If you run your application with the preceding argument, you will get the following output and cache status. The AspectJ agent advises all calls to the `Complex(int, int)` constructor.

```
Cache MISS for (1,2)
Cache MISS for (2,3)
Cache MISS for (3,5)
(1 + 2i) + (2 + 3i) = (3 + 5i)
Cache MISS for (5,8)
Cache HIT for (2,3)
Cache HIT for (3,5)
(5 + 8i) - (2 + 3i) = (3 + 5i)
```

Load-Time Weaving by Spring Load-Time Weaver

Spring has several load-time weavers for different runtime environments. To turn on a suitable load-time weaver for your Spring application, you need only to declare the empty XML element `<context:load-time-weaver>`. This element is defined in the context schema.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <context:load-time-weaver />
    ...
</beans>
```

Spring will be able to detect the most suitable load-time weaver for your runtime environment. Some Java EE application servers have class loaders that support the Spring load-time weaver mechanism, so there's no need to specify a Java agent in their startup commands.

However, for a simple Java application, you still require a weaving agent provided by Spring to enable load-time weaving. You have to specify the Spring agent in the VM argument of the startup command.

```
java -javaagent:c:lib/spring-instrument-3.2.0.jar -jar Recipe_3_19_iii-1.0.jar
```

However, if you run your application, you will get the following output and cache status.

```

Cache MISS for (3,5)
(1 + 2i) + (2 + 3i) = (3 + 5i)
Cache HIT for (3,5)
(5 + 8i) - (2 + 3i) = (3 + 5i)
```

This is because the Spring agent advises only the `Complex(int, int)` constructor calls made by beans declared in the Spring IoC container. As the complex operands are created in the `Main` class, the Spring agent will not advise their constructor calls.

3-20. Configuring AspectJ Aspects in Spring

Problem

Aspects used in the AspectJ framework are instantiated by the AspectJ framework itself. Therefore you have to retrieve the aspect instances from the AspectJ framework to configure them.

Solution

Each AspectJ aspect provides a static factory method called `aspectOf()`, which allows you to access the current aspect instance. In the Spring IoC container, you can declare a bean created by a factory method by specifying the `factory-method` attribute.

How It Works

For instance, you can allow the cache map of `ComplexCachingAspect` to be configured via a setter method and delete its instantiation from the constructor.

```

package com.apress.springrecipes.calculator;
...
import java.util.Collections;
import java.util.Map;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public void setCache(Map<String, Complex> cache) {
        this.cache = Collections.synchronizedMap(cache);
    }
    ...
}

```

To configure this property in the Spring IoC container you need to use XML configuration because there's no explicit factory annotation to call a factory method — other than creating a Java config class to setup the factory. So let's declare a bean in XML created by the factory method `aspectOf()`.

```

<bean class="com.apress.springrecipes.calculator.ComplexCachingAspect"
    factory-method="aspectOf">
    <property name="cache">
        <map>
            <entry key="2,3">
                <bean class="com.apress.springrecipes.calculator.Complex">
                    <constructor-arg value="2" />
                    <constructor-arg value="3" />
                </bean>
            </entry>

            <entry key="3,5">
                <bean class="com.apress.springrecipes.calculator.Complex">
                    <constructor-arg value="3" />
                    <constructor-arg value="5" />
                </bean>
            </entry>
        </map>
    </property>
</bean>

```

Tip You may wonder why your `ComplexCachingAspect` has a static factory method `aspectOf()` that you have not declared. This method is woven by AspectJ at load time to allow you to access the current aspect instance. So, if you are using Spring IDE, it may give you a warning because it cannot find this method in your class.

Finally, because the mechanism used by the static factory method `aspectOf()` is provided by AspectJ, to run the application you use AspectJ's weaver:

```
java -javaagent:lib/aspectjweaver-1.7.2.jar -jar Recipe_3_20-1.0.jar
```

3-21. Inject POJOs into Domain Objects with AOP Problem

Beans declared in the Spring IoC container can wire themselves to one another through Spring's dependency injection capability. However, objects created outside the Spring IoC container cannot wire themselves to Spring beans via configuration. You have to perform the wiring manually with programming code.

Solution

Objects created outside the Spring IoC container are usually domain objects. They are often created using the new operator or from the results of database queries.

To inject a Spring bean into domain objects created outside Spring, you need the help of AOP. Actually, the injection of Spring beans is also a kind of crosscutting concern. As the domain objects are not created by Spring, you cannot use Spring AOP for injection. Spring supplies an AspectJ aspect specialized for this purpose. You can enable this aspect in the AspectJ framework.

How It Works

Suppose you have a global formatter to format complex numbers. This formatter accepts a pattern for formatting and uses the standard `@Component` and `@Value` annotations to instantiate a POJO.

```
package com.apress.springrecipes.calculator;

@Component
public class ComplexFormatter {

    @Value("(a + bi)")
    private String pattern;

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }

    public String format(Complex complex) {
        return pattern.replaceAll("a", Integer.toString(complex.getReal()))
            .replaceAll("b", Integer.toString(complex.getImaginary()));
    }
}
```

In the `Complex` class, you want to use this formatter in the `toString()` method to convert a complex number into a string. It exposes a setter method for `ComplexFormatter`.

```
package com.apress.springrecipes.calculator;

public class Complex {
```

```

private int real;
private int imaginary;
...
private ComplexFormatter formatter;

public void setFormatter(ComplexFormatter formatter) {
    this.formatter = formatter;
}

public String toString() {
    return formatter.format(this);
}
}

```

However, because `Complex` objects are not instantiated by the Spring IoC container, they cannot be configured for dependency injection in the regular manner. Spring includes `AnnotationBeanConfigurerAspect` in its aspect library to configure the dependencies of any objects, even if they were not created by the Spring IoC container.

First of all, you have to annotate your object type with the `@Configurable` annotation to declare that this type of object is configurable.

```

package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.context.annotation.Scope;

@Configuration
@Component
@Scope("prototype")
public class Complex {
    ...
    @Autowired
    public void setFormatter(ComplexFormatter formatter) {
        this.formatter = formatter;
    }
}

```

In addition to the `@Configurable` annotation, you decorate the POJO with the standard `@Component`, `@Scope` and `@Autowired` annotations so the bean gets its standard Spring behaviors. However, the `@Configurable` annotation is the most important configuration piece and for it Spring defines a convenient XML element, `<context:spring-configured>`, for you to enable the mentioned aspect.

```

<beans ...>
    ...
    <context:load-time-weaver />
    <context:spring-configured />

    <context:component-scan
        base-package="com.apress.springrecipes.calculator"/>
</beans>

```

When a class with the @Configurable annotation is instantiated, the aspect will look for a prototype-scoped bean definition whose type is the same as this class. Then, it will configure the new instances according to this bean definition. If there are properties declared in the bean definition, the new instances will also have the same properties set by the aspect.

Finally, to run the application you weave the Aspect it into your classes at load time with the Spring agent.

```
java -javaagent:c:/lib/spring-instrument-3.2.0.jar -jar Recipe_3_21-1.0.jar
```

3-22. Concurrency with Spring and TaskExecutors

Problem

You want to build a threaded, concurrent program with Spring but don't know what approach to use, since there's no standard approach.

Solution

Use Spring's TaskExecutor abstraction. This abstraction provides numerous implementations for many environments, including basic Java SE Executor implementations, the CommonJ WorkManager implementations, and custom implementations. In Spring all the implementations are unified and can be cast to Java SE's Executor interface, too.

How It Works

Threading is a difficult issue which can be particularly tedious to implement using standard threading in the Java SE environment. Concurrency is another important aspect of server-side components but has little to no standardization in the enterprise Java space. In fact some parts of the Java Enterprise Edition specifications forbid the explicit creation and manipulation of threads.

In the Java SE landscape, many options have been introduced over the years to deal with threading and concurrency. First, there was the standard `java.lang.Thread` support present since day one and Java Development Kit (JDK) 1.0. Java 1.3 saw the introduction of `java.util.TimerTask` to support doing some sort of work periodically. Java 5 debuted the `java.util.concurrent` package, as well as a reworked hierarchy for building thread pools, oriented around the `java.util.concurrent.Executor`.

The application programming interface (API) for Executor is simple:

```
package java.util.concurrent;
public interface Executor {
    void execute(Runnable command);
}
```

`ExecutorService`, a subinterface, provides more functionality for managing threads and provides support to raise events to threads, such as `shutdown()`. There are several implementations that have shipped with the JDK since Java SE 5.0. Many of them are available via static factory methods in the `java.util.concurrent` package. What follows are several examples using Java SE classes.

The `ExecutorService` class provides a `submit()` method, which returns a `Future<T>` object. An instance of `Future<T>` can be used to track the progress of a thread, that's usually executing asynchronously. You can call `Future.isDone()` or `Future.isCancelled()` to determine whether the job is finished or cancelled, respectively. When you use the `ExecutorService` and `submit()` inside a `Runnable` instance whose `run` method has no return type, calling `get()` on the returned `Future` returns null, or the value specified on submission:

```
Runnable task = new Runnable(){
    public void run(){
        try{
            Thread.sleep( 1000 * 60 ) ;
        }
```

```

        System.out.println("Done sleeping for a minute, returning! " );
    } catch (Exception ex) { /* ... */ }
}
};

ExecutorService executorService = Executors.newCachedThreadPool();
if(executorService.submit(task, Boolean.TRUE).get().equals( Boolean.TRUE ))
    System.out.println( "Job has finished!");

```

With this background information, we can explore some of the characteristics of the various implementations. For example, the following is a class designed to mark the passage of time using Runnable:

```

package com.apress.springrecipes.spring3.executors;

import java.util.Date;
import org.apache.commons.lang.exception.ExceptionUtils;

public class DemonstrationRunnable implements Runnable {
    public void run() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println(
                ExceptionUtils.getFullStackTrace(e));
        }
        System.out.println(Thread.currentThread().getName());
        System.out.printf("Hello at %s \n", new Date());
    }
}

```

You'll use the same instance when you explore Java SE Executors and Spring's TaskExecutor support:

```

package com.apress.springrecipes.spring3.executors;
import java.util.Date;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class ExecutorsDemo {

    public static void main(String[] args) throws Throwable {
        Runnable task = new DemonstrationRunnable();

        // will create a pool of threads and attempt to
        // reuse previously created ones if possible
        ExecutorService cachedThreadPoolExecutorService = Executors
            .newCachedThreadPool();
        if (cachedThreadPoolExecutorService.submit(task).get() == null)
            System.out.printf("The cachedThreadPoolExecutorService "
                + "has succeeded at %s \n", new Date());
    }
}

```

```

// limits how many new threads are created, queuing the rest
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(100);
if (fixedThreadPool.submit(task).get() == null)
    System.out.printf("The fixedThreadPool has " +
        "succeeded at %s \n",
        new Date());

// doesn't use more than one thread at a time
ExecutorService singleThreadExecutorService = Executors
    .newSingleThreadExecutor();
if (singleThreadExecutorService.submit(task).get() == null)
    System.out.printf("The singleThreadExecutorService " +
        "has succeeded at %s \n", new Date());

// support sending a job with a known result
ExecutorService es = Executors.newCachedThreadPool();
if (es.submit(task, Boolean.TRUE).get().equals(Boolean.TRUE))
    System.out.println("Job has finished!");

// mimic TimerTask
ScheduledExecutorService scheduledThreadExecutorService = Executors
    .newScheduledThreadPool(10);
if (scheduledThreadExecutorService.schedule(
    task, 30, TimeUnit.SECONDS).get() == null)
    System.out.printf("The scheduledThreadExecutorService " +
        "has succeeded at %s \n", new Date());

// this doesn't stop until it encounters
// an exception or its cancel()ed
scheduledThreadExecutorService.scheduleAtFixedRate(task, 0, 5,
    TimeUnit.SECONDS);

}

}

```

If you use the `submit()` method version of the `ExecutorService` that accepts `Callable<T>`, then `submit()` returns whatever was returned from the main `call()` method in `Callable`. The interface for `Callable` is the following:

```

package java.util.concurrent;

public interface Callable<V> {
    V call() throws Exception;
}

```

In the Java EE landscape, different approaches for solving these sorts of problems have been created, since Java EE by design restricts the handling of threads.

Quartz (a job scheduling framework) was among the first solutions to fill this thread feature gap with a solution that provided scheduling and concurrency. JCA 1.5 (or the J2EE Connector Architecture) is another specification that provides a primitive type of gateway for integration functionality supports ad-hoc concurrency. With JCA components are notified about incoming messages and respond concurrently. JCA 1.5 provides primitive, limited enterprise service bus—similar to integration features, without nearly as much of the finesse of something like SpringSource's Spring Integration framework.

The requirement for concurrency wasn't lost on application server vendors, though. Many other initiatives came to the forefront. For example, in 2003, IBM and BEA jointly created the Timer and WorkManager APIs, which eventually became JSR-237 and was then merged with JSR-236 to focus on how to implement concurrency in a managed environment. The Service Data Object (SDO) Specification, JSR-235, also had a similar solution. In addition, open source implementations of the CommonJ API have sprung up in recent years to achieve the same solution.

The issue is that there's no portable, standard, simple way of controlling threads and providing concurrency for components in a managed environment, similar to the case of Java SE solutions.

Spring provides a unified solution via the `org.springframework.core.task.TaskExecutor` interface. The `TaskExecutor` abstraction extends `java.util.concurrent.Executor` which is part of Java 1.5.

In fact, the `TaskExecutor` interface is used quite a bit internally in the Spring framework. For example, for Spring Quartz integration (which supports threading) and the message-driven POJO container support there's wide use of `TaskExecutor`:

```
// the Spring abstraction
package org.springframework.core.task;

import java.util.concurrent.Executor;

public interface TaskExecutor extends Executor {
    void execute(Runnable task);
}
```

In some places, the various solutions mirror the functionality provided by the core JDK options. In others, they're quite unique and provide integrations with other frameworks such as with CommonJ WorkManager. These integrations usually take the form of a class that can exist in the target framework but that you can manipulate just like any other `TaskExecutor` abstraction.

Although there's support for adapting an existing Java SE Executor or ExecutorService as a `TaskExecutor`, this isn't so important in Spring because the base class for `TaskExecutor` is an `Executor`, anyway. In this way, the `TaskExecutor` in Spring bridges the gap between various solutions on Java EE and Java SE.

Next, let's see a simple example of the `TaskExecutor`, using the same `Runnable` defined previously. The client for the code is a simple Spring POJO, into which you've injected various instances of `TaskExecutor` with the sole aim of submitting `Runnable`:

```
package com.apress.springrecipes.executors;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

import org.springframework.core.task.SimpleAsyncTaskExecutor;
import org.springframework.core.task.SyncTaskExecutor;
import org.springframework.core.task.support.TaskExecutorAdapter;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.scheduling.concurrent.ScheduledExecutorFactoryBean;

public class SpringExecutorsDemo {
    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");
        SpringExecutorsDemo demo = context.getBean(
```

```

        "springExecutorsDemo", SpringExecutorsDemo.class);
demo.submitJobs();
}

@Autowired
private SimpleAsyncTaskExecutor asyncTaskExecutor;

@Autowired
private SyncTaskExecutor syncTaskExecutor;

@Autowired
private TaskExecutorAdapter taskExecutorAdapter;

/* No need, since the scheduling and submission
   is already configured, and done in the application context
   via Factory bean
@Autowired
private ScheduledExecutorFactoryBean concurrentTaskScheduler;
*/

@Autowired
private ThreadPoolTaskExecutor threadPoolTaskExecutor;

@Autowired
private DemonstrationRunnable task;

public void submitJobs() {
    syncTaskExecutor.execute(task);
    taskExecutorAdapter.submit(task);
    asyncTaskExecutor.submit(task);

    /* will do 100 at a time,
       then queue the rest, ie,
       should take round 5 seconds total
    */
    for (int i = 0; i < 500; i++)
        threadPoolTaskExecutor.submit(task);
}
}

```

The application context demonstrates the creation of these various TaskExecutor implementations. Most are so simple that you could create them manually. Only in one case do you delegate to a factory bean to automatically trigger the execution:

```

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

```

```

xsi:schemaLocation="
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.2.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.2.xsd">

<context:annotation-config />
<!-- sample Runnable -->
<bean
    id="task"    class="com.apress.springrecipes.
                           executors.DemonstrationRunnable" />

<!-- TaskExecutors -->
<bean
    class="org.springframework.core.task.support.TaskExecutorAdapter">
<constructor-arg>
    <bean
        class="java.util.concurrent.Executors"
        factory-method="newCachedThreadPool" />
</constructor-arg>
</bean>

<bean
    class="org.springframework.core.task.SimpleAsyncTaskExecutor"
    p:daemon="false" />

<bean
    class="org.springframework.core.task.SyncTaskExecutor" />

<bean class="org.springframework.scheduling.concurrent.ScheduledExecutorFactoryBean"
id="concurrentTaskScheduler">
    <property name="scheduledExecutorTasks">
        <list>
            <bean class="org.springframework.scheduling.concurrent.ScheduledExecutorTask"
                  p:period="1000"
                  p:runnable-ref="task"/>
        </list>
    </property>
</bean>

<bean
    class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"
    p:corePoolSize="50"
    p:daemon="false"
    p:waitForTasksToCompleteOnShutdown="true"
    p:maxPoolSize="100"
    p:allowCoreThreadTimeOut="true" />

<!-- client bean -->
<bean
    id="springExecutorsDemo"

```

```

class="com.apress.springrecipes.
    executors.SpringExecutorsDemo" />
</beans>

```

The previous code shows different implementations of the TaskExecutor interface. The first bean, the TaskExecutorAdapter instance, is a simple wrapper around a `java.util.concurrent.Executors` instance so you can deal with it in terms of the Spring TaskExecutor interface. You use Spring here to configure an instance of an Executor and pass it in as the constructor argument.

`SimpleAsyncTaskExecutor` provides a new Thread for each submitted job. It does no thread pooling or reuse. Each job submitted runs asynchronously in a thread.

`SyncTaskExecutor` is the simplest of the implementations of TaskExecutor. Submission of a job is synchronous and tantamount to launching a Thread, running it, and then using `join()` to connect it immediately. It's effectively the same as manually invoking the `run()` method in the calling thread, skipping threading all together.

`ScheduledExecutorFactoryBean` automatically triggers jobs defined as `ScheduledExecutorTask` beans. You can specify a list of `ScheduledExecutorTask` instances to trigger multiple jobs simultaneously. A `ScheduledExecutorTask` instance can accept a period to space out the execution of tasks.

The last example is `ThreadPoolTaskExecutor`, which is a full-on thread pool implementation built on `java.util.concurrent.ThreadPoolExecutor`.

If you want to build applications using the CommonJ WorkManager/TimerManager support available in application servers like IBM WebSphere, you can use `org.springframework.scheduling.commonj.WorkManagerTaskExecutor`. This class delegates to a reference to the CommonJ Work Manager available inside of WebSphere. Usually, you'll provide it with a JNDI reference to the appropriate resource. This works well enough (such as with Geronimo), but extra effort is required with JBoss or GlassFish. Spring provides classes that delegate to the JCA support provided on those servers: for GlassFish, use `org.springframework.jca.work.glassfish.GlassFishWorkManagerTaskExecutor`; for JBoss, use `org.springframework.jca.work.jboss.JBossWorkManagerTaskExecutor`.

The TaskExecutor support provides a powerful way to access scheduling services on an application server via a unified interface. If you're looking for more robust (albeit much more heavyweight) support that can be deployed on any app server (e.g., Tomcat and Jetty), you might consider Spring's Quartz support.

Summary

In this chapter, you learned about Spring's core tasks associated with annotations. Instead of using the classical Spring XML POJO instantiation process, you learned how Spring supports the `@Configuration` and `@Bean` annotations to instantiate POJO via a Java config class. You also learned how to use the `@Component` annotation to administer POJOs with Spring. In addition, you learned about the `@Repository`, `@Service`, and `@Controller` annotations which provide more specific behavior than the `@Component` annotation.

You also learned how to reference POJOs from other POJOs, as well as how to use the `@Autowired` annotation which can automatically associate POJOs by either type or name. In addition, you also explored how the standard `@Resource` and `@Inject` annotations work to reference POJOs via autowiring, instead of using the Spring specific `@Autowired` annotation.

You then learned how to set a Spring POJOs scope with the `@Scope` annotation. You also learned how Spring can read external resources and use this data in the context of POJO configuration and creation using the `@PropertySource` and `@Value` annotations. In addition, you learned how Spring supports different languages in POJOs through the use of i18n resource bundles.

Next, you learned how to customize the initialization and destruction of POJOs with the `initMethod` and `destroyMethod` attributes of a `@Bean` annotation, as well as the `@PostConstruct` and `@PreDestroy` annotations. In addition, you learned how to do lazy initialization with the `@PreDestroy` annotation and define initialization dependencies with the `@DependsOn` annotation.

You then learned about Spring post processors to validate and modify POJO values, including how to use the `@Required` annotation. Next, you explored how to work with Spring environments and profiles to load different sets of POJOs, including how to use the `@Profile` annotation.

Next, you explored aspect oriented programming in the context of Spring and learned how to create aspects, pointcuts, and advices. This included the use of the `@Aspect` annotation, as well as the `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing`, and `@Around` annotations.

Next, you learned how to access AOP join point information and apply it to different program execution points. And then you learned how to specify aspect precedence with the `@Order` annotations, followed by how to reuse aspect pointcut definition.

In this chapter, you also learned how to write AspectJ pointcut expressions, as well as how to apply the concept of AOP 'introductions' so a POJO can inherit behaviors from multiple implementation classes at the same time. You also learned how to introduce states to POJOs with AOP, as well as how to apply the technique of load-time weaving.

Finally, you learned how to configure AspectJ aspects in Spring, how to inject POJOs into domain objects, as well as how to deal with concurrency with Spring and TaskExecutors.

CHAPTER 4

Spring @MVC

MVC is one of the most important modules of the Spring framework. It builds on the powerful Spring IoC container and makes extensive use of the container features to simplify its configuration.

Model-view-controller (MVC) is a common design pattern in UI design. It decouples business logic from UIs by separating the roles of model, view, and controller in an application. *Models* are responsible for encapsulating application data for views to present. *Views* should only present this data, without including any business logic. *Controllers* are responsible for receiving requests from users and invoking back-end services for business processing. After processing, back-end services may return some data for views to present. Controllers collect this data and prepare models for views to present. The core idea of the MVC pattern is to separate business logic from UIs to allow them to change independently without affecting each other.

In a Spring MVC application, models usually consist of domain objects that are processed by the service layer and persisted by the persistence layer. Views are usually JSP templates written with Java Standard Tag Library (JSTL). However, it's also possible to define views as PDF files, Excel files, RESTful web services or even Flex interfaces the last of which are often dubbed a Rich Internet Application (RIA).

Upon finishing this chapter, you will be able to develop Java web applications using Spring MVC. You will also understand Spring MVC's common controller and view types, including, what has become the de facto use of annotations for creating controllers as of Spring 3.0. Moreover, you will understand the basic principles of Spring MVC, which will serve as the foundations for more advanced topics covered in the upcoming chapters.

4-1. Developing a Simple Web Application with Spring MVC

Problem

You want to develop a simple web application with Spring MVC to learn the basic concepts and configurations of this framework.

Solution

The central component of Spring MVC is a controller. In the simplest Spring MVC application, a controller is the only servlet you need to configure in a Java web deployment descriptor (i.e., the `web.xml` file or a `ServletContainerInitializer`). A Spring MVC controller—often referred to as a Dispatcher Servlet—implements one of Sun's core Java EE design patterns called *front controller*. It acts as the front controller of the Spring MVC framework, and every web request must go through it so that it can manage the entire request-handling process.

When a web request is sent to a Spring MVC application, a controller first receives the request. Then it organizes the different components configured in Spring's web application context or annotations present in the controller itself, all needed to handle the request. Figure 4-1 shows the primary flow of request handling in Spring MVC.

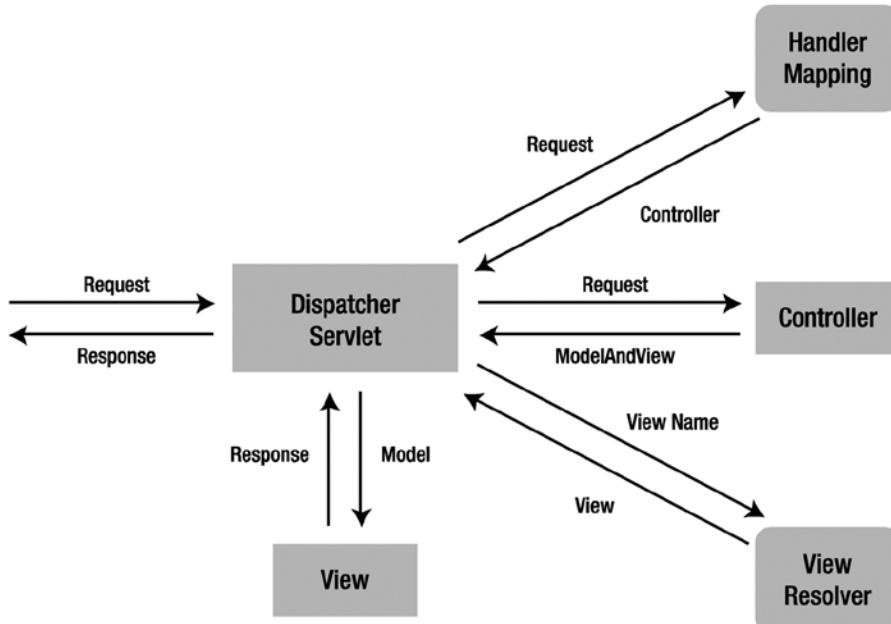


Figure 4-1. Primary flow of request handling in Spring MVC

To define a controller class in Spring 4.0, a class has to be marked with the `@Controller` annotation. In contrast to other framework controllers or earlier Spring versions, an annotated controller class need not implement a framework-specific interface or extend a framework-specific base class.

For example, prior to Spring 3.0 one of a series of classes, such as `AbstractController`, were used to give a class the behavior of a Dispatcher Servlet. Starting from Spring 2.5, annotated classes for defining Dispatcher Servlets became available. As of Spring 3.0, these series of classes for giving a class the behavior of a Dispatcher Servlet have been deprecated in favor of annotated classes.

When a `@Controller` annotated class (i.e., a controller class) receives a request, it looks for an appropriate handler method to handle the request. This requires that a controller class map each request to a handler method by one or more handler mappings. In order to do so, a controller class's methods are decorated with the `@RequestMapping` annotation, making them handler methods.

The signature for these handler methods—as you can expect from any standard class—is open ended. You can specify an arbitrary name for a handler method and define a variety of method arguments. Equally, a handler method can return any of a series of values (e.g., `String` or `void`), depending on the application logic it fulfills.

As the book progresses, you will encounter the various method arguments that can be used in handler methods using the `@RequestMapping` annotation. The following is only a partial list of valid argument types, just to give you an idea.

- `HttpServletRequest` or `HttpServletResponse`
- Request parameters of arbitrary type, annotated with `@RequestParam`
- Model attributes of arbitrary type, annotated with `@ModelAttribute`
- Cookie values included in an incoming request, annotated with `@CookieValue`

- `Map` or `ModelMap`, for the handler method to add attributes to the model
- `Errors` or `BindingResult`, for the handler method to access the binding and validation result for the command object
- `SessionStatus`, for the handler method to notify its completion of session processing

Once the controller class has picked an appropriate handler method, it invokes the handler method's logic with the request. Usually, a controller's logic invokes back-end services to handle the request. In addition, a handler method's logic is likely to add or remove information from the numerous input arguments (e.g., `HttpServletRequest`, `Map`, `Errors`, or `SessionStatus`) that will form part of the ongoing Spring MVC flow.

After a handler method has finished processing the request, it delegates control to a view, which is represented as the handler method's return value. To provide a flexible approach, a handler method's return value doesn't represent a view's implementation (e.g., `user.jsp` or `report.pdf`) but rather a *logical* view (e.g., `user` or `report`)—note the lack of file extension.

A handler method's return value can be either a `String`—representing a logical view name—or `void`, in which case a default logical view name is determined on the basis of a handler method's or controller's name.

In order to pass information from a controller to a view, it's irrelevant that a handler's method returns a logical view name—`String` or a `void`—since the handler method input arguments will be available to a view.

For example, if a handler method takes a `Map` and `SessionStatus` objects as input parameters—modifying their contents inside the handler method's logic—these same objects will be accessible to the view returned by the handler method.

When the controller class receives a view, it resolves the *logical* view name into a specific view implementation (e.g., `user.jsp` or `report.pdf`) by means of a view resolver. A view resolver is a bean configured in the web application context that implements the `ViewResolver` interface. Its responsibility is to return a specific view implementation (HTML, JSP, PDF, or other) for a logical view name.

Once the controller class has resolved a view name into a view implementation, per the view implementation's design, it renders the objects (e.g., `HttpServletRequest`, `Map`, `Errors`, or `SessionStatus`) passed by the controller's handler method. The view's responsibility is to display the objects added in the handler method's logic to the user.

How It Works

Suppose you are going to develop a court reservation system for a sports center. The UIs of this application are web-based so that users can make online reservations through the Internet. You want to develop this application using Spring MVC. First of all, you create the following domain classes in the `domain` subpackage:

```
package com.apress.springrecipes.court.domain;
...
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.court.domain;
```

```

public class Player {

    private String name;
    private String phone;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.court.domain;

public class SportType {

    private int id;
    private String name;

    // Constructors, Getters and Setters
    ...
}

```

Then you define the following service interface in the service subpackage to provide reservation services to the presentation layer:

```

package com.apress.springrecipes.court.service;
...
public interface ReservationService {

    public List<Reservation> query(String courtName);
}

```

In a production application, you should implement this interface with database persistence. But for simplicity's sake, you can store the reservation records in a list and hard-code several reservations for testing purposes:

```

package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");

    private List<Reservation> reservations;

    public ReservationServiceImpl() {
        reservations = new ArrayList<>();
        reservations.add(new Reservation("Tennis #1",
            new GregorianCalendar(2008, 0, 14).getTime(), 16,
            new Player("Roger", "N/A"), TENNIS));
        reservations.add(new Reservation("Tennis #2",
            new GregorianCalendar(2008, 0, 14).getTime(), 20,
            new Player("James", "N/A"), TENNIS));
    }
}

```

```

public List<Reservation> query(String courtName) {
    List<Reservation> result = new ArrayList<>();
    for (Reservation reservation : reservations) {
        if (reservation.getCourtName().equals(courtName)) {
            result.add(reservation);
        }
    }
    return result;
}
}

```

Setting up a Spring MVC Application

Next, you need to create a Spring MVC application layout. In general, a web application developed with Spring MVC is set up in the same way as a standard Java web application, except that you have to add a couple of configuration files and required libraries specific to Spring MVC.

The Java EE specification defines the valid directory structure of a Java web application made up of a Web Archive or WAR file. For example, you have to provide a web deployment descriptor (i.e., `web.xml`) in the WEB-INF root or 1 or more classes implementing `ServletContainerInitializer`. The class files and JAR files for this web application should be put in the WEB-INF/classes and WEB-INF/lib directories, respectively.

For your court reservation system, you create the following directory structure. Note that the highlighted files are Spring-specific configuration files.

Note To develop a web application with Spring MVC, you have to add all the normal Spring dependencies (see Chapter 1 for more information) as well as the Spring Web and Spring MVC dependencies to your CLASSPATH. If you are using Maven, add the following dependencies to your Maven Project:

```

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${spring.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
</dependency>

```

```

court/
    css/
    images/
    WEB-INF/
        classes/
        lib/*.jar
        jsp/

```

```
welcome.jsp
reservationQuery.jsp
court-service.xml
court-servlet.xml
web.xml
```

The files outside the WEB-INF directory are directly accessible to users via URLs, so the CSS files and image files must be put there. When using Spring MVC, the JSP files act as templates. They are read by the framework for generating dynamic content, so the JSP files should be put inside the WEB-INF directory to prevent direct access to them. However, some application servers don't allow the files inside WEB-INF to be read by a web application internally. In that case, you can only put them outside the WEB-INF directory.

Creating the Configuration Files

The web deployment descriptor web.xml is the essential configuration file for a Java web application. In this file, you define the servlets for your application and how web requests are mapped to them. For a Spring MVC application, you only have to define a single DispatcherServlet instance that acts as the front controller for Spring MVC, although you are allowed to define more than one if required.

In large applications, it can be convenient to use multiple DispatcherServlet instances. This allows DispatcherServlet instances to be designated to specific URLs, making code management easier and letting individual team members work on an application's logic without getting in each other's way.

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_3_0.xsd">

  <display-name>Court Reservation System</display-name>

  <servlet>
    <servlet-name>court</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

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

In this web deployment descriptor, you define a servlet of type DispatcherServlet. This is the core servlet class in Spring MVC that receives web requests and dispatches them to appropriate handlers. You set this servlet's name to court and map all URLs using a / (slash), with the slash representing the root directory. Note that the URL pattern can be set to more granular patterns. In larger application's it can make more sense to delegate patterns among various servlets, but for simplicity all URLs in the application are delegated to the single court servlet.

Another purpose of the servlet name is for DispatcherServlet to decide which file to load for Spring MVC configurations. By default, a look is made for a file by joining the servlet name with -servlet.xml as the file name. You can explicitly specify a configuration file in the contextConfigLocation servlet parameter. With the preceding

setting, the court servlet loads the Spring MVC configuration file `court-servlet.xml` by default. This file should be a standard Spring bean configuration file, as shown in the following:

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

Later, you can configure Spring MVC with a series of Spring beans declared in this configuration file. You may also declare other application components such as data access objects and service objects in this file. However, it's not a good practice to mix beans of different layers in a single configuration file. Instead, you should declare one bean configuration file per layer (e.g., `court-persistence.xml` for the persistence layer and `court-service.xml` for the service layer). For example, `court-service.xml` should include the following service object:

```
<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="reservationService"
        class="com.apress.springrecipes.court.service.ReservationServiceImpl" />
</beans>
```

In order for Spring to load your configuration files besides `court-servlet.xml`, you need to define the servlet listener `ContextLoaderListener` in `web.xml`. By default, it loads the bean configuration file `/WEB-INF/applicationContext.xml`, but you can specify your own in the context parameter `contextConfigLocation`. You can specify multiple configuration files by separating their locations with either commas or spaces.

```
<web-app ...>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/court-service.xml</param-value>
    </context-param>

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

Note that `ContextLoaderListener` loads the specified bean configuration files into the root application context, while each `DispatcherServlet` instance loads its configuration file into its own application context and refers to the root application context as its parent. So, the context loaded by each `DispatcherServlet` instance can access and even override beans declared in the root application context (but not vice versa). However, the contexts loaded by the `DispatcherServlet` instances cannot access each other.

Activating Spring MVC annotation scanning

Before you create an application's controllers, you have to set up the web application for classes to be scanned for the presence of `@Controller` and `@RequestMapping` annotations. Only then can they operate as controllers. First, for Spring to auto-detect annotations, you have to enable Spring's component scanning feature through the `<context:component-scan>` element.

In addition to this statement, since Spring MVC's `@RequestMapping` annotation maps URL requests to controller classes and their corresponding handler methods, this requires additional statements in a web application's context. To make this work, you have to register a `RequestMappingHandlerMapping` instance and an `RequestMappingHandlerAdapter` instance in the web application context. These instances process the `@RequestMapping` annotations at the class level and the method level, respectively.

To enable support for annotation-based controllers, include the following configuration in the `court-servlet.xml` file:

```

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

    <context:component-scan
        base-package="com.apress.springrecipes.court.web" />
    <bean class="org.springframework.web.servlet.mvc.method.annotation.
    RequestMappingHandlerMapping" />
    <bean class="org.springframework.web.servlet.mvc.method.annotation.
    RequestMappingHandlerAdapter" />
</beans>

```

Notice the element `<context:component-scan>` with a `base-package` value of `com.apress.springrecipes.court.web`. This package corresponds to the same one used in the Spring MVC controller, which is illustrated next.

Next, the `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` bean classes are preregistered in the web application context by default.

Once you have the basic context configuration file for scanning Spring MVC annotations, you can proceed to creating the controller class itself, as well as finishing the configuration set-up for `court-servlet.xml`.

Creating Spring MVC Controllers

An annotation-based controller class can be an arbitrary class that doesn't implement a particular interface or extend a particular base class. You can annotate it with the `@Controller` annotation. There can be one or more handler methods defined in a controller to handle single or multiple actions. The signature of the handler methods is flexible enough to accept a range of arguments.

The `@RequestMapping` annotation can be applied to the class level or the method level. The first mapping strategy is to map a particular URL pattern to a controller class, and then a particular HTTP method to each handler method:

```

package com.apress.springrecipes.court.web;
...
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

```

```

import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;

@Controller
@RequestMapping("/welcome")
public class WelcomeController {

    @RequestMapping(method = RequestMethod.GET)
    public String welcome(Model model) {
        Date today = new Date();
        model.addAttribute("today", today);
        return "welcome";
    }
}

```

This controller creates a `java.util.Date` object to retrieve the current date, and then adds it to the input `Model` object as an attribute so the target view can display it.

Since you've already activated annotation scanning on the `com.apress.springrecipes.court.web` package declared inside the `court-servlet.xml` file, the annotations for the controller class are detected upon deployment.

The `@Controller` annotation defines the class as a Spring MVC controller. The `@RequestMapping` annotation is more interesting since it contains properties and can be declared at the class or handler method level. The first value used in this class—`("/welcome")`—is used to specify the URL on which the controller is actionable, meaning any request received on the `/welcome` URL is attended by the `WelcomeController` class.

Once a request is attended by the controller class, it delegates the call to the default HTTP GET handler method declared in the controller. The reason for this behavior is that every initial request made on a URL is of the HTTP GET kind. So when the controller attends a request on the `/welcome` URL it subsequently delegates to the default HTTP GET handler method for processing.

The annotation `@RequestMapping(method = RequestMethod.GET)` is used to decorate the `welcome` method as the controller's default HTTP GET handler method. It's worth mentioning that if no default HTTP GET handler method is declared, a `ServletException` is thrown. Hence the importance of a Spring MVC controller having at a minimum a URL route and default HTTP GET handler method.

Another variation to this approach can be declaring both values—URL route and default HTTP GET handler method—in the `@RequestMapping` annotation used at the method level. This declaration is illustrated next:

```

@Controller
public class WelcomeController {

    @RequestMapping(value = "/welcome", method=RequestMethod.GET)
    public String welcome(Model model) {
        ...
    }
}

```

This last declaration is equivalent to the earlier one. The `value` attribute indicates the URL to which the handler method is mapped and the `method` attribute defines the handler method as the controller's default HTTP GET method.

This last controller illustrates the basic principles of Spring MVC. However, a typical controller may invoke back-end services for business processing. For example, you can create a controller for querying reservations of a particular court as follows:

```

package com.apress.springrecipes.court.web;
...

```

```

import com.apress.springrecipes.court.domain.Reservation;
import com.apress.springrecipes.court.service.ReservationService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.ui.Model;

@Controller
@RequestMapping("/reservationQuery")
public class ReservationQueryController {

    private ReservationService reservationService;

    @Autowired
    public void ReservationQueryController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public void setupForm() {
    }

    @RequestMapping(method = RequestMethod.POST)
    public String sumbitForm(@RequestParam("courtName") String courtName, Model model) {
        List<Reservation> reservations = java.util.Collections.emptyList();
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
        model.addAttribute("reservations", reservations);
        return "reservationQuery";
    }
}

```

This controller equally relies on the `@Controller` annotation to indicate the class in question is a Spring MVC controller. A new addition though is the `@Autowired` annotation assigned to the class's constructor. This allows a class's constructor to instantiate its fields from declarations made in the application's configuration files (i.e., `court-service.xml`). So for example, in this last controller an attempt is made to locate a bean named `reservationService` to instantiate the field by the same name.

If you recall from earlier, the `court-service.xml` file was used to define a service bean by this same name. This allows the service bean to be injected into the controller class and assigned to the field implicitly. Without the `@Autowired` annotation, the service bean can be injected explicitly inside the `court-servlet.xml` configuration using a statement like the following:

```

<bean
    class="com.apress.springrecipes.court.web.ReservationQueryController">
    <property name="reservationService" ref="reservationService" />
</bean>

```

Thus the `@Autowired` annotation saves you time by not having to inject properties using XML. Continuing with the controller class statements, you can find the `@RequestMapping("/reservationQuery")` statement used to indicate that any request on the `/reservationQuery` URL be attended by the controller.

As outlined earlier, the controller then looks for a default HTTP GET handler method. Since the public void `setupForm()` method is assigned the necessary `@RequestMapping` annotation for this purpose, it's called next.

Unlike the previous default HTTP GET handler method, notice that this method has no input parameters, no logic, and has a void return value. This means two things. By having no input parameters and no logic, a view only displays data hard-coded in the implementation template (e.g., JSP), since no data is being added by the controller. By having a void return value, a default view name based on the request URL is used, therefore since the requesting URL is `/reservationQuery` a return view named `reservationQuery` is assumed.

The remaining handler method is decorated with the `@RequestMapping(method = RequestMethod.POST)` annotation. At first sight, having two handler methods with only the class level `/reservationQuery` URL statement can be confusing, but it's really simple. One method is invoked when HTTP GET requests are made on the `/reservationQuery` URL, the other when HTTP POST requests are made on the same URL.

The majority of requests in web applications are of the HTTP GET kind, where as requests of the HTTP POST kind are generally made when a user submits an HTML form. So revealing more of the application's view (which we will describe shortly), one method is called when the HTML form is initially loaded (i.e., HTTP GET), where as the other is called when the HTML form is submitted (i.e., HTTP POST).

Looking closer at the HTTP POST default handler method, notice the two input parameters. First the `@RequestParam("courtName") String courtName` declaration, used to extract a request parameter named `courtName`. In this case, the HTTP POST request comes in the form `/reservationQuery?courtName=<value>`, this declaration makes said value available in the method under the variable named `courtName`. And second the `Model` declaration, used to define an object in which to pass data onto the returning view.

The logic executed by the handler method consists of using the controller's `reservationService` to perform a query using the `courtName` variable. The results obtained from this query are assigned to the `Model` object, that will later become available to the returning view for display.

Finally, note that the method returns a view named `reservationQuery`. This method could have also returned `void`, just like the default HTTP GET, and have been assigned to the same `reservationQuery` default view on account of the requesting URL. Both approaches are identical.

Now that you are aware of how Spring MVC controllers are constituted, it's time to explore the views to which a controller's handler methods delegate their results.

Creating JSP Views

Spring MVC supports many types of views for different presentation technologies. These include: JSPs, HTML, PDF, Excel worksheets (XLS), XML, JSON, Atom and RSS feeds, JasperReports, and other third-party view implementations.

In a Spring MVC application, views are most commonly JSP templates written with JSTL. When the `DispatcherServlet`—defined in an application's `web.xml` file—receives a view name returned from a handler, it resolves the *logical* view name into a view implementation for rendering. For example, you can configure the `InternalResourceViewResolver` bean, in this case inside `court-servlet.xml`, of a web application's context to resolve view names into JSP files in the `/WEB-INF/jsp/` directory:

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

By using this last configuration, a *logical* view named `reservationQuery` is delegated to a view implementation located at `/WEB-INF/jsp/reservationQuery.jsp`. Knowing this you can create the following JSP template for the welcome controller, naming it `welcome.jsp` and putting it in the `/WEB-INF/jsp/` directory:

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
Welcome to Court Reservation System</h2>
Today is <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />.
</body>
</html>
```

In this JSP template, you make use of the `fmt` tag library in JSTL to format the `today` model attribute into the pattern `yyyy-MM-dd`. Don't forget to include the `fmt` tag library definition at the top of this JSP template.

Next, you can create another JSP template for the reservation query controller and name it `reservationQuery.jsp` to match the view name:

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

<html>
<head>
<title>Reservation Query</title>
</head>

<body>
<form method="post">
Court Name
<input type="text" name="courtName" value="${courtName}" />
<input type="submit" value="Query" />
</form>

<table border="1">
<tr>
<th>Court Name</th>
<th>Date</th>
<th>Hour</th>
<th>Player</th>
</tr>
<c:forEach items="${reservations}" var="reservation">
<tr>
<td>${reservation.courtName}</td>
<td><fmt:formatDate value="${reservation.date}" pattern="yyyy-MM-dd" /></td>
```

```

<td>${reservation.hour}</td>
<td>${reservation.player.name}</td>
</tr>
</c:forEach>
</table>
</body>
</html>

```

In this JSP template, you include a form for users to input the court name they want to query, and then use the `<c:forEach>` tag to loop the `reservations` model attribute to generate the result table.

Deploying the Web Application

In a web application's development process, we strongly recommend installing a local Java EE application server that comes with a web container for testing and debugging purposes. For the sake of easy configuration and deployment, we have chosen Apache Tomcat 7.0.x as the web container.

The deployment directory for this web container is located under the `webapps` directory. By default, Tomcat listens on port 8080 and deploys applications onto a context by the same name of an application WAR. Therefore, if you package the application in a WAR named `court.war`, the welcome controller and the reservation query controller can be accessed through the following URLs:

`http://localhost:8080/court/welcome`
`http://localhost:8080/court/reservationQuery`

Reducing the amount of XML

The current configuration is a classic configuration, it is all XML based and isn't using namespaces. Namespaces can help you minimize the amount of XML you need to write. To minimize the XML we can leverage the `mvc` namespace, we could use `<mvc:annotation-driven />` instead of declaring the `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` explicitly. The configuration would then look like the following:

```

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

    <!-- Scanning enablement on package -->
    <context:component-scan base-package="com.apress.springrecipes.court.web" />

    <!-- Annotation handlers (Applied by default to ALL @controllers -->
    <mvc:annotation-driven />

```

```
<!-- Views mapped in JSPs under /WEB-INF/jsp -->
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
</beans>
```

The `<mvc:annotation-driven>` tag registers, among other things, the `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter`.

As of Spring 3.1 it is also possible to use a Java based configuration approach which we can leverage to eliminate the XML (we will only be left with the `<context:component-scan />`for now). The Java based configuration looks like this

```
package com.apress.springrecipes.court.web.config;

// ... Imports omitted

@Configuration
@EnableWebMvc
public class WebConfiguration {

    @Bean
    public ViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

The important thing to notice here is the `@EnableWebMvc` annotation, which takes care of registering the `RequestMappingHandlerMapping` and `RequestMappingHandlerAdapter` as does the `<mvc:annotation-driven />`tag. We also added the `@Configuration` annotation so that Spring knows that this class needs to be used to configure our application. Our XML is now reduced to only a `<context:component-scan />`which will pickup our `@Configuration` annotated class. We could do the same for our service configuration and move that to Java configuration.

```
package com.apress.springrecipes.court.service.config;

import com.apress.springrecipes.court.service.ReservationService;
import com.apress.springrecipes.court.service.ReservationServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ServiceConfiguration {

    @Bean
    public ReservationService reservationService() {
        return new ReservationServiceImpl();
    }
}
```

Now we would need to modify our court-service.xml to do component scanning of the service sub package to detect this configuration class.

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

    <!-- Scanning enablement on package -->
    <context:component-scan base-package="com.apress.springrecipes.court.service"/>
</beans>
```

If we rebuild and redeploy our application it should still work.

As of the release of the Servlet 3.0 specification it isn't necessary to have a web.xml file anymore, it became optional. As hinted earlier we can use an implementation of a `ServletContainerInitializer` (or multiple) to configure our application.

Instead of implementing our own we are going to leverage the convenient Spring implementation the `SpringServletContainerInitializer`. This class is an implementation of the `ServletContainerInitializer` interface and scans the classpath for implementations of a `WebApplicationInitializer` interface. Luckily Spring provides some convenience implementations of this interface which we can leverage for our application we are going to use the `AbstractAnnotationConfigDispatcherServletInitializer` to reduce our Spring XML to zero.

Create a class which extends `AbstractAnnotationConfigDispatcherServletInitializer` and implements the necessary methods.

```
package com.apress.springrecipes.court.web;

import com.apress.springrecipes.court.service.config.ServiceConfiguration;
import com.apress.springrecipes.court.web.config.WebConfiguration;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class CourtApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[] {ServiceConfiguration.class};
    }

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

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

We need to configure the root-context and the servlet-context, respectively the ContextLoaderListener and DispatcherServlet. We do this by implementing the getRootConfigClasses and getServletConfigClasses methods. The first is for the ContextLoaderListener and this we pass the ServiceConfiguration class we created earlier. The second is for the DispatcherServlet and that we pass the WebConfiguration class (which requires a minor addition). Finally we need to map the servlet to 1 or more URLs for this we implement the getServletMappings method.

```
@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration { ... }
```

As we want to eliminate XML as much as possible we also need to move the component-scanning to Java configuration. For this we use the @ComponentScan tag, which replaces the <context:component-scan /> tag.

With the configuration above we can remove the court-service.xml and court-servlet.xml however we, sadly, still need the web.xml. This is due to some omissions in the Servlet 3.0 spec, not everything that can be configured in the web.xml has a Java based alternative yet. For instance the welcome pages can only be specified in XML not in Java, the same goes for the display-name and error-pages (which we will see later). However the following is what is left of the web.xml and all the XML we have left in our application.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <display-name>Court Reservation System</display-name>

  <welcome-file-list>
    <welcome-file>welcome</welcome-file>
  </welcome-file-list>

</web-app>
```

Note From this point on the recipes will use Java-based configuration, however if you prefer an XML-based approach we also included the XML equivalent of the configuration. See the court-servlet.xml file in the project.

4-2. Mapping requests with @RequestMapping

Problem

When DispatcherServlet receives a web request, it attempts to dispatch requests to the various controllers classes that have been declared with the @Controller annotation. The dispatching process depends on the various @RequestMapping annotations declared in a controller class and its handler methods. You want to define a strategy for mapping requests using the @RequestMapping annotation.

Solution

In a Spring MVC application, web requests are mapped to handlers by one or more `@RequestMapping` annotations declared in controller classes.

Handler mappings match URLs according to their paths relative to the context path (i.e., the web application context's deployed path) and the servlet path (i.e., the path mapped to `DispatcherServlet`). So for example, in the URL `http://localhost:8080/court/welcome` the path to match is `/welcome`, as the context path is `/court` and there's no servlet path—recall the servlet path declared as `/` in `web.xml`.

How It Works

Mapping requests by method

The simplest strategy for using `@RequestMapping` annotations is to decorate the handler methods directly. For this strategy to work, you have to declare each handler method with the `@RequestMapping` annotation containing a URL pattern. If a handler's `@RequestMapping` annotation matches a request's URL, `DispatcherServlet` it dispatches the request to this handler for it to handle the request.

```
@Controller
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("/member/add")
    public String addMember(Model model) {
        model.addAttribute("member", new Member());
        model.addAttribute("guests", memberService.list());
        return "memberList";
    }

    @RequestMapping(value={"/member/remove","/member/delete"}, method=RequestMethod.GET)
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect:";
    }
}
```

This last listing illustrates how each handler method is mapped to a particular URL using the `@RequestMapping` annotation. The second handler method illustrates the assignment of multiple URLs, so both `/member/remove` and `/member/delete` trigger the execution of the handler method. By default, it's assumed all incoming requests to URLs are of the HTTP GET kind.

Mapping requests by class

The `@RequestMapping` annotation can also be used to decorate a controller class. This allows handler methods to either forgo the use of `@RequestMapping` annotations, as illustrated in the `ReservationQueryController` controller in recipe 4-1, or use finer grained URLs with their own `@RequestMapping` annotation. For broader URL matching, the `@RequestMapping` annotation also supports the use wildcards (i.e., `*`).

The following listing illustrates the use of URL wildcards in a `@RequestMapping` annotation, as well as finer grained URL matching on `@RequestMapping` annotations for handler methods.

```

@Controller
@RequestMapping("/member/*")
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("add")
    public String addMember(Model model) {
        model.addAttribute("member", new Member());
        model.addAttribute("guests", memberService.list());
        return "memberList";
    }

    @RequestMapping(value={"remove","delete"}, method=RequestMethod.GET)
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect:";
    }

    @RequestMapping("display/{user}")
    public String removeMember(
        @RequestParam("memberName") String memberName,
        @PathVariable("user") String user) {
        ....
    }

    @RequestMapping
    public void memberList() {
        ....
    }

    public void memberLogic(String memberName) {
        ....
    }
}

```

Note the class level @RequestMapping annotation uses a URL wildcard: /member/* . This in turn delegates all requests under the /member/ URL to the controller's handler methods.

The first two handler methods make use of the @RequestMapping annotation. The addMember() method is invoked when an HTTP GET request is made on the /member/add URL. Whereas the removeMember() method is invoked when an HTTP GET request is made on either the /member/remove or /member/delete URL.

The third handler method uses the special notation {path_variable} to specify its @RequestMapping value. By doing so, a value present in the URL can be passed as input to the handler method. Notice the handler method declares @PathVariable("user") String user. In this manner, if a request is received in the form member/display/jdoe , the handler method has access to the user variable with a jdoe value. This is mainly a facility that allows you to avoid tinkering with a handler's request object and an approach that is especially helpful when you design RESTful web services.

The fourth handler method also uses the @RequestMapping annotation, but in this case lacks a URL value. Since the class level uses the /member/* URL wildcard, this handler method is executed as a catch-all. So any URL request (e.g., /member/abcdefg or /member/randomroute) triggers this method. Note the void return value, that in turn makes the handler method default to a view by its name (i.e., memberList).

The last method—memberLogic—lacks any @RequestMapping annotations, this means the method is a utility for the class and has no influence on Spring MVC.

Mapping requests by HTTP request type

By default, @RequestMapping annotations handle all types of incoming requests. It is however, in most cases, not wanted that the same method is executed for both a GET and POST request. To differentiate on HTTP request, it's necessary to specify the type explicitly in the @RequestMapping annotation as follows:

```
@RequestMapping(value= "processUser", method = RequestMethod.POST)
public String submitForm(@ModelAttribute("member") Member member, ↴
                        BindingResult result, Model model) {
    ....
}
```

The extent to which you require specifying a handler method's HTTP type depends on how and what is interacting with a controller. For the most part, web browsers perform the bulk of their operations using HTTP GET and HTTP POST requests. However, other devices or applications (e.g., RESTful web services) may require support for other HTTP request types.

In all, there are eight different HTTP request types: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS and CONNECT. However, support for handling all these request types goes beyond the scope of an MVC controller, since a web server, as well as the requesting party need to support such HTTP request types. Considering the majority of HTTP requests are of the GET or POST kind, you will rarely if ever require implementing support for these additional HTTP request types.

WHERE ARE THE URL EXTENSIONS LIKE .HTML AND .JSP ?

You might have noticed that in all the URLs specified in @RequestMapping annotations, there was no trace of a file extension like .html or .jsp. This is good practice in accordance with MVC design, even though it's not widely adopted.

A controller should not be tied to any type of extension that is indicative of a view technology, like HTML or JSP. This is why controllers return logical views and also why matching URLs should be declared without extensions.

In an age where it's common to have applications serve the same content in different formats, such as XML, JSON, PDF, or XLS (Excel). It should be left to a view resolver to inspect the extension provided in a request—if any—and determine which view technology to use.

In this short introduction, you've seen how a resolver is configured in an MVC's configuration file (`*-servlet.xml`) to map logical views to JSP files, all without every using a URL file extension like `.jsp`.

In later recipes, you will learn how Spring MVC uses this same non-extension URL approach to serve content using different view technologies.

4-3. Intercepting Requests with Handler Interceptors

Problem

Servlet filters defined by the Servlet API can pre-handle and post-handle every web request before and after it's handled by a servlet. You want to configure something with similar functions as filters in Spring's web application context to take advantage of the container features.

Moreover, sometimes you may want to pre-handle and post-handle web requests that are handled by Spring MVC handlers, and manipulate the model attributes returned by these handlers before they are passed to the views.

Solution

Spring MVC allows you to intercept web requests for pre-handling and post-handling through *handler interceptors*. Handler interceptors are configured in Spring's web application context, so they can make use of any container features and refer to any beans declared in the container. A handler interceptor can be registered for particular URL mappings, so it only intercepts requests mapped to certain URLs.

Each handler interceptor must implement the `HandlerInterceptor` interface, which contains three callback methods for you to implement: `preHandle()`, `postHandle()`, and `afterCompletion()`. The first and second methods are called before and after a request is handled by a handler. The second method also allows you to get access to the returned `ModelAndView` object, so you can manipulate the model attributes in it. The last method is called after the completion of all request processing (i.e., after the view has been rendered).

How It Works

Suppose you are going to measure each web request's handling time by each request handler and allow the views to show this time to the user. You can create a custom handler interceptor for this purpose:

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class MeasurementInterceptor implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);
        return true;
    }
}
```

```

public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler,
    ModelAndView modelAndView) throws Exception {
    long startTime = (Long) request.getAttribute("startTime");
    request.removeAttribute("startTime");

    long endTime = System.currentTimeMillis();
    modelAndView.addObject("handlingTime", endTime - startTime);
}

public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
}
}

```

In the `preHandle()` method of this interceptor, you record the start time and save it to a request attribute. This method should return true, allowing `DispatcherServlet` to proceed with request handling. Otherwise, `DispatcherServlet` assumes that this method has already handled the request, so `DispatcherServlet` returns the response to the user directly. Then, in the `postHandle()` method, you load the start time from the request attribute and compare it with the current time. You can calculate the total duration and then add this time to the model for passing to the view. Finally, as there is nothing for the `afterCompletion()` method to do, you can leave its body empty.

When implementing an interface, you must implement all the methods even though you may not have a need for all of them. A better way is to extend the interceptor adapter class instead. This class implements all the interceptor methods by default. You can override only the methods that you need.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.handler.HandlerInterceptorAdapter;

public class MeasurementInterceptor extends HandlerInterceptorAdapter {

    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        ...
    }

    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        ...
    }
}

```

To register an interceptor you need to modify the WebConfiguration which was created in the first recipe. You need to have it extend WebMvcConfigurerAdapter and override the addInterceptors method. The method gives you access to the InterceptorRegistry which you can use the add interceptors. The modified class looks like the following.

```
@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(measurementInterceptor());
    }

    @Bean
    public MeasurementInterceptor measurementInterceptor() {
        return new MeasurementInterceptor();
    }

    ...
}
```

Now you can show this time in welcome.jsp to verify this interceptor's functionality. As WelcomeController doesn't have much to do, you may likely see that the handling time is 0 milliseconds. If this is the case, you may add a sleep statement to this class to see a longer handling time.

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
...
<hr />Handling time : ${handlingTime} ms
</body>
</html>
```

By default HandlerInterceptors apply to all @Controllers however sometimes you want to discriminate on which controllers interceptors are applied. The namespace as well as the Java-based configuration allow for interceptors to be mapped to particular URLs. It is only a matter of configuration. Below is the Java configuration of this.

```
package com.apress.springrecipes.court.web.config;
// Other imports omitted for brevity
import com.apress.springrecipes.court.web.ExtensionInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
```

```

public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(measurementInterceptor());
        registry.addInterceptor(summaryReportInterceptor()).addPathPatterns("/reservationSummary*");
    }

    @Bean
    public MeasurementInterceptor measurementInterceptor() {
        return new MeasurementInterceptor();
    }

    @Bean
    public ExtensionInterceptor summaryReportInterceptor() { return new ExtensionInterceptor();}

    @Bean
    public ViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}

```

First there is the addition of the interceptor bean `summaryReportInterceptor`. The structure of the backing class for this bean is identical to that of the `measurementInterceptor`. (i.e., it implements the `HandlerInterceptor` interface). However, this interceptor performs logic that should be restricted to a particular controller which is mapped to the `/reservationSummary` URI.

When registering an interceptor we can specify which URLs it maps to, by default this takes a ANT-style expression. We pass this pattern into the `addPathPatterns` method, there is also an `excludePathPatterns` method which you can use to exclude the interceptor for certain URLs.

4-4. Resolving User Locales

Problem

In order for your web application to support internationalization, you have to identify each user's preferred locale and display contents according to this locale.

Solution

In a Spring MVC application, a user's locale is identified by a locale resolver, which has to implement the `LocaleResolver` interface. Spring MVC comes with several `LocaleResolver` implementations for you to resolve locales by different criteria. Alternatively, you may create your own custom locale resolver by implementing this interface.

You can define a locale resolver by registering a bean of type `LocaleResolver` in the web application context. You must set the bean name of the locale resolver to `localeResolver` for `DispatcherServlet` to auto-detect. Note that you can register only one locale resolver per `DispatcherServlet`.

How It Works

Resolving Locales by an HTTP Request Header

The default locale resolver used by Spring is `AcceptHeaderLocaleResolver`. It resolves locales by inspecting the `accept-language` header of an HTTP request. This header is set by a user's web browser according to the locale setting of the underlying operating system. Note that this locale resolver cannot change a user's locale because it is unable to modify the locale setting of the user's operating system.

Resolving Locales by a Session Attribute

Another option of resolving locales is by `SessionLocaleResolver`. It resolves locales by inspecting a predefined attribute in a user's session. If the session attribute doesn't exist, this locale resolver determines the default locale from the `accept-language` HTTP header.

```
@Bean
public LocaleResolver localeResolver () {
    SessionLocaleResolver localeResolver = new SessionLocaleResolver();
    localeResolver.setDefaultLocale(new Locale("en"));
    return localeResolver;
}
```

You can set the `defaultLocale` property for this resolver in case the session attribute doesn't exist. Note that this locale resolver is able to change a user's locale by altering the session attribute that stores the locale.

Resolving Locales by a Cookie

You can also use `CookieLocaleResolver` to resolve locales by inspecting a cookie in a user's browser. If the cookie doesn't exist, this locale resolver determines the default locale from the `accept-language` HTTP header.

```
@Bean
public LocaleResolver localeResolver() {
    return new CookieLocaleResolver();
}
```

The cookie used by this locale resolver can be customized by setting the `cookieName` and `cookieMaxAge` properties. The `cookieMaxAge` property indicates how many seconds this cookie should be persisted. The value `-1` indicates that this cookie will be invalid after the browser is closed.

```
@Bean
public LocaleResolver localeResolver() {
    CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
    cookieLocaleResolver.setCookieName("language");
    cookieLocaleResolver.setCookieMaxAge(3600);
    cookieLocaleResolver.setDefaultLocale(new Locale("en"));
    return cookieLocaleResolver;
}
```

You can also set the `defaultLocale` property for this resolver in case the cookie doesn't exist in a user's browser. This locale resolver is able to change a user's locale by altering the cookie that stores the locale.

Changing a User's Locale

In addition to changing a user's locale by calling `LocaleResolver.setLocale()` explicitly, you can also apply `LocaleChangeInterceptor` to your handler mappings. This interceptor detects if a special parameter is present in the current HTTP request. The parameter name can be customized with the `paramName` property of this interceptor. If such a parameter is present in the current request, this interceptor changes the user's locale according to the parameter value.

```
package com.apress.springrecipes.court.web.config;

import org.springframework.web.servlet.LocaleResolver;
import org.springframework.web.servlet.i18n.CookieLocaleResolver;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

import java.util.Locale;

// Other imports omitted

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(measurementInterceptor());
        registry.addInterceptor(localeChangeInterceptor());
        registry.addInterceptor(summaryReportInterceptor()).addPathPatterns("/reservationSummary*");
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor localeChangeInterceptor = new LocaleChangeInterceptor();
        localeChangeInterceptor.setParamName("language");
        return localeChangeInterceptor;
    }

    @Bean
    public LocaleResolver localeResolver() {
        CookieLocaleResolver cookieLocaleResolver = new CookieLocaleResolver();
        cookieLocaleResolver.setCookieName("language");
        cookieLocaleResolver.setCookieMaxAge(3600);
        cookieLocaleResolver.setDefaultLocale(new Locale("en"));
        return cookieLocaleResolver;
    }
    ...
}
```

Now a user's locale can be changed by any URLs with the `language` parameter. For example, the following two URLs change the user's locale to English for the United States, and to German, respectively:

```
http://localhost:8080/court/welcome?language=en_US
http://localhost:8080/court/welcome?language=de
```

Then you can show the HTTP response object's locale in `welcome.jsp` to verify the locale interceptor's configuration:

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

<html>
<head>
<title>Welcome</title>
</head>

<body>
...
<br />Locale : ${pageContext.response.locale}
</body>
</html>
```

4-5. Externalizing Locale-Sensitive Text Messages

Problem

When developing an internationalized web application, you have to display your web pages in a user's preferred locale. You don't want to create different versions of the same page for different locales.

Solution

To avoid creating different versions of a page for different locales, you should make your web page independent of the locale by externalizing locale-sensitive text messages. Spring is able to resolve text messages for you by using a message source, which has to implement the `MessageSource` interface. Then your JSP files can use the `<spring:message>` tag, defined in Spring's tag library, to resolve a message given the code.

How It Works

You can define a message source by registering a bean of type `MessageSource` in the web application context. You must set the bean name of the message source to `messageSource` for `DispatcherServlet` to auto-detect. Note that you can register only one message source per `DispatcherServlet`.

The `ResourceBundleMessageSource` implementation resolves messages from different resource bundles for different locales. For example, you can register it in `WebConfiguration` to load resource bundles whose base name is `messages`:

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

Then you create two resource bundles, `messages.properties` and `messages_de.properties`, to store messages for the default and German locales. These resource bundles should be put in the root of the classpath.

```
welcome.title=Welcome
welcome.message=Welcome to Court Reservation System

welcome.title=Willkommen
welcome.message=Willkommen zum Spielplatz-Reservierungssystem
```

Now, in a JSP file such as `welcome.jsp`, you can use the `<spring:message>` tag to resolve a message given the code. This tag automatically resolves the message according to a user's current locale. Note that this tag is defined in Spring's tag library, so you have to declare it at the top of your JSP file.

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

<html>
<head>
<title><spring:message code="welcome.title" text="Welcome" /></title>
</head>

<body>
<h2><spring:message code="welcome.message"
    text="Welcome to Court Reservation System" /></h2>
...
</body>
</html>
```

In `<spring:message>`, you can specify the default text to output when a message for the given code cannot be resolved.

4-6. Resolving Views by Names

Problem

After a handler has finished handling a request, it returns a logical view name. In which case the `DispatcherServlet` has to delegate control to a view template so the information is rendered. You want to define a strategy for `DispatcherServlet` to resolve views by their logical names.

Solution

In a Spring MVC application, views are resolved by one or more view resolver beans declared in the web application context. These beans have to implement the `ViewResolver` interface for `DispatcherServlet` to auto-detect them. Spring MVC comes with several `ViewResolver` implementations for you to resolve views using different strategies.

How It Works

Resolving Views Based on a template's name and location

The basic strategy of resolving views is to map them to a template's name and location directly. The view resolver `InternalResourceViewResolver` maps each view name to an application's directory by means of a prefix and a suffix declaration. To register `InternalResourceViewResolver`, you can declare a bean of this type in the web application context.

```
@Bean
public ViewResolver viewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

For example, `InternalResourceViewResolver` resolves the view names `welcome` and `reservationQuery` in the following way:

```
welcome '/WEB-INF/jsp/welcome.jsp
reservationQuery '/WEB-INF/jsp/reservationQuery.jsp
```

The type of the resolved views can be specified by the `viewClass` property. By default, `InternalResourceViewResolver` resolves view names into view objects of type `JstlView` if the JSTL library (i.e., `jstl.jar`) is present in the classpath. So, you can omit the `viewClass` property if your views are JSP templates with JSTL tags.

`InternalResourceViewResolver` is simple, but it can only resolve internal resource views that can be forwarded by the Servlet API's `RequestDispatcher` (e.g., an internal JSP file or a servlet). As for other view types supported by Spring MVC, you have to resolve them using other strategies.

Resolving Views from an XML Configuration File

Another strategy for resolving views is to declare them as Spring beans and resolve them by their bean names. You can declare the view beans in the same configuration file as the web application context, but it's better to isolate them in a separate configuration file. By default, `XmlViewResolver` loads view beans from `/WEB-INF/views.xml`, but this location can be overridden through the `location` property.

```
Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration extends WebMvcConfigurerAdapter implements ResourceLoaderAware {

    private ResourceLoader resourceLoader;

    @Bean
    public ViewResolver viewResolver() {
        XmlViewResolver viewResolver = new XmlViewResolver();
        viewResolver.setLocation(resourceLoader.getResource("/WEB-INF/court-views.nl"));
        return viewResolver;
    }
}
```

```

@Override
public void setResourceLoader(ResourceLoader resourceLoader) {
    this.resourceLoader=resourceLoader;
}

```

Note the implementation of the `ResourceLoaderAware` interface, we need to load resources as the `location` property takes an argument of the type `Resource`. In a Spring XML file the conversion from `String` to `Resource` is handled for us, however when using Java-based configuration we have to do some additional work.

In the `court-views.xml` configuration file, you can declare each view as a normal Spring bean by setting the class name and properties. In this way, you can declare any types of views (e.g., `RedirectView` and even custom view types).

```

<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="welcome"
          class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/welcome.jsp" />
    </bean>

    <bean id="reservationQuery"
          class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/reservationQuery.jsp" />
    </bean>

    <bean id="welcomeRedirect"
          class="org.springframework.web.servlet.view.RedirectView">
        <property name="url" value="welcome" />
    </bean>
</beans>

```

Resolving Views from a Resource Bundle

In addition to an XML configuration file, you can declare view beans in a resource bundle.

`ResourceBundleViewResolver` loads view beans from a resource bundle in the classpath root. Note that `ResourceBundleViewResolver` can also take advantage of the resource bundle capability to load view beans from different resource bundles for different locales.

```

@Bean
public ViewResolver viewResolver() {
    ResourceBundleViewResolver viewResolver = new ResourceBundleViewResolver();
    viewResolver.setBasename("court-views");
    return viewResolver;
}

```

As you specify `court-views` as the base name of `ResourceBundleViewResolver`, the resource bundle is `court-views.properties`. In this resource bundle, you can declare view beans in the format of properties. This type of declaration is equivalent to the XML bean declaration.

```
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

reservationQuery.(class)=org.springframework.web.servlet.view.JstlView
reservationQuery.url=/WEB-INF/jsp/reservationQuery.jsp

welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome
```

Resolving Views with Multiple Resolvers

If you have a lot of views in your web application, it is often insufficient to choose only one view-resolving strategy. Typically, `InternalResourceViewResolver` can resolve most of the internal JSP views, but there are usually other types of views that have to be resolved by `ResourceBundleViewResolver`. In this case, you have to combine both strategies for view resolution.

```
@Bean
public ViewResolver viewResolver() {
    ResourceBundleViewResolver viewResolver = new ResourceBundleViewResolver();
    viewResolver.setOrder(0);
    viewResolver.setBasename("court-views");
    return viewResolver;
}

@Bean
public ViewResolver internalResourceViewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setOrder(1);
    viewResolver.setPrefix("/WEB-INF/jsp/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

When choosing more than one strategy at the same time, it's important to specify the resolving priority. You can set the `order` properties of the view resolver beans for this purpose. The lower order value represents the higher priority. Note that you should assign the lowest priority to `InternalResourceViewResolver` because it always resolves a view no matter whether it exists or not. So, other resolvers will have no chance to resolve a view if they have lower priorities.

Now the resource bundle `court-views.properties` should only contain the views that can't be resolved by `InternalResourceViewResolver` (e.g., the redirect views):

```
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome
```

The Redirect Prefix

If you have `InternalResourceViewResolver` configured in your web application context, it can resolve redirect views by using the `redirect` prefix in the view name. Then the rest of the view name is treated as the redirect URL. For example, the view name `redirect:welcome` triggers a redirect to the relative URL `welcome`. You may also specify an absolute URL in the view name.

4-7. Views and Content Negotiation

Problem

You are relying on extension-less URLs in your controllers—`welcome` and not `welcome.html` or `welcome.pdf`. You want to devise a strategy so the correct content and type is returned for all requests.

Solution

When a request is received for a web application, it contains a series of properties that allow the processing framework, in this case Spring MVC, to determine the correct content and type to return to the requesting party. The main two properties include:

- The URL extension provided in a request
- The HTTP Accept header

For example, if a request is made to a URL in the form `/reservationSummary.xml`, a controller is capable of inspecting the extension and delegating it to a logical view representing an XML view.

However, the possibility can arise for a request to be made to a URL in the form `/reservationSummary`. Should this request be delegated to an XML view or an HTML view? Or perhaps some other type of view? It's impossible to tell through the URL. But instead of deciding on a default view for such requests, a request can be inspected for its HTTP Accept header to decide what type of view is more appropriate.

Inspecting HTTP Accept headers in a controller can be a messy process. So Spring MVC supports the inspection of headers through the `ContentNegotiatingViewResolver` allowing view delegation to be made based on either a URL file extension or HTTP Accept header value.

How It Works

The first thing you need to realize about Spring MVC content negotiation is that it's configured as a resolver, just like those illustrated in the previous recipe "Resolving Views by Names."

The Spring MVC content negotiating resolver is based on the `ContentNegotiatingViewResolver` class. But before we describe how it works, we will illustrate how to configure and integrate it with other resolvers.

```
@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Autowired
    private ContentNegotiationManager contentNegotiationManager;
```

```

@Override
public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
    Map<String, MediaType> mediatypes = new HashMap<>();
    mediatypes.put("html", MediaType.TEXT_HTML);
    mediatypes.put("pdf", new MediaType("application/pdf"));
    mediatypes.put("xls", new MediaType("application/vnd.ms-excel"));
    mediatypes.put("xml", MediaType.APPLICATION_XML);
    mediatypes.put("json", MediaType.APPLICATION_JSON);
    configurer.mediaTypes(mediatypes);
}

@Bean
public ContentNegotiatingViewResolver contentNegotiatingViewResolver() {
    ContentNegotiatingViewResolver viewResolver = new ContentNegotiatingViewResolver();
    viewResolver.setContentNegotiationManager(contentNegotiationManager);
    return viewResolver;
}

```

First of all we need to configure content negotiation, the default configuration adds a `ContentNegotiationManager`, which can be configured by implementing the `configureContentNegotiation` method. To get access to the configured `ContentNegotiationManager` we can simply autowire it in our configuration class.

Turning our attention back to the `ContentNegotiatingViewResolver` resolver. This configuration sets up the resolver to have the highest priority among all resolvers, which **is necessary to make the content negotiating resolver work**. The reason for this resolver having the highest priority is that it does not resolve views themselves, but rather delegates them to other view resolvers (which it automatically detects). Since a resolver that does not resolve views can be confusing, we will elaborate with an example.

Let's assume a controller receives a request for `/reservationSummary.xml`. Once the handler method finishes, it sends control to a logical view named `reservation`. At this point Spring MVC resolvers come into play, the first of which is the `ContentNegotiatingViewResolver` resolver, since it has the highest priority.

The `ContentNegotiatingViewResolver` resolver first determines the media type for a request based on the following criteria:

It checks a request path extension (e.g., `.html`, `.xml`, or `.pdf`) against the default media types (e.g., `text/html`) specified by the `mediaTypes` map in the configuration of the `ContentNegotiatingManager` bean.

If a request path has an extension but no match can be found in the default `mediaTypes` section attempt is made to determine an extension's media type using `FileTypeMap` belonging to Java Activation Framework.

If no extension is present in a request path, the HTTP `Accept` header of the request is used.

For the case of a request made on `/reservationSummary.xml`, the media type is determined in step 1 to be `application/xml`. However, for a request made on a URL like `/reservationSummary`, the media type is not determined until step 3.

The HTTP `Accept` header contains values like `Accept: text/html` or `Accept:application/pdf`, these values help the resolver determine the media type a requester is expecting, given that no extension is present in the requesting URL.

At this juncture, the `ContentNegotiatingViewResolver` resolver has a media type and logical view named `reservation`. Based on this information, an iteration is performed over the remaining resolvers—based on their order—to determine what view best matches the logical name based on the detected media type.

This process allows you to have multiple logical views with the same name, each supporting a different media type (e.g., HTML, PDF, or XLS), with `ContentNegotiatingViewResolver` resolving which is the best match.

In such cases a controller's design is further simplified, since it won't be necessary to hard-code the logical view necessary to create a certain media type (e.g., `pdfReservation`, `xlsReservation`, or `htmlReservation`), but instead a single view (e.g., `reservation`), letting the `ContentNegotiatingViewResolver` resolver determine the best match.

A series of outcomes for this process can be the following:

- The media type is determined to be `application/pdf`. If the resolver with the highest priority (lower order) contains a mapping to a logical view named `reservation`, but such a view does not support the `application/pdf` type, no match occurs—the lookup process continues onto the remaining resolvers.
- The media type is determined to be `application/pdf`. The resolver with the highest priority (lower order) containing a mapping to a logical view named `reservation` and having support for `application/pdf` is matched.
- The media type is determined to be `text/html`. There are four resolvers with a logical view named `reservation`, but the views mapped to the two resolvers with highest priority do not support `text/html`. It's the remaining resolver containing a mapping for a view named `reservation` that supports `text/html` that is matched.

This search process for views automatically take place on all the resolvers configured in an application. It's also possible to configure—within the `ContentNegotiatingViewResolver` bean—default views and resolvers, in case you don't want to fall-back on configurations made outside the `ContentNegotiatingViewResolver` resolver.

Recipe 4-13, “Creating Excel and PDF Views,” will illustrate a controller that relies on the `ContentNegotiatingViewResolver` resolver to determine an application’s views.

4-8. Mapping Exceptions to Views

Problem

When an unknown exception occurs, your application server usually displays the evil exception stack trace to the user. Your users have nothing to do with this stack trace and complain that your application is not user friendly. Moreover, it's also a potential security risk, as you may expose the internal method call hierarchy to users.

Though a web application's `web.xml` can be configured to display friendly JSP pages in case an HTTP error or class exception occur, Spring MVC supports a more robust approach to managing views for class exceptions.

Solution

In a Spring MVC application, you can register one or more exception resolver beans in the web application context to resolve uncaught exceptions. These beans have to implement the `HandlerExceptionResolver` interface for `DispatcherServlet` to auto-detect them. Spring MVC comes with a simple exception resolver for you to map each category of exceptions to a view.

How It Works

Suppose your reservation service throws the following exception due to a reservation not being available:

```
package com.apress.springrecipes.court.service;
...
public class ReservationNotFoundException extends RuntimeException {
    private String courtName;
    private Date date;
    private int hour;
    // Constructors and Getters
    ...
}
```

To resolve uncaught exceptions, you can write your custom exception resolver by implementing the `HandlerExceptionResolver` interface. Usually, you'll want to map different categories of exceptions into different error pages. Spring MVC comes with the exception resolver `SimpleMappingExceptionResolver` for you to configure the exception mappings in the web application context. For example, you can register the following exception resolver in `WebConfiguration`:

```

@Override
public void configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
exceptionResolvers) {
    exceptionResolvers.add(handlerExceptionResolver());
}

@Bean
public HandlerExceptionResolver handlerExceptionResolver() {
    Properties exceptionMapping = new Properties();
    exceptionMapping.setProperty(
        ReservationNotAvailableException.class.getName(),
        "reservationNotAvailable");

    SimpleMappingExceptionResolver exceptionResolver = new SimpleMappingExceptionResolver();
    exceptionResolver.setExceptionMappings(exceptionMapping);
    exceptionResolver.setDefaultValue("error");
    return exceptionResolver;
}

```

In this exception resolver, you define the logical view name `reservationNotAvailable` for `ReservationNotAvailableException`. You can add any number of exception classes using the `exceptionMappings` property, all the way down to the more general exception class `java.lang.Exception`. In this manner, depending on the type of class exception, a user is served a view in accordance with the exception.

The property `defaultValue`, is used to define a default view named `error`, used in case an exception class not mapped in the `exceptionMapping` element is raised.

Addressing the corresponding views, if the `InternalResourceViewResolver` is configured in your web application context, the following `reservationNotAvailable.jsp` page is shown in case of a reservation not being available:

```

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

<html>
<head>
<title>Reservation Not Available</title>
</head>

<body>
Your reservation for ${exception.courtName} is not available on
<fmt:formatDate value="${exception.date}" pattern="yyyy-MM-dd" />at
${exception.hour}:00.
</body>
</html>

```

In an error page, the exception instance can be accessed by the variable `${exception}`, so you can show the user more details on this exception.

It's a good practice to define a default error page for any unknown exceptions. You can use `<property name="defaultErrorView" value="error"/>` to define a default view or map a page to the key `java.lang.Exception`

as the last entry of the mapping, so it will be shown if no other entry has been matched before. Then you can create this view's JSP—`error.jsp`—as follows:

```
<html>
<head>
<title>Error</title>
</head>

<body>
An error has occurred. Please contact our administrator for details.
</body>
</html>
```

Mappings exceptions using `@ExceptionHandler`

Instead of configuring a `HandlerExceptionResolver` we can also annotate a method with `@ExceptionHandler`. It works in a similar way as the `@RequestMapping` annotation.

```
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation") // Command name class was used in earlier Spring versions
public class ReservationFormController {
    @ExceptionHandler(ReservationNotAvailableException.class)
    public String handle(ReservationNotAvailableException ex) {
        return "reservationNotAvailable";
    }

    @ExceptionHandler
    public String handleDefault(Exception e) {
        return "error";
    }
    ...
}
```

We have here two methods annotated `@ExceptionHandler`. The first is for handling the specific `ReservationNotAvailableException`, the second is the general (catch-all) exception handling method. You also don't have to specify a `HandlerExceptionResolver` in the `WebConfiguration` anymore.

Methods annotated with `@ExceptionHandler` can have a variety of return types (like the `@RequestMapping` methods), here we just return the name of the view which needs to be rendered, but we could also have returned a `ModelAndView`, a `View` etc.

Although using `@ExceptionHandler` annotated methods is very powerful and flexible there is a drawback when you put them in controllers. Those methods will only work in the controller they are defined in, so if we have an exception occurring in another controller (for instance the `WelcomeController`) these methods won't be called. Generic exception handling methods have to be moved to a separate class and that class has to be annotated with `@ControllerAdvice`

```
@ControllerAdvice
public class ExceptionHandlingAdvice {

    @ExceptionHandler(ReservationNotAvailableException.class)
    public String handle(ReservationNotAvailableException ex) {
        return "reservationNotAvailable";
    }
}
```

```

@ExceptionHandler
public String handleDefault(Exception e) {
    return "error";
}
}

```

This class will apply to all controllers in the application context, hence the name @ControllerAdvice.

4-9. Handling Forms with Controllers

Problem

In a web application, you often have to deal with forms. A form controller has to show a form to a user and also handle the form submission. Form handling can be a complex and variable task.

Solution

When a user interacts with a form, it requires support for two operations from a controller. First when a form is initially requested, it asks the controller to show a form by an HTTP GET request, that renders the form view to the user. Then when the form is submitted, an HTTP POST request is made to handle things like validation and business processing for the data present in the form.

If the form is handled successfully, it renders the success view to the user. Otherwise, it renders the form view again with errors.

How It Works

Suppose you want to allow a user to make a court reservation by filling out a form. To give you a better idea of the data handled by a controller, we will introduce the controller's view (i.e., the form) first.

Creating a form's views

Let's create the form view `reservationForm.jsp`. The form relies on Spring's form tag library, as this simplifies a form's data binding, display of error messages and the re-display of original values entered by the user in case of errors.

```

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

<html>
<head>
<title>Reservation Form</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

```

```

<body>
<form:form method="post" modelAttribute="reservation">
<form:errors path="*" cssClass="error" />
<table>
<tr>
<td>Court Name</td>
<td><form:input path="courtName" /></td>
<td><form:errors path="courtName" cssClass="error" /></td>
</tr>
<tr>
<td>Date</td>
<td><form:input path="date" /></td>
<td><form:errors path="date" cssClass="error" /></td>
</tr>
<tr>
<td>Hour</td>
<td><form:input path="hour" /></td>
<td><form:errors path="hour" cssClass="error" /></td>
</tr>
<tr>
<td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form:form>
</body>
</html>

```

The Spring `<form:form>` declares two attributes. The `method="post"` attribute used to indicate a form performs an HTTP POST request upon submission. And the `modelAttribute="reservation"` attribute used to indicate the form data is bound to a model named `reservation`. The first attribute should be familiar to you since it's used on most HTML forms. The second attribute will become clearer once we describe the controller that handles the form.

Bear in mind the `<form:form>` tag is rendered into a standard HTML before it's sent to a user, so it's not that the `modelAttribute="reservation"` is of use to a browser, the attribute is used as facility to generate the actual HTML form.

Next, you can find the `<form:errors>` tag, used to define a location in which to place errors in case a form does not meet the rules set forth by a controller. The attribute `path="*"` is used to indicate the display of all errors—given the wildcard `*`—where as the attribute `cssClass="error"` is used to indicate a CSS formatting class to display the errors.

Next, you can find the form's various `<form:input>` tags accompanied by another set of corresponding `<form:errors>` tags. These tags make use of the attribute `path` to indicate the form's fields, which in this case are `courtName`, `date` and `hour`.

The `<form:input>` tags are bound to properties corresponding to the `modelAttribute` by using the `path` attribute. They show the user the original value of the field, which will either be the bound property value or the value rejected due to a binding error. They must be used inside the `<form:form>` tag, which defines a form that binds to the `modelAttribute` by its name.

Finally, you can find the standard HTML tag `<input type="submit" />` that generates a 'Submit' button and trigger the sending of data to the server, followed by the `</form:form>` tag that closes out the form.

In case the form and its data are processed correctly, you need to create a success view to notify the user of a successful reservation. The `reservationSuccess.jsp` illustrated next serves this purpose.

```

<html>
<head>
<title>Reservation Success</title>
</head>

```

```
<body>
Your reservation has been made successfully.
</body>
</html>
```

It's also possible for errors to occur due to invalid values being submitted in a form. For example, if the date is not in a valid format, or an alphabetic character is presented for the hour, the controller is designed to reject such field values. The controller will then generate a list of selective error codes for each error to be returned to the form view, values that are placed inside the `<form:errors>` tag.

For example, for an invalid value input in the date field, the following error codes are generated by a controller:

```
typeMismatch.command.date
typeMismatch.date
typeMismatch.java.util.Date
typeMismatch
```

If you have a `ResourceBundleMessageSource` defined, you can include the following error messages in your resource bundle for the appropriate locale (e.g., `messages.properties` for the default locale):

```
typeMismatch.date=Invalid date format
typeMismatch.hour=Invalid hour format
```

The corresponding error codes and their values are what is returned to a user if a failure occurs processing form data.

Now that you know the structure of the views involved with a form, as well as the data handled by it, let's take a look at the logic that handles the submitted data (i.e., the reservation) in a form.

Creating a form's service processing

This is not the controller, but rather the service used by the controller to process the form's data reservation. First define a `make()` method in the `ReservationService` interface:

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException;
}
```

Then you implement this `make()` method by adding a `Reservation` item to the list that stores the reservations. You throw a `ReservationNotAvailableException` in case of a duplicate reservation.

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException {
        for (Reservation made : reservations) {
            if (made.getCourtName().equals(reservation.getCourtName()))
```

```
        && made.getDate().equals(reservation.getDate())
        && made.getHour() == reservation.getHour()) {
    throw new ReservationNotAvailableException(
        reservation.getCourtName(), reservation.getDate(),
        reservation.getHour());
}
}
reservations.add(reservation);
}
}
```

Now that you have a better understanding of the two elements that interact with a controller—a form's views and the reservation service class—let's create a controller to handle the court reservation form.

Creating a form's controller

A controller used to handle forms makes use of practically the same annotations you've already used in the previous recipes. So let's get right to the code.

```
package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {

    private ReservationService reservationService;

    @Autowired
    public ReservationFormController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(Model model) {
        Reservation reservation = new Reservation();
        model.addAttribute("reservation", reservation);
        return "reservationForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") Reservation reservation,
        BindingResult result, SessionStatus status) {
        reservationService.make(reservation);
        return "redirect:reservationSuccess";
    }
}
```

The controller starts by using the standard `@Controller` annotation, as well as the `@RequestMapping` annotation that allows access to the controller through the following URL:

```
http://localhost:8080/court/reservationForm
```

When you enter this URL in your browser, it will send an HTTP GET request to your web application. This in turn triggers the execution of the `setupForm` method, which is designated to attend this type of request based on its `@RequestMapping` annotation.

The `setupForm` method defines a `Model` object as an input parameter, which serves to send model data to the view (i.e., the form). Inside the handler method, an empty `Reservation` object is created that is added as an attribute to the controller's `Model` object. Then the controller returns the execution flow to the `reservationForm` view, which in this case is resolved to `reservationForm.jsp` (i.e., the form).

The most important aspect of this last method is the addition of empty `Reservation` object. If you analyze the form `reservationForm.jsp`, you will notice the `<form:form>` tag declares an attribute `modelAttribute="reservation"`. This means that upon rendering the view, the form expects an object named `reservation` to be available, which is achieved by placing it inside the handler method's `Model`. In fact further inspection reveals that the path values for each `<form:input>` tag correspond to the field names belonging to the `Reservation` object. Since the form is being loaded for the first time, it should be evident that an empty `Reservation` object is expected.

Another aspect that is vital to describe prior to analyzing the other controller handler method is the `@SessionAttributes("reservation")` annotation—declared at the top of the controller class. Since it's possible for a form to contain errors, it can be an inconvenience to lose whatever valid data was already provided by a user on every subsequent submission. To solve this problem, the `@SessionAttributes` is used to save a `reservation` field to a user's session, so that any future reference to the `reservation` field is in fact made on the same reference, whether a form is submitted twice or more times. This is also the reason why only a single `Reservation` object is created and assigned to the `reservation` field in the entire controller. Once the empty `Reservation` object is created—inside the HTTP GET handler method—all actions are made on the same object, since it's assigned to a user's session.

Now let's turn our attention to submitting the form for the first time. After you have filled in the form fields, submitting the form triggers an HTTP POST request, that in turn invokes the `submitForm` method—on account of this method's `@RequestMapping` value.

The input fields declared for the `submitForm` method are three. The `@ModelAttribute("reservation")` `Reservation reservation` used to reference the `reservation` object. The `BindingResult` object that contains newly submitted data by the user. And the `SessionStatus` object used in case it's necessary to access a user's session.

At this juncture, the handler method doesn't incorporate validation or perform access to a user's session, which is the purpose of the `BindingResult` object and `SessionStatus` object—I will describe and incorporate them shortly.

The only operation performed by the handler method is `reservationService.make(reservation);`. This operation invokes the `reservation` service using the current state of the `reservation` object. Generally, controller objects are first validated prior to performing this type of operation on them.

Finally, note the handler method returns a view named `redirect:reservationSuccess`. The actual name of the view in this case is `reservationSuccess`, which is resolved to the `reservationSuccess.jsp` page you created earlier.

The `redirect:` prefix in the view name is used to avoid a problem known as *duplicate form submission*.

When you refresh the web page in the form success view, the form you just submitted is resubmitted again. To avoid this problem, you can apply the *post/redirect/get* design pattern, which recommends redirecting to another URL after a form submission is handled successfully, instead of returning an HTML page directly. This is the purpose of prefixing a view name with `redirect:..`.

Initializing a model attribute object and pre-populating a form with values

The form is designed to let users make reservations. However, if you analyze the `Reservation` domain class, you will note the form is still missing two fields in order to create a complete `reservation` object. One of these fields is the `player` field, which corresponds to a `Player` object. Per the `Player` class definition, a `Player` object has both a `name` and `phone` fields.

So can the player field be incorporated into a form view and controller? Let's analyze the form view first:

```
<html>
<head>
<title>Reservation Form</title>
</head>
<body>
<form method="post" modelAttribute="reservation">
<table>
...
<tr>
<td>Player Name</td>
<td><form:input path="player.name" /></td>
<td><form:errors path="player.name" cssClass="error" /></td>
</tr>
<tr>
<td>Player Phone</td>
<td><form:input path="player.phone" /></td>
<td><form:errors path="player.phone" cssClass="error" /></td>
</tr>
<tr>
<td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

Using a straightforward approach, you add two additional `<form:input>` tags used to represent the Player object's fields. Though these forms declaration are simple, you also need to perform modifications to the controller. Recall that by using `<form:input>` tags, a view expects to have access to model objects passed by the controller, that match the path value for `<form:input>` tags.

Though the controller's HTTP GET handler method returns an empty reservation Reservation to this last view, the player property is null, so it causes an exception when rendering the form. To solve this problem, you have to initialize an empty Player object and assign it to the Reservation object returned to the view.

```
@RequestMapping(method = RequestMethod.GET)
public String setupForm(
    @RequestParam(required = false, value = "username") String username, Model model) {
    Reservation reservation = new Reservation();
    reservation.setPlayer(new Player(username, null));
    model.addAttribute("reservation", reservation);
    return "reservationForm";
}
```

In this case, after creating the empty Reservation object, the `setPlayer` method is used to assign it an empty Player object.

Further note that the creation of the Person object relies on the `username` value. This particular value is obtained from the `@RequestParam` input value which was also added to the handler method. By doing so, the Player object can be created with a specific `username` value passed in as a request parameter, resulting in the `username` form field being pre-populated with this value.

So for example, if a request to the form is made in the following manner:

```
http://localhost:8080/court/reservationForm?username=Roger
```

This allows the handler method to extract the `username` parameter to create the `Player` object, in turn pre-populating the form's `username` form field with a `Roger` value. It's worth noting that the `@RequestParam` annotation for the `username` parameter uses the property `required=false`, this allows a form request to be processed even if such a request parameter is not present.

Providing form Reference Data

When a form controller is requested to render the form view, it may have some types of reference data to provide to the form (e.g., the items to display in an HTML selection). Now suppose you want to allow a user to select the sport type when reserving a court—which is the final unaccounted field for the `Reservation` class.

```
<html>
<head>
<title>Reservation Form</title>
</head>

<body>
<form method="post" modelAttribute="reservation">
<table>
...
<tr>
    <td>Sport Type</td>
    <td>
        <form:select path="sportType" items="${sportTypes}"
                     itemValue="id" itemLabel="name" />
    </td>
    <td><form:errors path="sportType" cssClass="error" /></td>
</tr>
<tr>
    <td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

The `<form:select>` tag provides a way to generate a drop-down list of values passed to the view by the controller. Thus the form represents the `sportType` field as a set of HTML `<select>` elements, instead of the previous open-ended fields—`<input>`—that require a user to introduce text values.

Next, let's take a look at how the controller assigns the `sportType` field as a model attribute, the process is a little different than the previous fields.

First let's define the `getAllSportTypes()` method in the `ReservationService` interface for retrieving all available sport types:

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public List<SportType> getAllSportTypes();
}
```

Then you can implement this method by returning a hard-coded list:

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");

    public List<SportType> getAllSportTypes() {
        return Arrays.asList(new SportType[] { TENNIS, SOCCER });
    }
}
```

Now that you have an implementation that returns a hard-coded list of `SportType` objects, let's take a look at how the controller associates this list for it to be returned to the form view.

```
package com.apress.springrecipes.court.service;
....
@ModelAttribute("sportTypes")
public List<SportType> populateSportTypes() {
    return reservationService.getAllSportTypes();
}

@RequestMapping(method = RequestMethod.GET)
public String setupForm(
    @RequestParam(required = false, value = "username") String username, Model model) {
    Reservation reservation = new Reservation();
    reservation.setPlayer(new Player(username, null));
    model.addAttribute("reservation", reservation);
    return "reservationForm";
}
```

Notice that the `setupForm` handler method charged with returning the empty `Reservation` object to the form view remains unchanged.

The new addition and what is responsible for passing a `SportType` list as a model attribute to the form view is the method decorated with the `@ModelAttribute("sportTypes")` annotation.

The `@ModelAttribute` annotation is used to define global model attributes, available to any returning view used in handler methods. In the same way a handler method declares a `Model` object as an input parameter and assigns attributes that can be accessed in the returning view.

Since the method decorated with the `@ModelAttribute("sportTypes")` annotation has a return type of `List<SportType>` and makes a call to `reservationService.getAllSportTypes()`, the hard-coded TENNIS and SOCCER `SportType` objects are assigned to the model attribute named `sportTypes`. With this last model attribute used in the form view to populate a drop down list (i.e., `<form:select>` tag).

Binding Properties of Custom Types

When a form is submitted, a controller binds the form field values to model object's properties of the same name, in this case a `Reservation` object. However, for properties of custom types, a controller is not able to convert them unless you specify the corresponding property editors for them.

For example, the sport type selection field only submits the selected sport type ID—as this is the way HTML `<select>` fields operate. Therefore, you have to convert this ID into a `SportType` object with a property editor. First of all, you require the `getSportType()` method in `ReservationService` to retrieve a `SportType` object by its ID:

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public SportType getSportType(int sportTypeId);
}
```

For testing purposes, you can implement this method with a switch/case statement:

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public SportType getSportType(int sportTypeId) {
        switch (sportTypeId) {
            case 1:
                return TENNIS;
            case 2:
                return SOCCER;
            default:
                return null;
        }
    }
}
```

Then you create the `SportTypeConverter` class to convert a sport type ID into a `SportType` object. This converter requires `ReservationService` to perform the lookup.

```
package com.apress.springrecipes.court.domain;

import com.apress.springrecipes.court.service.ReservationService;
import org.springframework.core.convert.converter.Converter;

public class SportTypeConverter implements Converter<String, SportType> {

    private ReservationService reservationService;
```

```

public SportTypeConverter(ReservationService reservationService) {
    this.reservationService = reservationService;
}

@Override
public SportType convert(String source) {
    int sportTypeId = Integer.parseInt(source);
    SportType sportType = reservationService.getSportType(sportTypeId);
    return sportType;
}
}

```

Now that you have the supporting `SportTypeConverter` class required to bind form properties to a custom class like `SportType`, you need to associate it with the controller. For this purpose, we can use the `addFormatters` method from the `WebMvcConfigurer`.

By overriding this method in our `WebConfiguration` class custom types can be associated with a controller. This includes the `SportTypeConverter` class and other custom types like `Date`.

Though we didn't mention the date field earlier, it suffers from the same problem as the sport type selection field. A user introduces date fields as text values. In order for the controller to assign these text values to the `Reservation` object's date field, this requires the date fields be associated with a `Date` object. Given the `Date` class is part of the Java language, it won't be necessary to create special a class like `SportTypeConverter` for this purpose, the Spring framework already includes a custom class for this purpose.

Knowing you need to bind both the `SportTypeConverter` class and a `Date` class to the underlying controller, the following listing illustrates the modifications to the `WebConfiguration` class.

```

package com.apress.springrecipes.court.web.config;
...
import com.apress.springrecipes.court.domain.SportTypeConverter;
import com.apress.springrecipes.court.service.ReservationService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.format.FormatterRegistry;
import org.springframework.format.datetime.DateFormatter;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.court.web")
public class WebConfiguration extends WebMvcConfigurerAdapter {

    @Autowired
    private ReservationService reservationService;
    ...
    @Override
    public void addFormatters(FormatterRegistry registry) {
        registry.addFormatterForFieldType(Date.class, new DateFormatter("yyyy-MM-dd"));
        registry.addConverter(new SportTypeConverter(reservationService));
    }
}

```

The only field for this last class corresponds to `reservationService`, used to access the application's `ReservationService` bean. Note the use of the `@Autowired` annotation that enables the injection of the bean.

Next, you can find the `addFormatters` method used to bind the `Date` and `SportTypeConverter` classes. You can then find two calls to register the converter and formatter. These methods belong to the `FormatterRegistry` object, which is passed as an input parameter to `addFormatters` method.

The first call is used to bind a `Date` class to the `DateFormatter` class. The `DateFormatter` class is provided by the Spring framework and offers functionality to parse and print `Date` objects.

The second call is used to register the `SportTypeConverter` class. Since you created the `SportTypeConverter` class, you should be familiar that its only input parameter is a `ReservationService` bean.

By using this approach, every annotation-based controller (i.e., classes using the `@Controller` annotation) can have access to the same custom converters and formatters in their handler methods.

Validating Form Data

When a form is submitted, it's standard practice to validate the data provided by a user before a submission is successful. Spring MVC supports validation by means of a validator object that implements the `Validator` interface. You can write the following validator to check if the required form fields are filled, and if the reservation hour is valid on holidays and weekdays:

```
package com.apress.springrecipes.court.domain;
...
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import org.springframework.stereotype.Component;

public class ReservationValidator implements Validator {

    public boolean supports(Class clazz) {
        return Reservation.class.isAssignableFrom(clazz);
    }

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Court name is required.");
        ValidationUtils.rejectIfEmpty(errors, "date",
            "required.date", "Date is required.");
        ValidationUtils.rejectIfEmpty(errors, "hour",
            "required.hour", "Hour is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
            "required.playerName", "Player name is required.");
        ValidationUtils.rejectIfEmpty(errors, "sportType",
            "required.sportType", "Sport type is required.");

        Reservation reservation = (Reservation) target;
        Date date = reservation.getDate();
        int hour = reservation.getHour();
        if (date != null) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(date);
        }
    }
}
```

```
        if (calendar.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) {
            if (hour < 8 || hour > 22) {
                errors.reject("invalid.holidayHour", "Invalid holiday hour.");
            }
        } else {
            if (hour < 9 || hour > 21) {
                errors.reject("invalid.weekdayHour", "Invalid weekday hour.");
            }
        }
    }
}
```

In this validator, you use utility methods such as `rejectIfEmptyOrWhitespace()` and `rejectIfEmpty()` in the `ValidationUtils` class to validate the required form fields. If any of these form fields is empty, these methods will create a *field error* and bind it to the field. The second argument of these methods is the property name, while the third and fourth are the error code and default error message.

You also check whether the reservation hour is valid on holidays and weekdays. In case of invalidity, you should use the `reject()` method to create an *object error* to be bound to the reservation object, not to a field.

Since the validator class is annotated with the `@Component` annotation, Spring attempts to instantiate the class as a bean in accordance with the class name, in this case `reservationValidator`.

We need to register the validator as a bean in our context, for this we can simply add a @Bean annotated method to our WebConfiguration.

```
@Bean  
public ReservationValidator reservationValidator() {  
    return new ReservationValidator();  
}
```

Since validators may create errors during validation, you should define messages for the error codes for displaying to the user. If you have `ResourceBundleMessageSource` defined, you can include the following error messages in your resource bundle for the appropriate locale (e.g., `messages.properties` for the default locale):

```
required.courtName=Court name is required  
required.date=Date is required  
required.hour=Hour is required  
required.playerName=Player name is required  
required.sportType=Sport type is required  
invalid.holidayHour=Invalid holiday hour  
invalid.weekdayHour=Invalid weekday hour
```

To apply this validator, you need to perform the following modification to your controller:

```
package com.apress.springrecipes.court.service;
.....
    private ReservationService reservationService;
private ReservationValidator reservationValidator;

    @Autowired
    public ReservationFormController(ReservationService reservationService,
                                    ReservationValidator reservationValidator) {
```

```

        this.reservationService = reservationService;
this.reservationValidator = reservationValidator;
}

@RequestMapping(method = RequestMethod.POST)
public String submitForm(
    @ModelAttribute("reservation") @Validated Reservation reservation,
    BindingResult result, SessionStatus status) {
    if (result.hasErrors()) {
        return "reservationForm";
    } else {
        reservationService.make(reservation);
        return "redirect:reservationSuccess";
    }
}

@InitBinder
public void initBinder(WebDataBinder binder) {
    binder.setValidator(reservationValidator);
}

```

The first addition to the controller is the `ReservationValidator` field, that gives the controller access to an instance of the validator bean. By relying on the `@Autowired` annotation, a `ReservationValidator` bean is injected along with the pre-existing `ReservationService` bean.

The next modification takes place in the HTTP POST handler method, which is always called when a user submits a form. Next to the `@ModelAttribute` annotation there is now a `@Validated` annotation, this annotation triggers validation of the object. After the validation, the `result` parameter—`BindingResult` object—contains the results for the validation process. So next, a conditional based on the value of `result.hasErrors()` is made. If the validation class detects errors this value is true.

In case errors are detected in the validation process the method handler returns the view `reservationForm`, which corresponds to the same form that so a user can re-submit information. In case no errors are detected in the validation process, a call is made to perform the reservation—`reservationService.make(reservation)`;—followed by a redirection to the success view `reservationSuccess`.

The registration of the validator is done in the `@InitBinder` annotated method, the validator is set on the `WebDataBinder` so that it can be used after binding. To register the validator one needs to use the `setValidator` method.

Note The `WebDataBinder` can also be used to register additional `PropertyEditors` for type conversion. This can be used instead of registering global `PropertyEditors`, `Converters`, or `Formatters`.

Epiring a controller's Session Data

In order to support the possibility of a form being submitted multiple times and not loose data provided by a user in between submissions, the controller relies on the use of the `@SessionAttributes` annotation. By doing so, a reference to the `reservation` field represented as a `Reservation` object is saved between requests.

However, once a form is submitted successfully and a reservation is made, there is no point in keeping the `Reservation` object in a user's session. In fact, if a user revisits the form within a short period of time, there is a possibility remnants of this old `Reservation` object emerge if not removed.

Values assigned using the `@SessionAttributes` annotation can be removed using the `SessionStatus` object, an object that can be passed as an input parameter to handler methods. The following listing illustrates how to expire the controller's session data.

```
package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/reservationForm")
@SessionAttributes("reservation")
public class ReservationFormController {
    ...
    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("reservation") Reservation reservation,
        BindingResult result, SessionStatus status) {

        if (result.hasErrors()) {
            return "reservationForm";
        } else {
            reservationService.make(reservation);
            status.setComplete();
            return "redirect:reservationSuccess";
        }
    }
}
```

Once the handler method performs the reservation by calling `reservationService.make(reservation)`; and right before a user is redirected to a success page, it becomes an ideal time in which to expire a controller's session data. This is done by calling the `setComplete()` method on the `SessionStatus` object. It's that simple.

4-10. Bean validation with Annotations (JSR-303)

Problem

You want to validate Java beans in a web application using annotations based on the JSR-303 standard.

Solution

JSR-303 or bean validation is a specification whose objective is to standardize the validation of Java beans through annotations.

In the previous examples, you saw how the Spring framework supports an ad-hoc technique for validating beans. This requires you to extend one of the Spring framework's classes to create a validator class for a particular type of Java bean.

The objective of the JSR-303 standard is to use annotations directly in a Java bean class. This allows validation rules to be specified directly in the code they are intended to validate, instead of creating validation rules in separate classes—just like you did earlier using Spring a framework class.

How It Works

The first thing you need to do is decorate a Java bean with the necessary JSR-303 annotations. The following listing illustrates the `Reservation` domain class used in the court application decorated with JSR-303 annotations:

```
public class Reservation {

    @NotNull
    @Size(min = 4)
    private String courtName;

    @NotNull
    private Date date;

    @Min(9)
    @Max(21)
    private int hour;

    @Valid
    private Player player;

    @NotNull
    private SportType sportType;

    // Getter/Setter methods omitted for brevity
}
```

The `courtName` field is assigned two annotations. The `@NotNull` annotation, which indicates that a field cannot be null and the `@Size` annotation used to indicate a field has to have a minimum of two characters. The `date` and `sportType` fields are annotated with `@NotNull` as those are required.

The `hour` field is annotated with `@Min` and `@Max` because those are the lower and upper limits of the `hour` field.

Both the fields in the `Player` domain class are annotated with `@NotNull` to also trigger validation of the related object we have annotated it with `@Valid`.

Now that you know how a Java bean class is decorated with annotations belonging to the JSR-303 standard. Let's take a look at how these validator annotations are enforced in a controller.

```
package com.apress.springrecipes.court.service;
.....
private ReservationService reservationService;

@Autowired
public ReservationFormController(ReservationService reservationService) {
    this.reservationService = reservationService;
}

@RequestMapping(method = RequestMethod.POST)
public String submitForm(
    @ModelAttribute("reservation") @Valid Reservation reservation,
    BindingResult result, SessionStatus status) {
    if (result.hasErrors()) {
        return "reservationForm";
```

```

    } else {
        reservationService.make(reservation);
        return "redirect:reservationSuccess";
    }
}

```

The controller is almost similar to the one from recipe 4.9. The only difference is the absence of the `@InitBinder` annotated method. Spring MVC detects a `javax.validation.Validator` if that is on the classpath. We added `hibernate-validator` to the classpath and that is a validation implementation.

Next, you can find the controller's HTTP POST handler method used to handle the submission of user data. Since the handler method is expecting an instance of the `Reservation` object, which you decorated with JSR-303 annotations, you can validate its data.

The remainder of the `submitForm` method is exact the same as from recipe 4.9.

Note To use JSR-303 bean validation in a web application, you must add a dependency to an implementation to your CLASSPATH. If you are using Maven, add the following dependencies to your Maven Project:

```

<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>5.1.1.Final</version>
</dependency>

```

4-11. Creating Excel and PDF Views

Problem

Although HTML is the most common method of displaying web contents, sometimes your users may wish to export contents from your web application in Excel or PDF format. In Java, there are several libraries that can help generate Excel and PDF files. However, to use these libraries directly in a web application, you have to generate the files behind the scenes and return them to users as binary attachments. You have to deal with HTTP response headers and output streams for this purpose.

Solution

Spring integrates the generation of Excel and PDF files into its MVC framework. You can consider Excel and PDF files as special kinds of views, so you can consistently handle a web request in a controller and add data to a model for passing to Excel and PDF views. In this way, you have no need to deal with HTTP response headers and output streams.

Spring MVC supports generating Excel files using either the Apache POI library (<http://poi.apache.org/>) or the JExcelAPI library (<http://jexcelapi.sourceforge.net/>). The corresponding view classes are `AbstractExcelView` and `AbstractJExcelView`. PDF files are generated by the iText library (<http://www.lowagie.com/iText/>), and the corresponding view class is `AbstractPdfView`.

How It Works

Suppose your users wish to generate a report of the reservation summary for a particular day. They want this report to be generated in either, Excel, PDF, or the basic HTML format. For this report generation function, you need to declare a method in the service layer that returns all the reservations of a specified day:

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public List<Reservation> findByDate(Date date);
}
```

Then you provide a simple implementation for this method by iterating over all the made reservations:

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public List<Reservation> findByDate(Date date) {
        List<Reservation> result = new ArrayList<Reservation>();
        for (Reservation reservation : reservations) {
            if (reservation.getDate().equals(date)) {
                result.add(reservation);
            }
        }
        return result;
    }
}
```

Now you can write a simple controller to get the date parameters from the URL. The date parameter is formatted into a date object and passed to the service layer for querying reservations. The controller relies on the content negotiation resolver described in recipe 4-7 “Views and Content Negotiation,” therefore the controller returns a single logic view and lets the resolver determine if a report should be generated in Excel, PDF, or a default HTML web page.

```
package com.apress.springrecipes.court.web;
...
@Controller
@RequestMapping("/reservationSummary")
public class ReservationSummaryController {
    private ReservationService reservationService;

    @Autowired
    public ReservationSummaryController(ReservationService reservationService) {
        this.reservationService = reservationService;
    }
}
```

```

@RequestMapping(method = RequestMethod.GET)
public String generateSummary(
    @RequestParam(required = true, value = "date") String selectedDate,
    Model model) {
    List<Reservation> reservations = java.util.Collections.emptyList();
    try {
        Date summaryDate = new SimpleDateFormat("yyyy-MM-dd").parse(selectedDate);
        reservations = reservationService.findByDate(summaryDate);
    } catch (java.text.ParseException ex) {
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        ex.printStackTrace(pw);
        throw new ReservationWebException("Invalid date format for reservation summary",new
Date(),sw.toString());
    }
    model.addAttribute("reservations",reservations);
    return "reservationSummary";
}

```

{}

This controller only contains a default HTTP GET handler method. The first action performed by this method is creating an empty `Reservation` list to place the results obtained from the reservation service. Next, you can find a `try/catch` block that attempts to create a `Date` object from the `selectedDate` `@RequestParam`, as well as invoke the reservation service with the created `Date` object. If creating the a `Date` object fails, a custom Spring exception named `ReservationWebException` is thrown.

If no errors are raised in the `try/catch` block, the `Reservation` list is placed into the controller's `Model` object. Once this is done, the method returns control to `reservationSummary` view.

Note that the controller returns a single view, even though it supports PDF, XLS and HTML views. This is possible due to the `ContentNegotiatingViewResolver` resolver, that determines on the basis of this single view name which of these multiple views to use. See recipe 4-7, “Views and Content Negotiation,” for more information on this resolver.

Creating Excel Views

An Excel view can be created by extending the `AbstractExcelView` class (for Apache POI) or the `AbstractJExcelView` class (for JExcelAPI). Here, `AbstractExcelView` is used as an example. In the `buildExcelDocument()` method, you can access the model passed from the controller and also a precreated Excel workbook. Your task is to populate the workbook with the data in the model.

Note To generate Excel files with Apache POI in a web application, you must have the Apache POI dependencies on your CLASSPATH. If you are using Apache Maven, add the following dependencies to your Maven Project:

```

<dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi</artifactId>
    <version>3.10-FINAL</version>
</dependency>

```

```

package com.apress.springrecipes.court.web.view;
...
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.springframework.web.servlet.view.document.AbstractExcelView;

public class ExcelReservationSummary extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook workbook,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        HSSFSheet sheet = workbook.createSheet();

        HSSFRow header = sheet.createRow(0);
        header.createCell((short) 0).setCellValue("Court Name");
        header.createCell((short) 1).setCellValue("Date");
        header.createCell((short) 2).setCellValue("Hour");
        header.createCell((short) 3).setCellValue("Player Name");
        header.createCell((short) 4).setCellValue("Player Phone");

        int rowNum = 1;
        for (Reservation reservation : reservations) {
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell((short) 0).setCellValue(reservation.getCourtName());
            row.createCell((short) 1).setCellValue(
                dateFormat.format(reservation.getDate()));
            row.createCell((short) 2).setCellValue(reservation.getHour());
            row.createCell((short) 3).setCellValue(
                reservation.getPlayer().getName());
            row.createCell((short) 4).setCellValue(
                reservation.getPlayer().getPhone());
        }
    }
}

```

In the preceding Excel view, you first create a sheet in the workbook. In this sheet, you show the headers of this report in the first row. Then you iterate over the reservation list to create a row for each reservation.

As you have `@RequestMapping("/reservationSummary")` configured in your controller and the handler method requires date as a request parameter. You can access this Excel view through the following URL.

<http://localhost:8080/court/reservationSummary.xls?date=2009-01-14>

Creating PDF Views

A PDF view is created by extending the `AbstractPdfView` class. In the `buildPdfDocument()` method, you can access the model passed from the controller and also a precreated PDF document. Your task is to populate the document with the data in the model.

Note To generate PDF files with iText in a web application, you must have the iText library on your CLASSPATH. If you are using Apache Maven, add the following dependency to your Maven Project:

```
<dependency>
    <groupId>com.lowagie</groupId>
    <artifactId>iText</artifactId>
    <version>4.2.1</version>
</dependency>

package com.apress.springrecipes.court.web.view;
...
import org.springframework.web.servlet.view.document.AbstractPdfView;

import com.lowagie.text.Document;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class PdfReservationSummary extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document document,
        PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        Table table = new Table(5);

        table.addCell("Court Name");
        table.addCell("Date");
        table.addCell("Hour");
        table.addCell("Player Name");
        table.addCell("Player Phone");

        for (Reservation reservation : reservations) {
            table.addCell(reservation.getCourtName());
            table.addCell(dateFormat.format(reservation.getDate()));
            table.addCell(Integer.toString(reservation.getHour()));
            table.addCell(reservation.getPlayer().getName());
            table.addCell(reservation.getPlayer().getPhone());
        }
        document.add(table);
    }
}
```

As you have `@RequestMapping("/reservationSummary")` configured in your controller and the handler method requires date as a request parameter. You can access this PDF view through the following URL.

<http://localhost:8080/court/reservationSummary.pdf?date=2009-01-14>

Creating resolvers for Excel and PDF views

In recipe 4-6, “Resolving Views by Names,” you learned different strategies for resolving logical view names to specific view implementations. One of these strategies was resolving views from a resource bundle, this is the better-suited strategy for mapping logical view names to view implementations consisting of PDF or XLS classes.

Ensuring you have the `ResourceBundleViewResolver` bean configured in your web application context as a view resolver, you can then define views in the `views.properties` file included in a web application’s classpath root.

You can add the following entry to the `views.properties` in order to map the XLS view class to a logical view name:

```
reservationSummary.(class)=com.apress.springrecipes.court.web.view.ExcelReservationSummary
```

Since the application relies on the process of content negotiation, this implies that the same view name is mapped to multiple view technologies. In addition, since it’s not possible to have duplicate names in the same `views.properties` file, you need to create a separate file named `secondaryviews.properties` to map the PDF view class to a logical view name, as illustrated next:

```
reservationSummary.(class)=com.apress.springrecipes.court.web.view.PdfReservationSummary
```

Take note that this file—`secondaryviews.properties`—needs to be configured in its own `ResourceBundleViewResolver` resolver.

The property name—`reservationSummary`—corresponds to the views name returned by the controller. It’s the task of the `ContentNegotiatingViewResolver` resolver to determine which of these classes to use based on a user’s request. Once this is determined, the execution of the corresponding class generates either a PDF or XLS file.

Creating date based PDF and XLS file names

When a user makes a request for a PDF or XLS file using any of the following URLs:

```
http://localhost:8080/court/reservationSummary.pdf?date=2008-01-14
http://localhost:8080/court/reservationSummary.xls?date=2008-02-24
```

The browser prompts a user with a question like “Save as `reservationSummary.pdf`?” or “Save as `reservationSummary.xls`?” This convention is based on the URL a user is requesting a resource from. However, given that a user is also providing a date in the URL, a nice feature can be an automatic prompt in the form “Save as `ReservationSummary_2009_01_24.xls`?” or “Save as `ReservationSummary_2009_02_24.xls`?” This can be done by applying an interceptor to rewrite the returning URL. The following listing illustrates this interceptor:

```
package com.apress.springrecipes.court.web
...
public class ExtensionInterceptor extends HandlerInterceptorAdapter {
    public void postHandle(HttpServletRequest request,
                           HttpServletResponse response, Object handler,
                           ModelAndView modelAndView) throws Exception {
        // Report date is present in request
        String reportName = null;
        String reportDate = request.getQueryString().replace("date","",).replace("-","_");
        if(request.getServletPath().endsWith(".pdf")) {
            reportName= "ReservationSummary_" + reportDate + ".pdf";
        }
    }
}
```

```
if(request.getServletPath().endsWith(".xls")) {  
    reportName= "ReservationSummary_" + reportDate + ".xls";  
}  
if (reportName != null) {  
    // Set "Content-Disposition" HTTP Header  
    // so a user gets a pretty 'Save as' address  
    response.setHeader("Content-Disposition","attachment; filename="+reportName);  
}  
}  
}
```

The interceptor extracts the entire URL if it contains a .pdf or .xls extension. If it detects such an extension, it creates a value for the return file name in the form `ReservationSummary_<report_date>.(<.pdf| .xls>)`. To ensure a user receives a download prompt in this form, the HTTP header `Content-Disposition` is set with this file name format.

In order to deploy this interceptor and that it only be applied to the URL corresponding to the controller charged with generating PDF and XLS files, we advise you to look over recipe 4-3, “Intercepting Requests with Handler Interceptors,” which contains this particular configuration and more details about interceptor classes.

CONTENT NEGOTIATION AND SETTING HTTP HEADERS IN AN INTERCEPTOR

Though this application uses the `ContentNegotiatingViewResolver` resolver to select an appropriate view, the process of modifying a return URL is outside the scope of view resolvers.

Therefore, it's necessary to use an interceptor to manually inspect a request extension, as well as set the necessary HTTP headers to modify the outgoing URL.

Summary

In this chapter, you have learned how to develop a Java web application using the Spring MVC framework. The central component of Spring MVC is `DispatcherServlet`, which acts as a front controller that dispatches requests to appropriate handlers for them to handle requests.

In Spring MVC, controllers are standard Java classes that are decorated with the `@Controller` annotation. Throughout the various recipes, you learned how to leverage other annotations used in Spring MVC controllers, which included: `@RequestMapping` to indicate access URLs, `@Autowired` to automatically inject bean references and `@SessionAttributes` to maintain objects in a user's session, among many others.

You also learned how to incorporate interceptors into an application, which allow you to alter request and response objects in a controller. In addition, you explored how Spring MVC supports form processing, including data validation using both Spring validators and the JSR-303 bean validation standard.

You also explored how Spring MVC incorporates SpEL to facilitate certain configuration tasks and how Spring MVC supports different types of views for different presentation technologies. Finally, you also learned how Spring supports content negotiation in order to determine a view based on a request's extensions or HTTP headers.

CHAPTER 5

Spring REST

In this chapter, you will learn how Spring addresses Representational State Transfer, usually referred to by its acronym REST. REST has had an important impact on web applications since the term was coined by Roy Fielding (http://en.wikipedia.org/wiki/Roy_Fielding) in the year 2000.

Based on the foundations of the web's protocol Hypertext Transfer Protocol (HTTP), the architecture set forth by REST has become increasingly popular in the implementation of web services.

Web services in and of themselves have become the cornerstone for much machine-to-machine communication taking place on the Web. It's the fragmented technology choices (e.g., Java, Python, Ruby, .NET) made by many organizations that have necessitated a solution capable of bridging the gaps between these disparate environments. How is information in an application backed by Java accessed by one written in Python? How can a Java application obtain information from an application written in .NET? Web services fill this void.

There are various approaches to implementing web services, but RESTful web services have become the most common choice in web applications. They are used by some of the largest Internet portals (e.g., Google and Yahoo) to provide access to their information, used to back access to Ajax calls made by browsers, in addition to providing the foundations for the distribution of information like news feeds (e.g., RSS).

In this chapter, you will learn how Spring applications can use REST, so you can both access and provide information using this popular approach.

5-1. Publishing XML with REST Services

Problem

You want to publish a XML based REST service with Spring.

Solution

There are two possibilities when designing REST services in Spring. One involves *publishing* an application's data as a REST service, the other one involves *accessing* data from third-party REST services to be used in an application. This recipe describes how to publish an application's data as a REST service. Recipe 5-2 describes how to access data from third-party REST services.

Publishing an application's data as a REST service revolves around the use of the Spring MVC annotations `@RequestMapping` and `@PathVariable`. By using these annotations to decorate a Spring MVC handler method, a Spring application is capable of publishing an application's data as a REST service.

In addition, Spring supports a series of mechanisms to generate a REST service's payload. This recipe will explore the simplest mechanism, which involves the use of Spring's `MarshallingView` class. As the recipes in this chapter progress, you will learn about more advanced mechanisms supported by Spring to generate REST service payloads.

How It Works

Publishing a web application's data as a REST service, or as it's more technically known in web services parlance "creating an end point" is strongly tied to Spring MVC, which you explored in Chapter 4.

Since Spring MVC relies on the annotation `@RequestMapping` to decorate handler methods and define access points (i.e., URLs), it's the preferred way in which to define a REST service's end point.

Using a MarshallingView to produce XML

The following listing illustrates a Spring MVC controller class with a handler method that defines a REST service endpoint:

```
package com.apress.springrecipes.court.web;

import com.apress.springrecipes.court.domain.Members;
import com.apress.springrecipes.court.service.MemberService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class RestMemberController {

    private final MemberService memberService;

    @Autowired
    public RestMemberController(MemberService memberService) {
        super();
        this.memberService=memberService;
    }

    @RequestMapping("/members")
    public String getRestMembers(Model model) {
        Members members = new Members();
        members.addMembers(memberService.findAll());
        model.addAttribute("members", members);
        return "membertemplate";
    }
}
```

By using `@RequestMapping("/members")` to decorate a controller's handler method, a REST service endpoint is made accessible at `http://[host_name]/[app-name]/members`. You can observe that control is relinquished to a logical view named `membertemplate`. The following listing illustrates the declaration used to define the logical view named `membertemplate`:

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.court.web")
public class CourtRestConfiguration {

    @Bean
    public View membertemplate() {
        return new MarshallingView(jaxb2Marshaller());
    }

    @Bean
    public Marshaller jaxb2Marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setClassesToBeBound(Members.class, Member.class);
        return marshaller;
    }

    @Bean
    public ViewResolver viewResolver() {
        return new BeanNameViewResolver();
    }

    @Bean
    public MemberService memberService() {
        return new InMemoryMemberService();
    }
}

```

The `membertemplate` view is defined as a `MarshallingView` type, which is a general-purpose class that allows a response to be rendered using a marshaller. Marshalling is the process of transforming an in-memory representation of an object into a data format. Therefore, for this particular case, a marshaller is charged with transforming `Members` and `Member` objects into an XML data format.

The marshaller used by `MarshallingView` belongs to one of a series of XML marshallers provided by Spring—`Jaxb2Marshaller`. Other marshallers provided by Spring include `CastorMarshaller`, `JibxMarshaller`, `XmlBeansMarshaller`, and `XStreamMarshaller`.

Marshallers themselves also require configuration. We opted to use the `Jaxb2Marshaller` marshaller due to its simplicity and Java Architecture for XML Binding (JAXB) foundations. However, if you're more comfortable using the Castor XML framework, you might find it easier to use the `CastorMarshaller`, and you would likely find it easier to use the `XStreamMarshaller` if you're more at ease using XStream, with the same case applying for the rest of the available marshallers.

The Jaxb2Marshaller marshaller requires to be configured with either a property named `classesToBeBound` or `contextPath`. In the case of `classesToBeBound`, the classes assigned to this property, indicate the class (i.e., object) structure that is to be transformed into XML. The following listing illustrates the `Members` and `Member` classes assigned to the Jaxb2Marshaller:

```
package com.apress.springrecipes.court.domain;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Member {
    private String name;
    private String phone;
    private String email;

    public String getEmail() {
        return email;
    }

    public String getName() {
        return name;
    }

    public String getPhone() {
        return phone;
    }

    public void setEmail(String email) {
        this.email = email;
    }

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

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

The `Members` class:

```
package com.apress.springrecipes.court.domain;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
```

```

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Members {

    @XmlElement(name="member")
    private List<Member> members = new ArrayList<>();

    public List<Member> getMembers() {
        return members;
    }

    public void setMembers(List<Member> members) {
        this.members = members;
    }

    public void addMembers(Collection<Member> members) {
        this.members.addAll(members);
    }
}

```

Note the Member class is a POJO decorated with the `@XmlRootElement` annotation. This annotation allows the Jaxb2Marshaller marshaller to detect a class's (i.e., object's) fields and transform them into XML data (e.g., `name=John` into `<name>john</name>`, `email=john@doe.com` into `<email>john@doe.com</email>`).

To recap what's been described, this means that when a request is made to a URL in the form `http://[host_name]//app-name]/members.xml`, the corresponding handler is charged with creating a `Members` object, which is then passed to a logical view named `membertemplate`. Based on this last view's definition, a marshaller is used to convert a `Members` object into an XML payload that is returned to the REST service's requesting party. The XML payload returned by the REST service is illustrated in the following listing:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<members>
    <member>
        <email>marten@deinum.biz</email>
        <name>Marten Deinum</name>
        <phone>00-31-1234567890</phone>
    </member>
    <member>
        <email>john@doe.com</email>
        <name>John Doe</name>
        <phone>1-800-800-800</phone>
    </member>
    <member>
        <email>jane@doe.com</email>
        <name>Jane Doe</name>
        <phone>1-801-802-803</phone>
    </member>
</members>

```

This last XML payload represents a very simple approach to generating a REST service's response. As the recipes in this chapter progress, you will learn more sophisticated approaches, such as the ability to create widely used REST service payloads like RSS, Atom, and JSON.

If you look closely at the REST service end point or URL described in the previous paragraph, you'll note that it has an `.xml` extension. If you try another extension—or even omit the extension—this particular REST service may not be triggered. This last behavior is directly tied to Spring MVC and how it handles view resolution. It has nothing do with REST services per se.

By default, since the view associated with this particular REST service handler method returns XML, it's triggered by an `.xml` extension. This allows the same handler method to support multiple views. For example, it can be convenient for a request like `http://[host_name]/[app-name]/members.pdf` to return the same information in a PDF document, as well as a request like `http://[host_name]/[app-name]/members.html` to return content in HTML or a request like `http://[host_name]/[app-name]/members.xml` to return XML for a REST request.

So what happens to a request with no URL extension, like `http://[host_name]/[app-name]/members`? This also depends heavily on Spring MVC view resolution. For this purpose, Spring MVC supports a process called *content negotiation*, by which a view is determined based on a request's extension or HTTP headers.

Since REST service requests typically have HTTP headers in the form `Accept: application/xml`, Spring MVC configured to use content negotiation can determine to serve XML (REST) payloads to such requests even if requests are made extensionless. This also allows extensionless requests to be made in formats like HTML, PDF, and XLS, all simply based on HTTP headers. Recipe 4-7 in Chapter 4 discusses content negotiation.

Using `@ResponseBody` to produce XML

Using a MarshallingView to produce XML is one way of producing results, however when you want to have multiple representations (JSON for instance) of the same data (a list of Member objects) adding another view can be a cumbersome task. Instead we can rely on the Spring MVC `HttpMessageConverters` to convert an object to the representation requested by the user. The following listing shows the changes made to the `RestMemberController`:

```
@Controller
public class RestMemberController {
    ...
    @RequestMapping("/members")
    @ResponseBody
    public Members getRestMembers() {
        Members members = new Members();
        members.addMembers(memberService.findAll());
        return members;
    }
}
```

The first change is that we have now, additionally, annotated our controller method with `@ResponseBody`. This annotation tells Spring MVC that the result of the method should be used as the body of the response. As we want XML this marshaling is done by the `Jaxb2RootElementHttpMessageConverter` provided by Spring. The second change is that, due to the `@ResponseBody` annotation, we don't need the view name anymore but can simply return the `Members` object.

Tip When using Spring 4 instead of annotation the method with `@ResponseBody` you can also annotate your controller with `@RestController` instead of `@Controller` which would give the same result. This is especially convenient if you have a single controller with multiple methods.

These changes also allow us to clean up our configuration, as we don't need the `MarshallingView` and `Jaxb2Marshaller` anymore.

```
package com.apress.springrecipes.court.web.config;

import com.apress.springrecipes.court.service.InMemoryMemberService;
import com.apress.springrecipes.court.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.court.web")
public class CourtRestConfiguration {

    @Bean
    public MemberService memberService() {
        return new InMemoryMemberService();
    }
}
```

When the application is deployed and you do request the members from `http://localhost:8080/court/members.xml` it will yield the same results as before.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<members>
    <member>
        <email>marten@deinum.biz</email>
        <name>Marten Deinum</name>
        <phone>00-31-1234567890</phone>
    </member>
    <member>
        <email>john@doe.com</email>
        <name>John Doe</name>
        <phone>1-800-800-800</phone>
    </member>
    <member>
        <email>jane@doe.com</email>
        <name>Jane Doe</name>
        <phone>1-801-802-803</phone>
    </member>
</members>
```

Using `@PathVariable` to limit the results

It's common for REST service requests to have parameters. This is done to limit or filter a service's payload. For example, a request in the form `http://[host_name]/[app-name]/member/353/` can be used to retrieve information exclusively on member 353. Another variation can be a request like `http://[host_name]/[app-name]/reservations/07-07-2010/` to retrieve reservations made on the date 07-07-2010.

In order to use parameters for constructing a REST service in Spring, you use the `@PathVariable` annotation. The `@PathVariable` annotation is added as an input parameter to the handler method, per Spring's MVC conventions, in order for it to be used inside the handler method body. The following snippet illustrates a handler method for a REST service using the `@PathVariable` annotation.

```
Import org.springframework.web.bind.annotation.PathVariable;

@Controller
public class RestMemberController {
    ...
    @RequestMapping("/member/{memberid}")
    @ResponseBody
    public Member getMember(@PathVariable("memberid") long memberID) {
        return memberService.find(memberID);
    }
}
```

Notice the `@RequestMapping` value contains `{memberid}`. Values surrounded by `{ }` are used to indicate URL parameters are variables. Further note the handler method is defined with the input parameter `@PathVariable("memberid") long memberID`. This last declaration associates whatever `memberid` value forms part of the URL and assigns it to a variable named `memberID` that can be accessible inside the handler method.

Therefore, REST end points in the form `/member/353/` and `/member/777/` will be processed by this last handler method, with the `memberID` variable being assigned values of 353 and 777, respectively. Inside the handler method, the appropriate queries can be made for members 353 and 777—via the `memberID` variable—and returned as the REST service's payload.

A request to `http://localhost:8080/court/member/2` will result in a XML representation of the member with id 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<member>
    <email>john@doe.com</email>
    <name>John Doe</name>
    <phone>1-800-800-800</phone>
</member>
```

In addition to supporting the `{ }` notation, it's also possible to use a wildcard `*` notation for defining REST end points. This is often the case when a design team has opted to use expressive URLs (often called *pretty URLs*) or opts to use search engine optimization (SEO) techniques to make a REST URL search engine friendly. The following snippet illustrates a declaration for a REST service using the wildcard notation.

```
@RequestMapping("/member*/{memberid}")
@ResponseBody
public Member getMember(@PathVariable("memberid") long memberID) { ... }
```

In this case, the addition of a wildcard doesn't have any influence over the logic performed by the REST service. But it will match end point requests in the form `/member/John+Smith/353/` and `/member/Mary+Jones/353/`, which can have an important impact on end user readability or SEO.

It's also worth mentioning that data binding can be used in the definition of handler methods for REST end points. The following snippet illustrates a declaration for a REST service using data binding:

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false));
}

@RequestMapping("/reservations/{date}")
public void getReservation(@PathVariable("date") Date resDate) { ... }
```

In this case, a request in the form `http://[host_name]/[app-name]/reservations/07-07-2010/` is matched by this last handler method, with the value `07-07-2010` passed into the handler method—as the variable `resDate`—where it can be used to filter the REST web service payload.

Using the ResponseEntity to inform the client

The endpoint for retrieval of a single `Member` instance either returns a valid member or nothing at all. Both lead to a request which will send a HTTP Response code 200, which means OK, back to the client. However this is probably not what our uses will expect. When working with resources we should inform them of the fact that a resource cannot be found. Ideally we would want to return a HTTP Response Code 404, which indicates not found. The following code snippets show the modified `getMember` method.

```
package com.apress.springrecipes.court.web;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
...
@Controller
public class RestMemberController {
...
    @RequestMapping("/member/{memberid}")
    @ResponseBody
    public ResponseEntity<Member> getMember(@PathVariable("memberid") long memberID) {
        Member member = memberService.find(memberID);
        if (member != null) {
            return new ResponseEntity<Member>(member, HttpStatus.OK);
        }
        return new ResponseEntity(HttpStatus.NOT_FOUND);
    }
}
```

The return value of the method has been changed to `ResponseEntity<Member>`. The `ResponseEntity` is a class in Spring MVC that acts as a wrapper for an object to be used as the body of the result together with a HTTP status code.

When we find a `Member` it is returned together with a `HttpStatus.OK`, the latter corresponds to a HTTP status code of 200. When there is no result we return the `HttpStatus.NOT_FOUND`, corresponding to the HTTP Status Code 404, not found.

5-2. Publishing JSON with REST services

Problem

You want to publish a JSON (JavaScript Object Notation) based REST service with Spring.

Solution

JSON has blossomed into a favorite payload format for REST services. However, unlike most REST service payloads, which rely on XML markup, JSON is different in the sense that its content is a special notation based on the JavaScript language.

For this recipe, in addition to relying on Spring's REST support, we will also use the `MappingJackson2JsonView` class that forms part of Spring to facilitate the publication of JSON content.

Note The `MappingJackson2JsonView` class depends on the presence of the Jackson JSON processor library, version 2, which can be downloaded at <http://wiki.fasterxml.com/JacksonDownload>. If you're using Maven, add the following dependency to your project:

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.4.0</version>
</dependency>
```

WHY PUBLISH JSON?

If your Spring application's incorporate Ajax designs, it's very likely that you'll find yourself designing REST services that publish JSON as their payload. This is mainly due to the limited processing capabilities in browsers.

Although browsers can process and extract information from REST services that publish XML payloads, it's not very efficient. By instead delivering payloads in JSON, which is based on a language for which browsers have a native interpreter—JavaScript—the processing and extraction of data becomes more efficient.

Unlike RSS and Atom feeds, which are standards, JSON has no specific structure it needs to follow—except its syntax which you'll explore shortly. Therefore, a JSON element's payload structure is likely to be determined in coordination with the team members charged with an application's Ajax design.

How It Works

The first thing you need to do is determine the information you wish to publish as a JSON payload. This information can be located in a RDBMS or text file, accessed through JDBC or ORM, inclusively be part of a Spring bean or some other type of construct. Describing how to obtain this information would go beyond the scope of this recipe, so we will assume you'll use whatever means you deem appropriate to access it.

In case you're unfamiliar with JSON, the following snippet illustrates a fragment of this format:

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup ←
                    languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

As you can observe, a JSON payload consists of text and separators like {, }, [,], : and ". We won't go into details about using one separator over another, but it suffices to say this type of syntax makes it easier for a JavaScript engine to access and manipulate data than if it was to process it in an XML type format.

Using a MappingJackson2JsonView to produce XML

Since you've already explored how to publish data using a REST service in recipes 5-1 and 5-3, we'll cut to the chase and show you the actual handler method needed in a Spring MVC controller to achieve this process.

```
@RequestMapping("/members")
public String getRestMembers(Model model) {
    // Return view jsonmembertemplate. Via resolver the view
    // will be mapped to a jackson ObjectMapper bound to the Member class

    Members members = new Members();
    members.addMembers(memberService.findAll());
    model.addAttribute("members", members);
    return "jsonmembertemplate";
}
```

You probably notice that it is quite similar to the controller method mentioned in Recipe 5-1. The only difference is that we return a different name for the view. The name of the view we are returning, **jsonmembertemplate**, is different and maps to a **MappingJackson2JsonView**. This view we need to configure in our configuration class.

```

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.court.web")
public class CourtRestConfiguration {

    ...
    @Bean
    public View jsonmembertemplate() {
        MappingJackson2JsonView view = new MappingJackson2JsonView();
        view.setPrettyPrint(true);
        return view;
    }
}

```

The `MappingJackson2JsonView` uses the Jackson2 library to convert objects to and from JSON. It uses a `JacksonObjectMapper` instance for the conversion. When a request is made to `http://localhost:8080/court/members.json` the controller method will be invoked and a JSON representation will be returned.

```

{
    "members" : [
        {
            "name" : "Marten Deinum",
            "phone" : "00-31-1234567890",
            "email" : "marten@deinum.biz"
        },
        {
            "name" : "John Doe",
            "phone" : "1-800-800-800",
            "email" : "john@doe.com"
        },
        {
            "name" : "Jane Doe",
            "phone" : "1-801-802-803",
            "email" : "jane@doe.com"
        }
    ]
}

```

Actually this JSON will be produced by each call to `/members` or `/members.*` (for instance `/members.xml` will also produce JSON). Let's add the method and view from Recipe 5-1 to our controller

```

@Controller
public class RestMemberController {
    ...
    @RequestMapping(value="/members", produces=MediaType.APPLICATION_XML_VALUE)
    public String getRestMembersXml(Model model) {
        Members members = new Members();
        members.addMembers(memberService.findAll());
        model.addAttribute("members", members);
        return "xmlmembertemplate";
    }
}

```

```

@RequestMapping(value="/members", produces= MediaType.APPLICATION_JSON_VALUE)
public String getRestMembersJson(Model model) {
    Members members = new Members();
    members.addMembers(memberService.findAll());
    model.addAttribute("members", members);
    return "jsonmembertemplate";
}
}

```

We have now a `getMembersXml` and `getMembersJson` method, both are basically the same with this distinction that they return a different view name. Notice the `produces` attribute on the `@RequestMapping` annotation this is used to determine which method to call `/members.xml` will now produce XML whereas `/members.json` will produce JSON.

Although this approach works duplicating all the methods for the different supported view types isn't a feasible solution for enterprise applications. We could create a helper method to reduce the duplication but we would still need a lot of boilerplate due to the differences in the `@RequestMapping` annotations.

Using `@ResponseBody` to produce JSON

Using a `MappingJackson2JsonView` to produce JSON is one way of producing results, however as mentioned in the previous section can be troublesome, especially with multiple supported view types. Instead we can rely on the Spring MVC `HttpMessageConverters` to convert an object to the representation requested by the user. The following listing shows the changes made to the `RestMemberController`:

```

@Controller
public class RestMemberController {
...
    @RequestMapping("/members")
    @ResponseBody
    public Members getRestMembers() {
        Members members = new Members();
        members.addMembers(memberService.findAll());
        return members;
    }
}

```

The first change is that we have now, additionally, annotated our controller method with `@ResponseBody`. This annotation tells Spring MVC that the result of the method should be used as the body of the response. As we want JSON this marshaling is done by the `Jackson2JsonMessageConverter` provided by Spring. The second change is that, due to the `@ResponseBody` annotation, we don't need the view name anymore but can simply return the `Members` object.

Tip When using Spring 4 instead of annotation the method with `@ResponseBody` you can also annotate your controller with `@RestController` instead of `@Controller` which would give the same result. This is especially convenient if you have a single controller with multiple methods.

These changes also allow us to clean up our configuration, as we don't need the `MappingJackson2JsonView` anymore.

```
package com.apress.springrecipes.court.web.config;

import com.apress.springrecipes.court.service.InMemoryMemberService;
import com.apress.springrecipes.court.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.court.web")
public class CourtRestConfiguration {

    @Bean
    public MemberService memberService() {
        return new InMemoryMemberService();
    }
}
```

When the application is deployed and you do request the members from `http://localhost:8080/court/members.json` it will give the same results as before.

```
{
    "members" : {
        "members" : [ {
            "name" : "Marten Deinum",
            "phone" : "00-31-1234567890",
            "email" : "marten@deinum.biz"
        }, {
            "name" : "John Doe",
            "phone" : "1-800-800-800",
            "email" : "john@doe.com"
        }, {
            "name" : "Jane Doe",
            "phone" : "1-801-802-803",
            "email" : "jane@doe.com"
        } ]
    }
}
```

You probably noticed that the `RestMemberController` and `CourtRestConfiguration` are now exactly the same as the in Recipe 5-1. When calling `http://localhost:8080/court/members.xml` you will get XML.

How is this possible without any additional configuration? Spring MVC will detect what is on the classpath it automatically detects JAXB2, Jackson (1 and 2), and Rome (see Recipe 5-4) it will register the appropriate `HttpMessageConverter` for the available technologies.

5-3. Accessing a REST Service with Spring

Problem

You want to access a REST service from a third party (e.g., Google, Yahoo, another business partner) and use its payload inside a Spring application.

Solution

Accessing a third-party REST service inside a Spring application revolves around the use of the Spring `RestTemplate` class.

The `RestTemplate` class is designed on the same principles as the many other Spring `*Template` classes (e.g., `JdbcTemplate`, `JmsTemplate`), providing a simplified approach with default behaviors for performing lengthy tasks.

This means the processes of invoking a REST service and using its returning payload are streamlined in Spring applications.

How It Works

Before describing the particularities of the `RestTemplate` class, it's worth exploring the life cycle of a REST service, so you're aware of the actual work the `RestTemplate` class performs. Exploring the life cycle of a REST service can best be done from a browser, so open your favorite browser on your workstation to get started.

The first thing that's needed is a REST service endpoint. We are going to reuse the endpoint we created in Recipe 5-2. This endpoint should be available at `http://localhost:8080/court/members.xml` (or `.json`). If you load this last REST service end point on your browser, the browser performs a `GET` request, which is one of the most popular HTTP requests supported by REST services. Upon loading the REST service, the browser displays a responding payload like the following:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<members>
    <member>
        <email>marten@deinum.biz</email>
        <name>Marten Deinum</name>
        <phone>00-31-1234567890</phone>
    </member>
    <member>
        <email>john@doe.com</email>
        <name>John Doe</name>
        <phone>1-800-800-800</phone>
    </member>
    <member>
        <email>jane@doe.com</email>
        <name>Jane Doe</name>
        <phone>1-801-802-803</phone>
    </member>
</members>
```

This last payload represents a well-formed XML fragment, which is in line with most REST service's responses. The actual meaning of the payload is highly dependent on a REST service. In this case, the XML tags (`<Members>`, `<Member>` etc.) are definitions set forth by ourselves, while the character data enclosed in each XML tag represents information related to a REST service's request.

It's the task of a REST service consumer (i.e., you) to know the payload structure—sometimes referred to as *vocabulary*—of a REST service to appropriately process its information. Though this last REST service relies on what can be considered a custom vocabulary, a series of REST services often rely on standardized vocabularies (e.g., RSS), which make the processing of REST service payloads uniform. In addition, it's also worth noting that some REST services provide Web Application Description Language (WADL) contracts to facilitate the discovery and consumption of payloads.

Now that you're familiar with a REST service's life cycle using your browser, we can take a look at how to use the `RestTemplate` class in order to incorporate a REST service's payload into a Spring application.

Given that the `RestTemplate` class is designed to call REST services, it should come as no surprise that its main methods are closely tied to REST's underpinnings, which are the HTTP protocol's methods: HEAD, GET, POST, PUT, DELETE, and OPTIONS. Table 5-1 contains the main methods supported by the `RestTemplate` class.

Table 5-1. *RestTemplate* class methods based on HTTP protocol's request methods

Method	Description
<code>headForHeaders(String, Object...)</code>	Performs an HTTP HEAD operation
<code>getForObject(String, Class, Object...)</code>	Performs an HTTP GET operation and return the result as a type of the given class.
<code>getForObject(String, Class, Object...)</code>	Performs an HTTP GET operation and return a <code>ResponseEntity</code> .
<code>postForLocation(String, Object, Object...)</code>	Performs an HTTP POST operation and return the value of the location header.
<code>postForObject(String, Object, Class, Object...)</code>	Performs an HTTP POST operation and return the result as a type of the given class.
<code>postForEntity(String, Object, Class, Object...)</code>	Performs an HTTP POST operation and return a <code>ResponseEntity</code> .
<code>put(String, Object, Object...)</code>	Performs an HTTP PUT operation
<code>delete(String, Object...)</code>	Performs an HTTP DELETE operation
<code>optionsForAllow(String, Object...)</code>	Performs an HTTP OPTIONS operation
<code>execute(String, HttpMethod, RequestCallback, ResponseExtractor, Object...)</code>	Can perform any HTTP operation with the exception of CONNECT

As you can observe in Table 5-1, the `RestTemplate` class methods are prefixed with a series of HTTP protocol methods that include HEAD, GET, POST, PUT, DELETE, and OPTIONS. In addition, the `execute` method serves as a general-purpose method that can perform any HTTP operation, including the more esoteric HTTP protocol TRACE method, albeit not the CONNECT method, the last of which is not supported by the underlying `HttpMethod` enum used by the `execute` method.

Note By far the most common HTTP method used in REST services is GET, since it represents a safe operation to obtain information (i.e., it doesn't modify any data). On the other hand, HTTP methods such as PUT, POST, and DELETE are designed to modify a provider's information, which makes them less likely to be supported by a REST service provider. For cases in which data modification needs to take place, many providers opt for the SOAP protocol, which is an alternative mechanism to using REST services.

Now that you're aware of the `RestTemplate` class methods, we can move on to invoking the same REST service you did with your browser previously, except this time using Java code from the Spring framework. The following listing illustrates a class that accesses the REST service and returns its contents to the `System.out`.

```
package com.apress.springrecipes.court;

import org.springframework.web.client.RestTemplate;

public class Main {

    public static void main(String[] args) throws Exception {
        final String uri = "http://localhost:8080/court/members.json";
        RestTemplate restTemplate = new RestTemplate();
        String result = restTemplate.getForObject(uri, String.class);
        System.out.println(result);
    }
}
```

Caution Some REST service providers restrict access to their data feeds depending on the requesting party. Access is generally denied by relying on data present in a request (e.g., HTTP headers or IP address). So depending on the circumstances, a provider can return an access denied response even when a data feed appears to be working in another medium (e.g., you might be able to access a REST service in a browser but get an accessed denied response when attempting to access the same feed from a Spring application). This depends on the terms of use set forth by a REST provider.

The first line marked in bold declares the import statement needed to access the `RestTemplate` class within a class's body. First we need to create an instance of the `RestTemplate`.

Next, you can find a call made to the `getForObject` method that belongs to the `RestTemplate` class, which as described in Table 5-1 is used to perform an HTTP GET operation—just like the one performed by a browser to obtain a REST service's payload. There are two important aspects related to this last method, its response and its parameters.

The response of calling the `getForObject` method is assigned to a `String` object. This means the same output you saw on your browser for this REST service (i.e., the XML structure) is assigned to a `String`. Even if you've never processed XML in Java, you're likely aware that extracting and manipulating data as a Java `String` is not an easy task. In other words, there are classes better suited for processing XML data, and with it a REST service's payload, than a `String` object. For the moment just keep this in mind; other recipes in the chapter illustrate how to better extract and manipulate the data obtained from a REST service.

The parameters passed to the `getForObject` method consist of the actual REST service end point. The first parameter corresponds to the URL (i.e., endpoint) declaration. Notice the URL is identical to the one used when you relied on a browser to call it.

When executed the output will be the same as in the browser except that it is now printed in the console.

Retrieving data from a parameterized URL

The previous section showed how we can call a URI to retrieve data, but what about a URI that requires parameters. We don't want to hardcode parameters into the URL. With the `RestTemplate` we can use a URL with placeholders, these placeholders will be replaced with actual values upon execution. Placeholders are defined using { and }, just as with a request mapping (See Recipes 5-1 and 5-2).

The URI `http://localhost:8080/court/member/{memberId}` is an example of such a parameterized URI. To be able to call this method we need to pass in a value for the placeholder, we can do this by using a Map and pass that as the third parameter to the `getForObject` method of the `RestTemplate`.

```
public class Main {

    public static void main(String[] args) throws Exception {
        final String uri = "http://localhost:8080/court/member/{memberId}";
        Map<String, String> params = new HashMap<>();
        params.put("memberId", "1");
        RestTemplate restTemplate = new RestTemplate();
        String result = restTemplate.getForObject(uri, String.class, params );
        System.out.println(result);
    }
}
```

This last snippet makes use of the `HashMap` class—part of the Java collections framework—creating an instance with the corresponding REST service parameters, which is later passed to the `getForObject` method of the `RestTemplate` class. The results obtained by passing either a series of `String` parameters or a single `Map` parameter to the various `RestTemplate` methods is identical.

Retrieving data as mapped object

Instead of returning a `String` to be used in our application we can also (re)use our `Members` and `Member` classes to map the result. Instead of passing in `String.class` as the second parameter pass `Members.class` and the response will be mapped onto this class.

```
package com.apress.springrecipes.court;

import com.apress.springrecipes.court.domain.Members;
import org.springframework.web.client.RestTemplate;

public class Main {

    public static void main(String[] args) throws Exception {
        final String uri = "http://localhost:8080/court/members.xml";
        RestTemplate restTemplate = new RestTemplate();
        Members result = restTemplate.getForObject(uri, Members.class);
        System.out.println(result);
    }
}
```

The `RestTemplate` makes use of the same `HttpMessageConverter` infrastructure as a controller with `@ResponseBody` marked methods. As Jaxb2 (as well as Jackson) is automatically detected mapping to a Jaxb mapped object is quite easy.

5-4. Publishing RSS and Atom feeds

Problem

You want to publish an RSS or Atom feed in a Spring application.

Solution

RSS and Atom feeds have become a popular means by which to publish information. Access to these types of feeds is provided by means of a REST service, which means building a REST service is a prerequisite to publishing RSS and Atom feeds.

In addition to relying on Spring's REST support, it's also convenient to rely on a third-party library especially designed to deal with the particularities of RSS and Atom feeds. This makes it easier for a REST service to publish this type of XML payload. For this last purpose, we will use Project Rome, an open source library available at <http://rometools.github.io/rome/>.

Note Project Rome depends on the JDOM library that can be downloaded at <http://www.jdom.org/>. If you are using Maven, you can add the following dependency to your pom.xml file:

```
<dependency>
  <groupId>org.jdom</groupId>
  <artifactId>jdom</artifactId>
  <version>1.1</version>
</dependency>
```

Tip Even though RSS and Atom feeds are often categorized as news feeds, they have surpassed this initial usage scenario of providing just news. Nowadays, RSS and Atom feeds are used to publish information related to blogs, weather, travel, and many other things in a cross-platform manner (i.e., using XML). Hence, if you require publishing information of any sort that's to be accessible in a cross-platform manner, doing so as RSS or Atom feeds can be an excellent choice given their wide adoption (e.g., many applications support them and many developers know their structure).

How It Works

The first thing you need to do is determine the information you wish to publish as an RSS or Atom news feed. This information can be located in an RDBMS or text file, accessed through JDBC or ORM, inclusively be part of a Spring bean or some other type of construct. Describing how to obtain this information would go beyond the scope of this recipe, so we will assume you'll use whatever means you deem appropriate to access it.

Once you've pinpointed the information you wish to publish, it's necessary to structure it as either an RSS or Atom feed, which is where Project Rome comes into the picture.

In case you're unfamiliar with an Atom feed's structure, the following snippet illustrates a fragment of this format:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>Example Feed</title>
    <link href="http://example.org/" />
    <updated>2010-08-31T18:30:02Z</updated>
    272103_1_En
        <name>John Doe</name>
    </author>
    <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>
    <entry>
        <title>Atom-Powered Robots Run Amok</title>
        <link href="http://example.org/2010/08/31/atom03"/>
        <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
        <updated>2010-08-31T18:30:02Z</updated>
        <summary>Some text.</summary>
    </entry>
</feed>
```

The following snippet illustrates a fragment of an RSS feed's structure:

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
    <channel>
        <title>RSS Example</title>
        <description>This is an example of an RSS feed</description>
        <link>http://www.example.org/link.htm</link>
        <lastBuildDate>Mon, 28 Aug 2006 11:12:55 -0400 </lastBuildDate>
        <pubDate>Tue, 31 Aug 2010 09:00:00 -0400</pubDate>
        <item>
            <title>Item Example</title>
            <description>This is an example of an Item</description>
            <link>http://www.example.org/link.htm</link>
            <guid isPermaLink="false"> 1102345</guid>
            <pubDate>Tue, 31 Aug 2010 09:00:00 -0400</pubDate>
        </item>
    </channel>
</rss>
```

As you can observe from these last two snippets, RSS and Atom feeds are just XML payloads that rely on a series of elements to publish information. Though going into the finer details of either an RSS or Atom feed structure would require a book in itself, both formats possess a series of common characteristics; chief among them are these:

- They have a metadata section to describe the contents of a feed. (e.g., the 272103_1_En and <title> elements for the Atom format and the <description> and <pubDate> elements for the RSS format).
- They have recurring elements to describe information (e.g., the <entry> element for the Atom feed format and the <item> element for the RSS feed format). In addition, each recurring element also has its own set of elements with which to further describe information.
- They have multiple versions. RSS versions include 0.90, 0.91 Netscape, 0.91 Userland, 0.92, 0.93, 0.94, 1.0, and 2.0. Atom versions include 0.3 and 1.0.

Project Rome allows you to create a feed's metadata section, recurring elements, as well as any of the previously mentioned versions, from information available in Java code (e.g., Strings, Maps, or other such constructs).

Now that you're aware of the structure of an RSS and Atom feed, as well as the role Project Rome plays in this recipe, let's take a look at a Spring MVC controller charged with presenting a feed to an end user.

```
package com.apress.springrecipes.court.web;

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

import com.apress.springrecipes.court.feeds.TournamentContent;

import com.apress.springrecipes.court.domain.Member;
import java.util.List;
import java.util.Date;
import java.util.ArrayList;

@Controller
public class FeedController {

    @RequestMapping("/atomfeed")
    public String getAtomFeed(Model model) {
        List<TournamentContent> tournamentList = new ArrayList<TournamentContent>();
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Australian Open", "www.australianopen.com"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Roland Garros", "www.rolandgarros.com"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "Wimbledon", "www.wimbledon.org"));
        tournamentList.add(TournamentContent.generateContent("ATP", ←
            new Date(), "US Open", "www.usopen.org"));
        model.addAttribute("feedContent", tournamentList);
        return "atomfeedtemplate";
    }

    @RequestMapping("/rssfeed")
    public String getRSSFeed(Model model) {
        List<TournamentContent> tournamentList = new ArrayList<TournamentContent>();
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "World Cup", "www.fifa.com/worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "U-20 World Cup", "www.fifa.com/u20worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "U-17 World Cup", "www.fifa.com/u17worldcup/"));
        tournamentList.add(TournamentContent.generateContent("FIFA", ←
            new Date(), "Confederations Cup", "www.fifa.com/confederationscup/"));
        model.addAttribute("feedContent", tournamentList);
        return "rssfeedtemplate";
    }
}
```

This last Spring MVC controller has two handler methods. One called `getAtomFeed()`, which is mapped to a URL in the form `http://[host_name]/[app-name]/atomfeed`, and another called `getRSSFeed()`, which is mapped to a URL in the form `http://[host_name]/[app-name]/rssfeed`.

Each handler method defines a List of `TournamentContent` objects, where the backing class for a `TournamentContent` object is a POJO. This List is then assigned to the handler method's `Model` object in order for it to become accessible to the returning view. The returning logical views for each handler methods are `atomfeedtemplate` and `rssfeedtemplate`, respectively. These logical views are defined in the following manner inside a Spring configuration class:

```
package com.apress.springrecipes.court.web.config;

import com.apress.springrecipes.court.feeds.AtomFeedView;
import com.apress.springrecipes.court.feeds.RSSFeedView;
import com.apress.springrecipes.court.service.InMemoryMemberService;
import com.apress.springrecipes.court.service.MemberService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.HandlerMapping;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.court.web")
public class CourtRestConfiguration {

    @Bean
    public AtomFeedView atomfeedtemplate() {
        return new AtomFeedView();
    }

    @Bean
    public RSSFeedView rssfeedtemplate() {
        return new RSSFeedView();
    }

    ...
}
```

As you can observe, each logical view is mapped to a class. Each of these classes is charged with implementing the necessary logic to build either an Atom or RSS view. If you recall from Chapter 4, you used an identical approach (i.e., using classes) for implementing PDF and Excel views.

In the case of Atom and RSS views, Spring comes equipped with two classes specially equipped and built on the foundations of Project Rome. These classes are `AbstractAtomFeedView` and `AbstractRssFeedView`. Such classes provide the foundations to build an Atom or RSS feed, without dealing in the finer details of each of these formats.

The following listing illustrates the AtomFeedView class which implements the AbstractAtomFeedView class and is used to back the atomfeedtemplate logical view:

```

package com.apress.springrecipes.court.feeds;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sun.syndication.feed.atom.Feed;
import com.sun.syndication.feed.atom.Entry;
import com.sun.syndication.feed.atom.Content;

import org.springframework.web.servlet.view.feed.AbstractAtomFeedView;

import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;

public class AtomFeedView extends AbstractAtomFeedView {
    protected void buildFeedMetadata(Map model, Feed feed, HttpServletRequest request) {
        feed.setId("tag:tennis.org");
        feed.setTitle("Grand Slam Tournaments");
        List<TournamentContent> tournamentList = (List<TournamentContent>)model.get("feedContent");
        for (TournamentContent tournament : tournamentList) {
            Date date = tournament.getPublicationDate();
            if (feed.getUpdated() == null || date.compareTo(feed.getUpdated()) > 0) {
                feed.setUpdated(date);
            }
        }
    }

    protected List buildFeedEntries(Map model,
                                    HttpServletRequest request, HttpServletResponse response) throws Exception {
        List<TournamentContent> tournamentList = ←
            (List<TournamentContent>)model.get("feedContent");
        List<Entry> entries = new ArrayList<Entry>(tournamentList.size());
        for (TournamentContent tournament : tournamentList) {
            Entry entry = new Entry();
            String date = String.format("%1$tY-%1$tm-%1$td", ←
                tournament.getPublicationDate());
            entry.setId(String.format("tag:tennis.org,%s:%d", date, ←
                tournament.getId()));
            entry.setTitle(String.format("%s - Posted by %s", ←
                tournament.getName(), tournament.getAuthor()));
            entry.setUpdated(tournament.getPublicationDate());
            Content summary = new Content();

```

```

        summary.setValue(String.format("%s - %s", tournament.getName(), tournament.getLink()));
        entry.setSummary(summary);
        entries.add(entry);
    }
    return entries;
}
}

```

The first thing to notice about this class is that it imports several Project Rome classes from the `com.sun.syndication.feed.atom` package, in addition to implementing the `AbstractAtomFeedView` class provided by the Spring framework. In doing so, the only thing that's needed next is to provide a feed's implementation details for two methods inherited from the `AbstractAtomFeedView` class: `buildFeedMetadata` and `buildFeedEntries`.

The `buildFeedMetadata` has three input parameters. A `Map` object which represents the data used to build the feed (i.e., data assigned inside the handler method, in this case a `List` of `TournamentContent` objects), a `Feed` object based on a Project Rome class that is used to manipulate the feed itself, and an `HttpServletRequest` object in case it's necessary to manipulate the HTTP request.

Inside the `buildFeedMetadata` method, you can observe several calls are made to the `Feed` object's setter methods (e.g., `setId`, `setTitle`, `setUpated`). Two of these calls are made using hard-coded strings, while another is made with a value determined after looping over a feed's data (i.e., the `Map` object). All these calls represent the assignment of an Atom feed's metadata information.

Note Consult Project Rome's API if you want to assign more values to an Atom feed's metadata section, as well as specify a particular Atom version. The default version is Atom 1.0.

The `buildFeedEntries` method also has three input parameters: a `Map` object that represents the data used to build the feed (i.e., data assigned inside the handler method, in this case a `List` of `TournamentContent` objects), an `HttpServletRequest` object in case it's necessary to manipulate the HTTP request, and an `HttpServletResponse` object in case it's necessary to manipulate the HTTP response. It's also important to note the `buildFeedEntries` method returns a `List` objects, which in this case corresponds to a `List` of `Entry` objects based on a Project Rome class and containing an Atom feed's recurring elements.

Inside the `buildFeedEntries` method, you can observe that the `Map` object is accessed to obtain the `feedContent` object assigned inside the handler method. Once this is done, an empty `List` of `Entry` objects is created. Next, a loop is performed on the `feedContent` object, which contains a list of a `List` of `TournamentContent` objects, and for each element, an `Entry` object is created that is assigned to the top-level `List` of `Entry` objects. Once the loop is finished, the method returns a filled `List` of `Entry` objects.

Note Consult Project Rome's API if you want to assign more values to an Atom feed's recurring elements section.

Upon deploying this last class, in addition to the previously cited Spring MVC controller, accessing a URL in the form `http://[host_name]/[app-name]/atomfeed.atom` (or `http://[host_name]/atomfeed.xml`) would result in the following response:

```

<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title>Grand Slam Tournaments</title>
    <id>tag:tennis.org</id>

```

```

<updated>2010-03-04T20:51:50Z</updated>
<entry>
  <title>Australian Open - Posted by ATP</title>
  <id>tag:tennis.org,2010-03-04:0</id>
  <updated>2010-03-04T20:51:50Z</updated>
  <summary>Australian Open - www.australianopen.com</summary>
</entry>
<entry>
  <title>Roland Garros - Posted by ATP</title>
  <id>tag:tennis.org,2010-03-04:1</id>
  <updated>2010-03-04T20:51:50Z</updated>
  <summary>Roland Garros - www.rolandgarros.com</summary>
</entry>
<entry>
  <title>Wimbledon - Posted by ATP</title>
  <id>tag:tennis.org,2010-03-04:2</id>
  <updated>2010-03-04T20:51:50Z</updated>
  <summary>Wimbledon - www.wimbledon.org</summary>
</entry>
<entry>
  <title>US Open - Posted by ATP</title>
  <id>tag:tennis.org,2010-03-04:3</id>
  <updated>2010-03-04T20:51:50Z</updated>
  <summary>US Open - www.usopen.org</summary>
</entry>
</feed>

```

Turning your attention to the remaining handler method—getRSSFeed—from the previous Spring MVC controller charged with building an RSS feed, you’ll see that the process is similar to the one just described for building Atom feeds.

The handler methods also creates a `List` of `TournamentContent` objects, which is then assigned to the handler method’s `Model` object for it to become accessible to the returning view. The returning logical view in this case though, now corresponds to one named `rssfeedtemplate`. As described earlier, this logical view is mapped to a class named `RssFeedView`.

The following listing illustrates the `RssFeedView` class, which implements the `AbstractRssFeedView` class:

```

package com.apress.springrecipes.court.feeds;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.sun.syndication.feed.rss.Channel;
import com.sun.syndication.feed.rss.Item;
import org.springframework.web.servlet.view.feed.AbstractRssFeedView;

import java.util.Date;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;

```

```

public class RSSFeedView extends AbstractRssFeedView {
    protected void buildFeedMetadata(Map model, Channel feed, HttpServletRequest request) {
        feed.setTitle("World Soccer Tournaments");
        feed.setDescription("FIFA World Soccer Tournament Calendar");
        feed.setLink("tennis.org");
        List<TournamentContent> tournamentList = (List<TournamentContent>)model.get("feedContent");
        for (TournamentContent tournament : tournamentList) {
            Date date = tournament.getPublicationDate();
            if (feed.getLastBuildDate() == null || date.compareTo(feed.getLastBuildDate()) > 0) {
                feed.setLastBuildDate(date);
            }
        }
    }

    protected List buildFeedItems(Map model,
        HttpServletRequest request, HttpServletResponse response)
    throws Exception {
        List<TournamentContent> tournamentList = (List<TournamentContent>)model.get("feedContent");
        List<Item> items = new ArrayList<Item>(tournamentList.size());
        for (TournamentContent tournament : tournamentList) {
            Item item = new Item();
            String date = String.format("%1$tY-%1$tm-%1$td", tournament.getPublicationDate());
            item.setAuthor(tournament.getAuthor());
            item.setTitle(String.format("%s - Posted by %s", tournament.getName(), tournament.getAuthor()));
            item.setPubDate(tournament.getPublicationDate());
            item.setLink(tournament.getLink());
            items.add(item);
        }
        return items;
    }
}

```

The first thing to notice about this class is that it imports several Project Rome classes from the `com.sun.syndication.feed.rss` package, in addition to implementing the `AbstractRssFeedView` class provided by the Spring framework. Once it does so, the only thing that's needed next is to provide a feed's implementation details for two methods inherited from the `AbstractRssFeedView` class: `buildFeedMetadata` and `buildFeedItems`.

The `buildFeedMetadata` method is similar in nature to the one by the same name used in building an Atom feed. Notice the `buildFeedMetadata` method manipulates a `Channel` object based on a Project Rome class, which is used to build RSS feeds, instead of a `Feed` object, which is used to build Atom feeds. The setter method calls made on the `Channel` object (e.g., `setTitle`, `setDescription`, `setLink`) represent the assignment of an RSS feed's metadata information.

The `buildFeedItems` method, which differs in name to its Atom counterpart `buildFeedEntries`, is so named because an Atom feed's recurring elements are called *entries* and an RSS feed's recurring elements are *items*. Naming conventions aside, their logic is similar.

Inside the `buildFeedItems` method, you can observe that the `Map` object is accessed to obtain the `feedContent` object assigned inside the `handler` method. Once this is done, an empty `List` of `Item` objects is created. Next, a loop is performed on the `feedContent` object, which contains a list of a `List` of `TournamentContent` objects, and for each element, an `Item` object is created which is assigned to the top level `List` of `Item` objects. Once the loop is finished, the method returns a filled `List` of `Item` objects.

Note Consult Project Rome's API if you want to assign more values to an RSS feed's metadata and recurring element sections, as well as specify a particular RSS version. The default version is RSS 2.0.

When you deploy this last class, in addition to the previously cited Spring MVC controller, accessing a URL in the form `http://[host_name]/rssfeed.rss` (or `http://[host_name]/rssfeed.xml`) results in the following response:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>World Soccer Tournaments</title>
    <link>tennis.org</link>
    <description>FIFA World Soccer Tournament Calendar</description>
    <lastBuildDate>Thu, 04 Mar 2010 21:45:08 GMT</lastBuildDate>
    <item>
      <title>World Cup - Posted by FIFA</title>
      <link>www.fifa.com/worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      272103_1_EnFIFA</author>
    </item>
    <item>
      <title>U-20 World Cup - Posted by FIFA</title>
      <link>www.fifa.com/u20worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      272103_1_EnFIFA</author>
    </item>
    <item>
      <title>U-17 World Cup - Posted by FIFA</title>
      <link>www.fifa.com/u17worldcup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      272103_1_EnFIFA</author>
    </item>
    <item>
      <title>Confederations Cup - Posted by FIFA</title>
      <link>www.fifa.com/confederationscup/</link>
      <pubDate>Thu, 04 Mar 2010 21:45:08 GMT</pubDate>
      272103_1_EnFIFA</author>
    </item>
  </channel>
</rss>
```

Summary

In this chapter, you have learned how to develop and access REST services using Spring. REST services are closely tied to Spring MVC, whereby a controller acts to dispatch requests made to REST services, as well as access third-party REST services to use this information for application content.

You learned how REST services leverage annotations used in Spring MVC controllers, which included `@RequestMapping` to indicate service end points, as well as `@PathVariable` to specify access parameters for filtering a service's payload. In addition, you learned about Spring's XML marshallers, such as `Jaxb2Marshaller`, which allow application objects to be transformed into XML and be output as a REST service's payload.

You also learned about Spring's `RestTemplate` class and how it supports the series of HTTP protocol methods that include HEAD, GET, POST, PUT, and DELETE—all of which allow you to access and perform operations on third-party REST services directly from the context of a Spring application.

Finally you explored how to publish Atom and RSS feeds in a Spring application by leveraging the Project Rome API.

CHAPTER 6



Spring Social

Social networking is everywhere and most Internet users have one or more social networking accounts. People tweet to share what they are doing or how they feel about a subject; they share pictures on Facebook and Instagram, and write blogs using Tumblr. More and more social networks are appearing every day.

As the owner of a website it can be beneficial to add integration with those social networks, allowing users to easily post links or to filter and show how people think.

Spring Social tries to have a unified API to connect to those different networks and an extension model. Spring Social itself provides integration for Facebook, Twitter, and LinkedIn; however, there are a lot of community projects providing support for different social networks (like Tumblr, Weibo, and Instagram to name a few).

Spring Social can be split into three parts. First there is the Connect Framework, which handles the authentication and connection flow with the underlying social network. Next, is the `ConnectController` is the controller doing the OAuth exchange between the service provider, the consumer (the application), and the user of the application. Finally there is the `SocialAuthenticationFilter` which integrates Spring Social with Spring Security (see Chapter 8) to allow users to sign in with their social network account.

6-1. Setting Up Spring Social

Problem

You want to use Spring Social in your application.

Solution

Add Spring Social to your dependencies and enable Spring Social in your configuration.

How It Works

Spring Social consists of several modules apart from all the different modules for each service provider (like Twitter, Facebook, GitHub, etc.). To be able to use Spring Social you will need to add those to your application's dependencies. Table 6-1 shows the available modules.

Table 6-1. Overview of Spring Social Modules

Module	Description
spring-social-core	Core module of Spring Social, contains the main and shared infrastructure classes.
spring-social-config	Spring Social configuration module, makes it easier to configure (parts) of Spring Social
spring-social-web	Web integration for Spring Social contains filters and controllers for easy use.
spring-social-security	Integration with Spring Security (see chapter 8).

To dependencies are in the group org.springframework.social, this chapter will cover every module (core, config, web, and security) in different recipes. This section will cover the basic setup of Spring Social. At the moment of writing 1.1.0.RELEASE was the latest version of Spring Social available. Add the following dependencies (when using maven).

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-core</artifactId>
    <version>1.1.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-config</artifactId>
    <version>1.1.0.RELEASE</version>
</dependency>
```

With the dependencies added Spring Social can now be setup.

```
package com.apress.springrecipes.social.config;

import com.apress.springrecipes.social.StaticUserIdSource;
import org.springframework.context.annotation.*;
import org.springframework.core.env.Environment;
import org.springframework.social.config.annotation.EnableSocial;
import org.springframework.social.config.annotation.SocialConfigurerAdapter;

@Configuration
@EnableSocial
@PropertySource("classpath:/application.properties")
public class SocialConfig extends SocialConfigurerAdapter {

    @Override
    public StaticUserIdSource getUserIdSource() {
        return new StaticUserIdSource();
    }
}
```

To enable Spring Social simply add the `@EnableSocial` annotation to a `@Configuration` annotated class. This annotation will trigger loading of the configuration of Spring Social. It will detect any instance of `SocialConfigurer` beans, these beans are used for further configuration of Spring Social. Those are used to add the configuration for 1 or more service providers.

The SocialConfig extends SocialConfigurerAdapter which is an implementation of a SocialConfigurer, as you can see there is an overridden method getUserIdSource which returns a StaticUserIdSource. Spring Social requires an instance of a UserIdSource to determine the current user. This user is used to lookup any connections with service providers. These connections are stored in a, per user, ConnectionRepository. The ConnectionRepository to use is determined by the UsersConnectionRepository, which uses the current user for that. The default configured UsersConnectionRepository is the InMemoryUsersConnectionRepository.

Finally you are going to load a properties file from the classpath, this properties file contains the api keys for your application to use for service providers. Instead of putting them in a properties file you could also hardcode them into your code.

For the time being you are going to use the StaticUserIdSource to determine the current user.

```
package com.apress.springrecipes.social;

import org.springframework.social.UserIdSource;

public class StaticUserIdSource implements UserIdSource {

    private static final String DEFAULT_USERID = "anonymous";

    private String userId = DEFAULT_USERID;

    @Override
    public String getUserId() {
        return this.userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }
}
```

The StaticUserIdSource implements UserIdSource and returns a preset userId, although this works for now but in a real application you want to be able to store the connection information on a per user basis.

6-2. Connecting to Twitter

Problem

You want your application to have access to Twitter.

Solution

Register your application with Twitter and configure Spring Social to make use of the application credentials to get access to Twitter.

How It Works

Before you can have your application use Twitter you need to register your application with Twitter. After this registration you will have credentials (API key and API secret) to identify your application.

Register an Application on Twitter

To register an application with Twitter go to <https://dev.twitter.com> and look in the right-hand top corner for your avatar, now from the dropdown menu select My applications (see Figure 6-1).



Figure 6-1. Selecting my applications on twitter

After selection My Applications the Application Management page will appear. On this page is a button which allows you to create new apps (see Figure 6-2).

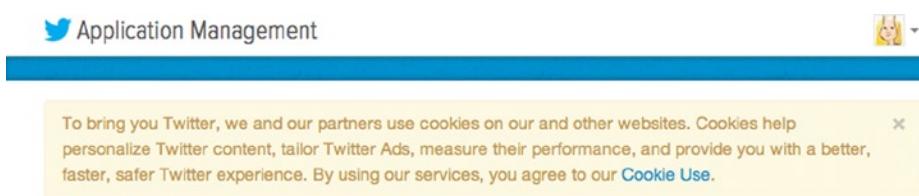


Figure 6-2. Application Management Page

On this page, press the button to open the screen (see Figure 6-3) to register your application.

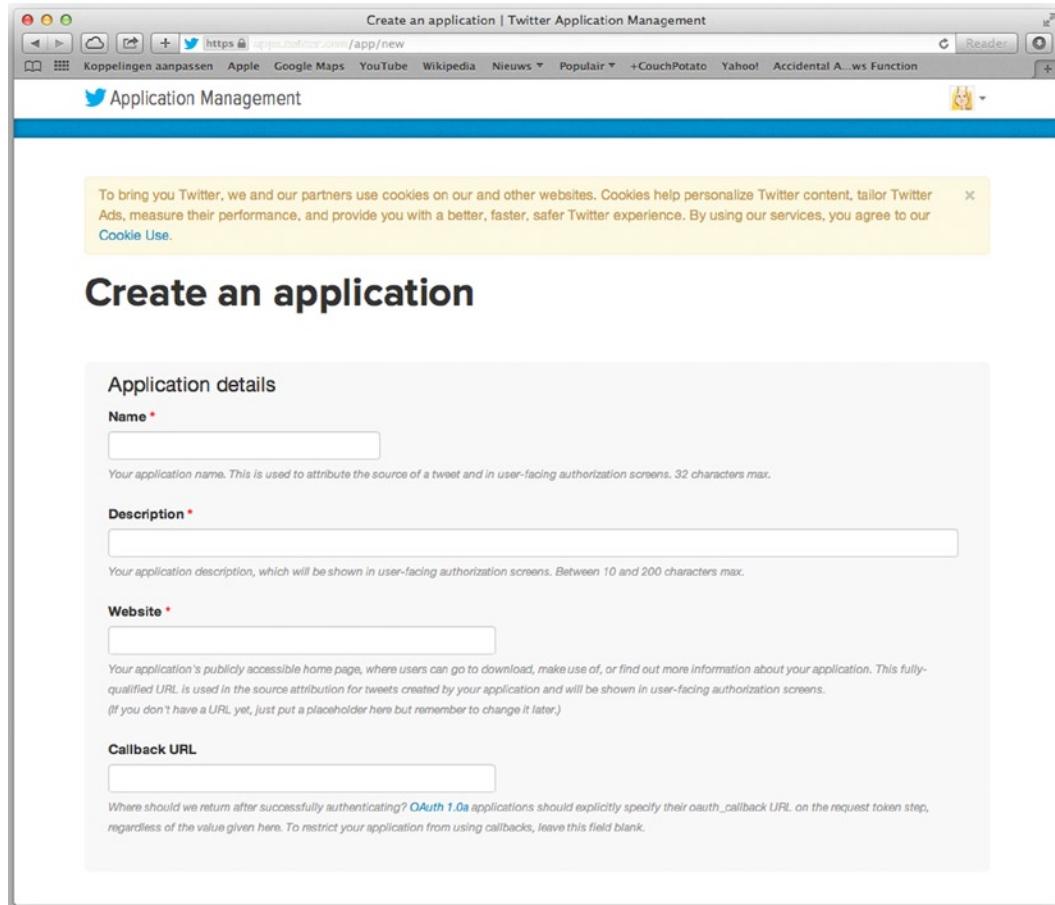


Figure 6-3. Register a new application

Now in this screen you must enter a name and description of your application and it also requires a URL of the website on which this application is going to be used. When using Spring Social it is also important that you fill-out the callback url field as we need callbacks, the actual value doesn't really matter (unless you use a very old version of OAuth).

After accepting the terms and conditions and pressing the final create button you will be taken to your application settings page and you successfully created your application.

To be able to connect Spring Social to Twitter you need to know your API key and API secret, those can be found on the API Keys tab of your application settings (see Figures 6-4 and 6-5).

SRSocialDemo

[Test OAuth](#)


Figure 6-4. Application Settings page

SRSocialDemo

The screenshot shows the 'API Keys' tab selected. A red box highlights the 'API key' and 'API secret' fields, which are currently empty. Below them, other application details are listed: Access level (Read-only (modify app permissions)), Owner (mdeinum), and Owner ID (86901511).

Figure 6-5. API key and API secret needed to connect Spring Social

Configure Spring Social to Connect with Twitter

Now that you have an API key and API secret you can configure Spring Social to connect to Twitter. First create a properties file (for instance application.properties) to hold your API key and API secret so that we can easily retrieve it when we need it.

```
twitter.appId=<your-twitter-API-key-here>
twitter.appSecret=<your-twitter-API-secret-here>
```

To connect to Twitter you need to add a TwitterConnectionFactory which will use the application id and secret when requested to connect to Twitter.

```
package com.apress.springrecipes.social.config;

import org.springframework.core.env.Environment;
import org.springframework.social.config.annotation.ConnectionFactoryConfigurer;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.ConnectionRepository;
import org.springframework.social.twitter.api.Twitter;
import org.springframework.social.twitter.connect.TwitterConnectionFactory;
```

```

@Configuration
@EnableSocial
@PropertySource("classpath:/application.properties")
public class SocialConfig extends SocialConfigurerAdapter {
    ...
    @Configuration
    public static class TwitterConfigurer extends SocialConfigurerAdapter {
        @Override
        public void addConnectionFactories(
            ConnectionFactoryConfigurer connectionFactoryConfigurer,
            Environment env) {
            connectionFactoryConfigurer.addConnectionFactory(
                new TwitterConnectionFactory(
                    env.getRequiredProperty("twitter.appId"),
                    env.getRequiredProperty("twitter.appSecret")));
        }
    }
}

```

The SocialConfigurer interface has the callback method addConnectionFactories which allows you to add ConnectionFactory instances to use to Spring Social. For Twitter there is the TwitterConnectionFactory which takes two arguments, the first is the API key, the second is the API secret. Both constructor arguments come from the properties file that is read. Of course you could also hardcode the values into the configuration.

The connection to Twitter has been made, although you could use the raw underlying connection it isn't really recommended to do so. Instead use the TwitterTemplate which makes it easier to work with the Twitter API.

The configuration above adds a TwitterTemplate to the application context. Notice the @Scope annotation. It is important that this bean is request scoped. For each request the actual connection to Twitter might differ as potentially every request is for a different user. Hence the request scoped bean. The ConnectionRepository that is injected into the method is determined based on the id of the current user, which is retrieved using the UserIdSource you configured earlier.

Note Although the sample uses a separate configuration class to configure Twitter as a service provider you can also add it to the main SocialConfig class. However it can be desirable to separate the global Spring Social configuration from the specific service provider setup.

6-3. Connecting to Facebook

Problem

You want your application to have access to Facebook.

Solution

Register your application with Facebook and configure Spring Social to make use of the application credentials to get access to Facebook.

How It Works

Before you can have your application use Facebook you first need to register your application with Facebook. After this registration you will have credentials (API key and API secret) to identify your application. To be able to register an application on Facebook you have to have a Facebook account and have to be registered as a developer. (This recipe assumes you already have been registered as a developer with Facebook. If not go to <http://developers.facebook.com> and click the Register Now button and fill out the wizard).

Register an Application on Facebook

Start by going to <http://developers.facebook.com> and click the Apps menu on top of the page and select Create a New App (see Figure 6-6).

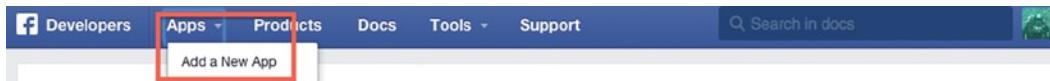
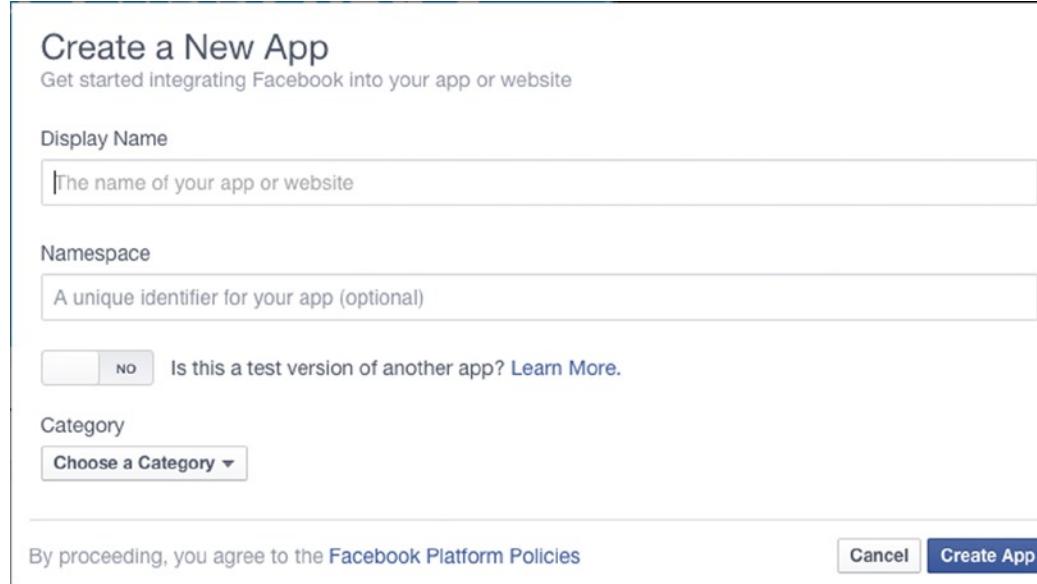


Figure 6-6. First steps in registering a new app

This will open a screen (see Figure 6-7) which allows you to fill in some details about your application.



The screenshot shows the 'Create a New App' form on the Facebook developer portal. It includes fields for Display Name, Namespace, Test Status, Category selection, and a link to Platform Policies, along with 'Cancel' and 'Create App' buttons.

Create a New App
Get started integrating Facebook into your app or website

Display Name

Namespace

NO Is this a test version of another app? [Learn More.](#)

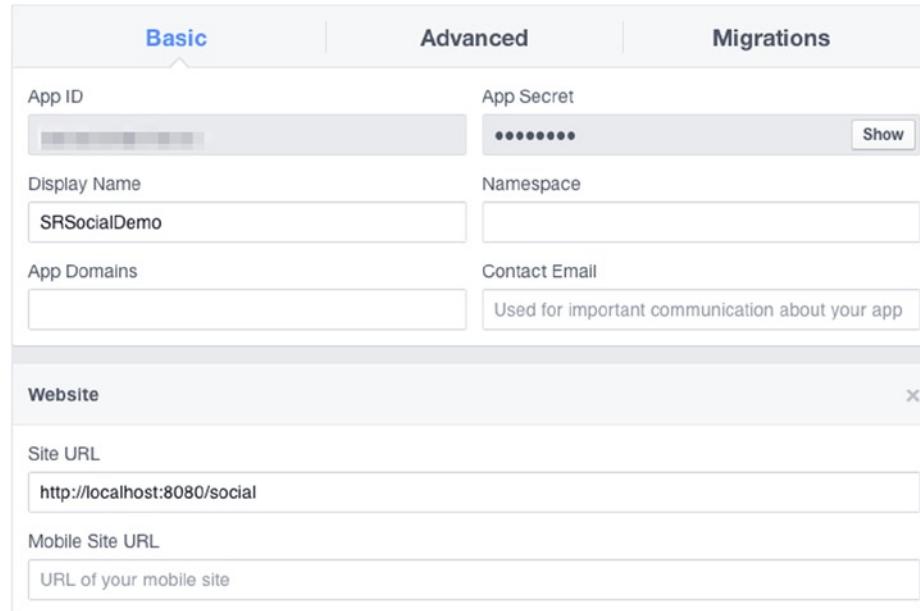
Category
[Choose a Category ▾](#)

By proceeding, you agree to the [Facebook Platform Policies](#)

[Cancel](#) [Create App](#)

Figure 6-7. Create a new App window

The name of your application can be anything as long as it doesn't contain the word "face" or "book". The namespace is used by the Facebook Graph API it is limit to only contain lowercase characters and of course has to be unique over all Facebook applications. Finally you need to select the category in which your application belongs. When you decided on all these things press the Create App button, which will take you to your application page (see Figure 6-8).



The screenshot shows the 'Basic' tab of the Facebook Settings page. It displays the App ID, App Secret (with a 'Show' button), Display Name (set to 'SRSocialDemo'), Namespace, App Domains, Contact Email, Website URL (set to 'http://localhost:8080/social'), and Mobile Site URL.

Basic		Advanced	Migrations
App ID	App Secret		
<input type="text" value="XXXXXXXXXX"/>	<input type="text" value="*****"/> Show		
Display Name	Namespace		
<input type="text" value="SRSocialDemo"/>	<input type="text"/>		
App Domains	Contact Email		
<input type="text"/>	<input type="text" value="Used for important communication about your app"/>		
Website X			
Site URL <input type="text" value="http://localhost:8080/social"/>			
Mobile Site URL <input type="text" value="URL of your mobile site"/>			

Figure 6-8. Facebook Settings page

On this page navigate to the Settings tab.

On the settings page enter the URL of the site your app is going to be part of. In this exercise that is <http://localhost:8080/social>. If this URL isn't present, authorization will not be granted and the connection will be never made.

Configure Spring Social to Connect with Facebook

This Facebook settings page also contains the application id and secret needed by the application to connect to Facebook. Put them in the `application.properties` file.

```
facebook appId=<your app-id here>
facebook appSecret=<your app-secret here>
```

Assuming Spring Social is already setup (see Recipe 6-1) it is a matter of adding a `FacebookConnectionFactory` and `FacebookTemplate` for easy access.

```
package com.apress.springrecipes.social.config;

import org.springframework.social.facebook.api.Facebook;
import org.springframework.social.facebook.connect.FacebookConnectionFactory;

@Configuration
@EnableSocial
@PropertySource("classpath:/application.properties")
public class SocialConfig extends SocialConfigurerAdapter {
    ...
    @Configuration
    public static class FacebookConfiguration extends SocialConfigurerAdapter {
        @Override
        public void addConnectionFactories(
            ConnectionFactoryConfigurer connectionFactoryConfigurer,
            Environment env) {
            connectionFactoryConfigurer.addConnectionFactory(
                new FacebookConnectionFactory(
                    env.getRequiredProperty("facebook.appId"),
                    env.getRequiredProperty("facebook.appSecret")));
        }
        ...
    }
}
```

The FacebookConnectionFactory needs the application id and secret. Both properties are added to the application.properties file and are available through the Environment object.

The bean configuration above adds a bean named facebookTemplate to the application context. Notice the @Scope annotation. It is important that this bean is request scoped. For each request the actual connection to Facebook might differ as potentially every request is for a different user. Hence the request scoped bean. The ConnectionRepository that is injected into the method is determined based on the id of the current user, which is retrieved using the UserIdSource you configured earlier (see Recipe 6-1).

Note Although the sample uses a separate configuration class to configure Facebook as a service provider you can also add it to the main SocialConfig class. However it can be desirable to separate the global Spring Social configuration from the specific service provider setup.

6-4. Showing Service Provider Connection Status

Problem

You want to display the status of the connections of the used service providers.

Solution

Configure the ConnectController and use it to show the status to the user.

How It Works

Spring Social comes with a ConnectController which takes care of connecting and disconnecting to a service provider but you can also show the status (connected or not) of the current user for the used service providers. The ConnectController uses several REST URLs to either show, add, or remove the connection for the given user (see Table 6-2).

Table 6-2. ConnectController URL mapping

URL	Method	Description
/connect	GET	Display the connection status of all available service providers. Will return connect/status as the name of the view to render.
/connect/{provider}	GET	Display the connection status of the given provider. When connected will return connect/{provider}Connected else will return connect/{provider}Connect as the view to render.
/connect/{provider}	POST	Starts the connection flow with the given provider.
/connect/{provider}	DELETE	Deletes all connections for the current user with the given provider.

To be able to use the controller you first need to configure Spring MVC (see Chapter 4). For this add the following configuration.

```
package com.apress.springrecipes.social.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.social.connect.ConnectionFactoryLocator;
import org.springframework.social.connect.ConnectionRepository;
import org.springframework.social.connect.web.ConnectController;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@EnableWebMvc
@ComponentScan({"com.apress.springrecipes.social.web"})
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public ViewResolver internalResourceViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
    }
}
```

You need to enable Spring MVC using `@EnableWebMvc`, and add a `ViewResolver` so the jsp pages can be picked up. Finally you want to show the `index.jsp` when the application starts up.

Next add the `ConnectController` to the `WebConfig`, this controller needs `ConnectionFactoryLocator` and `ConnectionRepository` as constructor arguments. To access those simply add them as method arguments.

```
@Bean
public ConnectController connectController(
    ConnectionFactoryLocator connectionFactoryLocator,
    ConnectionRepository connectionRepository) {
    return new ConnectController(connectionFactoryLocator, connectionRepository);
}
```

The `ConnectController` will listen to the URLs as listed in Table 6-2. Now add two views in the `/WEB-INF/views` directory. The first is the main index the second is the status overview page.

First create the `index.jsp`

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
<head>
    <title>Hello Spring Social</title>
</head>
<body>
<h3>Connections</h3>
    Click <a href="

```

Next create the status.jsp in the /WEB-INF/views/connect directory.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
    <title>Spring Social - Connections</title>
</head>
<body>
<h3>Spring Social - Connections</h3>

<c:forEach items="${providerIds}" var="provider">
    <h4>$provider</h4>
    <c:if test="${not empty connectionMap[provider]}>
        You are connected to ${provider} as ${connectionMap[provider][0].displayName}
    </c:if>
    <c:if test="${empty connectionMap[provider]}>
        <div>
            You are not yet connected to ${provider}. Click <a href="

```

The status page will iterate over all available providers and will determine if there is an existing connection for the current user for that service provider (twitter, facebook, etc.). The ConnectController will make the list of providers available under the providerIds attribute and the connectionMap holds the connections of the current user.

Now to bootstrap the application you will need to create a WebApplicationInitializer that will register a ContextLoaderListener and DispatcherServlet to handle the requests.

```
package com.apress.springrecipes.social;

import com.apress.springrecipes.social.config.SocialConfig;
import com.apress.springrecipes.social.config.WebConfig;
import org.springframework.web.filter.DelegatingFilterProxy;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

import javax.servlet.Filter;
```

```

public class SocialWebApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{SocialConfig.class};
    }

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

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

```

This will bootstrap the application. The `SocialConfig` will be loaded by the `ContextLoaderListener` and the `WebConfig` will be loaded by the `DispatcherServlet`. To be able to handle requests there needs to be a servlet mapping, for this the mapping will be `/`.

Now that everything is configured the application can be deployed and accessed by the url `http://localhost:8080/social`. This will show the index page. Clicking the link will show the connection status page, which initially will show that the current user isn't connected.

Connecting to a Service Provider

Now when clicking on a link to connect to a service provider the user will be sent to the `/connect/{provider}` url. When there isn't a connection the `connect/{provider}Connect` page will be rendered or the `connect/{provider}Connected` page would be shown.

To be able to use the `ConnectController` to connect to Twitter you need to add the `twitterConnect.jsp` and `twitterConnected.jsp`. For Facebook it would be `facebookConnect.jsp` and `facebookConnected.jsp`. The same pattern applies to all other service provider connectors for Spring Social (like GitHub, FourSquare, LinkedIn, ...).

First add the `twitterConnect.jsp` to the `/WEB-INF/views/connect` directory.

```

<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
<head>
    <title>Spring Social - Connect to Twitter</title>
</head>
<body>
<h3>Connect to Twitter</h3>

<form action="

```

Notice the form tag which POSTs the form back to the same URL. When pressing the submit button you will be redirected to Twitter which will ask for your permission to allow this application to access your Twitter profile. (Replace this with facebook to connect to Facebook.).

Next add the `twitterConnected.jsp` to the `/WEB-INF/views/connect` directory. This is the page that will be displayed when you are already connected to Twitter but also when you return from Twitter after authorizing the application.

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<html>
<head>
    <title>Spring Social - Connected to Twitter</title>
</head>
<body>
<h3>Connected to Twitter</h3>

<p>
    You are now connected to your Twitter account.
    Click <a href=<spring:url value='/connect' />>here</a> to see your Connection Status.
</p>
</body>
</html>
```

When these pages are added reboot the application and navigate to the status page. Now when clicking the Connect to Twitter link you will send to the `twitterConnect.jsp`. After clicking the Connect to Twitter button you will be shown the Twitter authorize application page (see Figure 6-9).

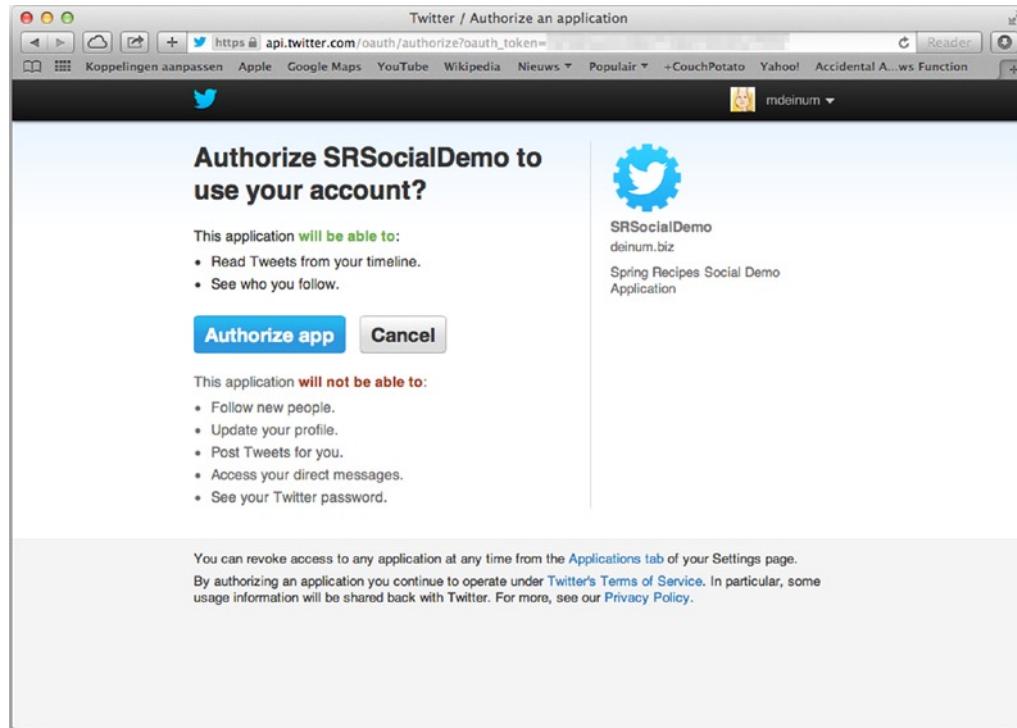


Figure 6-9. Twitter Authorize Page

After authorizing the application you will be returned to the `twitterConnect.jsp` telling you that you have successfully connected to Twitter. When returning to the status page you will see that you are connected to Twitter with your nickname.

For Facebook or any other service provider follow the same steps of adding the `{provider}Connect` and `{provider}Connected` page and Spring Social will be able to connect to that provider, given that you also added the correct service provider connector and configuration.

6-5. Using the Twitter API

Problem

You want to use the Twitter API.

Solution

Use the Twitter object to access the Twitter API.

How It Works

Each service provider has its own API using Twitter. There is an object implementing the Twitter interface which represents the Twitter API in Java; for Facebook an object implementing the Facebook interface is available. In Recipe 6-2 you already set up the connection to Twitter and the `TwitterTemplate`.

The `TwitterTemplate` exposes various parts of the Twitter API (see Table 6-3).

Table 6-3. Exposed operations of the Twitter API

Operations	Description
<code>blockOperations()</code>	Block and unblock users
<code>directMessageOperations()</code>	Reading and sending direct messages
<code>friendOperations()</code>	Retrieving a user's list of friends and followers and following/unfollowing users
<code>geoOperations()</code>	Working with locations
<code>listOperations()</code>	Maintaining, subscribing to, and unsubscribing from user lists
<code>searchOperations()</code>	Searching tweets and viewing search trends
<code>streamingOperations()</code>	Receive tweets as they are created via Twitter's Streaming API
<code>timelineOperations()</code>	Reading timelines and posting tweets
<code>userOperations()</code>	Retrieving user profile data
<code>restOperations()</code>	The underlying <code>RestTemplate</code> if a part of the API hasn't been exposed through the other APIs

It might be that for certain operations your application requires more access than read-only. If you want to send tweets you need read-write access and to be able to access direct messages you need Read, Write, and access to direct messages.

To post a status update you would use the `timelineOperations()` and then the `updateStatus()` method. Depending on your needs it either takes a simple String, the status, or a value object `TweetData` holding the status and other information like location, if it is a reply to another tweet and optionally any resources like images.

A simple controller could look like the following.

```
package com.apress.springrecipes.social.web;

import org.springframework.social.twitter.api.Twitter;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/twitter")
public class TwitterController {

    private final Twitter twitter;

    public TwitterController(Twitter twitter) {
        this.twitter = twitter;
    }

    @RequestMapping(method = RequestMethod.GET)
    public String index() {
        return "twitter";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String tweet(@RequestParam("status") String status) {
        twitter.timelineOperations().updateStatus(status);
        return "redirect:/twitter";
    }
}
```

The controller needs the Twitter API through the `TwitterTemplate`. The `TwitterTemplate` implements the `Twitter` interface. As you might recall from Recipe 6-2 the API is request scoped. You get a scoped proxy hence the usage of the `Twitter` interface. The `tweet` method receives a parameter and passes that on to Twitter.

6-6. Using a Persistent `UsersConnectionRepository`

Problem

You want to persist the users' connection data to survive server restarts.

Solution

Use the `JdbcUsersConnectionRepository` instead of the default `InMemoryUsersConnectionRepository`.

How It Works

By default Spring Social automatically configures an `InMemoryUsersConnectionRepository` for storing the connection information for a user. However this doesn't work in a cluster nor does it survive server restarts. To solve this problem it is possible to use a database to store the connection information. This is enabled by the `JdbcUsersConnectionRepository`.

The `JdbcUsersConnectionRepository` requires a database containing a table named `UserConnection` containing a certain number of columns. Luckily Spring Social contains a DDL script, `JdbcUsersConnectionRepository.sql`, which you can use to create the table.

First add a `DataSource` to point to the database of your choice. In this case Derby is used but any database would do.

```
@Bean
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setUrl(env.getRequiredProperty("dataSource.url"));
    dataSource.setDriverClassName(env.getRequiredProperty("dataSource.driverClassName"));
    dataSource.setUsername(env.getProperty("dataSource.username"));
    dataSource.setPassword(env.getProperty("dataSource.password"));
    return dataSource;
}
```

Notice the `dataSource.*` properties, which are used to configure the URL, JDBC Driver, and username/password. Add the properties to the `application.properties` file.

```
dataSource.password=app
dataSource.username=app
dataSource.driverClassName=org.apache.derby.jdbc.ClientDriver
dataSource.url=jdbc:derby://localhost:1527/social;create=true
```

Now before starting the application you need to start Derby to accept connections. Assuming that you have Derby already available you can start it by running the `startNetworkServer` command which can be found in the `bin` directory of the Derby installation.

If you want automatic creation of the desired database table you will need to add a `DataSourceInitializer` and have it execute the `JdbcUsersConnectionRepository.sql` file.

```
@Bean
public DataSourceInitializer databasePopulator() {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScript(
        new ClassPathResource("org/springframework/social/connect/jdbc/JdbcUsersConnectionRepository.sql"));
    populator.setContinueOnError(true); // Continue in case the create script already ran

    DataSourceInitializer initializer = new DataSourceInitializer();
    initializer.setDatabasePopulator(populator);
    initializer.setDataSource(dataSource());
    return initializer;
}
```

This `DataSourceInitializer` is executed at application startup and will execute all the scripts handed to it. By default it will stop application startup as soon as an error is encountered. To stop this set the `continueOnError` property to true.

Now that the DataSource is setup and configured the final step is to add the JdbcUsersConnectionRepository to the SocialConfig class.

```
package com.apress.springrecipes.social.config;

import org.springframework.social.connect.jdbc.JdbcUsersConnectionRepository;
...
@Configuration
@EnableSocial
@PropertySource("classpath:/application.properties")
public class SocialConfig extends SocialConfigurerAdapter {

    @Override
    public UsersConnectionRepository getUsersConnectionRepository(
        ConnectionFactoryLocator connectionFactoryLocator) {
        return new JdbcUsersConnectionRepository(
            dataSource(), connectionFactoryLocator, Encryptors.noOpText());
    }

    @Bean
    public DataSource dataSource() { ... }
...
}
```

The JdbcUsersConnectionRepository takes three constructor arguments. The first is the DataSource, the second is the passed in ConnectionFactoryLocator, and the last argument is a TextEncryptor. The TextEncryptor is a class from the Spring Security crypto module and is used to encrypt the accessToken, secret, and, when available, the refresh token. The encryption is needed as when the data is stored as plain text and the data would be compromised. The tokens could be used to gain access to your profile information.

For testing however it can be handy to use the noOpText encryptor which, as the name implies, does no encryption. For real production you want to use a TextEncryptor which uses a password and salt to encrypt the values.

When the JdbcUsersConnectionRepository is configured and the database has been started you can restart the application. At first glance nothing has changed; however, as soon as you grant access to, for instance, Twitter, this access will survive application restarts. You can also query the database and see that the information is stored in the USERCONNECTION table.

6-7. Integrating Spring Social and Spring Security

Problem

You want to allow users of your website to connect their social network accounts.

Solution

Use the spring-social-security project to integrate both frameworks.

How It Works

Before you can use the Spring Security integration for Spring Social you will need to add the spring-social-security module to your dependencies.

```
<dependency>
    <groupId>org.springframework.social</groupId>
    <artifactId>spring-social-security</artifactId>
    <version>1.1.0.RELEASE</version>
</dependency>
```

This dependency will pull in all other necessary dependencies for Spring Security.

Next let's setup Spring Security. It goes beyond this recipe to discuss Spring Security in detail. For that check Chapter 7. The setup for this recipe is as follows.

```
Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .loginPage("/signin")
            .failureUrl("/signin?param.error=bad_credentials")
            .loginProcessingUrl("/signin/authenticate").permitAll()
            .defaultSuccessUrl("/connect")
            .and()
            .logout().logoutUrl("/signout").permitAll();
    }

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        JdbcUserDetailsManager userDetailsService = new JdbcUserDetailsManager();
        userDetailsService.setDataSource(dataSource);
        userDetailsService.setEnableAuthorities(true);
        return userDetailsService;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService(null));
    }
}
```

The `@EnableWebMvcSecurity` annotation will enable security for Spring MVC applications. It registers beans needed for Spring Security to operate. To do further configuration, like setting up security rules, one or more `WebSecurityConfigurers` can be added. To make it easier there is a `WebSecurityConfigurerAdapter` which one can extend.

The `configure(HttpSecurity http)` method takes care of setting up security. This particular configuration wants a user to be authenticated for every call that is made. If a user isn't already authenticated (i.e., has logged in to the application) he will be prompted with a login form. You will also notice that the `loginPage`, `loginProcessingUrl`, and `logoutUrl` are modified. This is done so that they match the default URLs from Spring Social.

Note If you want to keep the Spring Security defaults configure the `SocialAuthenticationFilter` explicitly and set the `signupUrl` and `defaultFailureUrl` properties.

With the `configure(AuthenticationManagerBuilder auth)` you add a `AuthenticationManager` which is used to determine if a user exists and if the correct credentials were entered. The `UserDetailsService` used is a `JdbcUserDetailsService` which, next to being a `UserDetailsService`, can also add and remove users from the repository. This will be needed when you add Social signin to the application.

The `JdbcUserDetailsService` uses a `DataSource` to read and write the data and the `enableAuthorities` properties is set to `true` so that any roles the user gets from the application are added to the database as well. To bootstrap the database add the `create_users.sql` script to the database populator configured in the previous recipe.

```
@Bean
public DataSourceInitializer databasePopulator() {
    ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
    populator.addScript(
        new ClassPathResource("org/springframework/social/connect/jdbc/JdbcUsersConnectionRepository.sql"));
populator.addScript(new ClassPathResource("sql/create_users.sql"));
    populator.setContinueOnError(true); // Continue in case the create scripts already ran

    DataSourceInitializer initializer = new DataSourceInitializer();
    initializer.setDatabasePopulator(populator);
    initializer.setDataSource(dataSource());
    return initializer;
}
```

Next, to be able to render the custom login or signin page it needs to be added as a view controller to the `WebConfig`. This tells that a request to `/signin` should render the `signin.jsp` page.

```
package com.apress.springrecipes.social.config;

...
@EnableWebMvc
@ComponentScan({"com.apress.springrecipes.social.web"})
public class WebConfig extends WebMvcConfigurerAdapter {
...
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("index");
registry.addViewController("/signin").setViewName("signin");
    }
}
```

The `signin.jsp` is a simple JSP page rendering a username and password input field and a submit button.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<body>
    <c:url var="formLogin" value="/signin/authenticate" />
    <c:if test="${param.error eq 'bad_credentials'}">
        <div class="error">
            The login information was incorrect please try again.
        </div>
    </c:if>
    <form method="post" action="${formLogin}">
        <input type="hidden" name="_csrf" value="${_csrf.token}" />
        <table>
            <tr>
                <td><label for="username">Username</label></td>
                <td><input type="text" name="username"/></td>
            </tr>
            <tr>
                <td><label for="password">Password</label></td>
                <td><input type="password" name="password"/></td>
            </tr>
            <tr><td colspan="2"><button>Login</button></td> </tr>
        </table>
    </form>
</body>
</html>
```

Notice the hidden input which contains a CSFR token (Cross Site Forgery Request token). This is to prevent malicious websites or javascript code to post to our URL. When using Spring Security with Java Config this is enabled by default. It can be disabled with `http.csrf().disable()` in the `SecurityConfig` class.

There are two final configuration pieces left. First this configuration needs to be loaded and second a filter needs to be registered to apply the security to our request. For this modify the `SocialWebApplicationInitializer` class.

```
package com.apress.springrecipes.social;

import com.apress.springrecipes.social.config.SecurityConfig;
import com.apress.springrecipes.social.config.SocialConfig;
import com.apress.springrecipes.social.config.WebConfig;
import org.springframework.web.filter.DelegatingFilterProxy;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

import javax.servlet.Filter;

public class SocialWebApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{SecurityConfig.class, SocialConfig.class};
    }
}
```

```

@Override
protected Filter[] getServletFilters() {
    DelegatingFilterProxy springSecurityFilterChain = new DelegatingFilterProxy();
    springSecurityFilterChain.setTargetBeanName("springSecurityFilterChain");
    return new Filter[]{springSecurityFilterChain};
}
...
}

```

First notice that the `SecurityConfig` class is added to the `getRootConfigClasses` method, this will take care of the configuration class being loaded. Next the `getServletFilters` method is added. This method is used to register filters to requests that are going to be handled by the `DispatcherServlet`. Spring Security, by default, registers a `Filter` in the application context named `springSecurityFilterChain`. To have this executed you need to add a `DelegatingFilterProxy`. The `DelegatingFilterProxy` will look up a bean of the type `Filter` for the specified `targetBeanName`.

Using Spring Security to Obtain the Username

In the previous recipes you used a `UserIdSource` implementation that returned a static username. If you have an application that is already using Spring Security you could use the `AuthenticationNameUserIdSource` which uses the `SecurityContext` (from Spring Security) to obtain the username of the authenticated current user. That username in turn is used to store and lookup the users connections with the different service providers.

```

@Configuration
@EnableSocial
@PropertySource("classpath:/application.properties")
public class SocialConfig extends SocialConfigurerAdapter {

    @Override
    public UserIdSource getUserIdSource() {
        return new AuthenticationNameUserIdSource();
    }

    ...
}

```

Notice the construction of the `AuthenticationNameUserIdSource`. This is all that is needed to be able to retrieve the username from Spring Security. It will do a lookup of the `Authentication` object from the `SecurityContext` and return the `name` property of the `Authentication`.

When restarting the application you will be prompted with a login form. Now login as `user1` with password `user1`.

Using Spring Social for Sign In

Letting the current user connect his social networks is nice. It would be better if a user could use his social network account(s) to sign in to the application. Spring Social provides tight integration with Spring Security to enable this. There are a couple of additional parts that need to be setup for this.

First Spring Social needs to be integrated with Spring Security. For this the `SpringSocialConfigurer` can be used and applied to the Spring Security configuration.

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        ...
        http.apply(new SpringSocialConfigurer());
    }
    ...
}
```

The `SpringSocialConfigurer` needs a `SocialUserDetailsService`. This is used to lookup a user based on the user's id. For this recipe an implementation that delegates to a normal `UserDetailsService` will do. The `SpringSocialConfigurer` registers the `SocialAuthenticationFilter` which starts the authentication flow with the selected service provider. The filter listens, by default, to the `/auth/{provider}` URL, where `{provider}` points to the service provider being used (i.e., Twitter, Facebook, etc.).

```
package com.apress.springrecipes.social.security;

import org.springframework.dao.DataAccessException;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.social.security.SocialUser;
import org.springframework.social.security.SocialUserDetails;
import org.springframework.social.security.SocialUserDetailsService;
import org.springframework.util.Assert;

public class SimpleSocialUserDetailsService implements SocialUserDetailsService {

    private final UserDetailsService userDetailsService;

    public SimpleSocialUserDetailsService(UserDetailsService userDetailsService) {
        Assert.notNull(userDetailsService, "UserDetailsService cannot be null.");
        this.userDetailsService = userDetailsService;
    }

    @Override
    public SocialUserDetails loadUserById(String userId) throws UsernameNotFoundException,
    DataAccessException {
        UserDetails user = userDetailsService.loadUserByUsername(userId);
        return new SocialUser(user.getUsername(), user.getPassword(), user.getAuthorities());
    }
}
```

Next add the links for your configured service providers to the signin page.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html>
<body>
...
<!-- TWITTER SIGNIN -->
<c:url var="twitterSigin" value="/auth/twitter"/>
<p><a href="#">Sign in with Twitter</a></p>

<!-- FACEBOOK SIGNIN -->
<c:url var="twitterSigin" value="/auth/facebook"/>
<p><a href="#">Sign in with Facebook</a></p>

</body>
</html>
```

The `SimpleSocialUserDetailsService` delegates the actual lookup to a `UserDetailsService` which is passed in through the constructor. When a user is retrieved it uses the retrieved information to construct a `SocialUser` instance. Finally this bean needs to be added to the configuration.

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public SocialUserDetailsService socialUserDetailsService(UserDetailsService userDetailsService) {
        return new SimpleSocialUserDetailsService(userDetailsService);
    }
    ...
}
```

This will allow a user to sign in with his social networking accounts; however, the application needs to know which user the account belongs to. If a user cannot be located for the specific social network a user needs to be created. Basically the application needs a way for users to sign up for the application. By default the `SocialAuthenticationFilter` redirects the user to the `/signup` url. You can create a controller which is attached to this URL and renders a form allowing the user to create an account.

```
package com.apress.springrecipes.social.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.provisioning.UserDetailsManager;
import org.springframework.social.connect.Connection;
import org.springframework.social.connect.web.ProviderSignInUtils;
import org.springframework.social.security.SocialUser;
```

```

import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.context.request.WebRequest;

import java.util.Collections;

@Controller
public class SignupController {

    private final ProviderSignInUtils providerSignInUtils = new ProviderSignInUtils();
    private final UserDetailsManager userDetailsManager;

    @Autowired
    public SignupController(UserDetailsManager userDetailsManager) {
        this.userDetailsManager = userDetailsManager;
    }

    @RequestMapping(value="/signup", method=RequestMethod.GET)
    public SignupForm signupForm(WebRequest request) {
        Connection<?> connection = providerSignInUtils.getConnectionFromSession(request);
        if (connection != null) {
            return SignupForm.fromProviderUser(connection.fetchUserProfile());
        } else {
            return new SignupForm();
        }
    }

    @RequestMapping(value="/signup", method=RequestMethod.POST)
    public String signup(@Validated SignupForm form, BindingResult formBinding,
    WebRequest request) {
        if (formBinding.hasErrors()) {
            return null;
        }
        SocialUser user = createUser(form, formBinding);
        if (user != null) {
            SecurityContextHolder.getContext().setAuthentication(
            new UsernamePasswordAuthenticationToken(user.getUsername(), null, null));
            providerSignInUtils.doPostSignUp(user.getUsername(), request);
            return "redirect:/";
        }
        return null;
    }
}

```

```

private SocialUser createUser(SignupForm form, BindingResult errors) {
    SocialUser user = new SocialUser(
        form.getUsername(),
        form.getPassword(),
        Collections.singleton(new SimpleGrantedAuthority("ROLE_USER")));
    userDetailsManager.createUser(user);
    return user;
}
}

```

First the `signupForm` method will be called as the initial request will be a GET request to the `/signup` URL. The `signupForm` method checks if a connection attempt has been done. This is delegated to the `ProviderSignInUtils` provided by Spring Social. If that is the case the retrieved `UserProfile` is used to prepopulate a `SignupForm`.

```

package com.apress.springrecipes.social.web;

import org.springframework.social.connect.UserProfile;

public class SignupForm {

    private String username;

    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public static SignupForm fromProviderUser(UserProfile providerUser) {
        SignupForm form = new SignupForm();
        form.setUsername(providerUser.getUsername());
        return form;
    }
}

```

The HTML form used for filling in the two fields.

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Sign Up</title>
</head>
<body>
<h3>Sign Up</h3>

<form:form modelAttribute="signupForm" method="POST">

    <table>
        <tr><td><form:label path="username" /></td><td><form:input path="username"/></td></tr>
        <tr><td><form:label path="password" /></td><td><form:password path="password"/></td></tr>
        <tr><td colspan="2"><button>Sign Up</button></td></tr>
    </table>

</form:form>

</body>
</html>
```

Note There is no hidden input for the CSRF tag here. Spring Security integrates tightly with Spring MVC and this field will be added automatically when using the Spring Framework form tags.

After the user filled out the form the `signup` method will be called. This will create a user with the given username and password. After the user is created a `Connection` is added for the entered username. Now that the connection has been made the user is logged in to the application and on subsequent visits can use the social network connection to log in to the application.

Summary

In this chapter you explored Spring Social. The first step taken was to register an application with a service provider and use the generated API key and secret to connect the application to that service provider.

Next you looked into connecting a user's account to the application so that it can be used to access user information; however, this will also allow you to use the service provider's API. For Twitter you could query a timeline or look at someone's friends.

To make the connections to the service providers more useful they are stored in a JDBC-based storage.

Finally you looked at how Spring Social can integrate with Spring Security and how it can be used to allow a service provider to sign in to your application.

CHAPTER 7

Spring Security

In this chapter, you will learn how to secure applications using the Spring Security framework, a subproject of the Spring framework. Spring Security was initially known as Acegi Security, but its name has been changed since joining with the Spring Portfolio projects. Spring Security can be used to secure any Java application, but it's mostly used for web-based applications. Web applications, especially those that can be accessed through the Internet, are vulnerable to hacker attacks if they are not secured properly.

If you've already used Spring Security, be advised there have been several changes introduced in the 3.0 release of the Spring Security framework, which was released almost in tandem with the 3.0 release of the Spring framework (i.e., core). These changes include feature enhancements like support for annotations and OpenID, as well as a restructuring that includes class name changes and package partitioning (i.e., multiple JARs). This is especially important if you are running Spring Security 2.x code, on which the first edition of the book was based.

If, on the other hand, you've never handled security in an application, there are several terms and concepts that you must understand first. Authentication is the process of verifying a principal's identity against what it claims to be. A principal can be a user, a device, or a system, but most typically, it's a user. A principal has to provide evidence of identity to be authenticated. This evidence is called a credential, which is usually a password when the target principal is a user.

Authorization is the process of granting authority to an authenticated user so that this user is allowed to access particular resources of the target application. The authorization process must be performed after the authentication process. Typically, authorities are granted in terms of roles.

Access control means controlling access to an application's resources. It entails making a decision on whether a user is allowed to access a resource. This decision is called an access control decision, and it's made by comparing the resource's access attributes with the user's granted authorities or other characteristics.

After finishing this chapter, you will understand basic security concepts and know how to secure your web applications at the URL access level, the method invocation level, the view-rendering level, and the domain object level.

7-1. Securing URL Access

Problem

Many web applications have some particular URLs that are critically important and private. You must secure these URLs by preventing unauthorized access to them.

Solution

Spring Security enables you to secure a web application's URL access in a declarative way through simple configuration. It handles security by applying servlet filters to HTTP requests. You can configure these filters in Spring's bean configuration files using XML elements defined in the Spring Security schema. However, as servlet filters must be registered in the web deployment descriptor to take effect, you have to register a `DelegatingFilterProxy` instance in the web deployment descriptor, which is a servlet filter that delegates request filtering to a filter in Spring's application context.

Spring Security allows you to configure web application security through the `<http>` element. If your web application's security requirements are straightforward and typical, you can set this element's `auto-config` attribute to true so that Spring Security will automatically register and configure several basic security services, including the following:

- Form-based login service: This provides a default page that contains a login form for users to log into this application.
- Logout service: This provides a handler mapped with a URL for users to log out of this application.
- HTTP Basic authentication: This can process the Basic authentication credentials presented in HTTP request headers. It can also be used for authenticating requests made with remoting protocols and web services.
- Anonymous login: This assigns a principal and grants authorities to an anonymous user so that you can handle an anonymous user like a normal user.
- Remember-me support: This can remember a user's identity across multiple browser sessions, usually by storing a cookie in the user's browser.
- Servlet API integration: This allows you to access security information in your web application via standard Servlet APIs, such as `HttpServletRequest.isUserInRole()` and `HttpServletRequest.getUserPrincipal()`.

With these security services registered, you can specify the URL patterns that require particular authorities to access. Spring Security will perform security checks according to your configurations. A user must log into an application before accessing the secure URLs, unless these URLs are opened for anonymous access. Spring Security provides a set of authentication providers for you to choose from. An authentication provider authenticates a user and returns the authorities granted to this user.

How It Works

Suppose you are going to develop an online message board application for users to post their messages on. First, you create the domain class `Message` with three properties: `author`, `title`, and `body`:

```
package com.apress.springrecipes.board.domain;

public class Message {

    private Long id;
    private String author;
    private String title;
    private String body;

    // Getters and Setters
    ...
}
```

Next, you define the operations of your message board in a service interface, including listing all messages, posting a message, deleting a message, and finding a message by its ID:

```
package com.apress.springrecipes.board.service;
...
public interface MessageBoardService {

    public List<Message> listMessages();
    public void postMessage(Message message);
    public void deleteMessage(Message message);
    public Message findMessageById(Long messageId);
}
```

For testing purposes, let's implement this interface by using a list to store the posted messages. You can use the message posting time (in milliseconds) as a message's identifier. You also have to declare the `postMessage()` and `deleteMessage()` method as synchronized to make them thread-safe.

```
package com.apress.springrecipes.board.service;
...
public class MessageBoardServiceImpl implements MessageBoardService {

    private Map<Long, Message> messages = new LinkedHashMap<Long, Message>();

    public List<Message> listMessages() {
        return new ArrayList<Message>(messages.values());
    }

    public synchronized void postMessage(Message message) {
        message.setId(System.currentTimeMillis());
        messages.put(message.getId(), message);
    }

    public synchronized void deleteMessage(Message message) {
        messages.remove(message.getId());
    }

    public Message findMessageById(Long messageId) {
        return messages.get(messageId);
    }
}
```

Setting Up a Spring MVC Application That Uses Spring Security

To develop this application using Spring MVC as the web framework and Spring Security as the security framework, you first create the following directory structure for your web application.

Note Before using Spring Security, you have to have the relevant Spring Security JARs on your classpath. If you are using Maven, add the following dependencies to your Maven project. We include here a few extra dependencies that you will need on a case-by-case basis, including LDAP support and ACL support. In this book, we have used a variable - \${spring.security.version} to extract the version out. In this book, we are building against **3.2.4.RELEASE**.

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

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-ldap</artifactId>
<version>${spring.security.version}</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-config</artifactId>
<version>${spring.security.version}</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-web</artifactId>
<version>${spring.security.version}</version>
</dependency>

<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-taglibs</artifactId>
<version>${spring.security.version}</version>
</dependency>

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

The Spring configurations for this application are separated into three different files: `board-security.xml`, `board-service.xml`, and `board-servlet.xml`. Each of them configures a particular layer.

Creating the Configuration Files

In the web deployment descriptor (i.e., `web.xml`), you register `ContextLoaderListener` to load the root application context at startup and Spring MVC's `DispatcherServlet` to dispatch requests:

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/board-service.xml</param-value>
    </context-param>

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

    <servlet>
        <servlet-name>board</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
    </servlet>

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

If the root application context's configuration file doesn't have the default name (i.e., `applicationContext.xml`), or if you configure it with multiple configuration files, you'll have to specify the file locations in the `contextConfigLocation` context parameter. Also note that you have mapped the URL pattern `/` to `DispatcherServlet`, meaning everything under the application's root directory will be handled by this servlet.

In the web layer configuration file (i.e., `board-servlet.xml`), you define a view resolver to resolve view names into JSP files located in the `/WEB-INF/jsp/` directory. Later, you will have to configure your controllers in this file.

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

```
<context:component-scan base-package="com.apress.springrecipes.board.web" />
<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
</beans>
```

In the service layer configuration file (i.e., `board-service.xml`), you have to declare only the message board service:

```
<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="messageBoardService"
        class="com.apress.springrecipes.board.service.MessageBoardServiceImpl" />
</beans>
```

Creating the Controllers and Page Views

Suppose you have to implement a function for listing all messages posted on the message board. The first step is to create the following controller.

```
package com.apress.springrecipes.board.web;
...
@Controller
@RequestMapping("/messageList")
public class MessageListController {
    private MessageBoardService messageBoardService;
    @Autowired
    public MessageListController(MessageBoardService messageBoardService) {
        this.messageBoardService = messageBoardService;
    }
    @RequestMapping(method = RequestMethod.GET)
    public String generateList(Model model) {
        List<Message> messages = java.util.Collections.emptyList();
        messages = messageBoardService.listMessages();
        model.addAttribute("messages", messages);
        return "messageList";
    }
}
```

The controller is mapped to a URL in the form `/messageList`. The controller's main method—`generateList()`—obtains a list of messages from the `messageBoardService`, saves it to the model object under the `messages` named, and returns control to a logical view named `messageList`. In accordance with Spring MVC conventions, this last

logical view is mapped to the JSP /WEB-INF/jsp/messageList.jsp showing all the messages passed from the controller:

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

<html>
<head>
<title>Message List</title>
</head>

<body>
<c:forEach items="${messages}" var="message">
<table>
  <tr>
    <td>Author</td>
    <td>${message.author}</td>
  </tr>
  <tr>
    <td>Title</td>
    <td>${message.title}</td>
  </tr>
  <tr>
    <td>Body</td>
    <td>${message.body}</td>
  </tr>
  <tr>
    <td colspan="2">
      <a href="messageDelete?messageId=${message.id}">Delete</a>
    </td>
  </tr>
</table>
<hr />
</c:forEach>
<a href="messagePost.htm">Post</a>
</body>
</html>
```

Another function you have to implement is for users to post messages on the message board. You create the following form controller for this purpose:

```
package com.apress.springrecipes.board.web;
...

@Controller
@RequestMapping("/messagePost")
public class MessagePostController {

    private MessageBoardService messageBoardService;

    @Autowired
    public void MessagePostController(MessageBoardService messageBoardService) {
        this.messageBoardService = messageBoardService;
    }
}
```

```

@RequestMapping(method=RequestMethod.GET)
public String setupForm(Model model) {
    Message message = new Message();
    model.addAttribute("message", message);
    return "messagePost";
}

@RequestMapping(method=RequestMethod.POST)
public String onSubmit(@ModelAttribute("message") Message message, BindingResult result) {
    if (result.hasErrors()) {
        return "messagePost";
    } else {
        messageBoardService.postMessage(message);
        return "redirect:messageList";
    }
}

```

A user must have logged into the message board before posting a message. You can get a user's login name with the `getRemoteUser()` method defined in `HttpServletRequest`. This login name will be used as the message's author name.

You then create the form view `/WEB-INF/jsp/messagePost.jsp` with Spring's form tags for users to input message contents:

```

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

<html>
<head>
<title>Message Post</title>
</head>

<body>
<form:form method="POST" modelAttribute="message">
<table>
<tr>
    <td>Title</td>
    <td><form:input path="title" /></td>
</tr>
<tr>
    <td>Body</td>
    <td><form:textarea path="body" /></td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Post" /></td>
</tr>
</table>
</form:form>
</body>
</html>

```

The last function is to allow a user to delete a posted message by clicking the Delete link on the message list page. You create the following controller for this function:

```
package com.apress.springrecipes.board.web;
...
@Controller
@RequestMapping("/messageDelete")
public class MessageDeleteController {
    private MessageBoardService messageBoardService;

    @Autowired
    public void MessageDeleteController(MessageBoardService messageBoardService) {
        this.messageBoardService = messageBoardService;
    }

    @RequestMapping(method= RequestMethod.GET)
    public String messageDelete(@RequestParam(required = true, name = "messageId") Long messageId, Model model) {
        Message message = messageBoardService.findMessageById(messageId);
        messageBoardService.deleteMessage(message);
        model.addAttribute("messages", messageBoardService.listMessages());
        return "redirect:messageList";
    }
}
```

Now, you can deploy this application to a web container (e.g., Apache Tomcat 7.0). By default, Tomcat listens on port 8080, so if you deploy your application to the board context path, you can list all posted messages with the following URL:

<http://localhost:8080/board/messageList.htm>

Up to this time, you haven't configured any security service for this application, so you can access it directly without logging into it.

Securing URL Access

Now, let's secure this web application's URL access with Spring Security. First, you have to configure a `DelegatingFilterProxy` instance in `web.xml` to delegate HTTP request filtering to a filter defined in Spring Security:

```
<web-app ...>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/board-service.xml
            /WEB-INF/board-security.xml
        </param-value>
    </context-param>
    ...

```

```

<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>springSecurityFilterChain</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>

```

The responsibility of `DelegatingFilterProxy` is simply to delegate HTTP request filtering to a Spring bean that implements the `javax.servlet.Filter` interface. By default, it delegates to a bean whose name is the same as its `<filter-name>` property, but you can override the bean name in its `targetBeanName` init parameter. As Spring Security will automatically configure a filter chain with the name `springSecurityFilterChain` when you enable web application security, you can simply use this name for your `DelegatingFilterProxy` instance.

Although you can configure Spring Security in the same configuration file as the web and service layers, it's better to separate the security configurations in an isolated file (e.g., `board-security.xml`). Inside `web.xml`, you have to add the file's location to the `contextConfigLocation` context parameter for `ContextLoaderListener` to load it at startup. Then, you create it with the following content:

```

<beans:beans xmlns="http://www.springframework.org/schema/security"
    xmlns:beans="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
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security.xsd">

    <http auto-config="true">
        <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_ANONYMOUS" />
        <intercept-url pattern="/messagePost*" access="ROLE_USER" />
        <intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
    </http>

    <authentication-manager>
        <authentication-provider>
            <user-service>
                <user name="admin" password="secret" authorities="ROLE_ADMIN,ROLE_USER" />
                <user name="user1" password="1111" authorities="ROLE_USER" />
            </user-service>
        </authentication-provider>
    </authentication-manager>
</beans:beans>

```

You may find that this file looks a bit different from a normal bean configuration file. Normally, the default namespace of a bean configuration file is beans, so you can use the `<bean>` and `<property>` elements without the beans prefix. However, if you use this style to declare the Spring Security services, all security elements must be appended with the security prefix. Because the elements in a security configuration file are mostly Spring Security's, you can define security as the default namespace instead, so you can use them without the security prefix. If you do it this way, however, when you declare normal Spring beans in this file, you have to include the beans prefix for the `<bean>` and `<property>` elements.

Inside the `<http>` configuration element, you can restrict access to particular URLs with one or more `<intercept-url>` elements. Each `<intercept-url>` element specifies a URL pattern and a set of access attributes required to access the URLs. Remember that you must always include a wildcard at the end of a URL pattern. Failing to do so will make the URL pattern unable to match a URL that has request parameters. As a result, hackers could easily skip the security check by appending an arbitrary request parameter.

Access attributes are compared with a user's authorities to decide if this user can access the URLs. In most cases, access attributes are defined in terms of roles. For example, users with the `ROLE_USER` role, or anonymous users, who have the `ROLE_ANONYMOUS` role by default, are able to access the URL `/messageList` to list all messages. However, a user must have the `ROLE_USER` role to post a new message via the URL `/messagePost`. Only an administrator who has the `ROLE_ADMIN` role can delete messages via `/messageDelete`.

You can configure authentication services in the `<authentication-provider>` element, which is nested inside the `<authentication-manager>` element. Spring Security supports several ways of authenticating users, including authenticating against a database or an LDAP repository. It also supports defining user details in `<user-service>` directly for simple security requirements. You can specify a username, a password, and a set of authorities for each user.

Now, you can redeploy this application to test its security configurations. You can enter the request path `/messageList` to list all posted messages as usual, because it's open to anonymous users. But if you click the link to post a new message, you will be redirected to the default login page generated by Spring Security. You must log into this application with a correct username and password to post a message. Finally, to delete a message, you must log in as an administrator.

Using Java to configure Spring Security

Until now we have used XML to configure our application. However as of Spring Security 3.2 we can also use a full Java based configuration approach and the Servlet 3.0 specification also allows us to remove the `web.xml`, which allows us to do everything in Java. The `web.xml` we created in the previous samples can be replaced with a `ServletContainerInitializer` which gives us the opportunity to configure the `ServletContext` ourselves.

Spring provides us with the `SpringServletContainerInitializer` which allows us to add one or more `WebApplicationInitializer` which we can use to add servlets, filters, and listeners to the `ServletContext`.

```
package com.apress.springrecipes.board.web;

import com.apress.springrecipes.board.config.MessageBoardConfiguration;
import com.apress.springrecipes.board.web.config.MessageBoardSecurityConfiguration;
import com.apress.springrecipes.board.web.config.MessageBoardWebConfiguration;
import org.springframework.core.annotation.Order;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

@Order(1)
public class MessageBoardApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {

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

```

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

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

```

And the security WebApplicationInitializer.

```

package com.apress.springrecipes.board.web;

import org.springframework.core.annotation.Order;
import org.springframework.security.web.context.AbstractSecurityWebApplicationInitializer;

@Order(2)
public class MessageBoardSecurityInitializer extends AbstractSecurityWebApplicationInitializer { }

```

The MessageBoardApplicationInitializer takes care of creating a ContextLoaderListener and DispatcherServlet the code that does this is part of the AbstractAnnotationConfigDispatcherServletInitializer class. We only need to provide the configuration classes for the listener and servlet for this we need to implement the getRootConfigClasses and getServletConfigClasses method. Finally we need to supply the servlet mapping for this we let the getServletMappings method return /.

We also need to register a DelegatingFilterProxy to make Spring Security work, we can extend the AbstractSecurityWebApplicationInitializer base class for this. This class is provided by Spring Security, this class is also capable of constructing a ContextLoaderListener which can be convenient in an environment where you don't use Spring MVC but another web technology like JSF. That way you can work with a single WebApplicationInitializer.

Notice the @Order annotation on both classes it is important that the DelegatingFilterProxy is the first Filter in the chain of filters. However the Spring provided AbstractAnnotationConfigDispatcherServletInitializer registers any filters before all already configured filters. To make sure that Spring Security is setup correctly it should be the last WebApplicationInitializer to execute so that it can register the DelegatingFilterProxy as the first filter without the risk of being overwritten later on.

```

package com.apress.springrecipes.board.config;

import com.apress.springrecipes.board.service.MessageBoardService;
import com.apress.springrecipes.board.service.MessageBoardServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MessageBoardConfiguration {

    @Bean
    public MessageBoardService messageBoardService() {
        return new MessageBoardServiceImpl();
    }
}

```

The configuration class for the service isn't spectacular; it only contains the bean definition for our service class.

```
package com.apress.springrecipes.board.web.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan("com.apress.springrecipes.board.web")
@EnableWebMvc
public class MessageBoardWebConfiguration {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

To configure our DispatcherServlet we have the `MessageBoardWebConfiguration` class. In the `MessageBoardWebConfiguration` class you define a view resolver to resolve view names into JSP files located in the `/WEB-INF/jsp/` directory. We have a `@ComponentScan` (analogous to `<context:component-scan />`) to pick up our `@Controller` annotated beans. Finally we specify that we want to use Spring MVC by placing the `@EnableWebMvc` annotation.

To configure Spring Security we need to have a configuration class which is annotated with `@EnableWebSecurity` and for further configuration needs to be a `WebSecurityConfigurer`. The latter is needed to correctly setup the security configuration. Although you could implement your own class Spring Security provides `WebSecurityConfigurerAdapter` as a base class which takes away a lot of ground work.

```
package com.apress.springrecipes.board.web.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {
```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .antMatchers("/messageList*").hasAnyRole("USER", "ANONYMOUS")
            .antMatchers("/messagePost*").hasRole("USER")
            .antMatchers("/messageDelete*").hasRole("ADMIM")
        .and()
            .formLogin();
}

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("admin").password("secret").authorities("ROLE_ADMIN","ROLE_USER")
        .and().withUser("user1").password("1111").authorities("ROLE_USER");
}
}

```

We are overriding two `configure` methods from the superclass. The first is the one that takes an `HttpSecurity` as argument. This is analogous to the `<http>` element in xml. We use it to configure the protection for our URLs with the `antMatchers` on conjunction with the `hasAnyRole` or `hasRole` method. Notice the omission of the `ROLE_` prefix the configured prefix is automatically added to the roles.

The `configure` method taking the `AuthenticationManagerBuilder` is used to setup the `AuthenticationManager`, just like the `<authentication-manager>` in XML. For now we are going to use in memory authentication this is done by using the `inMemoryAuthentication` method and configuring the users with the `withUser` method.

7-2. Logging In to Web Applications

Problem

A secure application requires its users to log in before they can access certain secure functions. This is especially important for web applications running on the open Internet, because hackers can easily reach them. Most web applications have to provide a way for users to input their credentials to log in.

Solution

Spring Security supports multiple ways for users to log into a web application. It supports form-based login by providing a default web page that contains a login form. You can also provide a custom web page as the login page. In addition, Spring Security supports HTTP Basic authentication by processing the Basic authentication credentials presented in HTTP request headers. HTTP Basic authentication can also be used for authenticating requests made with remoting protocols and web services.

Some parts of your application may allow for anonymous access (e.g., access to the welcome page). Spring Security provides an anonymous login service that can assign a principal and grant authorities to an anonymous user so that you can handle an anonymous user like a normal user when defining security policies.

Spring Security also supports remember-me login, which is able to remember a user's identity across multiple browser sessions so that a user needn't log in again after logging in for the first time.

How It Works

To help you better understand the various login mechanisms in isolation, let's first disable HTTP auto-configuration by removing the `auto-config` attribute:

```
<http>
    <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_ANONYMOUS" />
    <intercept-url pattern="/messagePost*" access="ROLE_USER" />
    <intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
</http>
```

Note that the login services introduced next will be registered automatically if you enable HTTP auto-`config`. However, if you disable HTTP auto-`config` or you want to customize these services, you have to configure the corresponding XML elements explicitly.

HTTP Basic Authentication

The HTTP Basic authentication support can be configured via the `<http-basic>` element. When HTTP Basic authentication is required, a browser will typically display a login dialog or a specific login page for users to log in.

```
<http>
    ...
    <http-basic />
</http>
```

Or using the `httpBasic` method when using Java config.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .httpBasic();
    }
}
```

Note that when HTTP Basic authentication and form-based login are enabled at the same time, the latter will be used. So, if you want your web application users to log in with this authentication type, you should not enable form-based login.

Form-Based Login

The form-based login service will render a web page that contains a login form for users to input their login details and process the login form submission. It's configured via the `<form-login>` element:

```
<http>
    ...
    <form-login />
</http>
```

Or when using Java config the `formLogin` method.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        ...
        .formLogin();
    }
}
```

By default, Spring Security automatically creates a login page and maps it to the URL `/spring_security_login`. So, you can add a link to your application (e.g., in `messageList.jsp`) referring to this URL for login:

```
<a href=">Login</a>
```

If you don't prefer the default login page, you can provide a custom login page of your own. For example, you can create the following `login.jsp` file in the root directory of the web application. Note that you shouldn't put this file inside WEB-INF, which would prevent users from accessing it directly.

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

<html>
<head>
<title>Login</title>
</head>

<body>
<form method="POST" action="

```

Note that the form action URL and the input field names are Spring Security-specific. However, the action URL can be customized with the `login-url` attribute of `<form-login>`.

Now, you have to change the previous login link (i.e., `messageList.jsp`) to refer to this URL for login:

```
<a href=">Login</a>
```

In order for Spring Security to display your custom login page when a login is requested, you have to specify its URL in the `login-page` attribute:

```
<http>
  ...
    <form-login login-page="/login.jsp" />
</http>
```

Or the equivalent in Java based configuration.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        ...
        .formLogin().loginPage("/login.jsp");
    }
}
```

If the login page is displayed by Spring Security when a user requests a secure URL, the user will be redirected to the target URL once the login succeeds. However, if the user requests the login page directly via its URL, by default the user will be redirected to the context path's root (i.e., `http://localhost:8080/board/`) after a successful login. If you have not defined a welcome page in your web deployment descriptor, you may wish to redirect the user to a default target URL when the login succeeds:

```
<http>
  ...
    <form-login login-page="/login.jsp" default-target-url="/messageList" />
</http>
```

The equivalent in Java based configuration.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        ...
        .formLogin().loginPage("/login.jsp").defaultSuccessUrl("/messageList");
    }
}
```

If you use the default login page created by Spring Security, then when a login fails, Spring Security will render the login page again with the error message. However, if you specify a custom login page, you will have to configure the authentication-failure-url attribute to specify which URL to redirect to on login error. For example, you can redirect to the custom login page again with the error request parameter:

```
<http>
  ...
  <form-login login-page="/login.jsp" default-target-url="/messageList"
    authentication-failure-url="/login.jsp?error=true" />
</http>
```

The equivalent in Java based configuration.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        ...
        .formLogin()
            .loginPage("/login.jsp")
            .defaultSuccessUrl("/messageList")
            .failureUrl("login.jsp?error=true");
    }
}
```

Then your login page should test whether the error request parameter is present. If an error has occurred, you will have to display the error message by accessing the session scope attribute SPRING_SECURITY_LAST_EXCEPTION, which stores the last exception for the current user.

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

<html>
<head>
<title>Login</title>
</head>

<body>
<c:if test="${not empty param.error}">
    <font color="red">
        Login error. <br />
        Reason : ${sessionScope["SPRING_SECURITY_LAST_EXCEPTION"].message}
    </font>
</c:if>
...
</body>
</html>
```

The Logout Service

The logout service provides a handler to handle logout requests. It can be configured via the `<logout>` element:

```
<http>
  ...
  <logout />
</http>
```

For Java based configuration you can use the `logout` method.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .logout();
    }
}
```

By default, it's mapped to the URL `/j_spring_security_logout`, so you can add a link to a page referring to this URL for logout. Note that this URL can be customized with the `logout-url` attribute of `<logout>`.

```
<a href="Logout</a>
```

By default, a user will be redirected to the context path's root when the logout succeeds, but sometimes, you may wish to direct the user to another URL, which you can do as follows:

```
<http>
  ...
  <logout logout-success-url="/login.jsp" />
</http>
```

In Java config you would use the `logoutSuccessUrl` method to configure the URL to direct to.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .logout().logoutSuccessUrl("/login.jsp");
    }
}
```

Anonymous Login

The anonymous login service can be configured via the `<anonymous>` element or `anonymous()` in Java config, where you can customize the username and authorities of an anonymous user, whose default values are `anonymousUser` and `ROLE_ANONYMOUS`:

```
<http>
    <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_GUEST" />
    <intercept-url pattern="/messagePost*" access="ROLE_USER" />
    <intercept-url pattern="/messageDelete*" access="ROLE_ADMIN" />
    ...
    <anonymous username="guest" granted-authority="ROLE_GUEST" />
</http>
```

When using Java config:

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .anonymous().principal("guest").authorities("ROLE_GUEST");
    }
}
```

Remember-Me Support

Remember-me support can be configured via the `<remember-me>` element or `rememberMe()` method in Java config. By default, it encodes the username, password, remember-me expiration time, and a private key as a token, and stores it as a cookie in the user's browser. The next time the user accesses the same web application, this token will be detected so that the user can log in automatically.

```
<http>
    ...
    <remember-me />
</http>
```

When using Java config:

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            ...
            .and()
            .rememberMe();
    }
}
```

However, static remember-me tokens can cause security issues, because they may be captured by hackers. Spring Security supports rolling tokens for more advanced security needs, but this requires a database to persist the tokens. For details about rolling remember-me token deployment, please refer to the Spring Security reference documentation.

7-3. Authenticating Users

Problem

When a user attempts to log into your application to access its secure resources, you have to authenticate the user's principal and grant authorities to this user.

Solution

In Spring Security, authentication is performed by one or more *authentication providers*, connected as a chain. If any of these providers authenticates a user successfully, that user will be able to log into the application. If any provider reports that the user is disabled or locked or that the credential is incorrect, or if no provider can authenticate the user, then the user will be unable to log into this application.

Spring Security supports multiple ways of authenticating users and includes built-in provider implementations for them. You can easily configure these providers with the built-in XML elements. Most common authentication providers authenticate users against a user repository storing *user details* (e.g., in an application's memory, a relational database, or an LDAP repository).

When storing user details in a repository, you should avoid storing user passwords in clear text, because that makes them vulnerable to hackers. Instead, you should always store encrypted passwords in your repository. A typical way of encrypting passwords is to use a one-way hash function to encode the passwords. When a user enters a password to log in, you apply the same hash function to this password and compare the result with the one stored in the repository. Spring Security supports several algorithms for encoding passwords (including MD5 and SHA) and provides built-in password encoders for these algorithms.

If you retrieve a user's details from a user repository every time a user attempts to log in, your application may incur a performance impact. This is because a user repository is usually stored remotely, and it has to perform some kinds of queries in response to a request. For this reason, Spring Security supports caching user details in local memory and storage to save you the overhead of performing remote queries.

How It Works

Authenticating Users with In-Memory Definitions

If you have only a few users in your application and you seldom modify their details, you can consider defining the user details in Spring Security's configuration file so that they will be loaded into your application's memory:

```
<authentication-manager>
    <authentication-provider>
        <user-service>
            <user name="admin" password="secret" authorities="ROLE_ADMIN,ROLE_USER" />
            <user name="user1" password="1111" authorities="ROLE_USER" />
            <user name="user2" password="2222" disabled="true" authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

You can define user details in `<user-service>` with multiple `<user>` elements. For each user, you can specify a username, a password, a disabled status, and a set of granted authorities. A disabled user cannot log into an application.

Spring Security also allows you to externalize user details in a properties file, such as `/WEB-INF/users.properties`:

```
<authentication-manager>
    <authentication-provider>
        <user-service properties="/WEB-INF/users.properties" />
    </authentication-provider>
</authentication-manager>
```

Then, you can create the specified properties file and define the user details in the form of properties:

```
admin=secret,ROLE_ADMIN,ROLE_USER
user1=1111,ROLE_USER
user2=2222,disabled,ROLE_USER
```

Each property in this file represents a user's details. The property key is the username, and the property value is divided into several parts separated by commas. The first part is the password, and the second part is the enabled status, which is optional; and the default status is enabled. The following parts are the authorities granted to the user.

Authenticating Users with In-Memory Definitions using Java Config

If you have only a few users in your application and you seldom modify their details, you can consider defining the user details in Spring Security's configuration file so that they will be loaded into your application's memory:

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("admin").password("secret").authorities("ROLE_ADMIN","ROLE_USER").and()
            .withUser("user1").password("1111").authorities("ROLE_USER").and()
            .withUser("user2").password("2222").disabled(true).authorities("ROLE_USER");
    }
}
```

You can define user details with the `inMemoryAuthentication` using the `withUser` method you can define the users. For each user, you can specify a username, a password, a disabled status, and a set of granted authorities. A disabled user cannot log into an application.

Authenticating Users Against a Database

More typically, user details should be stored in a database for easy maintenance. Spring Security has built-in support for querying user details from a database. By default, it queries user details, including authorities, with the following SQL statements:

```
SELECT username, password, enabled
FROM   users
WHERE  username = ?
```

```
SELECT username, authority
FROM   authorities
WHERE  username = ?
```

In order for Spring Security to query user details with these SQL statements, you have to create the corresponding tables in your database. For example, you can create them in the board schema of Apache Derby with the following SQL statements:

```
CREATE TABLE USERS (
    USERNAME  VARCHAR(10)      NOT NULL,
    PASSWORD   VARCHAR(32)      NOT NULL,
    ENABLED    SMALLINT,
    PRIMARY KEY (USERNAME)
);

CREATE TABLE AUTHORITIES (
    USERNAME  VARCHAR(10)      NOT NULL,
    AUTHORITY  VARCHAR(10)      NOT NULL,
    FOREIGN KEY (USERNAME) REFERENCES USERS
);
```

Next, you can input some user details into these tables for testing purposes. The data for these two tables is shown in Tables 7-1 and 7-2.

Table 7-1. Testing User Data for the USERS Table

USERNAME	PASSWORD	ENABLED
Admin	Secret	1
user1	1111	1
user2	2222	0

Table 7-2. Testing User Data for the AUTHORITIES Table

USERNAME	AUTHORITY
Admin	ROLE_ADMIN
Admin	ROLE_USER
user1	ROLE_USER
user2	ROLE_USER

In order for Spring Security to access these tables, you have to declare a data source (e.g., in `board-service.xml`) for creating connections to this database.

```
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
    value="org.apache.derby.jdbc.ClientDriver" />
  <property name="url"
    value="jdbc:derby://localhost:1527/board;create=true" />
  <property name="username" value="app" />
  <property name="password" value="app" />
</bean>
```

Note To connect to a database in the Apache Derby server, you need the Derby client .jars, as well as the Spring JDBC support. If you are using Apache Maven, add the following dependencies to your project:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.10.2.0</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring.version}</version>
</dependency>
```

The final step is to configure an authentication provider that queries this database for user details. You can achieve this simply by using the `<jdbc-user-service>` element with a data source reference:

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource" />
  </authentication-provider>
</authentication-manager>
```

For Java Config use the `jdbcAuthentication` and pass it a `DataSource`.

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private DataSource dataSource;

    ...

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.jdbcAuthentication().dataSource(dataSource);
    }
}
```

However, in some cases, you may already have your own user repository defined in a legacy database. For example, suppose that the tables are created with the following SQL statements, and that all users in the `MEMBER` table have the enabled status:

```
CREATE TABLE MEMBER (
    ID          BIGINT      NOT NULL,
    USERNAME    VARCHAR(10) NOT NULL,
    PASSWORD    VARCHAR(32) NOT NULL,
    PRIMARY KEY (ID)
);

CREATE TABLE MEMBER_ROLE (
    MEMBER_ID   BIGINT      NOT NULL,
    ROLE        VARCHAR(10) NOT NULL,
    FOREIGN KEY (MEMBER_ID) REFERENCES MEMBER
);
```

Suppose you have the legacy user data stored in these tables as shown in Tables 7-3 and 7-4.

Table 7-3. Legacy User Data in the `MEMBER` Table

ID	USERNAME	PASSWORD
1	Admin	Secret
2	user1	1111

Table 7-4. Legacy User Data in the `MEMBER_ROLE` Table

MEMBER_ID	ROLE
1	ROLE_ADMIN
1	ROLE_USER
2	ROLE_USER

Fortunately, Spring Security also supports using custom SQL statements to query a legacy database for user details. You can specify the statements for querying a user's information and authorities in the `users-by-username-query` and `authorities-by-username-query` attributes:

```
<jdbc-user-service data-source-ref="dataSource"
    users-by-username-query=
        "SELECT username, password, 'true' as enabled
         FROM member
        WHERE username = ?"
    authorities-by-username-query=
        "SELECT member.username, member_role.role as authorities
         FROM member, member_role
        WHERE member.username = ? AND member.id = member_role.member_id" />
```

The equivalent in Java config using `usersByUsernameQuery` and `authoritiesByUsernameQuery` methods:

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.jdbcAuthentication()
            .dataSource(dataSource)
            .usersByUsernameQuery(
                "SELECT username, password, 'true' as enabled FROM member WHERE username = ?")
            .authoritiesByUsernameQuery(
                "SELECT member.username, member_role.role as authorities " +
                "FROM member, member_role " +
                "WHERE member.username = ? AND member.id = member_role.member_id");
    }
}
```

Encrypting Passwords

Until now, you have been storing user details with clear-text passwords. But this approach is vulnerable to hacker attacks, so you should encrypt the passwords before storing them. Spring Security supports several algorithms for encrypting passwords. For example, you can choose MD5 (Message-Digest algorithm 5), a one-way hash algorithm, to encrypt your passwords.

Note You may need a utility to calculate MD5 digests for your passwords. One such utility is Jacksum, which you can download from <http://sourceforge.net/projects/jacksum/> and extract to a directory of your choice. Then execute the following command to calculate a digest for a text:

```
java -jar jacksum.jar -a md5 -q "txt:secret"
```

Now, you can store the encrypted passwords in your user repository. For example, if you are using in-memory user definitions, you can specify the encrypted passwords in the password attributes. Then, you can configure a <password-encoder> element with a hashing algorithm specified in the hash attribute.

```
<authentication-manager>
    <authentication-provider>
        <password-encoder hash="md5" />
        <user-service>
            <user name="admin" password="5ebe2294ecd0e0f08eab7690d2a6ee69"
                  authorities="ROLE_ADMIN, ROLE_USER" />
            <user name="user1" password="b59c67bf196a4758191e42f76670ceba"
                  authorities="ROLE_USER" />
            <user name="user2" password="934b535800b1cba8f96a5d72f72f1611"
                  disabled="true" authorities="ROLE_USER" />
        </user-service>
    </authentication-provider>
</authentication-manager>
```

A password encoder is also applicable to a user repository stored in a database:

```
<authentication-manager>
    <authentication-provider>
        <password-encoder hash="md5" />
        <jdbc-user-service data-source-ref="dataSource" />
    </authentication-provider>
</authentication-manager>
```

In Java config you can specify the pass encoder using the `passwordEncoder` method on `AuthenticationManagerBuilder`:

```
@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.jdbcAuthentication()
            .dataSource(dataSource)
            .passwordEncoder(new Md5PasswordEncoder());
        ...
    }
}
```

Of course, you have to store the encrypted passwords in the database tables, instead of the clear-text passwords, as shown in Table 7-5.

Table 7-5. Testing User Data with Encrypted Passwords for the USERS Table

USERNAME	PASSWORD	ENABLED
Admin	5ebe2294ecd0e0f08eab7690d2a6ee69	1
user1	b59c67bf196a4758191e42f76670ceba	1
user2	934b535800b1cba8f96a5d72f72f1611	0

Authenticating Users Against an LDAP Repository

Spring Security also supports accessing an LDAP repository for authenticating users. First, you have to prepare some user data for populating the LDAP repository. Let's prepare the user data in the LDAP Data Interchange Format (LDIF), a standard plain-text data format for importing and exporting LDAP directory data. For example, create the `users.ldif` file containing the following contents:

```
dn: dc=springrecipes,dc=com
objectClass: top
objectClass: domain
dc: springrecipes

dn: ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=admin,ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: uidObject
objectclass: person
uid: admin
cn: admin
sn: admin
userPassword: secret

dn: uid=user1,ou=people,dc=springrecipes,dc=com
objectclass: top
objectclass: uidObject
objectclass: person
uid: user1
cn: user1
sn: user1
userPassword: 1111
```

```

dn: cn=admin,ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: groupOfNames
cn: admin
member: uid=admin,ou=people,dc=springrecipes,dc=com

dn: cn=user,ou=groups,dc=springrecipes,dc=com
objectclass: top
objectclass: groupOfNames
cn: user
member: uid=admin,ou=people,dc=springrecipes,dc=com
member: uid=user1,ou=people,dc=springrecipes,dc=com

```

Don't worry if you don't understand this LDIF file very well. You probably won't need to use this file format to define LDAP data often, because most LDAP servers support GUI-based configuration. This `users.ldif` file includes the following contents:

- The default LDAP domain, `dc=springrecipes,dc=com`
- The groups and people organization units for storing groups and users
- The `admin` and `user1` users with the passwords `secret` and `1111`
- The `admin` group (including the `admin` user) and the `user` group (including the `admin` and `user1` users)

For testing purposes, you can install an LDAP server on your local machine to host this user repository. For the sake of easy installation and configuration, we recommend installing OpenDS (<http://www.opends.org/>), a Java-based open source directory service engine that supports LDAP.

Note OpenDS supports two types of installation interfaces: command line and GUI. This example uses the command-line interface, so you have to download the ZIP distribution and extract it to an arbitrary directory (e.g., `C:\OpenDS-2.2.1`), and then execute the setup script from the root of this directory.

```
C:\OpenDS-2.2.0>setup --cli
```

```
OpenDS Directory Server 2.2.0
Please wait while the setup program initializes...
```

```
What would you like to use as the initial root user DN for the Directory
Server? [cn=Directory Manager]:
Please provide the password to use for the initial root user: ldap
Please re-enter the password for confirmation: ldap
```

```
On which port would you like the Directory Server to accept connections from
LDAP clients? [1389]:
```

```
On which port would you like the Administration Connector to accept
connections? [4444]:
```

What do you wish to use as the base DN for the directory data?

[dc=example,dc=com]:dc=springrecipes,dc=com

Options for populating the database:

- 1) Only create the base entry
- 2) Leave the database empty
- 3) Import data from an LDIF file
- 4) Load automatically-generated sample data

Enter choice [1]: 3

Please specify the path to the LDIF file containing the data to import: users.ldif

Do you want to enable SSL? (yes / no) [no]:

Do you want to enable Start TLS? (yes / no) [no]:

Do you want to start the server when the configuration is completed?

(yes/no) [yes]:

Enable OpenDS to run as a Windows Service? (yes / no) [no]:

Do you want to start the server when the configuration is completed?

(yes/no) [yes]:

What would you like to do?

- 1) Setup the server with the parameters above
- 2) Provide the setup parameters again
- 3) Cancel the setup

Enter choice [1]:

Configuring Directory Server Done.

Importing LDIF file users.ldif Done.

Starting Directory Server Done.

Note that the root user and password for this LDAP server are cn=Directory Manager and ldap, respectively.

Later, you will have to use this user to connect to this server.

After the LDAP server has started up, you can configure Spring Security to authenticate users against its repository.

Note To authenticate users against an LDAP repository, you have to have the Spring LDAP project on your CLASSPATH. If you are using Maven, add the following dependency to your Maven project:

```
<dependency>
    <groupId>org.springframework.ldap</groupId>
    <artifactId>spring-ldap-core</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>

<beans:beans ...>
    ...
    <authentication-manager>
        <authentication-provider>
            <password-encoder hash="{sha}" />
            <ldap-user-service server-ref="ldapServer"
                user-search-filter="uid={0}" user-search-base="ou=people"
                group-search-filter="member={0}" group-search-base="ou=groups" />
        </authentication-provider>
    </authentication-manager>

    <ldap-server id="ldapServer"
        url="ldap://localhost:1389/dc=springrecipes,dc=com"
        manager-dn="cn=Directory Manager" manager-password="ldap" />
</beans:beans>
```

You have to configure an `<ldap-user-service>` element to define how to search users from an LDAP repository. You can specify the search filters and search bases for searching users and groups via several attributes, whose values must be consistent with the repository's directory structure. With the preceding attribute values, Spring Security will search a user from the people organization unit with a particular user ID and search a user's groups from the groups organization unit. Spring Security will automatically insert the `ROLE_` prefix to each group as an authority.

As OpenDS uses SSHA (Salted Secure Hash Algorithm) to encode user passwords by default, you have to specify `{sha}` as the hash algorithm in `<password-encoder>`. Note that this value is different from sha, as it's specific to LDAP password encoding.

Finally, `<ldap-user-service>` has to refer to an LDAP server definition, which defines how to create connections to an LDAP server. You can specify the root user's username and password to connect to the LDAP server running on localhost.

Authenticating Users Against an LDAP Repository using Java Config

You have to configure the LDAP repository using the `ldapAuthentication` method. You can specify the search filters and search bases for searching users and groups via several callback methods, whose values must be consistent with the repository's directory structure. With the preceding attribute values, Spring Security will search a user from the people organization unit with a particular user ID and search a user's groups from the groups organization unit. Spring Security will automatically insert the `ROLE_` prefix to each group as an authority.

```

@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {
    ...
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .ldapAuthentication()
            .contextSource()
            .url("ldap://localhost:1389/dc=springrecipes,dc=com")
            .managerDn("cn=Directory Manager").managerPassword("ldap")
            .and()
            .userSearchFilter("uid={0}").userSearchBase("ou=people")
            .groupSearchFilter("member={0}").groupSearchBase("ou=groups")
            .passwordEncoder(new LdapShaPasswordEncoder())
            .passwordCompare().passwordAttribute("userPassword");
    }
}

```

As OpenDS uses SSHA (Salted Secure Hash Algorithm) to encode user passwords by default, you have to specify a `LdapShaPasswordEncoder` as the password encoder. Note that this value is different from `sha`, as it's specific to LDAP password encoding. We also need to specify the `passwordAttribute` because the password encoder needs to know which field in LDAP is the password.

Finally, we have to refer to an LDAP server definition, which defines how to create connections to an LDAP server. You can specify the root user's username and password to connect to the LDAP server running on localhost, configure the server using the `contextSource` method.

Caching User Details

Both `<jdbc-user-service>` and `<ldap-user-service>` support caching user details. First of all, you have to choose a cache implementation that provides a caching service. As Spring and Spring Security have built-in support for Ehcache (<http://ehcache.sourceforge.net/>), you can choose it as your cache implementation and create a configuration file for it (e.g., `ehcache.xml` in the classpath root) with the following contents:

```

<ehcache>
    <diskStore path="java.io.tmpdir"/>

    <defaultCache
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
    />

    <cache name="userCache"
        maxElementsInMemory="100"
        eternal="false"
        timeToIdleSeconds="600"
        timeToLiveSeconds="3600"
        overflowToDisk="true"
    />
</ehcache>

```

Note To use Ehcache to cache objects, you have to have the Ehcache library on your CLASSPATH. If you are using Maven, add the following dependency to your Maven project:

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <version>2.8.3</version>
  <artifactId>ehcache-core</artifactId>
</dependency>
```

This Ehcache configuration file defines two types of cache configurations. One is for the default, and the other is for caching user details. If the user cache configuration is used, a cache instance will cache the details of at most 100 users in memory. The cached users will overflow to disk when this limit is exceeded. A cached user will expire if it has been idle for 10 minutes or live for 1 hour after its creation.

To enable caching user details in Spring Security, you can set the `cache-ref` attribute of either `<jdbc-user-service>` or `<ldap-user-service>` to refer to a `UserCache` object. Spring Security comes with two `UserCache` implementations, `EhCacheBasedUserCache`, which has to refer to an Ehcache instance and a `SpringCacheBasedUserCache`, which uses Spring's caching abstraction.

```
<beans:beans ...>
  ...
  <authentication-manager>
    <authentication-provider>
      ...
      <ldap-user-service server-ref="ldapServer"
        user-search-filter="uid={0}" user-search-base="ou=people"
        group-search-filter="member={0}" group-search-base="ou=groups"
        cache-ref="userCache" />
    </authentication-provider>
  </authentication-manager>

  <beans:bean id="userCache" class="org.springframework.security.core.userdetails.cache.EhCacheBasedUserCache">
    <beans:property name="cache" ref="userEhCache" />
  </beans:bean>

  <beans:bean id="userEhCache"
    class="org.springframework.cache.ehcache.EhCacheFactoryBean">
    <beans:property name="cacheManager" ref="cacheManager" />
    <beans:property name="cacheName" value="userCache" />
  </beans:bean>
</beans:beans>
```

In Spring, an Ehcache instance can be created via `EhCacheFactoryBean` by providing a cache manager and a cache name. Spring also provides `EhCacheManagerFactoryBean` for you to create an Ehcache manager by loading a configuration file. By default, it loads `ehcache.xml` (located in the root of the classpath). As an Ehcache manager may be used by other service components, it should be defined in `board-service.xml`.

```
<bean id="cacheManager"
  class="org.springframework.security.core.userdetails.cache.EhCacheManagerFactoryBean" />
```

In Java based configuration, at the moment of writing, only the `jdbcAuthentication()` allows for easy configuration of a user cache. However configuration is quite similar to the one in XML. For a Spring Caching based cache solution (which still delegates to EhCache) we need to configure a `CacheManager` and make this aware of EhCache.

```
@Configuration
public class MessageBoardConfiguration {
    ...
    @Bean
    public EhCacheCacheManager cacheManager() {
        EhCacheCacheManager cacheManager = new EhCacheCacheManager();
        cacheManager.setCacheManager(ehCacheManager().get0bject());
        return cacheManager;
    }

    @Bean
    public EhCacheManagerFactoryBean ehCacheManager() {
        return new EhCacheManagerFactoryBean();
    }
}
```

This is best added to the configuration of the services as the caching can also be used for others means (see the recipes regarding Spring Caching). Now that we have the `CacheManager` setup we need to configure a `SpringCacheBasedUserCache`.

```
@Configuration
@EnableWebMvcSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private CacheManager cacheManager;

    @Bean
    public SpringCacheBasedUserCache userCache() throws Exception {
        Cache cache = cacheManager.getCache("userCache");
        return new SpringCacheBasedUserCache(cache);
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.jdbcAuthentication()
            .userCache(userCache())
    ...
}
```

Notice the autowiring of the `CacheManager` into the configuration class. We need access to it because we need to retrieve a `Cache` instance which we pass into the constructor of the `SpringCacheBasedUserCache`. We are going to use the cache named `userCache` (which we configured in the `ehcache.xml`). Finally we pass the configured `UserCache` into the `jdbcAuthentications userCache()`.

7-4. Making Access Control Decisions

Problem

In the authentication process, an application will grant a successfully authenticated user a set of authorities. When this user attempts to access a resource in the application, the application has to decide whether the resource is accessible with the granted authorities or other characteristics.

Solution

The decision on whether a user is allowed to access a resource in an application is called an access control decision. It is made based on the user's authentication status, and the resource's nature and access attributes. In Spring Security, access control decisions are made by access decision managers, which have to implement the `AccessDecisionManager` interface. You are free to create your own access decision managers by implementing this interface, but Spring Security comes with three convenient access decision managers based on the voting approach. They are shown in Table 7-6.

Table 7-6. Access Decision Managers That Come with Spring Security

Access Decision Manager	When to Grant Access
AffirmativeBased	At least one voter votes to grant access.
ConsensusBased	A consensus of voters votes to grant access.
UnanimousBased	All voters vote to abstain or grant access (no voter votes to deny access).

All these access decision managers require a group of *voters* to be configured for voting on access control decisions. Each voter has to implement the `AccessDecisionVoter` interface. A voter can vote to grant, abstain, or deny access to a resource. The voting results are represented by the `ACCESS_GRANTED`, `ACCESS_DENIED`, and `ACCESS_ABSTAIN` constant fields defined in the `AccessDecisionVoter` interface.

By default, if no access decision manager is specified explicitly, Spring Security will automatically configure an `AffirmativeBased` access decision manager with the following two voters configured:

`RoleVoter` votes for an access control decision based on a user's role. It will only process access attributes that start with the `ROLE_` prefix, but this prefix can be customized. It votes to grant access if the user has the same role as required to access the resource or to deny access if the user lacks any role required to access the resource.

If the resource does not have an access attribute starting with `ROLE_`, it will abstain from voting.

`AuthenticatedVoter` votes for an access control decision based on a user's authentication level. It will only process the access attributes `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY`. It votes to grant access if the user's authentication level is higher than the required attribute. From highest to lowest, authentication levels are fully authenticated, authentication remembered, and anonymously authenticated.

How It Works

By default, Spring Security will automatically configure an access decision manager if none is specified. This default access decision manager is equivalent to the one defined with the following bean configuration:

```
<bean id="accessDecisionManager"
      class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="decisionVoters">
      <list>
        <bean class="org.springframework.security.access.vote.RoleVoter" />
        <bean class="org.springframework.security.access.vote.AuthenticatedVoter" />
      </list>
    </property>
</bean>
```

This default access decision manager and its decision voters should satisfy most typical authorization requirements. However, if they don't satisfy yours, you can create your own. In most cases, you'll only need to create a custom voter. For example, you can create a voter to vote for a decision based on a user's IP address:

```
package com.apress.springrecipes.board.security;

import org.springframework.security.core.Authentication;
import org.springframework.security.access.ConfigAttribute;

import org.springframework.security.web.authentication.WebAuthenticationDetails;
import org.springframework.security.access.AccessDecisionVoter;

import java.util.Collection;

public class IpAddressVoter implements AccessDecisionVoter {

    public static final String IP_PREFIX = "IP_";
    public static final String IP_LOCAL_HOST = "IP_LOCAL_HOST";

    public boolean supports(ConfigAttribute attribute) {
        return attribute.getAttribute() != null
            && attribute.getAttribute().startsWith(IP_PREFIX);
    }

    public boolean supports(Class clazz) {
        return true;
    }

    public int vote(Authentication authentication, Object object,
                   Collection<ConfigAttribute> configList) {
        if (!(authentication.getDetails() instanceof WebAuthenticationDetails)) {
            return ACCESS_DENIED;
        }

        WebAuthenticationDetails details =
            (WebAuthenticationDetails) authentication.getDetails();
        String address = details.getRemoteAddress();
```

```

int result = ACCESS_ABSTAIN;
for (ConfigAttribute config : configList) {

    result = ACCESS_DENIED;
    if (IP_LOCAL_HOST.equals(config.getAttribute())) {
        if (address.equals("127.0.0.1") || address.equals("0:0:0:0:0:0:0:1")) {
            return ACCESS_GRANTED;
        }
    }
}
return result;
}
}

```

Note that this voter will only process the access attributes that start with the IP_ prefix. At the moment, it only supports the IP_LOCAL_HOST access attribute. If the user is a web client whose IP address is equal to 127.0.0.1 or 0:0:0:0:0:0:0:1—the last value being returned by networkless Linux workstations—this voter will vote to grant access. Otherwise, it will vote to deny access. If the resource does not have an access attribute starting with IP_, it will abstain from voting.

Next, you have to define a custom access decision manager that includes this voter. If you define this access decision manager in `board-security.xml`, you will have to include the beans prefix, because the default schema is security.

```

<beans:bean id="accessDecisionManager"
    class="org.springframework.security.access.vote.AffirmativeBased">
    <beans:property name="decisionVoters">
        <beans:list>
            <beans:bean
                class="org.springframework.security.access.vote.RoleVoter" />
            <beans:bean
                class="org.springframework.security.access.vote.AuthenticatedVoter" />
            <beans:bean class="com.apress.springrecipes.board.security.IpAddressVoter" />
        </beans:list>
    </beans:property>
</beans:bean>

```

Now, suppose you would like to allow users of the machine running the web container (i.e., the server administrators) to delete messages without logging in. You have to refer to this access decision manager from the `<http>` configuration element and add the access attribute IP_LOCAL_HOST to the URL pattern `/messageDelete.htm*`:

```

<http access-decision-manager-ref="accessDecisionManager">
    <intercept-url pattern="/messageList*" access="ROLE_USER,ROLE_GUEST" />
    <intercept-url pattern="/messagePost*" access="ROLE_USER" />
    <intercept-url pattern="/messageDelete*"
        access="ROLE_ADMIN,IP_LOCAL_HOST" />
    ...
</http>

```

Then, if you access this message board application from localhost, you needn't log in as an administrator to delete a posted message.

Using Expression to make Access Control Decisions

Although the `AccessDecisionVoters` allow for a certain degree of flexibility sometimes one wants more complex access control rules or be more flexible. With Spring Security it is also possible to use Springs Expression Language (SpEL) to create powerful access control rules.

To enable the use of expressions (by default this is disabled, unless Java configuration is used) you have to set the `use-expressions` attribute on the `<http>` element to `true`.

```
<http use-expressions="true">
    <intercept-url pattern="/messageList*" access="hasAnyRole('ROLE_USER','ROLE_ANONYMOUS')"/>
    <intercept-url pattern="/messagePost*" access="hasRole('ROLE_USER')"/>
    <intercept-url pattern="/messageDelete*" access="hasRole('ROLE_ADMIN')"/>
    <form-login />
</http>
```

One thing to notice is the value in the `access` attribute of the `<intercept-url>` elements. Those have to be rewritten to be valid expressions. Spring Security supports a couple of expressions out of the box (see Table 7-7 for a list). Using constructs like and, or, and not one can create very powerful and flexible expressions.

Table 7-7. Spring Security build in expressions

Expression	Description
<code>hasRole('role')</code>	Returns true if the current user has the given role (authority is the same as role)
<code>hasAuthority('authority')</code>	Returns true if the current user has at least one of the given roles (authority is the same as role)
<code>hasAnyRole('role1','role2')</code>	Returns true if the current user has the given ip-address.
<code>hasAnyAuthority('auth1','auth2')</code>	The current user
<code>hasIpAddress('ip-address')</code>	Access to the Spring Security authentication object.
<code>principal</code>	Always evaluates to true
<code>permitAll</code>	Always evaluates to false
<code>denyAll</code>	Returns true if the current user is anonymous
<code>isAnonymous()</code>	Returns true if the current user logged in by the means of remember-me functionality
<code>isRememberMe()</code>	Returns true if this is not an anonymous user
<code>isAuthenticated()</code>	Returns true if the user is not an anonymous nor a remember-me user.
<code>isFullyAuthenticated()</code>	

When `use-expressions` is set to `true` Spring Security will automatically configure an access decision manager with a `WebExpressionVoter`. This access decision manager is equivalent to the one defined with the following bean configuration.

```
<bean id="accessDecisionManager"
      class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="decisionVoters">
      <list>
```

```

<bean class="org.springframework.security.web.access.expression.WebExpressionVoter" />
</list>
</property>
</bean>

<http use-expressions="true">
...
<intercept-url pattern="/messageDelete*"
access="hasRole('ROLE_ADMIN') or hasIpAddress('127.0.0.1')
or hasIpAddress('0:0:0:0:0:0:0:1') />
</http>

```

The expression above would give access to deletion of a post if someone had the ADMIN role or was logged in on the local machine. In the previous section we needed to create our own custom AccessDecisionVoter, now you only have to write an expression.

```

@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/messageList*").hasAnyRole("USER", "GUEST")
                .antMatchers("/messagePost*").hasRole("USER")
                .antMatchers("/messageDelete*")
                .access("hasRole('ROLE_ADMIN') or hasIpAddress('127.0.0.1') or
hasIpAddress('0:0:0:0:0:0:0:1')")
            ...
    }
    ...
}

```

Expression can also be used with Java based configuration, for this you use the access method and write the expression inside it.

Although Spring Security has already several build in functions that can be used when creating expressions it is also possible to extend the functions with your own. For this we need to create a class which implements the SecurityExpressionOperations interface and register it with Spring Security. Although it would be possible to create a class which implements all the methods on this interface it is in general easier to extend the default when you want to add expressions.

```

package com.apress.springrecipes.board.security;

import org.springframework.security.core.Authentication;
import org.springframework.security.web.FilterInvocation;
import org.springframework.security.web.access.expression.WebSecurityExpressionRoot;

public class ExtendedWebSecurityExpressionRoot extends WebSecurityExpressionRoot {

    public ExtendedWebSecurityExpressionRoot(Authentication a, FilterInvocation fi) {

```

```

        super(a, fi);
    }

    public boolean localAccess() {
        return hasIpAddress("127.0.0.1") || hasIpAddress("0:0:0:0:0:0:0:1");
    }
}

```

We extended `WebSecurityExpressionRoot` which provides the default implementation and we added the method `localAccess()`. This method checks if we are logging in from the local machine. To make this class available for Spring Security we need to create `SecurityExpressionHandler`.

```

package com.apress.springrecipes.board.security;

import org.springframework.security.access.expression.SecurityExpressionOperations;
import org.springframework.security.authentication.AuthenticationTrustResolver;
import org.springframework.security.authentication.AuthenticationTrustResolverImpl;
import org.springframework.security.core.Authentication;
import org.springframework.security.web.FilterInvocation;
import org.springframework.security.web.access.expression.DefaultWebSecurityExpressionHandler;
import org.springframework.security.web.access.expression.WebSecurityExpressionRoot;

public class ExtendedWebSecurityExpressionHandler extends DefaultWebSecurityExpressionHandler {

    private AuthenticationTrustResolver trustResolver = new AuthenticationTrustResolverImpl();

    @Override
    protected SecurityExpressionOperations
        createSecurityExpressionRoot(Authentication authentication, FilterInvocation fi) {

        ExtendedWebSecurityExpressionRoot root =
            new ExtendedWebSecurityExpressionRoot(authentication, fi);
        root.setPermissionEvaluator(getPermissionEvaluator());
        root.setTrustResolver(trustResolver);
        root.setRoleHierarchy(getRoleHierarchy());
        return root;
    }

    @Override
    public void setTrustResolver(AuthenticationTrustResolver trustResolver) {
        this.trustResolver=trustResolver;
        super.setTrustResolver(trustResolver);
    }
}

```

We are extending the `DefaultWebSecurityExpressionHandler` which provides the default implementation. We override the `createSecurityExpressionRoot` method and let that create an instance of our `ExtendedWebSecurityExpressionRoot` class. As we need to add a couple of collaborators we call the `get` methods of the superclass. As there isn't a `getTrustResolver` method we need to create a new instance of that ourselves and implement the setter method.

```

@Configuration
@EnableWebSecurity
public class MessageBoardSecurityConfiguration extends WebSecurityConfigurerAdapter {

```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .expressionHandler(new ExtendedWebSecurityExpressionHandler())
        .antMatchers("/messageList*").hasAnyRole("USER", "GUEST")
        .antMatchers("/messagePost*").hasRole("USER")
        .antMatchers("/messageDelete*").access("hasRole('ROLE_ADMIN') or localAccess()")

```

We set our custom expression handler with the `expressionHandler` method. Now we can rewrite our expression using our `localAccess()` expression.

```

<http use-expressions="true" >
    <expression-handler ref="customExpressionHandler" />
    <intercept-url pattern="/messageDelete*" access="hasRole('ROLE_ADMIN') or localAccess()" />
    ...
</http>

<beans:bean id="customExpressionHandler"
            class="com.apress.springrecipes.board.security.
ExtendedWebSecurityExpressionHandler" />

```

When using XML configuration you have to wire up the custom implementation using the `<expression-handler />` element and create a bean for the `ExtendedWebSecurityExpressionHandler`.

7-5. Securing Method Invocations

Problem

As an alternative or a complement to securing URL access in the web layer, sometimes you may need to secure method invocations in the service layer. For example, in the case that a single controller has to invoke multiple methods in the service layer, you may wish to enforce fine-grained security controls on these methods.

Solution

Spring Security enables you to secure method invocations in a declarative way. First, you can embed a `<security:intercept-methods>` element in a bean definition to secure its methods. Alternatively, you can configure a global `<global-method-security>` element to secure multiple methods matched with AspectJ pointcut expressions. You can also annotate methods declared in a bean interface or an implementation class with the `@Secured` annotation and then enable security for them in `<global-method-security>` or using the `@EnableGlobalMethodSecurity` annotation.

How It Works

Securing Methods by Embedding a Security Interceptor

First, you can secure a bean's methods by embedding a `<security:intercept-methods>` element in the bean definition. For example, you can secure the methods of the `messageBoardService` bean defined in `board-service.xml`. As this element is defined in the `security` schema, you have to import it beforehand.

```

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

```

```

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

<bean id="messageBoardService"
    class="com.apress.springrecipes.board.service.MessageBoardServiceImpl">
    <security:intercept-methods
        access-decision-manager-ref="accessDecisionManager">
        <security:protect
            method="com.apress.springrecipes.board.service.MessageBoardService.listMessages"
            access="ROLE_USER,ROLE_GUEST" />
        <security:protect
            method="com.apress.springrecipes.board.service.MessageBoardService.postMessage"
            access="ROLE_USER" />
        <security:protect
            method="com.apress.springrecipes.board.service.MessageBoardService.deleteMessage"
            access="ROLE_ADMIN,IP_LOCAL_HOST" />
        <security:protect
            method="com.apress.springrecipes.board.service.MessageBoardService.findMessageById"
            access="ROLE_USER,ROLE_GUEST" />
    </security:intercept-methods>
</bean>
...
</beans>
```

In a bean's `<security:intercept-methods>`, you can specify multiple `<security:protect>` elements to specify access attributes for this bean's methods. You can match multiple methods by specifying a method name pattern with wildcards. If you would like to use a custom access decision manager, you can specify it in the `access-decision-manager-ref` attribute.

Securing Methods with Pointcuts

Second, you can define global pointcuts in `<global-method-security>` to secure methods using AspectJ pointcut expressions, instead of embedding a security interceptor in each bean whose methods require security. You should configure the `<global-method-security>` element in `board-security.xml` for centralizing security configurations. As the default namespace of this configuration file is `security`, you needn't specify a prefix for this element explicitly. You can also specify a custom access decision manager in the `access-decision-manager-ref` attribute.

```

<global-method-security
    access-decision-manager-ref="accessDecisionManager">
    <protect-pointcut
        expression="execution(* com.apress.springrecipes.board.service.*Service.list*(..))"
        access="ROLE_USER,ROLE_GUEST" />
    <protect-pointcut
        expression="execution(* com.apress.springrecipes.board.service.*Service.post*(..))"
        access="ROLE_USER" />
    <protect-pointcut
        expression="execution(* com.apress.springrecipes.board.service.*Service.delete*(..))"
        access="ROLE_ADMIN,IP_LOCAL_HOST" />
```

```
<protect-pointcut
    expression="execution(* com.apress.springrecipes.board.service.*Service.find*(..))"
    access="ROLE_USER,ROLE_GUEST" />
</global-method-security>
```

To test this approach, you have to delete the preceding `<security:intercept-methods>` element.

Securing Methods with Annotations

The third approach to securing methods is by annotating them with `@Secured`. For example, you can annotate the methods in `MessageBoardServiceImpl` with the `@Secured` annotation and specify the access attributes as its value, whose type is `String[]`.

```
package com.apress.springrecipes.board.service;
...
import org.springframework.security.access.annotation.Secured;

public class MessageBoardServiceImpl implements MessageBoardService {
    ...
    @Secured({"ROLE_USER", "ROLE_GUEST"})
    public List<Message> listMessages() {
        ...
    }

    @Secured("ROLE_USER")
    public synchronized void postMessage(Message message) {
        ...
    }

    @Secured({"ROLE_ADMIN", "IP_LOCAL_HOST"})
    public synchronized void deleteMessage(Message message) {
        ...
    }

    @Secured({"ROLE_USER", "ROLE_GUEST"})
    public Message findMessageById(Long messageId) {
        return messages.get(messageId);
    }
}
```

Then, in `<global-method-security>`, you have to enable security for methods annotated with `@Secured`.

```
<global-method-security secured-annotations="enabled"
    access-decision-manager-ref="accessDecisionManager" />
```

To do the same in Java config we have to add the `@EnableGlobalMethodSecurity` annotation to our configuration class.

```
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
public class MessageBoardConfiguration { ... }
```

Note It is important that you add the `@EnableGlobalMethodSecurity` annotation or `<global-method-security>` element to the application context configuration that contains the beans you want to secure!

7-6. Handling Security in Views

Problem

Sometimes, you may wish to display a user's authentication information, such as the principal name and the granted authorities, in the views of your web application. In addition, you would like to render the view contents conditionally according to the user's authorities.

Solution

Although you can write JSP scriptlets in your JSP files to retrieve authentication and authorization information through the Spring Security API, it's not an efficient solution. Spring Security provides a JSP tag library for you to handle security in JSP views. It includes tags that can display a user's authentication information and render the view contents conditionally according to the user's authorities.

How It Works

Displaying Authentication Information

Suppose you would like to display a user's principal name and granted authorities in the header of the message listing page (i.e., `messageList.jsp`). First, you have to import Spring Security's tag library definition.

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

<html>
<head>
<title>Message List</title>
</head>

<body>
Welcome! <security:authentication property="name" /></h2>

<security:authentication property="authorities" var="authorities" />
<ul>
<c:forEach items="${authorities}" var="authority">
  <li>${authority.authority}</li>
</c:forEach>
</ul>
<hr />
...
</body>
</html>
```

The `<security:authentication>` tag exposes the current user's Authentication object for you to render its properties. You can specify a property name or property path in its `property` attribute. For example, you can render a user's principal name through the `name` property.

In addition to rendering an authentication property directly, this tag supports storing the property in a JSP variable, whose name is specified in the `var` attribute. For example, you can store the `authorities` property, which contains the authorities granted to the user, in the JSP variable `authorities`, and render them one by one with a `<c:forEach>` tag. You can further specify the variable scope with the `scope` attribute.

Rendering View Contents Conditionally

If you would like to render view contents conditionally according to a user's authorities, you can use the `<security:authorize>` tag. For example, you can decide whether to render the message authors according to the user's authorities:

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

<html>
<head>
<title>Message List</title>
</head>

<body>
...
<c:forEach items="${messages}" var="message">
<table>
<security:authorize ifAllGranted="ROLE_ADMIN,ROLE_USER">
<tr>
<td>Author</td>
<td>${message.author}</td>
</tr>
</security:authorize>
...
</table>
<hr />
</c:forEach>
...
</body>
</html>
```

If you want the enclosing content to be rendered only when the user has been granted certain authorities at the same time, you have to specify them in the `ifAllGranted` attribute. Otherwise, if the enclosing content can be rendered with any of the authorities, you have to specify them in the `ifAnyGranted` attribute:

```
<security:authorize ifAnyGranted="ROLE_ADMIN,ROLE_USER">
<tr>
<td>Author</td>
<td>${message.author}</td>
</tr>
</security:authorize>
```

You can also render the enclosing content when a user has not been granted any of the authorities specified in the `ifNotGranted` attribute:

```
<security:authorize ifNotGranted="ROLE_GUEST">
<tr>
  <td>Author</td>
  <td>${message.author}</td>
</tr>
</security:authorize>
```

7-7. Handling Domain Object Security

Problem

Sometimes, you may have complicated security requirements that require handling security at the domain object level. That means you have to allow each domain object to have different access attributes for different principals.

Solution

Spring Security provides a module named `ACL` that allows each domain object to have its own *access control list (ACL)*. An `ACL` contains a domain object's *object identity* to associate with the object, and also holds multiple *access control entries (ACEs)*, each of which contains the following two core parts:

- **Permissions:** An ACE's permissions are represented by a particular bit mask, with each bit value for a particular type of permission. The `BasePermission` class predefines five basic permissions as constant values for you to use: `READ` (bit 0 or integer 1), `WRITE` (bit 1 or integer 2), `CREATE` (bit 2 or integer 4), `DELETE` (bit 3 or integer 8), and `ADMINISTRATION` (bit 4 or integer 16). You can also define your own using other unused bits.
- **Security Identity (SID):** Each ACE contains permissions for a particular SID. An SID can be a principal (`PrincipalSid`) or an authority (`GrantedAuthoritySid`) to associate with permissions.

In addition to defining the `ACL` object model, Spring Security defines APIs for reading and maintaining the model, and provides high-performance JDBC implementations for these APIs. To simplify `ACL`'s usages, Spring Security also provides facilities, such as access decision voters and JSP tags, for you to use `ACL` consistently with other security facilities in your application.

How It Works

Setting Up an `ACL` Service

Spring Security provides built-in support for storing `ACL` data in a relational database and accessing it with JDBC. First, you have to create the following tables in your database for storing `ACL` data:

```
CREATE TABLE ACL_SID(
    ID      BIGINT      NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    SID     VARCHAR(100) NOT NULL,
    PRINCIPAL SMALLINT   NOT NULL,
    PRIMARY KEY (ID),
    UNIQUE (SID, PRINCIPAL)
);
```

```

CREATE TABLE ACL_CLASS(
    ID      BIGINT      NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    CLASS   VARCHAR(100) NOT NULL,
    PRIMARY KEY (ID),
    UNIQUE (CLASS)
);

CREATE TABLE ACL_OBJECT_IDENTITY(
    ID          BIGINT      NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    OBJECT_ID_CLASS BIGINT      NOT NULL,
    OBJECT_ID_IDENTITY BIGINT      NOT NULL,
    PARENT_OBJECT    BIGINT,
    OWNER_SID        BIGINT,
    ENTRIES_INHERITING SMALLINT  NOT NULL,
    PRIMARY KEY (ID),
    UNIQUE (OBJECT_ID_CLASS, OBJECT_ID_IDENTITY),
    FOREIGN KEY (PARENT_OBJECT) REFERENCES ACL_OBJECT_IDENTITY,
    FOREIGN KEY (OBJECT_ID_CLASS) REFERENCES ACL_CLASS,
    FOREIGN KEY (OWNER_SID) REFERENCES ACL_SID
);

CREATE TABLE ACL_ENTRY(
    ID          BIGINT      NOT NULL GENERATED BY DEFAULT AS IDENTITY,
    ACL_OBJECT_IDENTITY BIGINT      NOT NULL,
    ACE_ORDER     INT        NOT NULL,
    SID          BIGINT      NOT NULL,
    MASK         INTEGER     NOT NULL,
    GRANTING     SMALLINT   NOT NULL,
    AUDIT_SUCCESS SMALLINT   NOT NULL,
    AUDIT_FAILURE SMALLINT   NOT NULL,
    PRIMARY KEY (ID),
    UNIQUE (ACL_OBJECT_IDENTITY, ACE_ORDER),
    FOREIGN KEY (ACL_OBJECT_IDENTITY) REFERENCES ACL_OBJECT_IDENTITY,
    FOREIGN KEY (SID) REFERENCES ACL_SID
);

```

Spring Security defines APIs and provides high-performance JDBC implementations for you to access ACL data stored in these tables, so you'll seldom have a need to access ACL data from the database directly.

As each domain object can have its own ACL, there may be a large number of ACLs in your application. Fortunately, Spring Security supports caching ACL objects. You can continue to use Ehcache as your cache implementation and create a new configuration for ACL caching in `ehcache.xml` (located in the classpath root).

```

<ehcache>
    ...
    <cache name="aclCache"
        maxElementsInMemory="1000"
        eternal="false"
        timeToIdleSeconds="600"
        timeToLiveSeconds="3600"
        overflowToDisk="true"
    />
</ehcache>

```

Next, you have to set up an ACL service for your application. However, as Spring Security doesn't support configuring the ACL module with XML schema-based configurations, you have to configure this module with a group of normal Spring beans. As the default namespace of `board-security.xml` is `security`, it's cumbersome to configure an ACL in this file using the standard XML elements in the `beans` namespace. For this reason, let's create a separate bean configuration file named `board-acl.xml`, which will store ACL-specific configurations, and add its location in the web deployment descriptor:

```
<web-app ...>
  ...
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/board-service.xml
      /WEB-INF/board-security.xml
      /WEB-INF/board-acl.xml
    </param-value>
  </context-param>
</web-app>
```

In an ACL configuration file, the core bean is an ACL service. In Spring Security, there are two interfaces that define operations of an ACL service: `AclService` and `MutableAclService`. `AclService` defines operations for you to read ACLs. `MutableAclService` is a subinterface of `AclService` that defines operations for you to create, update, and delete ACLs. If your application only needs to read ACLs, you can simply choose an `AclService` implementation, such as `JdbcAclService`. Otherwise, you should choose a `MutableAclService` implementation, such as `JdbcMutableAclService`.

```
<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="aclCache"
    class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
    <constructor-arg ref="aclEhCache" />
    <constructor-arg ref="permissionGrantingStrategy" />
    <constructor-arg ref="authorizationStrategy" />
  </bean>

  <bean id="aclEhCache"
    class="org.springframework.cache.ehcache.EhCacheFactoryBean">
    <property name="cacheManager" ref="cacheManager" />
    <property name="cacheName" value="aclCache" />
  </bean>

  <bean id="lookupStrategy"
    class="org.springframework.security.acls.jdbc.BasicLookupStrategy">
    <constructor-arg ref="dataSource" />
    <constructor-arg ref="aclCache" />
    <constructor-arg ref="authorizationStrategy" />
    <constructor-arg ref="permissionGrantingStrategy" />
  </bean>

  <bean id="authorizationStrategy"
    class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl">
    <constructor-arg>
      <bean id="adminRole"
```

```

        class="org.springframework.security.core.authority.SimpleGrantedAuthority">
        <constructor-arg value="ROLE_ADMIN" />
    </bean>
</constructor-arg>
</bean>

<bean id="permissionGrantingStrategy"
      class="org.springframework.security.acls.domain.DefaultPermissionGrantingStrategy">
<constructor-arg>
    <bean class="org.springframework.security.acls.domain.ConsoleAuditLogger" />
</constructor-arg>
</bean>

<bean id="aclService"
      class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
<constructor-arg ref="dataSource" />
<constructor-arg ref="lookupStrategy" />
<constructor-arg ref="aclCache" />
<property name="sidIdentityQuery" value="values identity_val_local()" />
</bean>
</beans>
```

The core bean definition in this ACL configuration file is the ACL service, which is an instance of `JdbcMutableAclService` that allows you to maintain ACLs. This class requires three constructor arguments. The first is a data source for creating connections to a database that stores ACL data. You should have a data source defined in `board-service.xml` beforehand so that you can simply refer to it here (assuming that you have created the ACL tables in the same database). The third constructor argument is a cache instance to use with an ACL, which you can configure using Ehcache as the back-end cache implementation.

The second argument—`sidIdentityQuery`—is a lookup strategy that performs lookup for an ACL service. Note, if you’re using HSQLDB, that the `sidIdentityQuery`’s property is not necessary, because it defaults to this database. If using another database—as in this case for Apache Derby—an explicit value is necessary.

The only implementation that comes with Spring Security is `BasicLookupStrategy`, which performs basic lookup using standard and compatible SQL statements. If you want to make use of advanced database features to increase lookup performance, you can create your own lookup strategy by implementing the `LookupStrategy` interface. A `BasicLookupStrategy` instance also requires a data source and a cache instance. Besides, it requires a constructor argument whose type is `AclAuthorizationStrategy`. This object determines whether a principal is authorized to change certain properties of an ACL, usually by specifying a required authority for each category of properties. For the preceding configurations, only a user who has the `ROLE_ADMIN` role can change an ACL’s ownership, an ACE’s auditing details, or other ACL and ACE details, respectively. Finally it needs a constructor argument whose type is `PermissionGrantingStrategy`. This objects responsibility it to check if the Acl grants access to the given Sid with the Permissions it has.

Finally, `JdbcMutableAclService` embeds standard SQL statements for maintaining ACL data in a relational database. However, those SQL statements may not be compatible with all database products. For example, you have to customize the identity query statement for Apache Derby.

Maintaining ACLs for Domain Objects

In your back-end services and DAOs, you can maintain ACLs for domain objects with the previously defined ACL service via dependency injection. For your message board, you have to create an ACL for a message when it is posted and delete the ACL when this message is deleted:

```
package com.apress.springrecipes.board.service;
...
```

```

import org.springframework.security.acls.model.MutableAcl;
import org.springframework.security.acls.model.MutableAclService;
import org.springframework.security.acls.domain.BasePermission;
import org.springframework.security.acls.model.ObjectIdentity;
import org.springframework.security.acls.domain.ObjectIdentityImpl;
import org.springframework.security.acls.domain.GrantedAuthoritySid;
import org.springframework.security.acls.domain.PrincipalSid;
import org.springframework.security.access.annotation.Secured;
import org.springframework.transaction.annotation.Transactional;

public class MessageBoardServiceImpl implements MessageBoardService {
    ...
    private MutableAclService mutableAclService;

    public void setMutableAclService(MutableAclService mutableAclService) {
        this.mutableAclService = mutableAclService;
    }

    @Transactional
    @Secured("ROLE_USER")
    public synchronized void postMessage(Message message) {
        ...
        ObjectIdentity oid = new ObjectIdentityImpl(Message.class, message.getId());
        MutableAcl acl = mutableAclService.createAcl(oid);
        acl.insertAce(0, BasePermission.ADMINISTRATION, new PrincipalSid(message.getAuthor()), true);
        acl.insertAce(1, BasePermission.DELETE, new GrantedAuthoritySid("ROLE_ADMIN"), true);
        acl.insertAce(2, BasePermission.READ, new GrantedAuthoritySid("ROLE_USER"), true);
        mutableAclService.updateAcl(acl);
    }

    @Transactional
    @Secured({"ROLE_ADMIN", "IP_LOCAL_HOST"})
    public synchronized void deleteMessage(Message message) {
        ...
        ObjectIdentity oid = new ObjectIdentityImpl(Message.class, message.getId());
        mutableAclService.deleteAcl(oid, false);
    }
}

```

When a user posts a message, you create a new ACL for this message at the same time, using the message ID as the ACL's object identity. When a user deletes a message, you delete the corresponding ACL as well. For a new message, you insert the following three ACEs into its ACL:

- The message author is permitted to administrate this message.
- A user who has the ROLE_ADMIN role is permitted to delete this message.
- A user who has the ROLE_USER role is permitted to read this message.

JdbcMutableAclService requires that the calling methods have transactions enabled so that its SQL statements can run within transactions. So, you annotate the two methods involving ACL maintenance with the @Transactional annotation and then define a transaction manager and <tx:annotation-driven> in board-service.xml. Also, don't forget to inject the ACL service into the message board service for it to maintain ACLs.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">
    ...
    <tx:annotation-driven />
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="messageBoardService"
        class="com.apress.springrecipes.board.service.MessageBoardServiceImpl">
        <property name="mutableAclService" ref="aclService" />
    </bean>
</beans>

```

Making Access Control Decisions using expressions

With an ACL for each domain object, you can use an object's ACL to make access control decisions on methods that involve this object. For example, when a user attempts to delete a posted message, you can consult this message's ACL about whether the user is permitted to delete this message.

Configuring ACL can be a daunting task, luckily you can use annotations and expressions to make your life easier. We can use the `@PreAuthorize` and `@PreFilter` annotations to check if someone is allowed to execute the method or use certain method arguments. The `@PostAuthorize` and `@PostFilter` can be used to check if a user is allowed to access the result or to filter results based on the ACL.

To enable the processing of these annotations you need to set the `pre-post-annotations` attribute of the `<global-method-security>` element to `enabled` or when using annotations, the `prePostEnabled` attribute of the `@EnableGlobalMethodSecurity` annotation to `true`.

```
<global-method-security pre-post-enabled="true" ... >
```

Or for Java based configuration:

```
@EnableGlobalMethodSecurity(prePostEnabled=true)
```

In addition we need to configure infrastructure components to be able to make decisions. We need to setup an `AclPermissionEvaluator` which is needed to evaluate the permission for an object.

```
<bean id="aclPermissionEvaluator" class="org.springframework.security.acls.AclPermissionEvaluator">
    <constructor-arg ref="aclService" />
</bean>
```

The equivalent in Java config:

```
package com.apress.springrecipes.board.web.config;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.Cache;
import org.springframework.cache.CacheManager;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.acls.AclPermissionEvaluator;
import org.springframework.security.acls.domain.AclAuthorizationStrategyImpl;
import org.springframework.security.acls.domain.ConsoleAuditLogger;
import org.springframework.security.acls.domain.DefaultPermissionGrantingStrategy;
import org.springframework.security.acls.domain.SpringCacheBasedAclCache;
import org.springframework.security.acls.jdbc.BasicLookupStrategy;
import org.springframework.security.acls.jdbc.JdbcMutableAclService;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;

import javax.sql.DataSource;

@Configuration
public class MessageBoardAclSecurityConfiguration {

    @Autowired
    private DataSource dataSource;

    @Autowired
    private CacheManager cacheManager;

    @Bean
    public SpringCacheBasedAclCache aclCache() {
        Cache cache = cacheManager.getCache("aclCache");
        return new SpringCacheBasedAclCache(cache, permissionGrantingStrategy(), authorizationStrategy());
    }

    @Bean
    public BasicLookupStrategy lookupStrategy() {
        return new BasicLookupStrategy(dataSource, aclCache(),
            authorizationStrategy(), permissionGrantingStrategy());
    }

    @Bean
    public DefaultPermissionGrantingStrategy permissionGrantingStrategy() {
        return new DefaultPermissionGrantingStrategy(new ConsoleAuditLogger());
    }

    @Bean
    public AclAuthorizationStrategyImpl authorizationStrategy() {
        return new AclAuthorizationStrategyImpl(new SimpleGrantedAuthority("ROLE_ADMIN"));
    }

    @Bean
    public JdbcMutableAclService jdbcMutableAclService() {
        return new JdbcMutableAclService(dataSource, lookupStrategy(), aclCache());
    }
}

```

```

    @Bean
    public AclPermissionEvaluator permissionEvaluator() {
        return new AclPermissionEvaluator(jdbcMutableAclService());
    }

}

```

This might look a little verbose but remember that we did not yet configure the collaborating classes in Java yet, until now ACL was done it in XML. We put the configuration in a separate class `MessageBoardAclSecurityConfiguration`. This class needs also be added to the `getRootConfigClasses()` method of the `MessageBoardApplicationInitializer` class.

```

@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class<?>[]{MessageBoardConfiguration.class,
                           MessageBoardSecurityConfiguration.class,
                           MessageBoardAclSecurityConfiguration.class};
}

```

The `AclPermissionEvaluator` requires an `AclService` to obtain the ACL for the objects it needs to check. When doing Java based configuration this is enough as the `PermissionEvaluator` will be automatically detected and wired to the `DefaultMethodSecurityExpressionHandler`. When using XML based configuration an extra step is needed, we need to explicitly configure the `DefaultMethodSecurityExpressionHandler` and we need to explicitly wire it to the `<global-method-security>` element.

```

<bean id="methodExpressionHandler" class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
    <property name="permissionEvaluator" ref="aclPermissionEvaluator" />
</bean>

<security:global-method-security pre-post-annotations="enabled">
    <security:expression-handler ref="methodExpressionHandler" />
</security:global-method-security>

```

Now everything is in place to use the annotations together with expressions to control our access.

```

public class MessageBoardServiceImpl implements MessageBoardService {

    @Transactional
    @PreAuthorize("hasPermission(#message, 'delete') or hasPermission(#message, 'admin')")
    public synchronized void deleteMessage(Message message) {
        messages.remove(message.getId());
        ObjectIdentity oid = new ObjectIdentityImpl(Message.class, message.getId());
        mutableAclService.deleteAcl(oid, false);
    }

    @PostFilter("hasPermission(filterObject, 'read')")
    public List<Message> listMessages() {
        return new ArrayList<Message>(messages.values());
    }
}

```

```
@PostAuthorize("hasPermission(returnObject, 'read')")
public Message findMessageById(Long messageId) {
    return messages.get(messageId);
}
```

You probably noticed the different annotations and the expressions inside these annotations. The `@PreAuthorize` annotation can be used to check if someone has the correct permissions to execute the method. The expression uses `#message` this refers to the method argument with the name `message`. The `hasPermission` expression is a built-in expression from Spring Security (see Table 7-7).

The `@PostFilter` annotation allows you to filter the collection and remove the elements someone isn't allowed to read. In the expression the keyword `filterObject` refers to an element in the collection, to remain in the collection the logged in user needs to have read permission.

`@PostAuthorize` can be used to check if a single return value can be used (i.e., if the user has the right permissions). To use the return value in an expression use the keyword `returnObject`.

Summary

In this chapter, you learned how to secure applications using Spring Security 3.2. It can be used to secure any Java application, but it's mostly used for web applications. The concepts of authentication, authorization, and access control are essential in the security area, so you should have a clear understanding of them.

You often have to secure critical URLs by preventing unauthorized access to them. Spring Security can help you to achieve this in a declarative way. It handles security by applying servlet filters, which can be configured with simple XML elements or java based configuration. If your web application's security requirements are simple and typical, you can enable the HTTP auto-config feature so that Spring Security will automatically configure the basic security services for you.

Spring Security supports multiple ways for users to log into a web application, such as form-based login and HTTP Basic authentication. It also provides an anonymous login service that allows you to handle an anonymous user just like a normal user. Remember-me support allows an application to remember a user's identity across multiple browser sessions.

Spring Security supports multiple ways of authenticating users and has built-in provider implementations for them. For example, it supports authenticating users against in-memory definitions, a relational database, and an LDAP repository. You should always store encrypted passwords in your user repository, because clear-text passwords are vulnerable to hacker attacks. Spring Security also supports caching user details locally to save you the overhead of performing remote queries.

Decisions on whether a user is allowed to access a given resource are made by access decision managers. Spring Security comes with three access decision managers that are based on the voting approach. All of them require a group of voters to be configured for voting on access control decisions.

Spring Security enables you to secure method invocations in a declarative way, either by embedding a security interceptor in a bean definition, or matching multiple methods with AspectJ pointcut expressions or annotations. Spring Security also allows you to display a user's authentication information in JSP views and render view contents conditionally according to a user's authorities.

Spring Security provides an ACL module that allows each domain object to have an ACL for controlling access. You can read and maintain an ACL for each domain object with Spring Security's high-performance APIs, which are implemented with JDBC. Spring Security also provides facilities such as access decision voters and JSP tags for you to use ACLs consistently with other security facilities.

CHAPTER 8



Spring Mobile

In the current world, more and more mobile devices exist. Most of these mobile devices can access the internet and access websites. Mobile devices might have a browser that lacks certain HTML or JavaScript features you might use on your website, you also might want to show a different website to your mobile users or maybe give them the choice.

You could write all the device detection routines yourself; however, Spring Mobile provides ways to detect the device and to act upon it.

Recipe 8-1. Device detection without Spring Mobile

Problem

You want to detect the type of device that connects to your website.

Solution

Create a Filter that detects the User-Agent of the incoming request and sets a request attribute so that it can be retrieved in a controller.

How It Works

First, let's take a look at the Filter implementation we need to do User-Agent based device detection.

```
package com.apress.springrecipes.mobile.web.filter;

import org.springframework.util.StringUtils;
import org.springframework.web.filter.OncePerRequestFilter;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class DeviceResolverRequestFilter extends OncePerRequestFilter {

    public static final String CURRENT_DEVICE_ATTRIBUTE = "currentDevice";
```

```

public static final String DEVICE_MOBILE = "MOBILE";
public static final String DEVICE_TABLET = "TABLET";
public static final String DEVICE_NORMAL = "NORMAL";

@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                FilterChain filterChain) throws ServletException, IOException {
    String userAgent = request.getHeader("User-Agent");
    String device = DEVICE_NORMAL;

    if (StringUtils.hasText(userAgent)) {
        userAgent = userAgent.toLowerCase();
        if (userAgent.contains("android")) {
            device = userAgent.contains("mobile") ? DEVICE_NORMAL : DEVICE_TABLET;
        } else if (userAgent.contains("ipad") || userAgent.contains("playbook") || userAgent.
        contains("kindle")) {
            device = DEVICE_TABLET;
        } else if (userAgent.contains("mobil") || userAgent.contains("ipod") || userAgent.
        contains("nintendo DS")) {
            device = DEVICE_MOBILE;
        }
    }
    request.setAttribute(CURRENT_DEVICE_ATTRIBUTE, device);
    filterChain.doFilter(request, response);
}
}

```

First there is the retrieval of the User-Agent header from the incoming request. When there is a value in there the filter needs to check what is in the header. There are some if/else constructs in there to do a basic detection of the type of device. There is a special case for Android as that can be a tablet or mobile.

When the filter determined what the type of device is, it is stored as a request attribute so that it is available to other components.

Next there is a controller and jsp to display some information on what is going on. The controller simply directs to a `home.jsp` which is located in the `WEB-INF/views` directory. A configured `InternalResourceViewResolver` takes care of resolving the name to an actual JSP (for more information check the recipes in Chapter 4).

```

package com.apress.springrecipes.mobile.web;

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

import javax.servlet.http.HttpServletRequest;

@Controller
public class HomeController {

```

```

    @RequestMapping("/home")
    public String index(HttpServletRequest request) {
        return "home";
    }
}

```

The home.jsp:

```

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

<h1>Welcome</h1>
<p>
    Your User-Agent header: <c:out value="${header['User-Agent']}"/>
</p>
<p>
    Your type of device: <c:out value="${requestScope.currentDevice}"/>
</p>

</body>
</html>

```

The JSP shows the User-Agent header (if any) and the type of device, which has been determined by your own implemented DeviceResolverRequestFilter.

Finally there is the configuration and bootstrapping logic.

```

package com.apress.springrecipes.mobile.web.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceView;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration {

    @Bean
    public ViewResolver viewResolver() {

```

```

        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }

}

```

Spring MVC is enabled by using the `@EnableWebMvc` annotation and the controller is picked up by the `@ComponentScan`.

For bootstrapping the application there is the `MobileApplicationInitializer` this bootstraps the `DispatcherServlet` and optionally the `ContextLoaderListener`.

```

package com.apress.springrecipes.mobile.web;

import com.apress.springrecipes.mobile.web.config.MobileConfiguration;
import com.apress.springrecipes.mobile.web.filter.DeviceResolverRequestFilter;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

import javax.servlet.Filter;

public class MobileApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

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

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {new DeviceResolverRequestFilter()};
    }

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

```

There are two things to notice here. First the earlier mentioned configuration class is passed to the `DispatcherServlet` by implementing the `getServletConfigClasses` method. Second the implementation of the `getServletFilters` method, which takes care of registering the filter and mapping it to the `DispatcherServlet`.

When the application is deployed, using `http://localhost:8080/mobile/home` will show you the User-Agent used and what type the filter thinks it is (see Figure 8-1).



Figure 8-1. Viewing the application in Chrome

Using Chrome on an iMac produces the above result. When using an iPhone 4 it looks like Figure 8-2:



Figure 8-2. Viewing the application using an iPhone 4

Note For testing different browsers, you can either use a tablet or mobile on your internal network or use a browser plugin like User-Agent switcher for Chrome or Firefox.

Although the filter does its job it is far from complete not to mention the mobile devices that don't match the rules (for instance the Kindle Fire has a different header than a normal Kindle device). It is also quite hard to maintain the list of rules and devices or to test with many devices. Using a library, like Spring Mobile, is much easier than to roll your own.

Recipe 8-2. Device detection with Spring Mobile

Problem

You want to detect the type of device that connects to your website and want to use Spring Mobile to help you with this.

Solution

Use the Spring Mobile DeviceResolver and helper classes to determine the type of device by either configuring the DeviceResolverRequestFilter or DeviceResolverHandlerInterceptor.

How It Works

Both the DeviceResolverRequestFilter and the DeviceResolverHandlerInterceptor delegate detection of the type of device to a DeviceResolver. Spring Mobile provides an implementation of that interface named the LiteDeviceResolver. The DeviceResolver returns a Device object which indicates the type, this Device is stored as a request attribute so that it can be used further down the chain. Spring Mobile comes with a single default implementation of the Device interface, LiteDevice.

Using the DeviceResolverRequestFilter

To use the DeviceResolverRequestFilter is a matter of adding it to the web application and mapping it to the servlet or requests that you want it to handle. For your application that means adding it to the getServletFilters method.

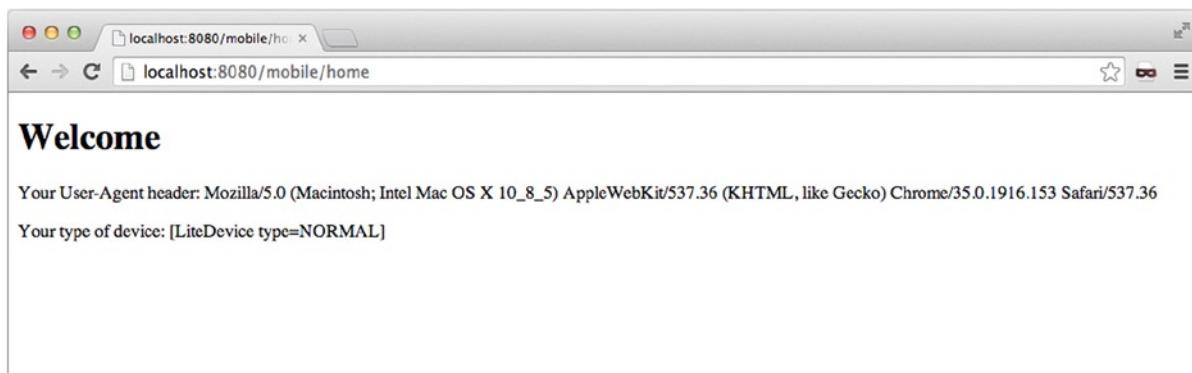
The advantage of using this filter is that it is possible to use it even outside a Spring based application. It could also be used in a, for instance, JSF based application.

```
package com.apress.springrecipes.mobile.web;
...
import org.springframework.mobile.device.DeviceResolverRequestFilter;

public class MobileApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
...
    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {new DeviceResolverRequestFilter()};
    }
}
```

This configuration registers the DeviceResolverRequestFilter and will automatically attach it to requests handled by the DispatcherServlet.

To test issue a request to <http://localhost:8080/mobile/home>, which should display something like the following:



The output for the device is the text as created for the `toString` method on the `LiteDevice` class provided by Spring Mobile.

Using the DeviceResolverHandlerInterceptor

When using Spring Mobile in a Spring MVC based application it is easier to work with the `DeviceResolverHandlerInterceptor`. This needs to be configured in your configuration class and needs to be registered with the `addInterceptors` helper method.

```
package com.apress.springrecipes.mobile.web.config;

import org.springframework.mobile.device.DeviceResolverHandlerInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration extends WebMvcConfigurerAdapter {
...
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new DeviceResolverHandlerInterceptor());
    }
}
```

The `MobileConfiguration` class extends `WebMvcConfigurerAdapter` from this class we can override the `addInterceptors` method. All the interceptors added to the registry will be added to the `HandlerMapping` beans in the application context. When the application is deployed and a request is done for <http://localhost:8080/mobile/home> the result should be the same as for the filter.

8-3. Using Site preferences

Problem

You want to allow the user to choose which type of site they visit with their device and store this for future reference.

Solution

Use the SitePreference support provided by Spring Mobile.

How It Works

Both the SitePreferenceRequestFilter and the SitePreferenceHandlerInterceptor delegate retrieval of the current SitePreference to a SitePreferenceHandler. The default implementation uses a SitePreferenceRepository to store the preferences, by default this is done in a cookie.

Using the SitePreferenceRequestFilter

To use the SitePreferenceRequestFilter is a matter of adding it to the web application and mapping it to the servlet or requests that you want it to handle. For your application that means adding it to the getServletFilters method.

The advantage of using a filter is that it is possible to use it even outside a Spring based application. It could also be used in a JSF-based application.

```
package com.apress.springrecipes.mobile.web;

import org.springframework.mobile.device.site.SitePreferenceRequestFilter;
...
public class MobileApplicationInitializer extends
AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] {
            new DeviceResolverRequestFilter(),
            new SitePreferenceRequestFilter()};
    }
    ...
}
```

Now that the SitePreferenceRequestFilter is registered it will inspect incoming requests. If a request has a parameter named site_preference it will use the passed in value (NORMAL, MOBILE, or TABLET) to set the SitePreference. The determined value is stored in a cookie and used for future reference, if a new value is detected the cookie value will be reset.

Modify the home.jsp pages to include the following that will display the current SitePreference:

```
<p>
    Your site preferences <c:out value="${requestScope.currentSitePreference}" />
</p>
```

When opening the page using the URL http://localhost:8080/mobile/home?site_preference=TABLET will set the SitePreference to TABLET.



Using the SitePreferenceHandlerInterceptor

When using Spring Mobile in a Spring MVC based application it is easier to work with the SitePreferenceHandlerInterceptor. This needs to be configured in your configuration class and needs to be registered with the addInterceptors helper method.

```
package com.apress.springrecipes.mobile.web.config;

import org.springframework.mobile.device.DeviceResolverHandlerInterceptor;
import org.springframework.mobile.device.site. SitePreferenceHandlerInterceptor;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration extends WebMvcConfigurerAdapter {
...
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new DeviceResolverHandlerInterceptor());
        registry.addInterceptor(new SitePreferenceHandlerInterceptor());
    }
}
```

The MobileConfiguration class extends WebMvcConfigurerAdapter from this class we can override the addInterceptors method. All the interceptors added to the registry will be added to the HandlerMapping beans in the application context. When the application is deployed and a request is done for http://localhost:8080/mobile/home?site_preference=TABLET the result should be the same as for the filter in the previous section.

8-4. Using the Device Information to Render Views

Problem

You want to render a different view based on the device or site preferences.

Solution

Use the current Device and SitePreferences to determine which view to render. This can be done manually or by using the LiteDeviceDelegatingViewResolver.

How It Works

Now that the type of Device is known it can be used to your advantage. First let's create some additional views for each type of device supported, put them respectively in a mobile and tablet directory under WEB-INF/views.

The source for mobile/home.jsp:

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

<h1>Welcome Mobile User</h1>
<p>
    Your User-Agent header: <c:out value="${header['User-Agent']}' />
</p>
<p>
    Your type of device: <c:out value="${requestScope.currentDevice}" />
</p>
<p>
    Your site preferences <c:out value="${requestScope.currentSitePreference}" />
</p>
</body>
</html>
```

The tablet/home.jsp:

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

```
<h1>Welcome Tablet User</h1>
<p>
    Your User-Agent header: <c:out value="${header['User-Agent']}" />
</p>
<p>
    Your type of device: <c:out value="${requestScope.currentDevice}" />
</p>
<p>
    Your site preferences <c:out value="${requestScope.currentSitePreference}" />
</p>
</body>
</html>
```

Now that the different views are in place we need to find a way to render the different views based on the Device that has been detected. One way would be to manually get access to the current device from the request and use that to determine which view to render.

```
package com.apress.springrecipes.mobile.web;

import org.springframework.mobile.device.Device;
import org.springframework.mobile.device.DeviceUtils;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;

@Controller
public class HomeController {

    @RequestMapping("/home")
    public String index(HttpServletRequest request) {
        Device device = DeviceUtils.getCurrentDevice(request);
        if (device.isMobile()) {
            return "mobile/home";
        } else if (device.isTablet()) {
            return "tablet/home";
        } else {
            return "home";
        }
    }
}
```

Spring Mobile has a `DeviceUtils` class that can be used to retrieve the current device. The current device is retrieved from a request attribute (`currentDevice`) which has been set by the filter or interceptor. The `Device` can be used to determine which view to render.

Getting the device in each method that needs it isn't very convenient. It would be a lot easier if it could be passed into the controller method as a method argument. For this there is the `DeviceHandlerMethodArgumentResolver` which can be registered and will resolve the method argument to the current device. To retrieve the current SitePreferences you can add the `SitePreferenceHandlerMethodArgumentResolver`.

```

package com.apress.springrecipes.mobile.web.config;

import org.springframework.mobile.device.DeviceHandlerMethodArgumentResolver;
...
import java.util.List;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration extends WebMvcConfigurerAdapter {

    ...
    @Override
    public void addArgumentResolvers(List<HandlerMethodArgumentResolver> argumentResolvers) {
        argumentResolvers.add(new DeviceHandlerMethodArgumentResolver());
        argumentResolvers.add(new SitePreferenceHandlerMethodArgumentResolver());
    }
}

```

Now that these have been registered the controller method can be simplified and the Device can be passed in as a method argument

```

package com.apress.springrecipes.mobile.web;

import org.springframework.mobile.device.Device;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;

@Controller
public class HomeController {

    @RequestMapping("/home")
    public String index(Device device) {
        if (device.isMobile()) {
            return "mobile/home";
        } else if (device.isTablet()) {
            return "tablet/home";
        } else {
            return "home";
        }
    }
}

```

The method signature changed from having a `HttpServletRequest` to a `Device`. That takes care of the lookup and will pass in the current device. However while this is more convenient than manual retrieval it is still quite an antiquated way to determine which view to render. Currently the preferences aren't taken into account but this could be added to the method signature as well and could be used to determine the preference. However it would complicate the detection algorithm.

Imagine this code in multiple controller methods and it will soon become a maintenance nightmare.

Spring Mobile ships with a `LiteDeviceDelegatingViewResolver` which can be used to add additional prefixes and/or suffixes to the view name, before it is passed on to the actual view resolver. It also takes into account the optional site preferences of the user.

```
package com.apress.springrecipes.mobile.web.config;

import org.springframework.mobile.device.view.LiteDeviceDelegatingViewResolver;
...

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration extends WebMvcConfigurerAdapter {
    ...
    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        viewResolver.setOrder(2);
        return viewResolver;
    }

    @Bean
    public ViewResolver mobileViewResolver() {
        LiteDeviceDelegatingViewResolver delegatingViewResolver =
            new LiteDeviceDelegatingViewResolver(viewResolver());
        delegatingViewResolver.setOrder(1);
        delegatingViewResolver.setMobilePrefix("mobile/");
        delegatingViewResolver.setTabletPrefix("tablet/");
        return delegatingViewResolver;
    }
}
```

The `LiteDeviceDelegatingViewResolver` takes a delegate view resolver as a constructor argument, the earlier configured `InternalResourceViewResolver` is passed in as the delegate. Also note the ordering of the view resolvers, you have to make sure that the `LiteDeviceDelegatingViewResolver` executes before any other view resolver. This way it has a chance to determine if a custom view for a particular device exists.

Next notice the configuration as the views for mobile devices are located in the `mobile` directory and for the tablet they are in the `tablet` directory. To add these directories to the view names the prefixes for those device types are set to their respective directories. Now when a controller returns `home` as the view name to select for a mobile device it would be turned into `mobile/home`. This modified name is passed on to the `InternalResourceViewResolver` which turns it into `/WEB-INF/views/mobile/home.jsp`, the page we actually want to render.

```
package com.apress.springrecipes.mobile.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
@Controller
public class HomeController {

    @RequestMapping("/home")
    public String index() {
        return "home";
    }
}
```

The controller is quite clean now. Its only concern is to return the name of the view. The determination of which view to render is left to the configured view resolvers. The `LiteDeviceDelegatingViewResolver` takes into account any `SitePreferences` when found.

8-5. Site Switching

Problem

Your mobile site is hosted on a different URL than your normal website.

Solution

Use Spring Mobile's site switching support to redirect to the appropriate part of your website.

How It Works

Spring Mobile comes with a `SiteSwitcherHandlerInterceptor` which you can use to switch to a mobile version of your site based on the detected Device. To configure the `SiteSwitcherHandlerInterceptor` there are a couple of factory methods which provide ready to use settings.

Table 8-1. Overview of factory methods on `SiteSwitcherHandlerInterceptor`

Factory Method	Description
mDot	Redirects to a domain starting with <code>m.</code> , For instance <code>http://www.yourdomain.com</code> would redirect to <code>http://m.yourdomain.com</code> .
dotMobi	Redirects to a domain ending with <code>.mobi</code> . A request to <code>http://www.yourdomain.com</code> would redirect to <code>http://www.yourdomain.mobi</code> .
urlPath	Setup different context roots for different devices. Will redirect to the configured URL path for that device. For instance <code>http://www.yourdomain.com</code> could be redirected to <code>http://www.yourdomain.com/mobile</code> .
standard	Most flexible configurable factory method allows to specify a domain to redirect to for the mobile, tablet and normal version of your website.

The SiteSwitcherHandlerInterceptor also provides the ability to use site preferences. When using the SiteSwitcherHandlerInterceptor it isn't needed to register the SitePreferencesHandlerInterceptor anymore as this is already taken care of.

Configuration is as simple as adding it to the list of interceptors you want to apply, the only thing to remember is that you need to place it after the DeviceResolverHandlerInterceptor as the device information is need to calculate the redirect.

```
package com.apress.springrecipes.mobile.web.config;
...
import org.springframework.mobile.device.switcher.SiteSwitcherHandlerInterceptor;

@Configuration
@EnableWebMvc
@ComponentScan("com.apress.springrecipes.mobile.web")
public class MobileConfiguration extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new DeviceResolverHandlerInterceptor());
        registry.addInterceptor(siteSwitcherHandlerInterceptor());
    }

    @Bean
    public SiteSwitcherHandlerInterceptor siteSwitcherHandlerInterceptor() {
        return SiteSwitcherHandlerInterceptor.mDot("yourdomain.com", true);
    }
}
...
```

Notice the bean declaration for the SiteSwitcherHandlerInterceptor the factory method mDot is used to create an instance. The method takes two arguments the first is the base domain name to use and the second is a boolean indicating whether tablets need to be considered a mobile device the default is false. This configuration would lead to redirecting request to the normal website from mobile devices to `m.yourdomain.com`.

```
@Bean
public SiteSwitcherHandlerInterceptor siteSwitcherHandlerInterceptor() {
    return SiteSwitcherHandlerInterceptor.dotMobi("yourdomain.com", true);
}
```

The configuration above uses the dotMobi factory method which takes two arguments. The first is the base domain name to use and the second is a boolean indicating if tables are to be considered as mobile device, the default is false. This would lead to redirecting request on our normal website from mobile devices to be redirected to `yourdomain.mobi`.

```
@Bean
public SiteSwitcherHandlerInterceptor siteSwitcherHandlerInterceptor() {
    return SiteSwitcherHandlerInterceptor.urlPath("/mobile", "/tablet", "/home");
}
```

The configuration above uses the `urlPath` factory method with three arguments. The first argument is the context root for mobile devices, the second the context root for tables. The final argument is the root path or your application. There are two more variations of the `urlPath` factory method: one which only takes only path for mobile devices and another which takes a path for mobile devices and a root path. The configuration above will lead to request from mobile devices to be redirected to `yourdomain.com/home/mobile` and for tables to `yourdomain.com/home/tablet`.

Finally there is the standard factory method, which is the most flexible and elaborate to configure.

```
@Bean
public SiteSwitcherHandlerInterceptor siteSwitcherHandlerInterceptor() {
    return SiteSwitcherHandlerInterceptor
        .standard("yourdomain.com", "mobile.yourdomain.com",
                  "tablet.yourdomain.com", "*.yourdomain.com");
}
```

The configuration above uses the standard factory method. It specifies a different domain for the normal, mobile, and tables versions of the website. Finally it specifies the domain name of the cookie to use for storing the site preferences. This is needed because of the different subdomains specified.

There are several other variations of the standard factory method which allow for a subset of configuration of what is shown above.

Summary

In this chapter you learned how to use Spring Mobile. It can be used to detect the device that is requesting a page or to allow the user to select a certain page based on preferences.

You learned how you can detect the users device using the `DeviceResolverRequestFilter` or `DeviceResolverHandlerInterceptor`. You also learned how you can use `SitePreferences` to allow the user to override the detected Device.

Next you looked at how you can use the device information and preferences to render a view for that device.

Finally you learned how to redirect the user to a different part of your website based on their device or site preferences.

In the next chapter you will explore how to integrate Spring with other frameworks like JSF.

CHAPTER 9



Spring with Other Web Frameworks

In this chapter, you will learn how to integrate the Spring framework with several popular web application frameworks, like JSF. Spring's powerful IoC container and enterprise support features make it very suitable for implementing the service and persistence layers of your Java EE applications. However, for the presentation layer, you have a choice between many different web frameworks. So, you often need to integrate Spring with whatever web application framework you are using. The integration mainly focuses on accessing beans declared in the Spring IoC container within these frameworks.

JavaServer Faces, or JSF (<http://java.sun.com/javaee/javaserverfaces/>), is an excellent component-based and event-driven web application framework included as part of the Java EE specification. You can use the rich set of standard JSF components and also develop custom components for reuse. JSF can cleanly separate presentation logic from UIs by encapsulating it in one or more managed beans. Due to its component-based approach and popularity, JSF is supported by a wide range of IDEs for visual development.

After finishing this chapter, you will be able to integrate Spring into web applications implemented with Servlet/JSP and popular web application frameworks like JSF.

9-1. Accessing Spring in Generic Web Applications

Problem

You would like to access beans declared in the Spring IoC container in a web application, regardless of which framework it uses.

Solution

A web application can load Spring's application context by registering the servlet listener `ContextLoaderListener`. This listener stores the loaded application context into the web application's servlet context. Later, a servlet, or any object that can access the servlet context, can also access Spring's application context through a utility method.

How It Works

Suppose you are going to develop a web application for users to find the distance (measured in kilometers) between two cities. First, you define the following service interface:

```
package com.apress.springrecipes.city;

public interface CityService {
    public double findDistance(String srcCity, String destCity);
}
```

For simplicity's sake, let's implement this interface by using a Java map to store the distance data. This map's keys are source cities while its values are nested maps that contain destination cities and their distances from the source city.

```
package com.apress.springrecipes.city;
...
public class CityServiceImpl implements CityService {

    private Map<String, Map<String, Double>>distanceMap;

    public void setDistanceMap(Map<String, Map<String, Double>>distanceMap) {
        this.distanceMap = distanceMap;
    }

    public double findDistance(String srcCity, String destCity) {
        Map<String, Double> destinationMap = distanceMap.get(srcCity);
        if (destinationMap == null) {
            throw new IllegalArgumentException("Source city not found");
        }
        Double distance = destinationMap.get(destCity);
        if (distance == null) {
            throw new IllegalArgumentException("Destination city not found");
        }
        return distance;
    }
}
```

Next the service needs to be configured by creating a class and annotating it with the `@Configuration` annotation and configure the bean appropriately. It could look like the following:

```
package com.apress.springrecipes.city.config;

import com.apress.springrecipes.city.CityService;
import com.apress.springrecipes.city.CityServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class DistanceConfiguration {

    @Bean
    public CityService cityService() {
        CityServiceImpl cityService = new CityServiceImpl();
        Map<String, Map<String, Double>>distances = new HashMap<>();
        Map<String, Double> newYorkDistances = new HashMap<>();
        newYorkDistances.put("London", 5574.0);
        newYorkDistances.put("Beijing", 10976.0);
        distances.put("New York", newYorkDistances);
        cityService.setDistanceMap(distances);
        return cityService;
    }
}
```

The CityServiceImpl is constructed and a Map is filled with some distance data.

Next you need a servlet to process the distance requests and a page to display the form and the results. When this servlet is accessed with the HTTP GET method, it simply displays the form. Later, when the form is submitted with the POST method, this servlet finds the distance between the two input cities and displays it in the form again.

Note To develop web applications that use the Servlet API, you have to include the Servlet API. If you are using Maven, add the following dependency to your project:

```
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>javax.servlet-api</artifactId>
<version>3.0.1</version>
</dependency>
```

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.CityService;
import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class DistanceServlet extends HttpServlet {

    private CityService cityService;

    @Override
    public void init() throws ServletException {
        WebApplicationContext context;
        context = WebApplicationContextUtils.getRequiredWebApplicationContext(getServletContext());
        cityService = context.getBean(CityService.class);
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        forward(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String srcCity = request.getParameter("srcCity");
        String destCity = request.getParameter("destCity");
```

```

        double distance = cityService.findDistance(srcCity, destCity);
        request.setAttribute("distance", distance);

        forward(request, response);
    }

private void forward(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher dispatcher = request.getRequestDispatcher("WEB-INF/jsp/distance.jsp");
    dispatcher.forward(request, response);
}
}

```

This servlet needs to access the `cityService` bean declared in the Spring IoC container to find distances. As Spring's application context is stored in the servlet context, you can retrieve it through the `WebApplicationContextUtils.getRequiredWebApplicationContext()` method by passing in a servlet context. The dependency lookup is done in the servlets `init` method.

To allow users to query distances between cities, you have to create a JSP file that contains a form. You can name it `distance.jsp` and put it in the `WEB-INF/jsp` directory to prevent direct access to it. There are two text fields in this form for users to input the source and destination cities. There's also a table grid for showing the actual distance.

```

<html>
<head>
<title>City Distance</title>
</head>

<body>
<form method="POST">
<table>
<tr>
    <td>Source City</td>
    <td><input type="text" name="srcCity" value="${param.srcCity}" /></td>
</tr>
<tr>
    <td>Destination City</td>
    <td><input type="text" name="destCity" value="${param.destCity}" /></td>
</tr>
<tr>
    <td>Distance</td>
    <td>${distance}</td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Find" /></td>
</tr>
</table>
</form>
</body>
</html>

```

Finally you need to tie everything together by loading the configuration using a `ContextLoaderListener` and we need to register the `DistanceServlet` and map it to the URL pattern `/distance`. For this create a class that implements the `ServletContainerInitializer` interface. This class is used by the Servlet container (Tomcat for instance) to bootstrap your application. This interface is part of the Servlet specification since version 3.0.

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.config.DistanceConfiguration;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

import javax.servlet.ServletContainerInitializer;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import java.util.Set;

public class DistanceApplicationInitializer implements ServletContainerInitializer {

    @Override
    public void onStartup(Set<Class<?>>c, ServletContext ctx) throws ServletException {
        // Register the ContextLoaderListener
        AnnotationConfigWebApplicationContext appContext =
            new AnnotationConfigWebApplicationContext();
        appContext.register(DistanceConfiguration.class);

        ctx.addListener(new ContextLoaderListener(appContext));

        // Register the Servlet
        ServletRegistration.Dynamic registration = ctx.addServlet("distance",
            new DistanceServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/distance");
    }
}
```

First an instance of the `AnnotationConfigWebApplicationContext` is created and is passed the `DistanceConfiguration` class. Next an instance of the `ContextLoaderListener` is created and handed the earlier created application context. The next part registers the servlet under the name `distance` and is mapped to the `/distance` url pattern. The `loadOnStartup` property makes sure the servlet is started as soon as the application starts.

Finally you need to create a file named `javax.servlet.ServletContainerInitializer` inside the `/META-INF/services` directory, the content of the file is the fully qualified classname of the `DistanceServletContainerInitializer`. The servlet container uses this file to determine which `ServletContainerInitializer` need to be executed.

```
com.apress.springrecipes.city.servlet.DistanceApplicationInitializer
```

Now, you can deploy this web application to a web container (e.g., Apache Tomcat 7.x). By default, Tomcat listens on port 8080, so if you deploy your application to the `city` context path, you can access it with the following URL once it has been started up: `http://localhost:8080/city/distance`.

When using Spring instead of implementing the `ServletContainerInitializer` you can also use the Spring-provided `WebApplicationInitializer` interface. Spring automatically detects the classes implementing this interface and will use them to bootstrap the application. Added benefit is that you don't need the `javax.servlet.ServletContainerInitializer` file anymore.

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.config.DistanceConfiguration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        // Register the ContextLoaderListener
        AnnotationConfigWebApplicationContext appContext =
            new AnnotationConfigWebApplicationContext();
        appContext.register(DistanceConfiguration.class);

        ctx.addListener(new ContextLoaderListener(appContext));

        // Register the Servlet
        ServletRegistration.Dynamic registration = ctx.addServlet("distance",
            new DistanceServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/distance");
    }
}
```

9-2. Using Spring in Your Servlets and Filters

Problem

The servlet specification provides servlets and filters. Servlets handle requests and responses and are responsible ultimately for producing output and acting on the inbound request. Filters are given the opportunity to react to the state of a given request and response before and after the servlet to which a request is destined has processed it. Filters provide the same effect as aspect-oriented programming, which lets you intercept and modify the state of a method invocation. Filters can be added in arbitrary depths to any existing servlet. It is possible, thus, to reuse filters to provide generic functionality to any servlet, such as gzip compression. These artifacts are declared in the `web.xml` file. The servlet container reads the configuration and instantiates the servlets or filters on your behalf and manages the life cycles of these objects inside the container. Because the life cycle is handled by the servlet container, and not by Spring, accessing the services of the Spring container—for example using traditional dependency injection and AOP—proves difficult. You can use `WebApplicationContextUtils` to look up and acquire the dependencies you need, but that defeats the value proposition of dependency injection—your code is required to acquire instances explicitly, which is not much better than using JNDI, for example. It is desirable to let Spring handle dependency injection for you and to let Spring manage the life cycles of your beans.

Solution

If you want to implement filter-like functionality but want to have full access to the Spring context's life cycle machinery and dependency injection, use the `DelegatingFilterProxy` class. Similarly, if you want to implement servlet-like functionality but want to have full access to the Spring context's life cycle machinery and dependency injection, use `HttpRequestHandlerServlet`. These classes are configured normally in `web.xml`, but they then delegate their obligations to a bean that you configure in the Spring application context.

How It Works

Servlets

Let's revisit our previous example. Suppose we wanted to rewrite the servlet functionality to leverage Spring's application context machinery and configuration. The `HttpRequestHandlerServlet` will handle this for us. It uses a little bit of indirection to achieve its work: you configure an instance of `org.springframework.web.context.support.HttpRequestHandlerServlet` and assign it a name. The servlet takes the name as configured and looks up a bean in the root Spring application context. Assuming the bean exists and that it implements the `HttpRequestHandler` interface, the servlet delegates all requests to that bean by invoking the `handleRequest` method.

You must first write a bean that implements the `org.springframework.web.HttpRequestHandler` interface. We will endeavor to replace our existing `DistanceServlet` with a POJO that implements the `HttpRequestHandler` interface. The logic is identical; it's just been reorganized a bit. The POJO's definition is as follows:

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.CityService;
import org.springframework.web.HttpRequestHandler;

import javax.servlet.Dispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class DistanceHttpRequestHandler implements HttpRequestHandler {

    private final CityService cityService;

    public DistanceHttpRequestHandler(CityService cityService) {
        this.cityService = cityService;
    }

    @Override
    public void handleRequest(final HttpServletRequest request, final HttpServletResponse response)
            throws ServletException, IOException {
        if (request.getMethod().toUpperCase().equals("POST")) {
            String srcCity = request.getParameter("srcCity");
            String destCity = request.getParameter("destCity");
            double distance = cityService.findDistance(srcCity, destCity);
            request.setAttribute("distance", distance);
        }
        forward(request, response);
    }
}
```

```

private void forward(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher dispatcher = request.getRequestDispatcher("WEB-INF/jsp/distance.jsp");
    dispatcher.forward(request, response);
}
}

```

Now, we must wire the bean up in the configuration class like so:

```

package com.apress.springrecipes.city.config;
...
import com.apress.springrecipes.city.servlet.DistanceHttpRequestHandler;

@Configuration
public class DistanceConfiguration {

    @Bean
    public CityService cityService() { ... }

    @Bean
    public DistanceHttpRequestHandler distance() {
        return new DistanceHttpRequestHandler(cityService());
    }
}

```

Here, we've configured our bean and injected a reference to the `CityServiceImpl` instance. The bean will be available under the name `distance` as Spring by default used the method name as the bean name. We need this to configure the `HttpRequestHandlerServlet`.

```

package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.config.DistanceConfiguration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.context.support.HttpRequestHandlerServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        // Register the ContextLoaderListener
        AnnotationConfigWebApplicationContext appContext =
            new AnnotationConfigWebApplicationContext();
        appContext.register(DistanceConfiguration.class);

        ctx.addListener(new ContextLoaderListener(appContext));
    }
}

```

```

// Register the Servlet
ServletRegistration.Dynamic registration = ctx.addServlet("distance",
    new HttpRequestHandlerServlet());
registration.setLoadOnStartup(1);
registration.addMapping("/distance");
}
}

```

The use of the bean is identical as in the previous application—all code referencing the /distance endpoint will continue to work from the perspective of a bean. Try it yourself by launching your browser and pointing it to <http://localhost:8080/city/distance?srcCity=New%20York&destCity=London>.

Filters

The Spring framework provides a similar feature for filters. We will demonstrate a suitably simple filter configuration that simply iterates through the inbound request's request attributes and lists them. Here, we will use a collaborating object, of type CityServiceRequestAuditor, whose function it is to enumerate the request parameters (conceivably, such a filter could be used to send the data to syslog, to a monitoring agent like Splunk™ or through JMX). The source code for CityServiceRequestAuditor is as follows:

```

package com.apress.springrecipes.city;

import java.util.Map;

public class CityServiceRequestAuditor {
    public void log(Map<String, String> attributes) {
        for (String k : attributes.keySet()) {
            System.out.println(String.format("%s=%s", k, attributes.get(k)));
        }
    }
}

```

In the servlet example, the HttpRequestHandlerServlet delegated to another object that implemented an interface—HttpRequestHandler—that was considerably simpler than that of a raw servlet. In the javax.servlet.Filter case, however, there is very little that can be done to simplify the interface, so we will instead delegate to an implementation of filter that's been configured using Spring.

Our filter implementation is as follows:

```

package com.apress.springrecipes.city.filter;

import com.apress.springrecipes.city.CityServiceRequestAuditor;

import javax.servlet.*;
import java.io.IOException;
import java.util.Map;

/**
 * This class is designed to intercept requests to the {@link
 * com.apress.springrecipes.city.CityServiceImpl} and log them
 */
public class CityServiceRequestFilter implements Filter {
    private CityServiceRequestAuditor cityServiceRequestAuditor;

```

```

@Override
public void init(final FilterConfig filterConfig) throws ServletException {
}

@Override
public void doFilter(final ServletRequest servletRequest,
                     final ServletResponse servletResponse,
                     final FilterChain filterChain)
throws IOException, ServletException {
    Map parameterMap = servletRequest.getParameterMap();

    this.cityServiceRequestAuditor.log(parameterMap);

    filterChain.doFilter(servletRequest, servletResponse);
}

@Override
public void destroy() {
}

public void setCityServiceRequestAuditor(
    final CityServiceRequestAuditor cityServiceRequestAuditor) {
    this.cityServiceRequestAuditor = cityServiceRequestAuditor;
}
}

```

It has a dependency on a bean of type `CityServiceRequestAuditor`. We will inject it and configure this bean in our Spring application context, below the previous configuration.

```

<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-3.0.xsd">

    ...

    <bean id="cityServiceRequestAuditor"
          class="com.apress.springrecipes.city.CityServiceRequestAuditor" />

    <bean id="cityServiceRequestFilter"
          class="com.apress.springrecipes.city.filter.CityServiceRequestFilter">
        <property name="cityServiceRequestAuditor" ref="cityServiceRequestAuditor" />
    </bean>

</beans>

```

Now, all that remains is to setup the Spring `org.springframework.web.filter.DelegatingFilterProxy` instance in `web.xml`. This configuration maps the filter to the distance servlet we configured earlier.

```

package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.config.DistanceConfiguration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.context.support.HttpRequestHandlerServlet;
import org.springframework.web.filter.DelegatingFilterProxy;

import javax.servlet.FilterRegistration;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        // Register the ContextLoaderListener
        AnnotationConfigWebApplicationContext appContext =
            new AnnotationConfigWebApplicationContext();
        appContext.register(DistanceConfiguration.class);

        ctx.addListener(new ContextLoaderListener(appContext));

        // Register the Servlet
        ServletRegistration.Dynamic registration = ctx.addServlet("distance",
            new HttpRequestHandlerServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/distance");

        //Register the Filter
        FilterRegistration.Dynamic filterReg = ctx.addFilter("cityServiceRequestFilter",
            new DelegatingFilterProxy());
        filterReg.addMappingForServletNames(null, false, "distance");

    }
}

```

Again, note the use of engineered coincidence; the name of the filter, `cityServiceRequestFilter`, is used to determine which bean in the root Spring application context to look up and delegate to. You might notice a bit of redundancy here: the filter interface exposes two methods for life cycle management—`init` and `destroy`—and the Spring context also provides life cycle management for your beans. The default behavior of the

`DelegatingFilterProxy` is to *not* delegate those life cycle methods to your bean, preferring instead to let the Spring life cycle machinery (`InitializingBean`, `@PostConstruct`, etc. for initialization and `DisposableBean`, `@PreDestroy`, etc. for destruction) work instead. If you'd like to have Spring invoke those life cycle methods for you, set the `targetFilterLifecycle` property to true on the filter:

```
public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        ...
        DelegatingFilterProxy delegatingFilter = new DelegatingFilterProxy();
        delegatingFilter.setTargetFilterLifecycle(true);
        FilterRegistration.Dynamic filterReg = ctx.addFilter("cityServiceRequestFilter",
        delegatingFilter);
        filterReg.addMappingForServletNames(null, false, "distance");

    }
}
```

9-3. Integrating Spring with JSF

Problem

You would like to access beans declared in the Spring IoC container in a web application developed with JSF.

Solution

A JSF application is able to access Spring's application context just like a generic web application (i.e., by registering the servlet listener `ContextLoaderListener` and accessing it from the servlet context). However, due to the similarity between Spring's and JSF's bean models, it's very easy to integrate them by registering the Spring-provided JSF variable resolver `DelegatingVariableResolver` (for JSF 1.1) or the `SpringBeanFacesELResolver` (for JSF 1.2 and greater), which can resolve JSF variables into Spring beans. Furthermore, you can even declare JSF managed beans in Spring's bean configuration file to centralize them with your Spring beans.

How It Works

Suppose you are going to implement your web application for finding city distances using JSF. First, you create the following directory structure for your web application.

Note Before you start developing a web application using JSF, you need a JSF implementation library. You can use the JSF Reference Implementation (JSF-RI) or a third-party implementation. If you are using Maven, add the following dependencies to your Maven project:

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
</dependency>

<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-api</artifactId>
    <version>2.2.7</version>
</dependency>

<dependency>
    <groupId>com.sun.faces</groupId>
    <artifactId>jsf-impl</artifactId>
    <version>2.2.7</version>
</dependency>

<dependency>
    <groupId>org.apache.taglibs</groupId>
    <artifactId>taglibs-standard-spec</artifactId>
    <version>1.2.1</version>
</dependency>

<dependency>
    <groupId>org.apache.taglibs</groupId>
    <artifactId>taglibs-standard-impl</artifactId>
    <version>1.2.1</version>
</dependency>
```

As of JSF 2.1 the necessary JSF components are registered by default. The FacesServlet, needed to handle web requests, is configured by default and is mapped to the following patterns /faces/*, *.faces and *.jsf. It is however very common to map it to *.xhtml also. Using our DistanceApplicationInitializer we can add a mapping for this pattern.

To load Spring's application context at startup, you also have to register the servlet listener ContextLoaderListener.

```
package com.apress.springrecipes.city.servlet;

import com.apress.springrecipes.city.config.DistanceConfiguration;
import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.ContextLoaderListener;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
```

```

import javax.servlet.ServletContext;
import javax.servlet.ServletException;

public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        // Register the ContextLoaderListener
        AnnotationConfigWebApplicationContext appContext =
            new AnnotationConfigWebApplicationContext();
        appContext.register(DistanceConfiguration.class);
        ctx.addListener(new ContextLoaderListener(appContext));

        // Configure facelets to use XHTML instead of jsp extension
        ctx.setInitParameter("javax.faces.DEFAULT_SUFFIX", ".xhtml");
        // Map the FacesServlet to *.xhtml
        ctx.getServletRegistration("FacesServlet").addMapping("*.xhtml");
    }
}

```

Notice the init parameter when FacesServlet receives a request, it will map this request to a file with the same name with the configured suffix (default is .jsp). For example, if you request the URL /distance.faces, then FacesServlet will load /distance.jsp accordingly. Instead of .jsp you want to switch it to .xhtml, for that set the javax.faces.DEFAULT_SUFFIX init parameter to .xhtml.

Next the default registered FacesServlet, named FacesServlet, is looked up and a mapping to *.xhtml is added.

The basic idea of JSF is to separate presentation logic from UIs by encapsulating it in one or more JSF managed beans. For your distance-finding function, you can create the following DistanceBean class for a JSF managed bean:

```

package com.apress.springrecipes.city.jsf;

import com.apress.springrecipes.city.CityService;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.RequestScoped;

@ManagedBean
@RequestScoped
public class DistanceBean {

    private String srcCity;
    private String destCity;
    private double distance;

    @ManagedProperty("#{cityService}")
    private CityService cityService;

    public String getSrcCity() {
        return srcCity;
    }
}

```

```

public String getDestCity() {
    return destCity;
}

public double getDistance() {
    return distance;
}

public void setSrcCity(String srcCity) {
    this.srcCity = srcCity;
}

public void setDestCity(String destCity) {
    this.destCity = destCity;
}

public void setCityService(CityService cityService) {
    this.cityService = cityService;
}

public void find() {
    distance = cityService.findDistance(srcCity, destCity);
}
}

```

Due to the @RequestScope annotation the scope of DistanceBean is request, which means a new bean instance will be created on each request. There are four properties defined in this bean. As your page has to show the srcCity, destCity, and distance properties, you define a getter method for each of them. Users can only input the srcCity and destCity properties, so they require a setter method as well. The back-end CityService bean is injected via a setter method, which is mandated when using a @ManagedProperty. When the find() method is called on this bean, it will invoke the back-end service to find the distance between these two cities and then store it in the distance property for subsequent display.

Then, you create distance.xhtml in the root of your web application context. You have to put it here because when FacesServlet receives a request, it will map this request to a file with the same name with the configured suffix (default is .jsp). For example, if you request the URL /distance.faces, then FacesServlet will load /distance.xhtml accordingly. (Remember that you configured the .xhtml suffix in the DistanceApplicationInitializer).

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html" xml:lang="en" lang="en">
<h:head>
    <title>City Distance</title>
</h:head>

<h:body>
<f:view>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputLabel for="srcCity">Source City</h:outputLabel>
            <h:inputText id="srcCity" value="#{distanceBean.srcCity}" />

```

```

<h:outputLabel for="destCity">Destination City</h:outputLabel>
<h:inputText id="destCity" value="#{distanceBean.destCity}"/>
<h:outputLabel>Distance</h:outputLabel>
<h:outputText value="#{distanceBean.distance}"/>
<h:commandButton value="Find" action="#{distanceBean.find()}"/>
</h:panelGrid>
</h:form>
</f:view>
</h:body>
</html>

```

This XHTML file contains an `<h:form>` component for users to input a source city and a destination city. These two fields are defined using two `<h:inputText>` components, whose values are bound to a JSF managed bean's properties. The distance result is defined using an `<h:outputText>` component because its value is read-only. Finally, you define an `<h:commandButton>` component whose action will be triggered on the server side when you click it.

Resolving Spring Beans in JSF

The JSF configuration file `faces-config.xml`, located in the root of WEB-INF, is where you configure your navigation rules and JSF managed beans. For this simple application with only one screen, there's no navigation rule to configure. You can simply configure the preceding `DistanceBean` here. Here is the configuration file we might use for JSF 1.2 and up (note the sample uses JSF 2.2):

```

<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.2" 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-facesconfig_2_2.xsd">

<application>
    <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
</application>

</faces-config>

```

Note that by registering the variable resolver the `SpringBeanFacesELResolver`, you can easily refer to a bean declared in Spring's application context as a JSF variable in the form of `#{{beanName}}`. This variable resolver will first attempt to resolve variables from the original JSF variable resolver. If a variable cannot be resolved, this variable resolver will look up Spring's application context for a bean with the same name.

Now, you can deploy this application to your web container and access it through the URL `http://localhost:8080/city/distance.xhtml`.

Declaring JSF Managed Beans in Spring's Bean Configuration File

By registering `DelegatingVariableResolver`, you can refer to beans declared in Spring from JSF managed beans. However, they are managed by two different containers: JSF's and Spring's. A better solution is to centralize them under the management of Spring's IoC container. Let's remove the managed bean declaration from the JSF configuration file and add the following Spring bean declaration in `applicationContext.xml`:

```

package com.apress.springrecipes.city.config;
...
import com.apress.springrecipes.city.jsf.DistanceBean;
import org.springframework.context.annotation.Scope;

@Configuration
public class DistanceConfiguration {

    @Bean
    public CityService cityService() { ... }

    @Bean
    @Scope("request")
    public DistanceBean distanceBean() {
        DistanceBean distanceBean = new DistanceBean();
        distanceBean.setCityService(cityService());
        return distanceBean;
    }
}

```

To enable the request bean scope in Spring's application context, you have to register `RequestContextListener` in the `DistanceApplicationInitializer`.

```

package com.apress.springrecipes.city.servlet;
...
import org.springframework.web.context.request.RequestContextListener;

public class DistanceApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext ctx) throws ServletException {
        ...
        ctx.addListener(new RequestContextListener());
    }
}

```

Summary

In this chapter, you have learned how to integrate Spring into web applications developed with Servlet/JSP and popular web application frameworks like JSF. The integration mainly focuses on accessing beans declared in the Spring IoC container within these frameworks.

In a generic web application, regardless of which framework it uses, you can register the Spring-provided servlet listener `ContextLoaderListener` to load Spring's application context into the servlet context of this web application. Later, a servlet or any object that can access the servlet context will also be able to access Spring's application context through a utility method.

For JSF, you can register the variable resolver `DelegatingVariableResolver` to resolve JSF variables into Spring beans. Furthermore, you can even declare JSF managed beans in Spring's bean configuration file to centralize them with Spring beans.

CHAPTER 10



Data Access

In chapter, you will learn how Spring can simplify your database access tasks (Spring can also simplify your NoSQL and BigData tasks, which is covered in Chapter 13). Data access is a common requirement for most enterprise applications, which usually require accessing data stored in relational databases. As an essential part of Java SE, Java Database Connectivity (JDBC) defines a set of standard APIs for you to access relational databases in a vendor-independent fashion.

The purpose of JDBC is to provide APIs through which you can execute SQL statements against a database. However, when using JDBC, you have to manage database-related resources by yourself and handle database exceptions explicitly. To make JDBC easier to use, Spring provides an abstraction framework for interfacing with JDBC. As the heart of the Spring JDBC framework, JDBC templates are designed to provide template methods for different types of JDBC operations. Each template method is responsible for controlling the overall process and allows you to override particular tasks of the process.

If raw JDBC doesn't satisfy your requirement or you feel your application would benefit from something slightly higher level, then Spring's support for ORM solutions will interest you. In this chapter, you will also learn how to integrate *object/relational mapping (ORM)* frameworks into your Spring applications. Spring supports most of the popular ORM (or data mapper) frameworks, including Hibernate, JDO, iBATIS, and the Java Persistence API (JPA). Classic TopLink isn't supported starting from Spring 3.0 (the JPA implementation's still supported, of course). However, the JPA support is varied and has support for many implementations of JPA, including the Hibernate and TopLink-based versions. The focus of this chapter will be on Hibernate and JPA. However, Spring's support for ORM frameworks is consistent, so you can easily apply the techniques in this chapter to other ORM frameworks as well.

ORM is a modern technology for persisting objects into a relational database. An ORM framework persists your objects according to the mapping metadata you provide (XML- or annotation-based), such as the mappings between classes and tables, properties and columns, and so on. It generates SQL statements for object persistence at runtime, so you needn't write database-specific SQL statements unless you want to take advantage of database-specific features or provide optimized SQL statements of your own. As a result, your application will be database independent, and it can be easily migrated to another database in the future. Compared to the direct use of JDBC, an ORM framework can significantly reduce the data access effort of your applications.

Hibernate is a popular open-source and high-performance ORM framework in the Java community. Hibernate supports most JDBC-compliant databases and can use specific dialects to access particular databases. Beyond the basic ORM features, Hibernate supports more advanced features such as caching, cascading, and lazy loading. It also defines a querying language called Hibernate Query Language (HQL) for you to write simple but powerful object queries.

JPA defines a set of standard annotations and APIs for object persistence in both the Java SE and Java EE platforms. JPA is defined as part of the EJB 3.0 specification in JSR-220. JPA is just a set of standard APIs that require a JPA-compliant engine to provide persistence services. You can compare JPA with the JDBC API and a JPA engine with a JDBC driver. Hibernate can be configured as a JPA-compliant engine through an extension module called Hibernate EntityManager. This chapter will mainly demonstrate JPA with Hibernate as the underlying engine.

Problems with Direct JDBC

Suppose you are going to develop an application for vehicle registration, whose major functions are the basic create, read, update, and delete (CRUD) operations on vehicle records. These records will be stored in a relational database and accessed with JDBC. First, you design the following `Vehicle` class, which represents a vehicle in Java:

```
package com.apress.springrecipes.vehicle;

public class Vehicle {

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    // Constructors, Getters and Setters
    ...
}
```

Setting Up the Application Database

Before developing your vehicle registration application, you have to set up the database for it. For the sake of low memory consumption and easy configuration, I have chosen Apache Derby (<http://db.apache.org/derby/>) as my database engine. Derby is an open-source relational database engine provided under the Apache License and implemented in pure Java.

Derby can run in either the embedded mode or the client/server mode. For testing purposes, the client/server mode is more appropriate because it allows you to inspect and edit data with any visual database tools that support JDBC—for example, the Eclipse Data Tools Platform (DTP).

Note To start the Derby server in the client/server mode, just execute the `startNetworkServer` script for your platform (located in the `bin` directory of the Derby installation).

After starting up the Derby network server on localhost, you can connect to it with the JDBC properties shown in Table 10-1.

Table 10-1. JDBC Properties for Connecting to the Application Database

Property	Value
Driver class	<code>org.apache.derby.jdbc.ClientDriver</code>
URL	<code>jdbc:derby://localhost:1527/vehicle;create=true</code>
Username	<code>app</code>
Password	<code>app</code>

Note You require Derby's client JDBC driver. If you are using Maven, add the following dependency to your project.

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.10.1.0</version>
</dependency>
```

The first time you connect to this database, the database instance `vehicle` will be created, if it did not exist before, because you specified `create=true` in the URL. Note that the specification of this parameter will not cause the re-creation of the database if it already exists.

Follow these steps to connect to Derby:

1. Open a shell on your platform.
2. Type `java -jar $DERBY_HOME/lib/derbyrun.jar ij` on Unix variants or `%DERBY_HOME%/lib/derbyrun.jar ij` on Windows.
3. Issue the command `CONNECT 'jdbc:derby://localhost:1527/vehicle;create=true';`.

You can provide any values for the username and password because Derby disables authentication by default. Next, you have to create the `VEHICLE` table for storing vehicle records with the following SQL statement. By default, this table will be created in the APP database `sAPP` database schema.

```
CREATE TABLE VEHICLE (
  VEHICLE_NO      VARCHAR(10)    NOT NULL,
  COLOR           VARCHAR(10),
  WHEEL           INT,
  SEAT            INT,
  PRIMARY KEY (VEHICLE_NO)
);
```

Understanding the Data Access Object Design Pattern

A typical design mistake made by inexperienced developers is to mix different types of logic (e.g., presentation logic, business logic, and data access logic) in a single large module. This reduces the module's reusability and maintainability because of the tight coupling it introduces. The general purpose of the Data Access Object (DAO) pattern is to avoid these problems by separating data access logic from business logic and presentation logic. This pattern recommends that data access logic be encapsulated in independent modules called data access objects.

For your vehicle registration application, you can abstract the data access operations to insert, update, delete, and query a vehicle. These operations should be declared in a DAO interface to allow for different DAO implementation technologies.

```
package com.apress.springrecipes.vehicle;

public interface VehicleDao {
    public void insert(Vehicle vehicle);
    public void update(Vehicle vehicle);
    public void delete(Vehicle vehicle);
    public Vehicle findByVehicleNo(String vehicleNo);
}
```

Most parts of the JDBC APIs declare throwing `java.sql.SQLException`. But because this interface aims to abstract the data access operations only, it should not depend on the implementation technology. So, it's unwise for this general interface to declare throwing the JDBC-specific `SQLException`. A common practice when implementing a DAO interface is to wrap this kind of exception with a runtime exception (either your own business `Exception` subclass or a generic one).

Implementing the DAO with JDBC

To access the database with JDBC, you create an implementation for this DAO interface (e.g., `JdbcVehicleDao`). Because your DAO implementation has to connect to the database to execute SQL statements, you may establish database connections by specifying the driver class name, database URL, username, and password. However, in JDBC 2.0 or higher, you can obtain database connections from a preconfigured `javax.sql.DataSource` object without knowing about the connection details.

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcVehicleDao implements VehicleDao {

    private DataSource dataSource;

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

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicle.getVehicleNo());
            ps.setString(2, vehicle.getColor());
            ps.setInt(3, vehicle.getWheel());
            ps.setInt(4, vehicle.getSeat());
            ps.executeUpdate();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
```

```

        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }

public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    Connection conn = null;
    try {
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicleNo);

        Vehicle vehicle = null;
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            vehicle = new Vehicle(rs.getString("VEHICLE_NO"),
                rs.getString("COLOR"), rs.getInt("WHEEL"),
                rs.getInt("SEAT"));
        }
        rs.close();
        ps.close();
        return vehicle;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}

public void update(Vehicle vehicle) {/* ... */}

public void delete(Vehicle vehicle) {/* ... */}
}

```

The vehicle insert operation is a typical JDBC update scenario. Each time this method is called, you obtain a connection from the data source and execute the SQL statement on this connection. Your DAO interface doesn't declare throwing any checked exceptions, so if a SQLException occurs, you have to wrap it with an unchecked RuntimeException. (There is a detailed discussion on handling exceptions in your DAOs later in this chapter). Don't forget to release the connection in the finally block. Failing to do so may cause your application to run out of connections.

Here, the update and delete operations will be skipped, because they are much the same as the insert operation from a technical point of view. For the query operation, you have to extract the data from the returned result set to build a vehicle object in addition to executing the SQL statement.

Configuring a Data Source in Spring

The `javax.sql.DataSource` interface is a standard interface defined by the JDBC specification that factories `Connection` instances. There are many data source implementations provided by different vendors and projects: C3PO and Apache Commons DBCP are popular open source options, and most application servers will provide their own implementation. It is very easy to switch between different data source implementations, because they implement the common `DataSource` interface. As a Java application framework, Spring also provides several convenient but less powerful data source implementations. The simplest one is `DriverManagerDataSource`, which opens a new connection every time one is requested.

```
package com.apress.springrecipes.vehicle.config;

import com.apress.springrecipes.vehicle.JdbcVehicleDao;
import com.apress.springrecipes.vehicle.VehicleDao;
import org.apache.derby.jdbc.ClientDriver;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration
public class VehicleConfiguration {

    @Bean
    public VehicleDao vehicleDao() {
        return new JdbcVehicleDao(dataSource());
    }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(ClientDriver.class.getName());
        dataSource.setUrl("jdbc:derby://localhost:1527/vehicle;create=true");
        dataSource.setUsername("app");
        dataSource.setPassword("app");
        return dataSource;
    }
}
```

`DriverManagerDataSource` is not an efficient data source implementation because it opens a new connection for the client every time it's requested. Another data source implementation provided by Spring is `SingleConnectionDataSource` (a `DriverManagerDataSource` subclass). As its name indicates, this maintains only a single connection that's reused all the time and never closed. Obviously, it is not suitable in a multithreaded environment.

Spring's own data source implementations are mainly used for testing purposes. However, many production data source implementations support connection pooling. For example, the Database Connection Pooling Services (DBCP) module of the Apache Commons Library has several data source implementations that support connection pooling. Of these, `BasicDataSource` accepts the same connection properties as `DriverManagerDataSource` and allows you to specify the initial connection size and maximum active connections for the connection pool.

```

@Bean
public DataSource dataSource() {
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName(ClientDriver.class.getName());
    dataSource.setUrl("jdbc:derby://localhost:1527/vehicle;create=true");
    dataSource.setUsername("app");
    dataSource.setPassword("app");
    dataSource.setInitialSize(2);
    dataSource.setMaxTotal(5);
    return dataSource;
}

```

Note To use the data source implementations provided by DBCP, you have to add them to your CLASSPATH. If you are using Maven, add the following dependency to your project:

```

<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.0</version>
</dependency>

```

Many Java EE application servers build in data source implementations that you can configure from the server console or in configuration files. If you have a data source configured in an application server and exposed for JNDI lookup, you can use `JndiDataSourceLookup` to look it up.

```

@Bean
public DataSource dataSource() {
    return new JndiDataSourceLookup().getDataSource("jdbc/VehicleDS");
}

```

Running the DAO

The following `Main` class tests your DAO by using it to insert a new vehicle to the database. If it succeeds, you can query the vehicle from the database immediately.

```

package com.apress.springrecipes.vehicle;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(VehicleConfiguration.class);

        VehicleDao vehicleDao = context.getBean(VehicleDao.class);
        Vehicle vehicle = new Vehicle("TEM0001", "Red", 4, 4);
        vehicleDao.insert(vehicle);
    }
}

```

```

        vehicle = vehicleDao.findByVehicleNo("TEM0001");
        System.out.println("Vehicle No: " + vehicle.getVehicleNo());
        System.out.println("Color: " + vehicle.getColor());
        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}

```

Now you can implement a DAO using JDBC directly. However, as you can see from the preceding DAO implementation, most of the JDBC code is similar and needs to be repeated for each database operation. Such redundant code will make your DAO methods much longer and less readable.

Taking It a Step Further

An alternative approach is to use an ORM (an object/relational mapping) tool, which lets you code the logic specifically for mapping an entity in your domain model to a database table. The ORM will, in turn, figure out how to write the logic to usefully persist your class's data to the database. This can be very liberating: you are suddenly beholden only to your business and domain model, not to whims of your database's SQL parser. The flip side, of course, is that you are also divesting yourself from the complete control over the communication between your client and the database—you have to trust that the ORM layer will do the right thing.

10-1. Using a JDBC Template to Update a Database

Problem

Using JDBC is tedious and fraught with redundant API calls, many of which could be managed for you. To implement a JDBC update operation, you have to perform the following tasks, most of which are redundant:

1. Obtain a database connection from the data source.
2. Create a `PreparedStatement` object from the connection.
3. Bind the parameters to the `PreparedStatement` object.
4. Execute the `PreparedStatement` object.
5. Handle `SQLException`.
6. Clean up the statement object and connection.

JDBC is a very low-level API, but with the JDBC template, the surface area of the API that you need to work with becomes more expressive (you spend less time in the weeds and more time working on your application logic) and is simpler to work with safely.

Solution

The `org.springframework.jdbc.core.JdbcTemplate` class declares a number of overloaded `update()` template methods to control the overall update process. Different versions of the `update()` method allow you to override different task subsets of the default process. The Spring JDBC framework predefines several callback interfaces to encapsulate different task subsets. You can implement one of these callback interfaces and pass its instance to the corresponding `update()` method to complete the process.

How It Works

Updating a Database with a Statement Creator

The first callback interface to introduce is `PreparedStatementCreator`. You implement this interface to override the statement creation task (task 2) and the parameter binding task (task 3) of the overall update process. To insert a vehicle into the database, you implement the `PreparedStatementCreator` interface as follows:

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.springframework.jdbc.core.PreparedStatementCreator;

public class InsertVehicleStatementCreator implements PreparedStatementCreator {

    private Vehicle vehicle;

    public InsertVehicleStatementCreator(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) VALUES (?, ?, ?, ?)";
        PreparedStatement ps = con.prepareStatement(sql);
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
    }
}
```

When implementing the `PreparedStatementCreator` interface, you will get the database connection as the `createPreparedStatement()` method's argument. All you have to do in this method is to create a `PreparedStatement` object on this connection and bind your parameters to this object. Finally, you have to return the `PreparedStatement` object as the method's return value. Notice that the method signature declares throwing `SQLException`, which means that you don't need to handle this kind of exception yourself.

Now, you can use this statement creator to simplify the vehicle insert operation. First of all, you have to create an instance of the `JdbcTemplate` class and pass in the data source for this template to obtain a connection from it. Then, you just make a call to the `update()` method and pass in your statement creator for the template to complete the update process.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(Vehicle vehicle) {
```

```

        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));
    }
}

```

Typically, it is better to implement the `PreparedStatementCreator` interface and other callback interfaces as inner classes if they are used within one method only. This is because you can get access to the local variables and method arguments directly from the inner class, instead of passing them as constructor arguments. The only constraint on such variables and arguments is that they must be declared as `final`.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(new PreparedStatementCreator() {

            public PreparedStatement createPreparedStatement(Connection conn)
                throws SQLException {
                String sql = "INSERT INTO VEHICLE "
                    + "(VEHICLE_NO, COLOR, WHEEL, SEAT) "
                    + "VALUES (?, ?, ?, ?)";
                PreparedStatement ps = conn.prepareStatement(sql);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
                return ps;
            }
        });
    }
}

```

Now, you can delete the preceding `InsertVehicleStatementCreator` class, because it will not be used anymore.

Updating a Database with a Statement Setter

The second callback interface, `PreparedStatementSetter`, as its name indicates, performs only the parameter binding task (task 3) of the overall update process.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;

```

```

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new PreparedStatementSetter() {

            public void setValues(PreparedStatement ps)
                throws SQLException {
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
            }
        });
    }
}

```

Another version of the `update()` template method accepts a SQL statement and a `PreparedStatementSetter` object as arguments. This method will create a `PreparedStatement` object for you from your SQL statement. All you have to do with this interface is to bind your parameters to the `PreparedStatement` object.

Updating a Database with a SQL Statement and Parameter Values

Finally, the simplest version of the `update()` method accepts a SQL statement and an object array as statement parameters. It will create a `PreparedStatement` object from your SQL statement and bind the parameters for you. Therefore, you don't have to override any of the tasks in the update process.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, vehicle.getVehicleNo(), vehicle.getColor(), vehicle.getWheel(),
        vehicle.getSeat());
    }
}

```

Of the three different versions of the `update()` method introduced, the last is the simplest because you don't have to implement any callback interfaces. Additionally, we've managed to remove all `setX` (`setInt`, `setString`, etc.)-style methods for parameterizing the query. In contrast, the first is the most flexible because you can do any preprocessing of the `PreparedStatement` object before its execution. In practice, you should always choose the simplest version that meets all your needs.

There are also other overloaded `update()` methods provided by the `JdbcTemplate` class. Please refer to Javadoc for details.

Batch Updating a Database

Suppose you want to insert a batch of vehicles into the database. If you call the `insert()` method multiple times, the update will be very slow as the SQL statement will be compiled repeatedly. So, it would be better to add a new method to the DAO interface for inserting a batch of vehicles.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles);
}
```

The `JdbcTemplate` class also offers the `batchUpdate()` template method for batch update operations. It requires a SQL statement and a `BatchPreparedStatementSetter` object as arguments. In this method, the statement is compiled (prepared) only once and executed multiple times. If your database driver supports JDBC 2.0, this method automatically makes use of the batch update features to increase performance.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insertBatch(final List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {
            ...
            public int getBatchSize() {
                return vehicles.size();
            }

            public void setValues(PreparedStatement ps, int i)
                    throws SQLException {
                Vehicle vehicle = vehicles.get(i);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
            }
        });
    }
}
```

You can test your batch insert operation with the following code snippet in the Main class:

```
package com.apress.springrecipes.vehicle;
...
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle1 = new Vehicle("TEM0002", "Blue", 4, 4);
        Vehicle vehicle2 = new Vehicle("TEM0003", "Black", 4, 6);
        vehicleDao.insertBatch(Arrays.asList(vehicle1, vehicle2));
    }
}
```

10-2. Using a JDBC Template to Query a Database

Problem

To implement a JDBC query operation, you have to perform the following tasks, two of which (tasks 5 and 6) are additional as compared to an update operation:

1. Obtain a database connection from the data source.
2. Create a `PreparedStatement` object from the connection.
3. Bind the parameters to the `PreparedStatement` object.
4. Execute the `PreparedStatement` object.
5. Iterate the returned result set.
6. Extract data from the result set.
7. Handle `SQLException`.
8. Clean up the statement object and connection.

The only steps relevant to your business logic, however, are the definition of the query and the extraction of the results from the result set! The rest is better handled by the JDBC template.

Solution

The `JdbcTemplate` class declares a number of overloaded `query()` template methods to control the overall query process. You can override the statement creation (task 2) and the parameter binding (task 3) by implementing the `PreparedStatementCreator` and `PreparedStatementSetter` interfaces, just as you did for the update operations. Moreover, the Spring JDBC framework supports multiple ways for you to override the data extraction (task 6).

How It Works

Extracting Data with Row Callback Handler

`RowCallbackHandler` is the primary interface that allows you to process the current row of the result set. One of the `query()` methods iterates the result set for you and calls your `RowCallbackHandler` for each row. So, the `processRow()` method will be called once for each row of the returned result set.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        final Vehicle vehicle = new Vehicle();
        jdbcTemplate.query(sql,
            new RowCallbackHandler() {
                public void processRow(ResultSet rs) throws SQLException {
                    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
                    vehicle.setColor(rs.getString("COLOR"));
                    vehicle.setWheel(rs.getInt("WHEEL"));
                    vehicle.setSeat(rs.getInt("SEAT"));
                }
            }, vehicleNo);
        return vehicle;
    }
}
```

As there will be one row returned for the SQL query at maximum, you can create a vehicle object as a local variable and set its properties by extracting data from the result set. For a result set with more than one row, you should collect the objects as a list.

Extracting Data with a Row Mapper

The `RowMapper<T>` interface is more general than `RowCallbackHandler`. Its purpose is to map a single row of the result set to a customized object, so it can be applied to a single-row result set as well as a multiple-row result set. From the viewpoint of reuse, it's better to implement the `RowMapper<T>` interface as a normal class than as an inner class. In the `mapRow()` method of this interface, you have to construct the object that represents a row and return it as the method's return value.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;
```

```
public class VehicleRowMapper implements RowMapper<Vehicle> {

    public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

As mentioned, `RowMapper<T>` can be used for either a single-row or multiple-row result set. When querying for a unique object like in `findByVehicleNo()`, you have to make a call to the `queryForObject()` method of `JdbcTemplate`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        return jdbcTemplate.queryForObject(sql, new VehicleRowMapper(), vehicleNo);
    }
}
```

Spring comes with a convenient `RowMapper<T>` implementation, `BeanPropertyRowMapper<T>`, which can automatically map a row to a new instance of the specified class. Note that the specified class must be a top-level class and must have a default or no-argument constructor. It first instantiates this class and then maps each column value to a property by matching their names. It supports matching a property name (e.g., `vehicleNo`) to the same column name or the column name with underscores (e.g., `VEHICLE_NO`).

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        return jdbcTemplate.queryForObject(sql, BeanPropertyRowMapper.newInstance(Vehicle.class),
            vehicleNo);
    }
}
```

Querying for Multiple Rows

Now, let's look at how to query for a result set with multiple rows. For example, suppose that you need a `findAll()` method in the DAO interface to get all vehicles.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public List<Vehicle> findAll();
}
```

Without the help of `RowMapper<T>`, you can still call the `queryForList()` method and pass in a SQL statement. The returned result will be a list of maps. Each map stores a row of the result set with the column names as the keys.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        List<Map<String, Object>> rows = jdbcTemplate.queryForList(sql);
        for (Map<String, Object> row : rows) {
            Vehicle vehicle = new Vehicle();
            vehicle.setVehicleNo((String) row.get("VEHICLE_NO"));
            vehicle.setColor((String) row.get("COLOR"));
            vehicle.setWheel((Integer) row.get("WHEEL"));
            vehicle.setSeat((Integer) row.get("SEAT"));
            vehicles.add(vehicle);
        }
        return vehicles;
    }
}
```

You can test your `findAll()` method with the following code snippet in the Main class:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        List<Vehicle> vehicles = vehicleDao.findAll();
        for (Vehicle vehicle : vehicles) {
            System.out.println("Vehicle No: " + vehicle.getVehicleNo());
            System.out.println("Color: " + vehicle.getColor());
```

```
        System.out.println("Wheel: " + vehicle.getWheel());
        System.out.println("Seat: " + vehicle.getSeat());
    }
}
```

If you use a `RowMapper<T>` object to map the rows in a result set, you will get a list of mapped objects from the `query()` method.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        return jdbcTemplate.query(sql, BeanPropertyRowMapper.newInstance(Vehicle.class), vehicleNo);
    }
}
```

Querying for a Single Value

Finally, let's consider to query for a single-row and single-column result set. As an example, add the following operations to the DAO interface:

```
package com.apress.springrecipes.vehicle;  
...  
public interface VehicleDao {  
    ...  
    public String getColor(String vehicleNo);  
    public int countAll();  
}
```

To query for a single string value, you can call the overloaded `queryForObject()` method, which requires an argument of `java.lang.Class` type. This method will help you to map the result value to the type you specified.

```
package com.apress.springrecipes.vehicle;  
...  
import org.springframework.jdbc.core.JdbcTemplate;  
  
public class JdbcVehicleDao implements VehicleDao {  
    ...  
    public String getColor(String vehicleNo) {  
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";  
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
  
        return jdbcTemplate.queryForObject(sql, String.class, vehicleNo);  
    }  
}
```

```

public int countAll() {
    String sql = "SELECT COUNT(*) FROM VEHICLE";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    return jdbcTemplate.queryForObject(sql, Integer.class);
}
}

```

Note Earlier versions of Spring provided the `queryForInt` and `queryForLong` methods, as of Spring 3.2 those methods are deprecated in favor of the `queryForObject` method.

You can test these two methods with the following code snippet in the `Main` class:

```

package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = context.getBean(VehicleDao.class);
        int count = vehicleDao.countAll();
        System.out.println("Vehicle Count: " + count);
        String color = vehicleDao.getColor("TEM0001");
        System.out.println("Color for [TEM0001]: " + color);
    }
}

```

10-3. Simplifying JDBC Template Creation

Problem

It's not efficient to create a new instance of `JdbcTemplate` every time you use it, because you have to repeat the creation statement and incur the cost of creating a new object.

Solution

The `JdbcTemplate` class is designed to be thread-safe, so you can declare a single instance of it in the IoC container and inject this instance into all your DAO instances. Furthermore, the Spring JDBC framework offers a convenient class, `org.springframework.jdbc.core.support.JdbcDaoSupport`, to simplify your DAO implementation. This class declares a `jdbcTemplate` property, which can be injected from the IoC container or created automatically from a data source, for example, `JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)`. Your DAO can extend this class to have this property inherited.

How It Works

Injecting a JDBC Template

Until now, you have created a new instance of `JdbcTemplate` in each DAO method. Actually, you can have it injected at the class level and use this injected instance in all DAO methods. For simplicity's sake, the following code shows only the change to the `insert()` method:

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

    private final JdbcTemplate jdbcTemplate;

    public JdbcVehicleDao (JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    ...
}
```

A JDBC template requires a data source to be set. You can inject this property by either a setter method or a constructor argument. Then, you can inject this JDBC template into your DAO.

```
@Configuration
public class VehicleConfiguration {
    ...
    @Bean
    public VehicleDao vehicleDao() {
        return new JdbcVehicleDao(jdbcTemplate());
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}
```

Extending the JdbcDaoSupport Class

The `org.springframework.jdbc.core.support.JdbcDaoSupport` class has a `setDataSource()` method and a `setJdbcTemplate()` method. Your DAO class can extend this class to have these methods inherited. Then, you can either inject a JDBC template directly or inject a data source for it to create a JDBC template. The following code fragment is taken from Spring's `JdbcDaoSupport` class:

```
package org.springframework.jdbc.core.support;
...
public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;

    public final void setDataSource(DataSource dataSource) {
        if( this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource() ){
            this.jdbcTemplate = createJdbcTemplate(dataSource);
            initTemplateConfig();
        }
    }
    ...
    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
    }
    ...
}
```

In your DAO methods, you can simply call the `getJdbcTemplate()` method to retrieve the JDBC template. You also have to delete the `dataSource` and `jdbcTemplate` properties, as well as their setter methods, from your DAO class, because they have already been inherited. Again, for simplicity's sake, only the change to the `insert()` method is shown.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcVehicleDao extends JdbcDaoSupport implements VehicleDao {

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        getJdbcTemplate().update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    ...
}
```

By extending `JdbcDaoSupport`, your DAO class inherits the `setDataSource()` method. You can inject a data source into your DAO instance for it to create a JDBC template.

```
@Configuration
public class VehicleConfiguration {
    ...
    @Bean
    public VehicleDao vehicleDao() {
        JdbcVehicleDao vehicleDao = new JdbcVehicleDao();
        vehicleDao.setDataSource(dataSource());
        return vehicleDao;
    }
}
```

10-4. Using Named Parameters in a JDBC Template

Problem

In classic JDBC usage, SQL parameters are represented by the placeholder `?` and are bound by position. The trouble with positional parameters is that whenever the parameter order is changed, you have to change the parameter bindings as well. For a SQL statement with many parameters, it is very cumbersome to match the parameters by position.

Solution

Another option when binding SQL parameters in the Spring JDBC framework is to use named parameters. As the term implies, named SQL parameters are specified by name (starting with a colon) rather than by position. Named parameters are easier to maintain and also improve readability. At runtime, the framework classes replace named parameters with placeholders. Named parameters are supported by the `NamedParameterJdbcTemplate`.

How It Works

When using named parameters in your SQL statement, you can provide the parameter values in a map with the parameter names as the keys.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;

public class JdbcVehicleDao extends NamedParameterJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("vehicleNo", vehicle.getVehicleNo());
        parameters.put("color", vehicle.getColor());
        parameters.put("wheel", vehicle.getWheel());
        parameters.put("seat", vehicle.getSeat());
    }
}
```

```

        getNamedParameterJdbcTemplate().update(sql, parameters);
    }
    ...
}

```

You can also provide a SQL parameter source, whose responsibility is to offer SQL parameter values for named SQL parameters. There are three implementations of the `SqlParameterSource` interface. The basic one is `MapSqlParameterSource`, which wraps a map as its parameter source. In this example, this is a net-loss compared to the previous example, as we've introduced one extra object—the `SqlParameterSource`:

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;

public class JdbcVehicleDao extends NamedParameterJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        ...
        SqlParameterSource parameterSource =
            new MapSqlParameterSource(parameters);

        getNamedParameterJdbcTemplate().update(sql, parameterSource);
    }
    ...
}

```

The power comes when we need an extra level of indirection between the parameters passed into the update method and the source of their values. For example, what if we want to get properties from a JavaBean? Here is where the `SqlParameterSource` intermediary starts to benefit us! `SqlParameterSource` is `BeanPropertySqlParameterSource`, which wraps a normal Java object as a SQL parameter source. For each of the named parameters, the property with the same name will be used as the parameter value.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;

public class JdbcVehicleDao extends NamedParameterJdbcDaoSupport implements
    VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";
    }
}

```

```

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(vehicle);

        getNamedParameterJdbcTemplate ().update(sql, parameterSource);
    }
    ...
}

```

Named parameters can also be used in batch update. You can provide either a Map array or a SqlParameterSource array for the parameter values.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcDaoSupport;

public class JdbcVehicleDao extends NamedParameterJdbcDaoSupport implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        List<SqlParameterSource> parameters = new ArrayList<SqlParameterSource>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new BeanPropertySqlParameterSource(vehicle));
        }

        getNamedParameterJdbcTemplate ().batchUpdate(sql,
            parameters.toArray(new SqlParameterSource[parameters.size()])));
    }
}

```

10-5. Handling Exceptions in the Spring JDBC Framework

Problem

Many of the JDBC APIs declare throwing `java.sql.SQLException`, a checked exception that must be caught. It's very troublesome to handle this kind of exception every time you perform a database operation. You often have to define your own policy to handle this kind of exception. Failure to do so may lead to inconsistent exception handling.

Solution

The Spring framework offers a consistent data access exception-handling mechanism for its data access module, including the JDBC framework. In general, all exceptions thrown by the Spring JDBC framework are subclasses of `org.springframework.dao.DataAccessException`, a type of `RuntimeException` that you are not forced to catch. It's the root exception class for all exceptions in Spring's data access module.

Figure 10-1 shows only part of the `DataAccessException` hierarchy in Spring's data access module. In total, there are more than 30 exception classes defined for different categories of data access exceptions.

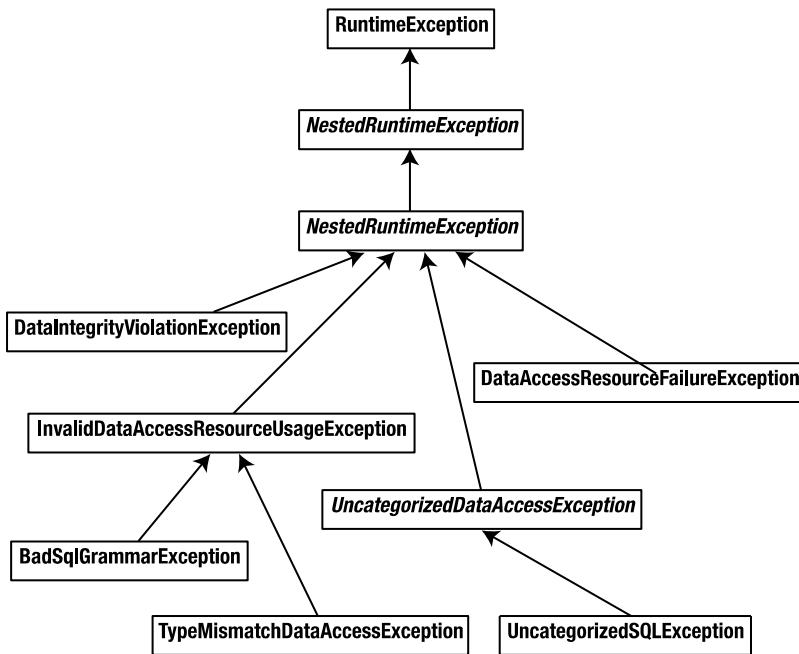


Figure 10-1. Common exception classes in the `DataAccessException` hierarchy

How It Works

Understanding Exception Handling in the Spring JDBC Framework

Until now, you haven't handled JDBC exceptions explicitly when using a JDBC template or JDBC operation objects. To help you understand the Spring JDBC framework's exception-handling mechanism, let's consider the following code fragment in the `Main` class, which inserts a vehicle. What happens if you insert a vehicle with a duplicate vehicle number?

```

package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = context.getBean(VehicleDao.class);
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        vehicleDao.insert(vehicle);
    }
}
  
```

If you run the method twice, or the vehicle has already been inserted into the database, it will throw a `DuplicateKeyException`, an indirect subclass of `DataAccessException`. In your DAO methods, you neither need to surround the code with a try/catch block nor declare throwing an exception in the method signature. This is because `DataAccessException` (and therefore its subclasses, including `DuplicateKeyException`) is an unchecked exception

that you are not forced to catch. The direct parent class of `DataAccessException` is `NestedRuntimeException`, a core Spring exception class that wraps another exception in a `RuntimeException`.

When you use the classes of the Spring JDBC framework, they will catch `SQLException` for you and wrap it with one of the subclasses of `DataAccessException`. As this exception is a `RuntimeException`, you are not required to catch it.

But how does the Spring JDBC framework know which concrete exception in the `DataAccessException` hierarchy should be thrown? It's by looking at the `errorCode` and `SQLState` properties of the caught `SQLException`. As a `DataAccessException` wraps the underlying `SQLException` as the root cause, you can inspect the `errorCode` and `SQLState` properties with the following catch block:

```
package com.apress.springrecipes.vehicle;
...
import java.sql.SQLException;
import org.springframework.dao.DataAccessException;
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = context.getBean(VehicleDao.class);
        Vehicle vehicle = new Vehicle("EX0001", "Green", 4, 4);
        try {
            vehicleDao.insert(vehicle);
        } catch (DataAccessException e) {
            SQLException sqle = (SQLException) e.getCause();
            System.out.println("Error code: " + sqle.getErrorCode());
            System.out.println("SQL state: " + sqle.getSQLState());
        }
    }
}
```

When you insert the duplicate vehicle again, notice that Apache Derby returns the following error code and SQL state:

```
Error code : -1
SQL state : 23505
```

If you refer to the Apache Derby reference manual, you will find the error code description shown in Table 10-2.

Table 10-2. Apache Derby's Error Code Description

SQL State	Message Text
23505	The statement was aborted because it would have caused a duplicate key value in a unique or primary key constraint or unique index identified by '<value>' defined on '<value>'.

How does the Spring JDBC framework know that state 23505 should be mapped to `DuplicateKeyException`? The error code and SQL state are database specific, which means different database products may return different codes for the same kind of error. Moreover, some database products will specify the error in the `errorCode` property, while others (like Derby) will do so in the `SQLState` property.

As an open Java application framework, Spring understands the error codes of most popular database products. Because of the large number of error codes, however, it can only maintain mappings for the most frequently encountered errors. The mapping is defined in the `sql-error-codes.xml` file, located in the `org.springframework.jdbc.support` package. The following snippet for Apache Derby is taken from this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 3.0//EN"
  "http://www.springframework.org/dtd/spring-beans-3.0.dtd">

<beans>
  ...
  <bean id="Derby" class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="databaseProductName">
      <value>Apache Derby</value>
    </property>
    <property name="useSqlStateForTranslation">
      <value>true</value>
    </property>
    <property name="badSqlGrammarCodes">
      <value>42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,42X07,42X08</value>
    </property>
    <property name="duplicateKeyCodes">
      <value>23505</value>
    </property>
    <property name="dataIntegrityViolationCodes">
      <value>22001,22005,23502,23503,23513,X0Y32</value>
    </property>
    <property name="dataAccessResourceFailureCodes">
      <value>04501,08004,42Y07</value>
    </property>
    <property name="cannotAcquireLockCodes">
      <value>40XL1</value>
    </property>
    <property name="deadlockLoserCodes">
      <value>40001</value>
    </property>
  </bean>
</beans>
```

Note that the `databaseProductName` property is used to match the database product name returned by `Connection.getMetaData().getDatabaseProductName()`. This enables Spring to know which type of database is currently connecting. The `useSqlStateForTranslation` property means that the `SQLState` property, rather than the `errorCode` property, should be used to match the error code. Finally, the `SQLErrorCodes` class defines several categories for you to map database error codes. The code 23505 lies in the `dataIntegrityViolationCodes` category.

Customizing Data Access Exception Handling

The Spring JDBC framework only maps well-known error codes. Sometimes, you may wish to customize the mapping yourself. For example, you might decide to add more codes to an existing category or define a custom exception for particular error codes.

In Table 10-2, the error code 23505 indicates a duplicate key error in Apache Derby. It is mapped by default to `DataIntegrityViolationException`. Suppose that you want to create a custom exception type, `MyDuplicateKeyException`, for this kind of error. It should extend `DataIntegrityViolationException` because it is also a kind of data integrity violation error. Remember that for an exception to be thrown by the Spring JDBC framework, it must be compatible with the root exception class `DataAccessException`.

```
package com.apress.springrecipes.vehicle;

import org.springframework.dao.DataIntegrityViolationException;

public class MyDuplicateKeyException extends DataIntegrityViolationException {

    public MyDuplicateKeyException(String msg) {
        super(msg);
    }

    public MyDuplicateKeyException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

By default, Spring will look up an exception from the `sql-error-codes.xml` file located in the `org.springframework.jdbc.support` package. However, you can override some of the mappings by providing a file with the same name in the root of the classpath. If Spring can find your custom file, it will look up an exception from your mapping first. However, if it does not find a suitable exception there, Spring will look up the default mapping.

For example, suppose that you want to map your custom `DuplicateKeyException` type to error code 23505. You have to add the binding via a `CustomSQLExceptionCodesTranslation` bean, and then add this bean to the `customTranslations` category.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <bean id="Derby"
          class="org.springframework.jdbc.support.SQLExceptionCodes">
        <property name="databaseProductName">
            <value>Apache Derby</value>
        </property>
        <property name="useSqlStateForTranslation">
            <value>true</value>
        </property>
        <property name="customTranslations">
            <list>
                <ref local="myDuplicateKeyTranslation" />
            </list>
        </property>
    </bean>
```

```

<bean id="myDuplicateKeyTranslation"
    class="org.springframework.jdbc.support.CustomSQLExceptionCodesTranslation">
    <property name="errorCodes">
        <value>23505</value>
    </property>
    <property name="exceptionClass">
        <value>
            com.apress.springrecipes.vehicle.MyDuplicateKeyException
        </value>
    </property>
</bean>
</beans>

```

Now, if you remove the try/catch block surrounding the vehicle insert operation and insert a duplicate vehicle, the Spring JDBC framework will throw a `MyDuplicateKeyException` instead.

However, if you are not satisfied with the basic code-to-exception mapping strategy used by the `SQLExceptionCodes` class, you may further implement the `SQLExceptionTranslator` interface and inject its instance into a JDBC template via the `setExceptionTranslator()` method.

10-6. Problems with Using ORM Frameworks Directly

Problem

You've decided to go to the next level—you have a sufficiently complex domain model, and manually writing all the code for each entity is getting tedious, so you begin to investigate a few alternatives, like Hibernate. You're stunned to find that while they're powerful, they can be anything but simple!

Solution

Let Spring lend a hand; it has facilities for dealing with ORM layers that rival those available for plain ol' JDBC access.

How It Works

Suppose you are developing a course management system for a training center. The first class you create for this system is `Course`. This class is called an *entity class* or a *persistent class* because it represents a real-world entity and its instances will be persisted to a database. Remember that for each entity class to be persisted by an ORM framework, a default constructor with no argument is required.

```

package com.apress.springrecipes.course;
...
public class Course {

    private Long id;
    private String title;
    private Date beginDate;
    private Date endDate;
    private int fee;

    // Constructors, Getters and Setters
    ...
}

```

For each entity class, you must define an identifier property to uniquely identify an entity. It's a best practice to define an auto-generated identifier because this has no business meaning and thus won't be changed under any circumstances. Moreover, this identifier will be used by the ORM framework to determine an entity's state. If the identifier value is null, this entity will be treated as a new and unsaved entity. When this entity is persisted, an insert SQL statement will be issued; otherwise, an update statement will. To allow the identifier to be null, you should choose a primitive wrapper type like `java.lang.Integer` and `java.lang.Long` for the identifier.

In your course management system, you need a DAO interface to encapsulate the data access logic. Let's define the following operations in the `CourseDao` interface:

```
package com.apress.springrecipes.course;
...
public interface CourseDao {
    public void store(Course course);
    public void delete(Long courseId);
    public Course findById(Long courseId);
    public List<Course> findAll();
}
```

Usually, when using ORM for persisting objects, the insert and update operations are combined into a single operation (e.g., `store`). This is to let the ORM framework (not you) decide whether an object should be inserted or updated.

In order for an ORM framework to persist your objects to a database, it must know the mapping metadata for the entity classes. You have to provide mapping metadata to it in its supported format. The native format for Hibernate is XML. However, because each ORM framework may have its own format for defining mapping metadata, JPA defines a set of persistent annotations for you to define mapping metadata in a standard format that is more likely to be reusable in other ORM frameworks.

Hibernate also supports the use of JPA annotations to define mapping metadata, so there are essentially three different strategies for mapping and persisting your objects with Hibernate and JPA:

- Using the Hibernate API to persist objects with Hibernate XML mappings
- Using the Hibernate API to persist objects with JPA annotations
- Using JPA to persist objects with JPA annotations

The core programming elements of Hibernate, JPA, and other ORM frameworks resemble those of JDBC. They are summarized in Table 10-3.

Table 10-3. Core Programming Elements for Different Data Access Strategies

Concept	JDBC	Hibernate	JPA
Resource	Connection	Session	EntityManager
Resource factory	DataSource	SessionFactory	EntityManagerFactory
Exception	SQLException	HibernateException	PersistenceException

In Hibernate, the core interface for object persistence is `Session`, whose instances can be obtained from a `SessionFactory` instance. In JPA, the corresponding interface is `EntityManager`, whose instances can be obtained from an `EntityManagerFactory` instance. The exceptions thrown by Hibernate are of type `HibernateException`, while those thrown by JPA may be of type `PersistenceException` or other Java SE exceptions like `IllegalArgumentException` and `IllegalStateException`. Note that all these exceptions are subclasses of `RuntimeException`, which you are not forced to catch and handle.

Persisting Objects Using the Hibernate API with Hibernate XML Mappings

To map entity classes with Hibernate XML mappings, you can provide a single mapping file for each class or a large file for several classes. Practically, you should define one for each class by joining the class name with `.hbm.xml` as the file extension for ease of maintenance. The middle extension `hbm` stands for “Hibernate metadata.”

The mapping file for the `Course` class should be named `Course.hbm.xml` and put in the same package as the entity class.

```
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.springrecipes.course">
  <class name="Course" table="COURSE">
    <id name="id" type="long" column="ID">
      <generator class="identity" />
    </id>
    <property name="title" type="string">
      <column name="TITLE" length="100" not-null="true" />
    </property>
    <property name="beginDate" type="date" column="BEGIN_DATE" />
    <property name="endDate" type="date" column="END_DATE" />
    <property name="fee" type="int" column="FEE" />
  </class>
</hibernate-mapping>
```

In the mapping file, you can specify a table name for this entity class and a table column for each simple property. You can also specify the column details such as column length, not-null constraints, and unique constraints. In addition, each entity must have an identifier defined, which can be generated automatically or assigned manually. In this example, the identifier will be generated using a table identity column.

Each application that uses Hibernate requires a global configuration file to configure properties such as the database settings (either JDBC connection properties or a data source’s JNDI name), the database dialect, the mapping metadata’s locations, and so on. When using XML mapping files to define mapping metadata, you have to specify the locations of the XML files. By default, Hibernate will read the `hibernate.cfg.xml` file from the root of the classpath. The middle extension `cfg` stands for “configuration.” If there is a `hibernate.properties` file on the classpath, that file will be consulted first and overridden by `hibernate.cfg.xml`.

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```

<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">
            org.apache.derby.jdbc.ClientDriver
        </property>
        <property name="connection.url">
            jdbc:derby://localhost:1527/course;create=true
        </property>
        <property name="connection.username">app</property>
        <property name="connection.password">app</property>
        <property name="dialect">org.hibernate.dialect.DerbyTenSevenDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>

        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>

```

Before you can persist your objects, you have to create tables in a database schema to store the object data. When using an ORM framework like Hibernate, you usually needn't design the tables by yourself. If you set the `hbm2ddl.auto` property to `update`, Hibernate can help you to update the database schema and create the tables when necessary. Naturally, you shouldn't enable this in production, but it can be a great speed boost for development.

Now, let's implement the DAO interface in the hibernate subpackage using the plain Hibernate API. Before you call the Hibernate API for object persistence, you have to initialize a Hibernate session factory (e.g., in the constructor).

```

package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        Configuration configuration = new Configuration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }

    public void store(Course course) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.getTransaction();
        try {
            tx.begin();
            session.saveOrUpdate(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
        }
    }
}

```

```

        throw e;
    } finally {
        session.close();
    }
}

public void delete(Long courseId) {
    Session session = sessionFactory.openSession();
    Transaction tx = session.getTransaction();
    try {
        tx.begin();
        Course course = (Course) session.get(Course.class, courseId);
        session.delete(course);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    } finally {
        session.close();
    }
}

public Course findById(Long courseId) {
    Session session = sessionFactory.openSession();
    try {
        return (Course) session.get(Course.class, courseId);
    } finally {
        session.close();
    }
}

public List<Course> findAll() {
    Session session = sessionFactory.openSession();
    try {
        Query query = session.createQuery("from Course");
        return query.list();
    } finally {
        session.close();
    }
}
}

```

The first step in using Hibernate is to create a Configuration object and ask it to load the Hibernate configuration file. By default, it loads hibernate.cfg.xml from the classpath root when you call the configure() method. Then, you build a Hibernate session factory from this Configuration object. The purpose of a session factory is to produce sessions for you to persist your objects.

In the preceding DAO methods, you first open a session from the session factory. For any operation that involves database update, such as saveOrUpdate() and delete(), you must start a Hibernate transaction on that session. If the operation completes successfully, you commit the transaction. Otherwise, you roll it back if any RuntimeException happens. For read-only operations such as get() and HQL queries, there's no need to start a transaction. Finally, you must remember to close a session to release the resources held by this session.

You can create the following `Main` class to test run all the DAO methods. It also demonstrates an entity's typical life cycle.

```
package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new HibernateCourseDao();

        Course course = new Course();
        course.setTitle("Core Spring");
        course.setBeginDate(new GregorianCalendar(2007, 8, 1).getTime());
        course.setEndDate(new GregorianCalendar(2007, 9, 1).getTime());
        course.setFee(1000);
        courseDao.store(course);

        List<Course> courses = courseDao.findAll();
        Long courseId = courses.get(0).getId();

        course = courseDao.findById(courseId);
        System.out.println("Course Title: " + course.getTitle());
        System.out.println("Begin Date: " + course.getBeginDate());
        System.out.println("End Date: " + course.getEndDate());
        System.out.println("Fee: " + course.getFee());

        courseDao.delete(courseId);
    }
}
```

Persisting Objects Using the Hibernate API with JPA Annotations

JPA annotations are standardized in the JSR-220 specification, so they're supported by all JPA-compliant ORM frameworks, including Hibernate. Moreover, the use of annotations will be more convenient for you to edit mapping metadata in the same source file.

The following `Course` class illustrates the use of JPA annotations to define mapping metadata:

```
package com.apress.springrecipes.course;
...
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "COURSE")
public class Course {
```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "ID")
private Long id;

@Column(name = "TITLE", length = 100, nullable = false)
private String title;

@Column(name = "BEGIN_DATE")
private Date beginDate;

@Column(name = "END_DATE")
private Date endDate;

@Column(name = "FEE")
private int fee;

// Constructors, Getters and Setters
...
}

```

Each entity class must be annotated with the `@Entity` annotation. You can assign a table name for an entity class in this annotation. For each property, you can specify a column name and column details using the `@Column` annotation. Each entity class must have an identifier defined by the `@Id` annotation. You can choose a strategy for identifier generation using the `@GeneratedValue` annotation. Here, the identifier will be generated by a table identity column.

Hibernate supports both native XML mapping files and JPA annotations as ways of defining mapping metadata. For JPA annotations, you have to specify the fully qualified names of the entity classes in `hibernate.cfg.xml` for Hibernate to read the annotations.

```

<hibernate-configuration>
  <session-factory>
    ...
    <mapping class="com.apress.springrecipes.course.Course" />
  </session-factory>
</hibernate-configuration>

```

Persisting Objects Using JPA with Hibernate as the Engine

In addition to persistent annotations, JPA defines a set of programming interfaces for object persistence. However, JPA is not a persistence implementation; you have to pick up a JPA-compliant engine to provide persistence services. Hibernate can be JPA-compliant through the Hibernate EntityManager extension module. With this extension, Hibernate can work as an underlying JPA engine to persist objects. This lets you retain both the valuable investment in Hibernate (perhaps it's faster or handles certain operations more to your satisfaction) and write code that is JPA-compliant and portable among other JPA engines. This can also be a useful way to transition a code base to JPA. New code is written strictly against the JPA APIs, and older code is transitioned to the JPA interfaces.

In a Java EE environment, you can configure the JPA engine in a Java EE container. But in a Java SE application, you have to set up the engine locally. The configuration of JPA is through the central XML file `persistence.xml`, located in the `META-INF` directory of the classpath root. In this file, you can set any vendor-specific properties for the underlying engine configuration.

Now, let's create the JPA configuration file `persistence.xml` in the `META-INF` directory of the classpath root. Each JPA configuration file contains one or more `<persistence-unit>` elements. A *persistence unit* defines a set of persistent classes and how they should be persisted. Each persistence unit requires a name for identification. Here, you assign the name `course` to this persistence unit.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0">

    <persistence-unit name="course">
        <properties>
            <property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml" />
        </properties>
    </persistence-unit>
</persistence>
```

In this JPA configuration file, you configure Hibernate as your underlying JPA engine by referring to the Hibernate configuration file located in the classpath root. However, because Hibernate EntityManager will automatically detect XML mapping files and JPA annotations as mapping metadata, you have no need to specify them explicitly. Otherwise, you will encounter an `org.hibernate.DuplicateMappingException`.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- Don't need to specify mapping files and annotated classes -->
        <!--
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
        <mapping class="com.apress.springrecipes.course.Course" />
        -->
    </session-factory>
</hibernate-configuration>
```

As an alternative to referring to the Hibernate configuration file, you can also centralize all the Hibernate configurations in `persistence.xml`.

```
<persistence ...>
    <persistence-unit name="course">
        <properties>
            <property name="hibernate.connection.driver_class"
                value="org.apache.derby.jdbc.ClientDriver" />
            <property name="hibernate.connection.url"
                value="jdbc:derby://localhost:1527/course;create=true" />
            <property name="hibernate.connection.username" value="app" />
            <property name="hibernate.connection.password" value="app" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.DerbyTenSevenDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
</persistence>'
```

In a Java EE environment, a Java EE container is able to manage the entity manager for you and inject it into your EJB components directly. But when you use JPA outside of a Java EE container (e.g., in a Java SE application), you have to create and maintain the entity manager by yourself.

Note To use Hibernate as the underlying JPA engine, you have to include the Hibernate Entity Manager libraries to your CLASSPATH. If you're using Maven, add the following dependency to your project:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>
```

Now, let's implement the `CourseDao` interface using JPA in a Java SE application. Before you call JPA for object persistence, you have to initialize an entity manager factory. The purpose of an entity manager factory is to produce entity managers for you to persist your objects.

```
package com.apress.springrecipes.course;
...
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public JpaCourseDao() {
        entityManagerFactory = Persistence.createEntityManagerFactory("course");
    }

    public void store(Course course) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            manager.merge(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }
}
```

```

public void delete(Long courseId) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    EntityTransaction tx = manager.getTransaction();
    try {
        tx.begin();
        Course course = manager.find(Course.class, courseId);
        manager.remove(course);
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}

public Course findById(Long courseId) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
        return manager.find(Course.class, courseId);
    } finally {
        manager.close();
    }
}

public List<Course> findAll() {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
        Query query = manager.createQuery("select course from Course course");
        return query.getResultList();
    } finally {
        manager.close();
    }
}
}

```

The entity manager factory is built by the static method `createEntityManagerFactory()` of the `javax.persistence.Persistence` class. You have to pass in a persistence unit name defined in `persistence.xml` for an entity manager factory.

In the preceding DAO methods, you first create an entity manager from the entity manager factory. For any operation that involves database update, such as `merge()` and `remove()`, you must start a JPA transaction on the entity manager. For read-only operations such as `find()` and JPA queries, there's no need to start a transaction. Finally, you must close an entity manager to release the resources.

You can test this DAO with the similar `Main` class, but this time, you instantiate the JPA DAO implementation instead.

```

package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new JpaCourseDao();
        ...
    }
}

```

In the preceding DAO implementations for both Hibernate and JPA, there are only one or two lines that are different for each DAO method. The rest of the lines are boilerplate routine tasks that you have to repeat. Moreover, each ORM framework has its own API for local transaction management.

10-7. Configuring ORM Resource Factories in Spring

Problem

When using an ORM framework on its own, you have to configure its resource factory with its API. For Hibernate and JPA, you have to build a session factory and an entity manager factory from the native Hibernate API and JPA. You have no choice but to manage these objects manually, without Spring's support.

Solution

Spring provides several factory beans for you to create a Hibernate session factory or a JPA entity manager factory as a singleton bean in the IoC container. These factories can be shared between multiple beans via dependency injection. Moreover, this allows the session factory and the entity manager factory to integrate with other Spring data access facilities, such as data sources and transaction managers.

How It Works

Configuring a Hibernate Session Factory in Spring

First of all, let's modify `HibernateCourseDao` to accept a session factory via dependency injection, instead of creating it directly with the native Hibernate API in the constructor.

```

package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}

```

Now, let's look at how to declare a session factory that uses XML mapping files in Spring. For this purpose, you have to enable the XML mapping file definition in `hibernate.cfg.xml` again.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- For Hibernate XML mappings -->
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Then, you create a bean configuration file for using Hibernate as the ORM framework (e.g., `beans-hibernate.xml` in the classpath root). You can declare a session factory that uses XML mapping files with the factory bean `LocalSessionFactoryBean`. You can also declare a `HibernateCourseDao` instance under Spring's management.

```
package com.apress.springrecipes.course.config;

import com.apress.springrecipes.course.CourseDao;
import com.apress.springrecipes.course.HibernateCourseDao;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.orm.hibernate4.LocalSessionFactoryBean;

@Configuration
public class CourseConfiguration {

    @Bean
    public CourseDao courseDao() {
        return new HibernateCourseDao(sessionfactory().getObjectType());
    }

    @Bean
    public LocalSessionFactoryBean sessionfactory() {
        LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
        sessionFactoryBean.setConfigLocation(new ClassPathResource("hibernate.cfg.xml"));
        return sessionFactoryBean;
    }
}
```

Note that you can specify the `configLocation` property for this factory bean to load the Hibernate configuration file. The `configLocation` property is of type `Resource`, and we want to load the file from the classpath, hence we construct a `ClassPathResource` with the name of the configuration file. The preceding factory bean loads the configuration file from the root of the classpath.

Now, you can modify the `Main` class to retrieve the `HibernateCourseDao` instance from the Spring IoC container.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(CourseConfiguration.class);
        CourseDao courseDao = context.getBean(CourseDao.class);
        ...
    }
}
```

The preceding factory bean creates a session factory by loading the Hibernate configuration file, which includes the database settings (either JDBC connection properties or a data source's JNDI name). Now, suppose you have a data source defined in the Spring IoC container. If you want to use this data source for your session factory, you can inject it into the `dataSource` property of `LocalSessionFactoryBean`. The data source specified in this property will override the database settings in the Hibernate configuration file. If this is set, the Hibernate settings should not define a connection provider to avoid meaningless double configuration.

```
@Configuration
public class CourseConfiguration {
    ...
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
        sessionFactoryBean.setDataSource(dataSource());
        sessionFactoryBean.setConfigLocation(new ClassPathResource("hibernate.cfg.xml"));
        return sessionFactoryBean;
    }

    @Bean
    public DataSource dataSource() {
        SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
        dataSource.setDriverClass(ClientDriver.class);
        dataSource.setUrl("jdbc:derby://localhost:1527/course;create=true");
        dataSource.setUsername("app");
        dataSource.setPassword("app");
        return dataSource;
    }
}
```

Or you can even ignore the Hibernate configuration file by merging all the configurations into `LocalSessionFactoryBean`. For example, you can specify the packages containing the JPA annotated classes in the `packagesToScan` property and other Hibernate properties such as the database dialect in the `hibernateProperties` property.

```
@Configuration
public class CourseConfiguration {
    ...
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
        sessionFactoryBean.setDataSource(dataSource());
        sessionFactoryBean.setPackagesToScan("com.apress.springrecipes.course");
        sessionFactoryBean.setHibernateProperties(hibernateProperties());
        return sessionFactoryBean;
    }
}
```

```

private Properties hibernateProperties() {
    Properties properties = new Properties();
    properties.put("hibernate.dialect", org.hibernate.dialect.DerbyTenSevenDialect.class.getName());
    properties.put("hibernate.show_sql", true);
    properties.put("hibernate.hbm2dll.auto", "update");
    return properties;
}
}

```

If you are in a project that still uses Hibernate mapping files you can use the `mappingLocations` property to specify the mapping files. `LocalSessionFactoryBean` also allows you take advantage of Spring's resource-loading support to load mapping files from various types of locations. You can specify the resource paths of the mapping files in the `mappingLocations` property, whose type is `Resource[]`.

```

@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
    sessionFactoryBean.setDataSource(dataSource());
    sessionFactoryBean.setMappingLocations(
        new ClassPathResource("com/apress/springrecipes/course/Course.hbm.xml "));
    sessionFactoryBean.setHibernateProperties(hibernateProperties());
    return sessionFactoryBean;
}

```

With Spring's resource-loading support, you can also use wildcards in a resource path to match multiple mapping files so that you don't need to configure their locations every time you add a new entity class. For this to work we need a `ResourcePatternResolver` in our configuration class. We can get this by using the `ResourcePatternUtils` and the `ResourceLoaderAware` interface. We implement the latter and use the `getResourcePatternResolver` method to get a `ResourcePatternResolver` based on the `ResourceLoader`.

```

@Configuration
public class CourseConfiguration implements ResourceLoaderAware {

    private ResourcePatternResolver resourcePatternResolver;
    ...
    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourcePatternResolver = ResourcePatternUtils.getResourcePatternResolver(resourceLoader);
    }
}

```

Now we can use the `ResourcePatternResolver` to resolve resource patterns to resources.

```

@Bean
public LocalSessionFactoryBean sessionFactory() throws IOException {
    LocalSessionFactoryBean sessionFactoryBean = new LocalSessionFactoryBean();
    Resource[] mappingResources =
        resourcePatternResolver.getResources("classpath:com/apress/springrecipes/course/*.hbm.xml");
    sessionFactoryBean.setMappingLocations(mappingResources);
    ...
    return sessionFactoryBean;
}

```

You can now delete the Hibernate configuration file (i.e., `hibernate.cfg.xml`) because its configurations have been ported to Spring.

Configuring a JPA Entity Manager Factory in Spring

First of all, let's modify `JpaCourseDao` to accept an entity manager factory via dependency injection, instead of creating it directly in the constructor.

```
package com.apress.springrecipes.course;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JpaCourseDao implements CourseDao {

    private final EntityManagerFactory entityManagerFactory;

    public JpaCourseDao (EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }
    ...
}
```

The JPA specification defines how you should obtain an entity manager factory in Java SE and Java EE environments. In a Java SE environment, an entity manager factory is created manually by calling the `createEntityManagerFactory()` static method of the `Persistence` class.

Let's create a bean configuration file for using JPA. Spring provides a factory bean, `LocalEntityManagerFactoryBean`, for you to create an entity manager factory in the IoC container. You must specify the persistence unit name defined in the JPA configuration file. You can also declare a `JpaCourseDao` instance under Spring's management.

```
package com.apress.springrecipes.course.config;

import com.apress.springrecipes.course.CourseDao;
import com.apress.springrecipes.course.JpaCourseDao;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

@Configuration
public class CourseConfiguration {

    @Bean
    public CourseDao courseDao() {
        return new JpaCourseDao(entityManagerFactory().getObject());
    }
}
```

```

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
        emf.setPersistenceUnitName("course");
        return emf;
    }
}

```

Now, you can test this JpaCourseDao instance with the Main class by retrieving it from the Spring IoC container.

```

package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(CourseConfiguration.class);
        CourseDao courseDao = context.getBean(CourseDao.class);
        ...
    }
}

```

In a Java EE environment, you can look up an entity manager factory from a Java EE container with JNDI. In Spring, you can perform a JNDI lookup by using the JndiLocatorDelegate object (which is simpler than constructing a JndiObjectFactoryBean which would also work).

```

@Bean
public EntityManagerFactory entityManagerFactory() throws NamingException {
    return JndiLocatorDelegate.createDefaultResourceRefLocator()
        .lookup("jpa/coursePU", EntityManagerFactory.class);
}

```

LocalEntityManagerFactoryBean creates an entity manager factory by loading the JPA configuration file (i.e., `persistence.xml`). Spring supports a more flexible way to create an entity manager factory by another factory bean, LocalContainerEntityManagerFactoryBean. It allows you to override some of the configurations in the JPA configuration file, such as the data source and database dialect. So, you can take advantage of Spring's data access facilities to configure the entity manager factory.

```

@Configuration
public class CourseConfiguration {
    ...
    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
        emf.setPersistenceUnitName("course");
        emf.setDataSource(dataSource());
        emf.setJpaVendorAdapter(jpaVendorAdapter());
        return emf;
    }
}

```

```

private JpaVendorAdapter jpaVendorAdapter() {
    HibernateJpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
    jpaVendorAdapter.setShowSql(true);
    jpaVendorAdapter.setGenerateDdl(true);
    jpaVendorAdapter.setDatabasePlatform(DerbyTenSevenDialect.class.getName());
    return jpaVendorAdapter;
}

@Bean
public DataSource dataSource() {
    SimpleDriverDataSource dataSource = new SimpleDriverDataSource();
    dataSource.setDriverClass(ClientDriver.class);
    dataSource.setUrl("jdbc:derby://localhost:1527/course;create=true");
    dataSource.setUsername("app");
    dataSource.setPassword("app");
    return dataSource;
}
}

```

In the preceding bean configurations, you inject a data source into this entity manager factory. It will override the database settings in the JPA configuration file. You can set a JPA vendor adapter to `LocalContainerEntityManagerFactoryBean` to specify JPA engine-specific properties. With Hibernate as the underlying JPA engine, you should choose `HibernateJpaVendorAdapter`. Other properties that are not supported by this adapter can be specified in the `jpaProperties` property.

Now your JPA configuration file (i.e., `persistence.xml`) can be simplified as follows because its configurations have been ported to Spring:

```

<persistence ...>
    <persistence-unit name="course" />
</persistence>

```

Spring also makes it possible to configure the JPA `EntityManagerFactory` without a `persistence.xml` if we want we can fully configure it in a Spring configuration file. Instead of a `persistenceUnitName` we need to specify the `packagesToScan` property. After this we can remove the `persistence.xml` completely.

```

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
    emf.setDataSource(dataSource());
    emf.setPackagesToScan("com.apress.springrecipes.course");
    emf.setJpaVendorAdapter(jpaVendorAdapter());
    return emf;
}

```

10-8. Persisting Objects with Spring's ORM Templates

Problem

When using an ORM framework on its own, you have to repeat certain routine tasks for each DAO operation. For example, in a DAO operation implemented with Hibernate or JPA, you have to open and close a session or an entity manager, and begin, commit, and roll back a transaction with the native API.

Solution

Spring's approach to simplifying an ORM framework's usage is the same as JDBC's—by defining template classes and DAO support classes. Also, Spring defines an abstract layer on top of different transaction management APIs. For different ORM frameworks, you only have to pick up a corresponding transaction manager implementation. Then, you can manage transactions for them in a similar way.

In Spring's data access module, the support for different data access strategies is consistent. Table 10-4 compares the support classes for JDBC, Hibernate, and JPA.

Table 10-4. Spring's Support Classes for Different Data Access Strategies

Support Class	JDBC	Hibernate	JPA
Template class	<code>JdbcTemplate</code>	<code>HibernateTemplate</code>	-
DAO support	<code>JdbcDaoSupport</code>	<code>HibernateDaoSupport</code>	-
Transaction	<code>DataSourceTransaction</code>	<code>HibernateTransaction</code>	<code>JpaTransactionManager</code>

Note Before Spring 4.0 there were also `JpaTemplate` and `JpaDaoSupport` classes; however they have been removed in favor of using plain JPA (see Recipe 10-10).

Spring defines the `HibernateTemplate` class (for Hibernate 3 and 4) to provide template methods for different versions of Hibernate minimize the effort involved in using them. The template methods in `HibernateTemplate` ensure that Hibernate will be opened and closed properly. They will also have native Hibernate transactions participate in Spring-managed transactions. As a result, you will be able to manage transactions declaratively for your Hibernate and JDBC DAOs without any boilerplate transaction code.

Note The template approach should be considered deprecated in favor of using either a plain `SessionFactory` or `EntityManager` (see respectively Recipes 10-9 and 10-10).

How It Works

Using a Hibernate Template

First, the `HibernateCourseDao` class can be simplified as follows with the help of Spring's `HibernateTemplate`:

```
package com.apress.springrecipes.course;
...
import org.springframework.orm.hibernate4.HibernateTemplate;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }

    @Transactional
    public void store(Course course) {
        hibernateTemplate.saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) hibernateTemplate.get(Course.class, courseId);
        hibernateTemplate.delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) hibernateTemplate.get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return hibernateTemplate.find("from Course");
    }
}
```

In this DAO implementation, you declare all the DAO methods to be transactional with the `@Transactional` annotation. Among these methods, `findById()` and `findAll()` are read-only. The template methods in `HibernateTemplate` are responsible for managing the sessions and transactions. If there are multiple Hibernate operations in a transactional DAO method, the template methods will ensure that they will run within the same session and transaction. As a result, you have no need to deal with the Hibernate API for session and transaction management.

The `HibernateTemplate` class is thread-safe, so you can declare a single instance of it in the bean configuration file for Hibernate and inject this instance into all Hibernate DAOs. A `HibernateTemplate` instance requires the `sessionFactory` property to be set. You can inject this property by either setter method or constructor argument.

```
@Configuration
@EnableTransactionManagement
public class CourseConfiguration {
```

```

@Bean
public CourseDao courseDao() {
    return new HibernateCourseDao(hibernateTemplate());
}

@Bean
public HibernateTemplate hibernateTemplate() {
    return new HibernateTemplate(sessionfactory().getobject());
}

@Bean
public LocalSessionFactoryBean sessionfactory() { ... }

}

@Bean
public PlatformTransactionManager transactionManager() {
    return new HibernateTransactionManager(sessionfactory().getobject());
}

```

To enable declarative transaction management for the methods annotated with `@Transactional`, you have to add the `@EnableTransactionManagement` annotation to your configuration class. By default, it will look for a transaction manager with the name `transactionManager`, so you have to declare a `HibernateTransactionManager` instance with that name. `HibernateTransactionManager` requires the session factory property to be set. It will manage transactions for sessions opened through this session factory.

Another advantage of `HibernateTemplate` is that they will translate native Hibernate exceptions into exceptions in Spring's `DataAccessException` hierarchy. This allows consistent exception handling for all the data access strategies in Spring. For instance, if a database constraint is violated when persisting an object, Hibernate will throw an `org.hibernate.exception.ConstraintViolationException`, while JPA will throw a `javax.persistence.EntityExistsException`. These exceptions will be translated by `HibernateTemplate` into a `DataIntegrityViolationException`, which is a subclass of Spring's `DataAccessException`.

If you want to get access to the underlying Hibernate session in `HibernateTemplate` in order to perform native Hibernate operations, you can implement the `HibernateCallback` interface and pass its instance to the `execute()` method of the template. This will give you a chance to use any implementation-specific features directly if there's not sufficient support already available from the template implementations.

```

hibernateTemplate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session) throws HibernateException, SQLException {
        // ... anything you can imagine doing can be done here.
        // Cache invalidation, for example...
    }
});

```

Extending the Hibernate DAO Support Classes

Your Hibernate DAO can extend `HibernateDaoSupport` to have the `setSessionFactory()` and `setHibernateTemplate()` methods inherited. Then, in your DAO methods, you can simply call the `getHibernateTemplate()` method to retrieve the template instance.

```

package com.apress.springrecipes.course;
...
import org.springframework.orm.hibernate4.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

```

```

public class HibernateCourseDao extends HibernateDaoSupport implements CourseDao {

    @Transactional
    public void store(Course course) {
        getHibernateTemplate().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = getHibernateTemplate().get(Course.class, courseId);
        return getHibernateTemplate().delete(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return getHibernateTemplate().get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getHibernateTemplate().find("from Course");
    }
}

```

Because `HibernateCourseDao` inherits the `setSessionFactory()` and `setHibernateTemplate()` methods, you can inject either of them into your DAO so that you can retrieve the `HibernateTemplate` instance. If you inject a session factory, you will be able to delete the `HibernateTemplate` declaration.

```

@Bean
public CourseDao courseDao() {
    HibernateCourseDao courseDao = new HibernateCourseDao();
    courseDao.setSessionFactory(sessionfactory().getObject());
    return courseDao;
}

```

10-9. Persisting Objects with Hibernate's Contextual Sessions

Problem

Spring's `HibernateTemplate` can simplify your DAO implementation by managing sessions and transactions for you. However, using `HibernateTemplate` means your DAO has to depend on Spring's API.

Tip This is the recommended approach instead of the template-based approach!

Solution

An alternative to Spring's `HibernateTemplate` is to use Hibernate's contextual sessions. In Hibernate 3, a session factory can manage contextual sessions for you and allows you to retrieve them by the `getCurrentSession()` method on `org.hibernate.SessionFactory`. Within a single transaction, you will get the same session for each `getCurrentSession()` method call. This ensures that there will be only one Hibernate session per transaction, so it works nicely with Spring's transaction management support.

How It Works

To use the contextual session approach, your DAO methods require access to the session factory, which can be injected via a setter method or a constructor argument. Then, in each DAO method, you get the contextual session from the session factory and use it for object persistence.

```
package com.apress.springrecipes.course;
...
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Transactional
    public void store(Course course) {
        sessionFactory.getCurrentSession().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) sessionFactory.getCurrentSession().get(Course.class, courseId);
        sessionFactory.getCurrentSession().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) sessionFactory.getCurrentSession().get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = sessionFactory.getCurrentSession().createQuery("from Course");
        return query.list();
    }
}
```

Note that all your DAO methods must be made transactional. This is required because Spring integrates with Hibernate through Hibernate's Contextual Session support. Spring has its own implementation of the `CurrentSessionContext` interface from Hibernate. It will attempt to find a transaction and then fail, complaining that no Hibernate session's been bound to the thread. You can achieve this by annotating each method or the entire class with `@Transactional`. This ensures that the persistence operations within a DAO method will be executed in the same transaction and hence by the same Session. Moreover, if a service layer component's method calls multiple DAO methods, and it propagates its own transaction to these methods, then all these DAO methods will run within the same session as well.

Caution When configuring Hibernate with Spring make sure not to set the `hibernate.current_session_context_class` property, as that will interfere with Springs' ability to properly manage the transactions. You should only set this property when you are in need of JTA transactions.

In the bean configuration file, you have to declare a `HibernateTransactionManager` instance for this application and enable declarative transaction management via `@EnableTransactionManagement`.

```
@Configuration
@EnableTransactionManagement
public class CourseConfiguration {

    @Bean
    public CourseDao courseDao() {
        return new HibernateCourseDao(sessionfactory().getobject());
    }
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new HibernateTransactionManager(sessionfactory().getobject());
    }
}
```

Remember that `HibernateTemplate` will translate the native Hibernate exceptions into exceptions in Spring's `DataAccessException` hierarchy. This allows consistent exception handling for different data access strategies in Spring. However, when calling the native methods on a Hibernate session, the exceptions thrown will be of native type `HibernateException`. If you want the Hibernate exceptions to be translated into Spring's `DataAccessException` for consistent exception handling, you have to apply the `@Repository` annotation to your DAO class that requires exception translation.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.stereotype.Repository;

@Repository
public class HibernateCourseDao implements CourseDao {
    ...
}
```

A PersistenceExceptionTranslationPostProcessor takes care of translating the native Hibernate exceptions into data access exceptions in Spring's DataAccessException hierarchy. This bean post processor will only translate exceptions for beans annotated with @Repository. When using Java based configuration this bean is automatically registered in the AnnotationConfigApplicationContext, hence there is no need to explicitly declare a bean for it.

In Spring, @Repository is a stereotype annotation. By annotating it, a component class can be auto-detected through component scanning. You can assign a component name in this annotation and have the session factory auto-wired by the Spring IoC container with @Autowired.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    @Autowired
    public HibernateCourseDao (SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

Then, you can simply add the @ComponentScan annotation and delete the original HibernateCourseDao bean declaration.

```
@Configuration
@EnableTransactionManagement
@ComponentScan("com.apress.springrecipes.course")
public class CourseConfiguration { ... }
```

10-10. Persisting Objects with JPA's Context Injection Problem

In a Java EE environment, a Java EE container can manage entity managers for you and inject them into your EJB components directly. An EJB component can simply perform persistence operations on an injected entity manager without caring much about the entity manager creation and transaction management.

Solution

Originally, the @PersistenceContext annotation is used for entity manager injection in EJB components. Spring can also interpret this annotation by means of a bean post processor. It will inject an entity manager into a property with this annotation. Spring ensures that all your persistence operations within a single transaction will be handled by the same entity manager.

How It Works

To use the context injection approach, you can declare an entity manager field in your DAO and annotate it with the `@PersistenceContext` annotation. Spring will inject an entity manager into this field for you to persist your objects.

```
package com.apress.springrecipes.course;
...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void store(Course course) {
        entityManager.merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = entityManager.find(Course.class, courseId);
        entityManager.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return entityManager.find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        TypedQuery<Course> query = entityManager.createQuery("from Course", Course.class);
        return query.getResultList();
    }
}
```

You can annotate each DAO method or the entire DAO class with `@Transactional` to make all these methods transactional. It ensures that the persistence operations within a DAO method will be executed in the same transaction and hence by the same entity manager.

In the bean configuration file, you have to declare a `JpaTransactionManager` instance and enable declarative transaction management via `@EnableTransactionManagement`. A `PersistenceAnnotationBeanPostProcessor` instance is registered automatically when using Java based config, to inject entity managers into properties annotated with `@PersistenceContext`.

```
package com.apress.springrecipes.course.config;

import com.apress.springrecipes.course.CourseDao;
import com.apress.springrecipes.course.JpaCourseDao;
import org.apache.derby.jdbc.ClientDriver;
import org.hibernate.dialect.DerbyTenSevenDialect;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.SimpleDriverDataSource;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.sql.DataSource;

@Configuration
@EnableTransactionManagement
public class CourseConfiguration {

    @Bean
    public CourseDao courseDao() {
        return new JpaCourseDao();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        LocalContainerEntityManagerFactoryBean emf = new LocalContainerEntityManagerFactoryBean();
        emf.setDataSource(dataSource());
        emf.setJpaVendorAdapter(jpaVendorAdapter());
        return emf;
    }

    private JpaVendorAdapter jpaVendorAdapter() {
        HibernateJpaVendorAdapter jpaVendorAdapter = new HibernateJpaVendorAdapter();
        jpaVendorAdapter.setShowSql(true);
        jpaVendorAdapter.setGenerateDdl(true);
        jpaVendorAdapter.setDatabasePlatform(DerbyTenSevenDialect.class.getName());
        return jpaVendorAdapter;
    }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new JpaTransactionManager(entityManagerFactory().getObject());
    }

    @Bean
    public DataSource dataSource() { ... }

}
```

The `PersistenceAnnotationBeanPostProcessor` can also inject the entity manager factory into a property with the `@PersistenceUnit` annotation. This allows you to create entity managers and manage transactions by yourself. It's no different from injecting the entity manager factory via a setter method.

```
package com.apress.springrecipes.course;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;

public class JpaCourseDao implements CourseDao {
    @PersistenceContext
    private EntityManager entityManager;

    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;
    ...
}
```

When calling native methods on a JPA entity manager, the exceptions thrown will be of native type `PersistenceException`, or other Java SE exceptions like `IllegalArgumentException` and `IllegalStateException`. If you want JPA exceptions to be translated into Spring's `DataAccessException`, you have to apply the `@Repository` annotation to your DAO class.

```
package com.apress.springrecipes.course;
...
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class JpaCourseDao implements CourseDao {
    ...
}
```

A `PersistenceExceptionTranslationPostProcessor` instance will translate the native JPA exceptions into exceptions in Spring's `DataAccessException` hierarchy. When using Java based configuration this bean is automatically registered in the `AnnotationConfigApplicationContext`, hence there is no need to explicitly declare a bean for it.

10-11. Simplify JPA with Spring Data JPA

Problem

Writing data access code, even with JPA, can be a tedious and repetitive task. You often need access to the `EntityManager` or `EntityManagerFactory` and have to create queries. Not to mention the fact that when one has a lot of dao's the repetitive declaration of `findById`, `findAll` methods for all different entities.

Solution

Spring Data JPA allows you, just as Spring itself, to focus on the parts that are important and not on the boilerplate needed to accomplish this. It also provides default implementations for the most commonly used data access methods (i.e., `findAll`, `delete`, `save` etc.).

How It Works

To use Spring Data JPA we have to extend one of its interfaces, these interfaces are detected and a default implementation of that repository is generated at runtime. In most cases it is enough to extend the `CrudRepository<T, ID>` interface.

```
package com.apress.springrecipes.course;

import com.apress.springrecipes.course.Course;
import org.springframework.data.repository.CrudRepository;

public interface CourseRepository extends CrudRepository<Course, Long>{}
```

This is enough to be able to do all necessary CRUD actions for the `Course` entity. When extending the Spring Data interfaces we have to specify the type, `Course`, and the type of the primary key, `Long`, this information is needed to generate the repository at runtime.

Note You could also extend `JpaRepository` which adds some JPA specific methods (`flush`, `saveAndFlush`) and provides query methods with paging/sorting capabilities.

Next we need to enable detection of the Spring Data enabled repositories for this we can use the `@EnableJpaRepositories` annotation provided by Spring Data JPA.

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories("com.apress.springrecipes.course")
public class CourseConfiguration { ... }
```

This will bootstrap Spring Data JPA and will construct a usable repository. By default all repository methods are marked with `@Transactional` so no additional annotations are needed.

Now, you can test this `CourseRepository` instance with the `Main` class by retrieving it from the Spring IoC container.

```
package com.apress.springrecipes.course.datajpa;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(CourseConfiguration.class);

        CourseRepository repository = context.getBean(CourseRepository.class);
        ...
    }
}
```

All other things like exception translation, transaction management, and easy configuration of your EntityManagerFactory still apply to Spring Data JPA based repositories. It just makes your life a lot easier and lets you focus on what is important.

Summary

This chapter discussed how to use Spring's support for JDBC, Hibernate, and JPA. You learned how to configure a DataSource to connect to a database and how to use Spring's JdbcTemplate and HibernateTemplate to rid your code of tedious boilerplate handling. You saw how to use the utility base classes to build DAO classes with JDBC and Hibernate, as well as how to use Spring's support for stereotype annotations and component scanning to easily build new DAOs and Services with a minimum of XML. The final recipe showed you how to simplify your data access code even more by using the power of Spring Data JPA.

In the next chapter, you will learn how to use transactions (i.e., for JMS or a database) with Spring to help ensure consistent state in your services.

CHAPTER 11

Spring Transaction Management

In this chapter, you will learn about the basic concept of transactions and Spring's capabilities in the area of transaction management. Transaction management is an essential technique in enterprise applications to ensure data integrity and consistency. Spring, as an enterprise application framework, provides an abstract layer on top of different transaction management APIs. As an application developer, you can use Spring's transaction management facilities without having to know much about the underlying transaction management APIs.

Like the bean-managed transaction (BMT) and container-managed transaction (CMT) approaches in EJB, Spring supports both programmatic and declarative transaction management. The aim of Spring's transaction support is to provide an alternative to EJB transactions by adding transaction capabilities to POJOs.

Programmatic transaction management is achieved by embedding transaction management code in your business methods to control the commit and rollback of transactions. You usually commit a transaction if a method completes normally and roll back a transaction if a method throws certain types of exceptions. With programmatic transaction management, you can define your own rules to commit and roll back transactions.

However, when managing transactions programmatically, you have to include transaction management code in each transactional operation. As a result, the boilerplate transaction code is repeated in each of these operations. Moreover, it's hard for you to enable and disable transaction management for different applications. If you have a solid understanding of AOP, you may already have noticed that transaction management is a kind of crosscutting concern.

Declarative transaction management is preferable to programmatic transaction management in most cases. It's achieved by separating transaction management code from your business methods via declarations. Transaction management, as a kind of crosscutting concern, can be modularized with the AOP approach. Spring supports declarative transaction management through the Spring AOP framework. This can help you to enable transactions for your applications more easily and define a consistent transaction policy. Declarative transaction management is less flexible than programmatic transaction management. Programmatic transaction management allows you to control transactions through your code—explicitly starting, committing, and joining them as you see fit. You can specify a set of transaction attributes to define your transactions at a fine level of granularity. The transaction attributes supported by Spring include the propagation behavior, isolation level, rollback rules, transaction timeout, and whether or not the transaction is read-only. These attributes allow you to further customize the behavior of your transactions.

Upon finishing this chapter, you will be able to apply different transaction management strategies in your application. Moreover, you will be familiar with different transaction attributes to finely define your transactions.

Programmatic transaction management is a good idea in certain cases where you don't feel the addition of Spring proxies is worth the trouble or negligible performance loss. Here, you might access the native transaction yourself and control the transaction manually. A more convenient option that avoids the overhead of Spring proxies is the `TransactionTemplate` class, which provides a template method around which a transactional boundary is started and then committed.

11-1. Problems with Transaction Management

Transaction management is an essential technique in enterprise application development to ensure data integrity and consistency. Without transaction management, your data and resources may be corrupted and left in an inconsistent state. Transaction management is particularly important for recovering from unexpected errors in a concurrent and distributed environment.

In simple words, a *transaction* is a series of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. If all the actions go well, the transaction should be committed permanently. In contrast, if any of them goes wrong, the transaction should be rolled back to the initial state as if nothing had happened.

The concept of transactions can be described with four key properties: *atomicity*, *consistency*, *isolation*, and *durability* (*ACID*).

- *Atomicity*: A *transaction* is an atomic operation that consists of a series of actions. The atomicity of a transaction ensures that the actions either complete entirely or take no effect at all.
- *Consistency*: Once all actions of a transaction have completed, the transaction is committed. Then your data and resources will be in a consistent state that conforms to business rules.
- *Isolation*: Because there may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
- *Durability*: Once a transaction has completed, its result should be durable to survive any system failure (imagine if the power to your machine was cut right in the middle of a transaction's commit). Usually, the result of a transaction is written to persistent storage.

To understand the importance of transaction management, let's begin with an example about purchasing books from an online bookshop. First, you have to create a new schema for this application in your database. If you are choosing Apache Derby as your database engine, you can connect to it with the JDBC properties shown in Table 11-1. For the examples in this book, we're using Derby 10.10.2.0.

Table 11-1. JDBC Properties for Connecting to the Application Database

Property	Value
Driver class	org.apache.derby.jdbc.ClientDriver
URL	jdbc:derby://localhost:1527/bookshop;create=true
Username	App
Password	App

With the preceding configuration, the database will be created for you because of the parameter on the JDBC URL: `create=true`. For your bookshop application, you need a place to store the data. You'll create a simple database to manage books and accounts.

The entity relational (ER) diagram for the tables looks like Figure 11-1.

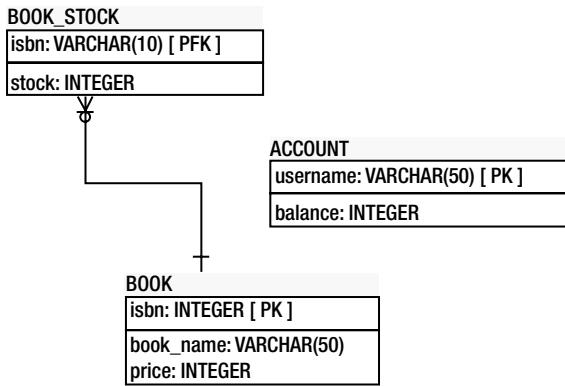


Figure 11-1. *BOOK_STOCK* describes how many given *BOOKs* exist

Now, let's create the SQL for the preceding model. You'll use the `ij` tool that ships with Derby. On a command line, proceed to the directory where Derby is installed (usually just where you unzipped it when you downloaded it.). Descend to the `bin` directory. If Derby's not already started, run `startNetworkServer` (or `startNetworkServer.bat` on Windows). Now, you need to log in and execute the SQL DDL. Background the Derby Server process or open up a second shell and return to the same `bin` directory in the Derby installation directory. Execute `ij`. In the shell, execute the following:

```
connect 'jdbc:derby://localhost:1527/bookshop;create=true' ;
```

Paste the following SQL into the shell and verify its success:

```

CREATE TABLE BOOK (
    ISBN      VARCHAR(50)  NOT NULL,
    BOOK_NAME VARCHAR(100) NOT NULL,
    PRICE     INT,
    PRIMARY KEY (ISBN)
);

CREATE TABLE BOOK_STOCK (
    ISBN      VARCHAR(50)  NOT NULL,
    STOCK     INT          NOT NULL,
    PRIMARY KEY (ISBN),
    CHECK (STOCK >=0)
);

CREATE TABLE ACCOUNT (
    USERNAME  VARCHAR(50)  NOT NULL,
    BALANCE   INT          NOT NULL,
    PRIMARY KEY (USERNAME),
    CHECK (BALANCE >=0)
);

```

A real-world application of this type would probably feature a price field with a decimal type, but using an `int` makes the programming simpler to follow, so leave it as an `int`.

The BOOK table stores basic book information such as the name and price, with the book ISBN as the primary key. The BOOK_STOCK table keeps track of each book's stock. The stock value is restricted by a CHECK constraint to be a positive number. Although the CHECK constraint type is defined in SQL-99, not all database engines support it. At the time of this writing, this limitation is mainly true of MySQL because Sybase, Derby, HSQL, Oracle, DB2, SQL Server, Access, PostgreSQL, and FireBird all support it. If your database engine doesn't support CHECK constraints, please consult its documentation for similar constraint support. Finally, the ACCOUNT table stores customer accounts and their balances. Again, the balance is restricted to be positive.

The operations of your bookshop are defined in the following BookShop interface. For now, there is only one operation: purchase().

```
package com.apress.springrecipes.bookshop;

public interface BookShop {

    public void purchase(String isbn, String username);
}
```

Because you will implement this interface with JDBC, you create the following JdbcBookShop class. To better understand the nature of transactions, let's implement this class without the help of Spring's JDBC support.

```
package com.apress.springrecipes.bookshop;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcBookShop implements BookShop {

    private DataSource dataSource;

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

    public void purchase(String isbn, String username) {
        Connection conn = null;
        try {
            conn = dataSource.getConnection();

            PreparedStatement stmt1 = conn.prepareStatement(
                "SELECT PRICE FROM BOOK WHERE ISBN = ?");
            stmt1.setString(1, isbn);
            ResultSet rs = stmt1.executeQuery();
            rs.next();
            int price = rs.getInt("PRICE");
            stmt1.close();
        }
    }
}
```

```
        PreparedStatement stmt2 = conn.prepareStatement(
                "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
                "WHERE ISBN = ?");
        stmt2.setString(1, isbn);
        stmt2.executeUpdate();
        stmt2.close();

        PreparedStatement stmt3 = conn.prepareStatement(
                "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
                "WHERE USERNAME = ?");
        stmt3.setInt(1, price);
        stmt3.setString(2, username);
        stmt3.executeUpdate();
        stmt3.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}
```

For the `purchase()` operation, you have to execute three SQL statements in total. The first is to query the book price. The second and third update the book stock and account balance accordingly.

Then, you can declare a bookshop instance in the Spring IoC container to provide purchasing services. For simplicity's sake, you can use `DriverManagerDataSource`, which opens a new connection to the database for every request.

Note To access a database running on the Derby server, you have to add the Derby client library to your CLASSPATH. If you're using Maven, add the following dependency to your project.

```
<dependency>
    <groupId>org.apache.derby</groupId>
    <artifactId>derbyclient</artifactId>
    <version>10.10.2.0</version>
</dependency>
```

```
package com.apress.springrecipes.bookshop.config;

import com.apress.springrecipes.bookshop.BookShop;
import com.apress.springrecipes.bookshop.JdbcBookShop;
import org.apache.derby.jdbc.ClientDriver;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
```

```

import javax.sql.DataSource;

@Configuration
public class BookstoreConfiguration {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(ClientDriver.class.getName());
        dataSource.setUrl("jdbc:derby://localhost:1527/bookstore;create=true");
        dataSource.setUsername("app");
        dataSource.setPassword("app");
        return dataSource;
    }

    @Bean
    public BookShop bookShop() {
        JdbcBookShop bookShop = new JdbcBookShop();
        bookShop.setDataSource(dataSource());
        return bookShop;
    }
}

```

To demonstrate the problems that can arise without transaction management, suppose you have the data shown in Tables 11-2, 11-3, and 11-4 entered in your bookshop database.

Table 11-2. Sample Data in the BOOK Table for Testing Transactions

ISBN	BOOK_NAME	PRICE
0001	The First Book	30

Table 11-3. Sample Data in the BOOK_STOCK Table for Testing Transactions

ISBN	STOCK
0001	10

Table 11-4. Sample Data in the ACCOUNT Table for Testing Transactions

USERNAME	BALANCE
user1	20

Then, write the following Main class for purchasing the book with ISBN 0001 by the user user1. Because that user's account has only \$20, the funds are not sufficient to purchase the book.

```

package com.apress.springrecipes.bookshop;

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

```

```

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        BookShop bookShop = (BookShop) context.getBean("bookShop");
        bookShop.purchase("0001", "user1");
    }
}

```

When you run this application, you will encounter a `SQLException`, because the `CHECK` constraint of the `ACCOUNT` table has been violated. This is an expected result because you were trying to debit more than the account balance. However, if you check the stock for this book in the `BOOK_STOCK` table, you will find that it was accidentally deducted by this unsuccessful operation! The reason is that you executed the second SQL statement to deduct the stock before you got an exception in the third statement.

As you can see, the lack of transaction management causes your data to be left in an inconsistent state. To avoid this inconsistency, your three SQL statements for the `purchase()` operation should be executed within a single transaction. Once any of the actions in a transaction fail, the entire transaction should be rolled back to undo all changes made by the executed actions.

Managing Transactions with JDBC Commit and Rollback

When using JDBC to update a database, by default, each SQL statement will be committed immediately after its execution. This behavior is known as *auto-commit*. However, it does not allow you to manage transactions for your operations.

JDBC supports the primitive transaction management strategy of explicitly calling the `commit()` and `rollback()` methods on a connection. But before you can do that, you must turn off auto-commit, which is turned on by default.

```

package com.apress.springrecipes.bookshop;
...
public class JdbcBookShop implements BookShop {
    ...
    public void purchase(String isbn, String username) {
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            conn.setAutoCommit(false);
            ...
            conn.commit();
        } catch (SQLException e) {
            if (conn != null) {
                try {
                    conn.rollback();
                } catch (SQLException e1) {}
            }
            throw new RuntimeException(e);
        }
    }
}

```

```
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }
}
```

The auto-commit behavior of a database connection can be altered by calling the `setAutoCommit()` method.

By default, auto-commit is turned on to commit each SQL statement immediately after its execution. To enable transaction management, you must turn off this default behavior and commit the connection only when all the SQL statements have been executed successfully. If any of the statements go wrong, you must roll back all changes made by this connection.

Now, if you run your application again, the book stock will not be deducted when the user's balance is insufficient to purchase the book.

Although you can manage transactions by explicitly committing and rolling back JDBC connections, the code required for this purpose is boilerplate code that you have to repeat for different methods. Moreover, this code is JDBC specific, so once you have chosen another data access technology, it needs to be changed also. Spring's transaction support offers a set of technology-independent facilities, including transaction managers (e.g., `org.springframework.transaction.PlatformTransactionManager`), a transaction template (e.g., `org.springframework.transaction.support.TransactionTemplate`), and transaction declaration support to simplify your transaction management tasks.

11-2. Choosing a Transaction Manager Implementation

Problem

Typically, if your application involves only a single data source, you can simply manage transactions by calling the `commit()` and `rollback()` methods on a database connection. However, if your transactions extend across multiple data sources or you prefer to make use of the transaction management capabilities provided by your Java EE application server, you may choose the Java Transaction API (JTA). Besides, you may have to call different proprietary transaction APIs for different object/relational mapping frameworks such as Hibernate and JPA.

As a result, you have to deal with different transaction APIs for different technologies. It would be hard for you to switch from one set of APIs to another.

Solution

Spring abstracts a general set of transaction facilities from different transaction management APIs. As an application developer, you can simply utilize Spring's transaction facilities without having to know much about the underlying transaction APIs. With these facilities, your transaction management code will be independent of any specific transaction technology.

Spring's core transaction management abstraction is based on the interface `PlatformTransactionManager`. It encapsulates a set of technology-independent methods for transaction management. Remember that a transaction manager is needed no matter which transaction management strategy (programmatic or declarative) you choose in Spring. The `PlatformTransactionManager` interface provides three methods for working with transactions:

- `TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException`
 - `void commit(TransactionStatus status) throws TransactionException;`
 - `void rollback(TransactionStatus status) throws TransactionException;`

How It Works

`PlatformTransactionManager` is a general interface for all Spring transaction managers. Spring has several built-in implementations of this interface for use with different transaction management APIs:

- If you have to deal with only a single data source in your application and access it with JDBC, `DataSourceTransactionManager` should meet your needs.
- If you are using JTA for transaction management on a Java EE application server, you should use `JtaTransactionManager` to look up a transaction from the application server. Additionally, `JtaTransactionManager` is appropriate for distributed transactions (transactions that span multiple resources). Note that while it's common to use a JTA transaction manager to integrate the application servers' transaction manager, there's nothing stopping you from using a stand-alone JTA transaction manager such as Atomikos.
- If you are using an object/relational mapping framework to access a database, you should choose a corresponding transaction manager for this framework, such as `HibernateTransactionManager` and `JpaTransactionManager`.

Figure 11-2 shows the common implementations of the `PlatformTransactionManager` interface in Spring.

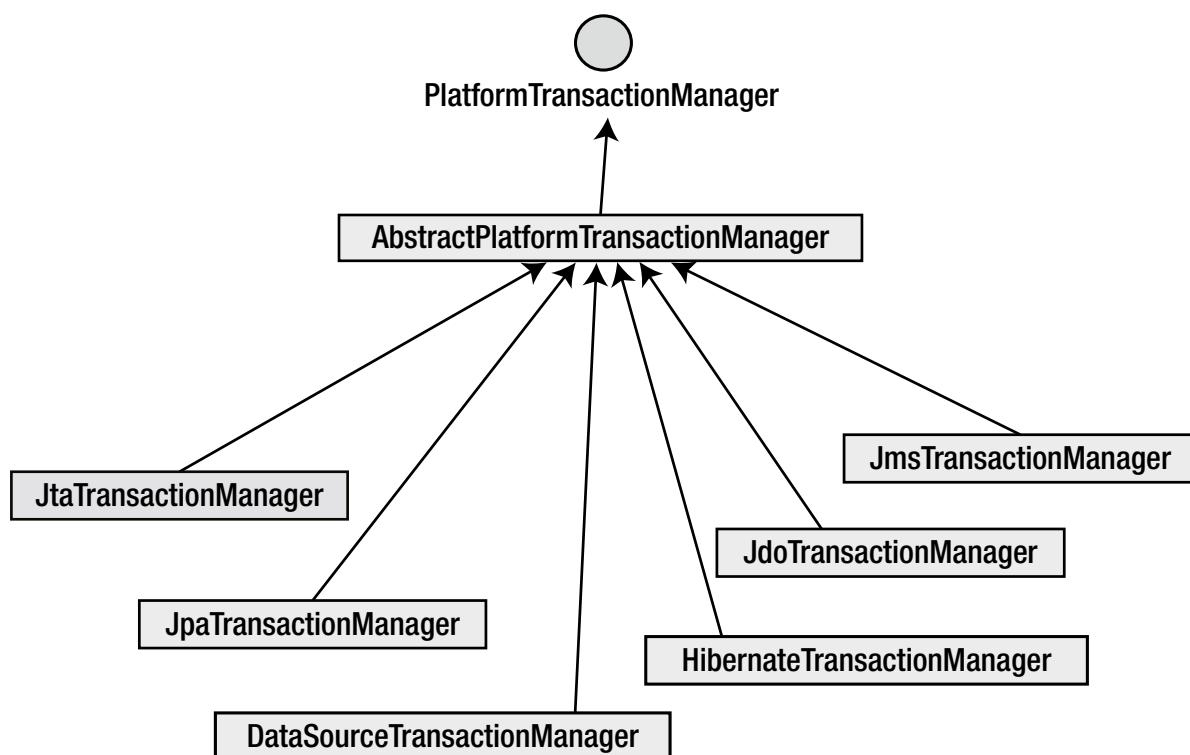


Figure 11-2. Common implementations of the `PlatformTransactionManager` interface

A transaction manager is declared in the Spring IoC container as a normal bean. For example, the following bean configuration declares a `DataSourceTransactionManager` instance. It requires the `dataSource` property to be set so that it can manage transactions for connections made by this data source.

```
@Bean
public PlatformTransactionManager transactionManager() {
    DataSourceTransactionManager transactionManager = new DataSourceTransactionManager()
        transactionManager.setDataSource(dataSource());
    return transactionManager;
}
```

11-3. Managing Transactions Programmatically with the Transaction Manager API

Problem

You need to precisely control when to commit and roll back transactions in your business methods, but you don't want to deal with the underlying transaction API directly.

Solution

Spring's transaction manager provides a technology-independent API that allows you to start a new transaction (or obtain the currently active transaction) by calling the `getTransaction()` method and manage it by calling the `commit()` and `rollback()` methods. Because `PlatformTransactionManager` is an abstract unit for transaction management, the methods you called for transaction management are guaranteed to be technology independent.

How It Works

To demonstrate how to use the transaction manager API, let's create a new class, `TransactionalJdbcBookShop`, which will make use of the Spring JDBC template. Because it has to deal with a transaction manager, you add a property of type `PlatformTransactionManager` and allow it to be injected via a setter method.

```
package com.apress.springrecipes.bookshop;

import org.springframework.dao.DataAccessViolationException;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```

```
public void purchase(String isbn, String username) {
    TransactionDefinition def = new DefaultTransactionDefinition();
    TransactionStatus status = transactionManager.getTransaction(def);

    try {
        int price = getJdbcTemplate().queryForInt(
            "SELECT PRICE FROM BOOK WHERE ISBN = ?",
            new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
            "WHERE ISBN = ?", new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
            "WHERE USERNAME = ?",
            new Object[] { price, username });

        transactionManager.commit(status);
    } catch (DataAccessException e) {
        transactionManager.rollback(status);
        throw e;
    }
}
```

Before you start a new transaction, you have to specify the transaction attributes in a transaction definition object of type `TransactionDefinition`. For this example, you can simply create an instance of `DefaultTransactionDefinition` to use the default transaction attributes.

Once you have a transaction definition, you can ask the transaction manager to start a new transaction with that definition by calling the `getTransaction()` method. Then, it will return a `TransactionStatus` object to keep track of the transaction status. If all the statements execute successfully, you ask the transaction manager to commit this transaction by passing in the transaction status. Because all exceptions thrown by the Spring JDBC template are subclasses of `DataAccessException`, you ask the transaction manager to roll back the transaction when this kind of exception is caught.

In this class, you have declared the transaction manager property of the general type `PlatformTransactionManager`. Now, you have to inject an appropriate transaction manager implementation. Because you are dealing with only a single data source and accessing it with JDBC, you should choose `DataSourceTransactionManager`. Here, you also wire a `dataSource` because the class is a subclass of Spring's `JdbcDaoSupport`, which requires it.

```
@Configuration
public class BookstoreConfiguration {
...
    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource());
        return transactionManager;
    }
}
```

```

@Bean
public BookShop bookShop() {
    TransactionalJdbcBookShop bookShop = new TransactionalJdbcBookShop();
    bookShop.setDataSource(dataSource());
bookShop.setTransactionManager(transactionManager());
    return bookShop;
}
}

```

11-4. Managing Transactions Programmatically with a Transaction Template

Problem

Suppose that you have a code block, but not the entire body, of a business method that has the following transaction requirements:

- Start a new transaction at the beginning of the block.
- Commit the transaction after the block completes successfully.
- Roll back the transaction if an exception is thrown in the block.

If you call Spring's transaction manager API directly, the transaction management code can be generalized in a technology-independent manner. However, you may not want to repeat the boilerplate code for each similar code block.

Solution

As with the JDBC template, Spring also provides a `TransactionTemplate` to help you control the overall transaction management process and transaction exception handling. You just have to encapsulate your code block in a callback class that implements the `TransactionCallback<T>` interface and pass it to the `TransactionTemplate`'s `execute` method for execution. In this way, you don't need to repeat the boilerplate transaction management code for this block. The template objects that Spring provides are lightweight and usually can be discarded or re-created with no performance impact. A JDBC template can be re-created on the fly with a `DataSource` reference, for example, and so too can a `TransactionTemplate` be re-created by providing a reference to a transaction manager. You can, of course, simply create one in your Spring application context, too.

How It Works

A `TransactionTemplate` is created on a transaction manager just as a JDBC template is created on a data source. A transaction template executes a transaction callback object that encapsulates a transactional code block. You can implement the callback interface either as a separate class or as an inner class. If it's implemented as an inner class, you have to make the method arguments final for it to access.

```

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

```

```

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void purchase(final String isbn, final String username) {
        TransactionTemplate transactionTemplate =
            new TransactionTemplate(transactionManager);

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {

            protected void doInTransactionWithoutResult(
                TransactionStatus status) {

                int price = getJdbcTemplate().queryForObject(
                    "SELECT PRICE FROM BOOK WHERE ISBN = ?",
                    new Object[] { isbn }, Integer.class);

                getJdbcTemplate().update(
                    "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
                    "WHERE ISBN = ?", new Object[] { isbn });

                getJdbcTemplate().update(
                    "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
                    "WHERE USERNAME = ?",
                    new Object[] { price, username });
            }
        });
    }
}

```

A `TransactionTemplate` can accept a transaction callback object that implements either the `TransactionCallback<T>` or an instance of the one implementer of that interface provided by the framework, the `TransactionCallbackWithoutResult` class. For the code block in the `purchase()` method for deducting the book stock and account balance, there's no result to be returned, so `TransactionCallbackWithoutResult` is fine. For any code blocks with return values, you should implement the `TransactionCallback<T>` interface instead. The return value of the callback object will finally be returned by the template's `T execute()` method. The main benefit is that the responsibility of starting, rolling back, or committing the transaction has been removed.

During the execution of the callback object, if it throws an unchecked exception (e.g., `RuntimeException` and `DataAccessException` fall into this category), or if you explicitly called `setRollbackOnly()` on the `TransactionStatus` argument in the `doInTransactionWithoutResult` method, the transaction will be rolled back. Otherwise, it will be committed after the callback object completes.

In the bean configuration file, the bookshop bean still requires a transaction manager to create a TransactionTemplate.

```
@Configuration
public class BookstoreConfiguration {
    ...
    @Bean
    public PlatformTransactionManager transactionManager() {
        DataSourceTransactionManager transactionManager = new DataSourceTransactionManager();
        transactionManager.setDataSource(dataSource());
        return transactionManager;
    }

    @Bean
    public BookShop bookShop() {
        TransactionalJdbcBookShop bookShop = new TransactionalJdbcBookShop();
        bookShop.setDataSource(dataSource());
        bookShop.setTransactionManager(transactionManager());
        return bookShop;
    }
}
```

You can also have the IoC container inject a transaction template instead of creating it directly. Because a transaction template handles all transactions, there's no need for your class to refer to the transaction manager any more.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.support.TransactionTemplate;

public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {

    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(
        TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    public void purchase(final String isbn, final String username) {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            protected void doInTransactionWithoutResult(TransactionStatus status) {
                ...
            }
        });
    }
}
```

Then you define a transaction template in the bean configuration file and inject it, instead of the transaction manager, into your bookshop bean. Notice that the transaction template instance can be used for more than one transactional bean because it is a thread-safe object. Finally, don't forget to set the transaction manager property for your transaction template.

```
package com.apress.springrecipes.bookshop.config;
...
import org.springframework.transaction.support.TransactionTemplate;

@Configuration
public class BookstoreConfiguration {
...
    @Bean
    public PlatformTransactionManager transactionManager() { ... }

    @Bean
    public TransactionTemplate transactionTemplate() {
        TransactionTemplate transactionTemplate = new TransactionTemplate();
        transactionTemplate.setTransactionManager(transactionManager());
        return transactionTemplate;
    }

    @Bean
    public BookShop bookShop() {
        TransactionalJdbcBookShop bookShop = new TransactionalJdbcBookShop();
        bookShop.setDataSource(dataSource());
        bookShop.setTransactionTemplate(transactionTemplate());
        return bookShop;
    }
}
```

11-5. Managing Transactions Declaratively with Transaction Advices

Problem

Because transaction management is a kind of crosscutting concern, you should manage transactions declaratively with the AOP approach available from Spring 2.x onward. Managing transactions manually can be tedious and error prone. It is simpler to specify, declaratively, what behavior you are expecting and to not prescribe *how* that behavior is to be achieved.

Solution

Spring (since version 2.0) offers a transaction advice that can be easily configured via the `<tx:advice>` element defined in the `tx` schema. This advice can be enabled with the AOP configuration facilities defined in the `aop` schema.

How It Works

To enable declarative transaction management, you can declare a transaction advice via the `<tx:advice>` element defined in the tx schema, so you have to add this schema definition to the `<beans>` root element beforehand. Once you have declared this advice, you need to associate it with a pointcut. Because a transaction advice is declared outside the `<aop:config>` element, it cannot link with a pointcut directly. You have to declare an advisor in the `<aop:config>` element to associate an advice with a pointcut.

Note Because Spring AOP uses the AspectJ pointcut expressions to define pointcuts, you have to include the AspectJ Weaver support on your CLASSPATH. If you're using Maven, add the following dependency to your project.

```

<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.0</version>
</dependency>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

    <tx:advice id="bookShopTxAdvice"
        transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="purchase"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <aop:pointcut id="bookShopOperation" expression=
            "execution(* com.apress.springrecipes.bookshop.BookShop.*(..))"/>
        <aop:advisor advice-ref="bookShopTxAdvice"
            pointcut-ref="bookShopOperation"/>
    </aop:config>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

```

```

<bean id="bookShop"
      class="com.apress.springrecipes.bookshop.TransactionalJdbcBookShop">
    <property name="dataSource" ref="dataSource"/>
  </bean>
</beans>

```

The preceding AspectJ pointcut expression matches all the methods declared in the BookShop interface. However, because Spring AOP is based on proxies, it can apply only to public methods. Thus only public methods can be made transactional with Spring AOP.

Each transaction advice requires an identifier and a reference to a transaction manager in the IoC container. If you don't specify a transaction manager explicitly, Spring will search the application context for a TransactionManager with a bean name of `transactionManager`. The methods that require transaction management are specified with multiple `<tx:method>` elements inside the `<tx:attributes>` element. The method name supports wildcards for you to match a group of methods. You can also define transaction attributes for each group of methods, but let's use the default attributes for simplicity's sake. The defaults are shown in Table 11-5.

Table 11-5. Attributes Used with `tx:attributes`

Attribute	Required	Default	Description
<code>name</code>	Yes	n/a	The name of the methods against which the advice will be applied. You can use wildcards (*).
<code>propagation</code>	No	REQUIRED	The propagation specification for the transaction.
<code>isolation</code>	No	DEFAULT	The isolation level specification for the transaction.
<code>timeout</code>	No	-1	How long (in seconds) the transaction will attempt to commit before it times out.
<code>read-only</code>	No	False	Tells the container whether the transaction is read-only or not. This is a Spring-specific setting. If you're used to standard Java EE transaction configuration, you won't have seen this setting before. Its meaning is different for different resources (e.g., databases have a different notion of "read-only" than a JMS queue does).
<code>rollback-for</code>	No	N/A	Comma-delimited list of fully qualified Exception types that, when thrown from the method, the transaction should roll back for.
<code>no-rollback-for</code>	No	N/A	A comma-delimited list of Exception types that, when thrown from the method, the transaction should ignore and not roll back for.

Now, you can retrieve the `bookShop` bean from the Spring IoC container to use. Because this bean's methods are matched by the pointcut, Spring will return a proxy that has transaction management enabled for this bean.

```

package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("bookstore-context.xml");

        BookShop bookShop = context.getBean(BookShop.class);
        bookShop.purchase("0001", "user1");
    }
}

```

11-6. Managing Transactions Declaratively with the @Transactional Annotation

Problem

Declaring transactions in the bean configuration file requires knowledge of AOP concepts such as pointcuts, advices, and advisors. Developers who lack this knowledge might find it hard to enable declarative transaction management.

Solution

In addition to declaring transactions in the bean configuration file with pointcuts, advices, and advisors, Spring allows you to declare transactions simply by annotating your transactional methods with `@Transactional` and enabling the `<tx:annotation-driven>` element. However, Java 1.5 or higher is required to use this approach. Note that although you could apply the annotation to an interface method, it's not a recommended practice.

How It Works

To define a method as transactional, you can simply annotate it with `@Transactional`. Note that you should only annotate public methods due to the proxy-based limitations of Spring AOP.

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Transactional;
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {

    @Transactional
    public void purchase(String isbn, String username) {

        int price = getJdbcTemplate().queryForInt(
            "SELECT PRICE FROM BOOK WHERE ISBN = ?",
            new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 "+
            "WHERE ISBN = ?", new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE ACCOUNT SET BALANCE = BALANCE - ? "+
            "WHERE USERNAME = ?", new Object[] { price, username });
    }
}
```

Note that, as we are extending `JdbcDaoSupport`, we no longer need the mutators for the `DataSource`; remove it from your DAO class.

You may apply the `@Transactional` annotation at the method level or the class level. When applying this annotation to a class, all of the public methods within this class will be defined as transactional. Although you can apply `@Transactional` to interfaces or method declarations in an interface, it's not recommended because it may not work properly with class-based proxies (i.e., CGLIB proxies).

In the bean configuration file, you only have to enable the `<tx:annotation-driven>` element and specify a transaction manager for it. That's all you need to make it work. Spring will advise methods with `@Transactional`, or methods in a class with `@Transactional`, from beans declared in the IoC container. As a result, Spring can manage transactions for these methods.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

    <tx:annotation-driven transaction-manager="transactionManager"/>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.spring.JdbcBookShop">
        <property name="dataSource" ref="dataSource"/>
    </bean>
</beans>
```

In fact, you can omit the `transaction-manager` attribute in the `<tx:annotation-driven>` element if your transaction manager has the name `transactionManager`. This element will automatically detect a transaction manager with this name. You have to specify a transaction manager only when it has a different name.

```
<beans ...>
    <tx:annotation-driven />
    ...
</beans>
```

Enable `@Transactional` with Java-based Configuration

When using Java-based configuration transaction management can be enabled by using the `@EnableTransactionManagement` annotation. This annotation is the equivalent of the `<tx:annotation-driven />` in XML based configuration.

```
@Configuration
@EnableTransactionManagement
public class BookstoreConfiguration { ... }
```

When using the `@EnableTransactionManagement` annotation it isn't possible to specify which transaction manager to use. When there is only one `PlatformTransactionManager` in your configuration this will be automatically detected and used for managing transactions.

For the situations where there are multiple PlatformTransactionManagers in your configuration, you have to specify on the @Transactional annotation which of the transaction managers to use for controlling the transaction. For this you can use the value attribute of the @Transactional annotation.

```
@Transaction("transaction-manager-1")
public void transactionalMethod() { ... }

@Transaction("transaction-manager-2")
public void otherTransactionalMethod() { ... }
```

11-7. Setting the Propagation Transaction Attribute

Problem

When a transactional method is called by another method, it is necessary to specify how the transaction should be propagated. For example, the method may continue to run within the existing transaction, or it may start a new transaction and run within its own transaction.

Solution

A transaction's propagation behavior can be specified by the *propagation* transaction attribute. Spring defines seven propagation behaviors, as shown in Table 11-6. These behaviors are defined in the org.springframework.transaction.TransactionDefinition interface. Note that not all types of transaction managers support all of these propagation behaviors. Their behavior is contingent on the underlying resource. Databases, for example, may support varying isolation levels, which constrains what propagation behaviors the transaction manager can support.

Table 11-6. Propagation Behaviors Supported by Spring

Propagation	Description
REQUIRED	If there's an existing transaction in progress, the current method should run within this transaction. Otherwise, it should start a new transaction and run within its own transaction.
REQUIRES_NEW	The current method must start a new transaction and run within its own transaction. If there's an existing transaction in progress, it should be suspended.
SUPPORTS	If there's an existing transaction in progress, the current method can run within this transaction. Otherwise, it is not necessary to run within a transaction.
NOT_SUPPORTED	The current method should not run within a transaction. If there's an existing transaction in progress, it should be suspended.
MANDATORY	The current method must run within a transaction. If there's no existing transaction in progress, an exception will be thrown.
NEVER	The current method should not run within a transaction. If there's an existing transaction in progress, an exception will be thrown.

(continued)

Table 11-6. (continued)

Propagation	Description
NESTED	If there's an existing transaction in progress, the current method should run within the nested transaction (supported by the JDBC 3.0 save point feature) of this transaction. Otherwise, it should start a new transaction and run within its own transaction. This feature is unique to Spring (whereas the previous propagation behaviors have analogs in Java EE transaction propagation). The behavior is useful for situations such as batch processing, in which you've got a long running process (imagine processing 1 million records) and you want to chunk the commits on the batch. So you commit every 10,000 records. If something goes wrong, you roll back the nested transaction and you've lost only 10,000 records' worth of work (as opposed to the entire 1 million).

How It Works

Transaction propagation happens when a transactional method is called by another method. For example, suppose a customer would like to check out all books to purchase at the bookshop cashier. To support this operation, you define the `Cashier` interface as follows:

```
package com.apress.springrecipes.bookshop;
...
public interface Cashier {
    public void checkout(List<String> isbns, String username);
}
```

You can implement this interface by delegating the purchases to a bookshop bean by calling its `purchase()` method multiple times. Note that the `checkout()` method is made transactional by applying the `@Transactional` annotation.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {
    private BookShop bookShop;

    public void setBookShop(BookShop bookShop) {
        this.bookShop = bookShop;
    }

    @Transactional
    public void checkout(List<String> isbns, String username) {
        for (String isbn : isbns) {
            bookShop.purchase(isbn, username);
        }
    }
}
```

Then define a cashier bean in your bean configuration file and refer to the bookshop bean for purchasing books.

```
@Configuration
@EnableTransactionManagement()
public class BookstoreConfiguration {
    ...
    @Bean
    public Cashier cashier() {
        BookShopCashier cashier = new BookShopCashier();
        cashier.setBookShop(bookShop());
        return cashier;
    }
}
```

To illustrate the propagation behavior of a transaction, enter the data shown in Tables 11-7, 11-8, and 11-9 in your bookshop database.

Table 11-7. Sample Data in the BOOK Table for Testing Propagation Behaviors

ISBN	BOOK_NAME	PRICE
0001	The First Book	30
0002	The Second Book	50

Table 11-8. Sample Data in the BOOK_STOCK Table for Testing Propagation Behaviors

ISBN	STOCK
0001	10
0002	10

Table 11-9. Sample Data in the ACCOUNT Table for Testing Propagation Behaviors

USERNAME	BALANCE
user1	40

The REQUIRED Propagation Behavior

When the user user1 checks out the two books from the cashier, the balance is sufficient to purchase the first book but not the second.

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {
```

```

public static void main(String[] args) {
    ...
    Cashier cashier = context.getBean(Cashier.class);
    List<String> isbnList = Arrays.asList(new String[] { "0001", "0002" });
    cashier.checkout(isbnList, "user1");
}
}

```

When the bookshop's `purchase()` method is called by another transactional method, such as `checkout()`, it will run within the existing transaction by default. This default propagation behavior is called REQUIRED. That means there will be only one transaction whose boundary is the beginning and ending of the `checkout()` method. This transaction will be committed only at the end of the `checkout()` method. As a result, the user can purchase none of the books. Figure 11-3 illustrates the REQUIRED propagation behavior.

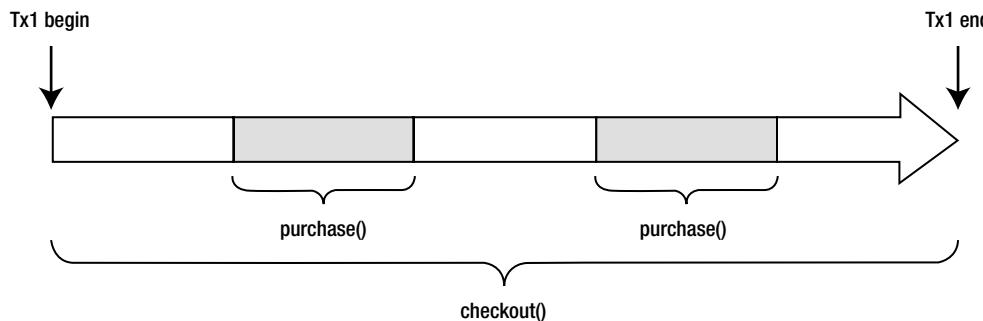


Figure 11-3. The REQUIRED transaction propagation behavior

However, if the `purchase()` method is called by a non-transactional method and there's no existing transaction in progress, it will start a new transaction and run within its own transaction.

The propagation transaction attribute can be defined in the `@Transactional` annotation. For example, you can set the REQUIRED behavior for this attribute as follows. In fact, this is unnecessary, because it's the default behavior.

```

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    @Transactional(propagation = Propagation.REQUIRED)
    public void purchase(String isbn, String username) {
        ...
    }
}

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

```

```
public class BookShopCashier implements Cashier {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void checkout(List<String> isbn, String username) {
        ...
    }
}
```

The REQUIRES_NEW Propagation Behavior

Another common propagation behavior is `REQUIRES_NEW`. It indicates that the method must start a new transaction and run within its new transaction. If there's an existing transaction in progress, it should be suspended first (as, for example, with the `checkout` method on `BookShopCashier`, with a propagation of `REQUIRED`).

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

In this case, there will be three transactions started in total. The first transaction is started by the `checkout()` method, but when the first `purchase()` method is called, the first transaction will be suspended and a new transaction will be started. At the end of the first `purchase()` method, the new transaction completes and commits. When the second `purchase()` method is called, another new transaction will be started. However, this transaction will fail and roll back. As a result, the first book will be purchased successfully, while the second will not. Figure 11-4 illustrates the `REQUIRES_NEW` propagation behavior.

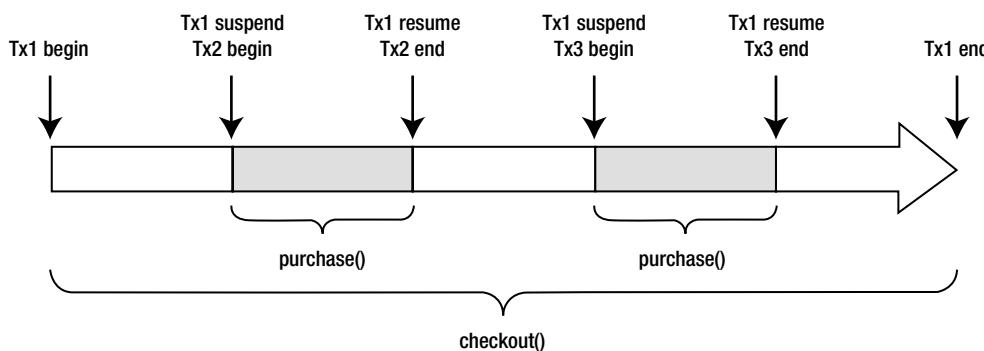


Figure 11-4. The `REQUIRES_NEW` transaction propagation behavior

Setting the Propagation Attribute in Transaction Advices and APIs

In a Spring transaction advice, the propagation transaction attribute can be specified in the `<tx:method>` element as follows:

```
<tx:advice ...>
  <tx:attributes>
    <tx:method name="..." propagation="REQUIRES_NEW"/>
  </tx:attributes>
</tx:advice>
```

In Spring's transaction management API, the propagation transaction attribute can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
```

11-8. Setting the Isolation Transaction Attribute

Problem

When multiple transactions of the same application or different applications are operating concurrently on the same dataset, many unexpected problems may arise. You must specify how you expect your transactions to be isolated from one another.

Solution

The problems caused by concurrent transactions can be categorized into four types:

- *Dirty read*: For two *transactions* T1 and T2, T1 reads a field that has been updated by T2 but not yet committed. Later, if T2 rolls back, the field read by T1 will be temporary and invalid.
- *Nonrepeatable read*: For two transactions T1 and T2, T1 reads a field and then T2 updates the field. Later, if T1 reads the same field again, the value will be different.
- *Phantom read*: For two transactions T1 and T2, T1 reads some rows from a table and then T2 inserts new rows into the table. Later, if T1 reads the same table again, there will be additional rows.
- *Lost updates*: For two transactions T1 and T2, they both select a row for update, and based on the state of that row, make an update to it. Thus, one overwrites the other when the second transaction to commit should have waited until the first one committed before performing its selection.

In theory, transactions should be completely isolated from each other (i.e., serializable) to avoid all the mentioned problems. However, this isolation level will have great impact on performance, because transactions have to run in serial order. In practice, transactions can run in lower isolation levels in order to improve performance.

A transaction's isolation level can be specified by the *isolation* transaction attribute. Spring supports five isolation levels, as shown in Table 11-10. These levels are defined in the `org.springframework.transaction.TransactionDefinition` interface.

Table 11-10. Isolation Levels Supported by Spring

Isolation	Description
DEFAULT	Uses the default isolation level of the underlying database. For most databases, the default isolation level is READ_COMMITTED.
READ_UNCOMMITTED	Allows a transaction to read uncommitted changes by other transactions. The dirty read, nonrepeatable read, and phantom read problems may occur.
READ_COMMITTED	Allows a transaction to read only those changes that have been committed by other transactions. The dirty read problem can be avoided, but the nonrepeatable read and phantom read problems may still occur.
REPEATABLE_READ	Ensures that a transaction can read identical values from a field multiple times. For the duration of this transaction, updates made by other transactions to this field are prohibited. The dirty read and nonrepeatable read problems can be avoided, but the phantom read problem may still occur.
SERIALIZABLE	Ensures that a transaction can read identical rows from a table multiple times. For the duration of this transaction, inserts, updates, and deletes made by other transactions to this table are prohibited. All the concurrency problems can be avoided, but the performance will be low.

Note that transaction isolation is supported by the underlying database engine but not an application or a framework. However, not all database engines support all these isolation levels. You can change the isolation level of a JDBC connection by calling the `setTransactionIsolation()` method on the `java.sql.Connection` interface.

How It Works

To illustrate the problems caused by concurrent transactions, let's add two new operations to your bookshop for increasing and checking the book stock.

```
package com.apress.springrecipes.bookshop;

public interface BookShop {
    ...
    public void increaseStock(String isbn, int stock);
    public int checkStock(String isbn);
}
```

Then, you implement these operations as follows. Note that these two operations should also be declared as transactional.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
}
```

```

@Transactional
public void increaseStock(String isbn, int stock) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + "- Prepare to increase book stock");

    getJdbcTemplate().update(
        "UPDATE BOOK_STOCK SET STOCK = STOCK + ? " +
        "WHERE ISBN = ?",
        new Object[] { stock, isbn });

    System.out.println(threadName + "- Book stock increased by " + stock);
    sleep(threadName);

    System.out.println(threadName + "- Book stock rolled back");
    throw new RuntimeException("Increased by mistake");
}

@Transactional
public int checkStock(String isbn) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName + "- Prepare to check book stock");

    int stock = getJdbcTemplate().queryForInt(
        "SELECT STOCK FROM BOOK_STOCK WHERE ISBN = ?",
        new Object[] { isbn });

    System.out.println(threadName + "- Book stock is " + stock);
    sleep(threadName);

    return stock;
}

private void sleep(String threadName) {
    System.out.println(threadName + "- Sleeping");
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {}
    System.out.println(threadName + "- Wake up");
}
}

```

To simulate concurrency, your operations need to be executed by multiple threads. You can track the current status of the operations through the `println` statements. For each operation, you print a couple of messages to the console around the SQL statement's execution. The messages should include the thread name for you to know which thread is currently executing the operation.

After each operation executes the SQL statement, you ask the thread to sleep for 10 seconds. As you know, the transaction will be committed or rolled back immediately once the operation completes. Inserting a sleep statement can help to postpone the commit or rollback. For the `increase()` operation, you eventually throw a `RuntimeException` to cause the transaction to roll back. Let's look at a simple client that runs these examples.

Before you start with the isolation level examples, enter the data from Tables 11-11 and 11-12 into your bookshop database. (Note that the `ACCOUNT` table isn't needed in this example.)

Table 11-11. Sample Data in the BOOK Table for Testing Isolation Levels

ISBN	BOOK_NAME	PRICE
0001	The First Book	30

Table 11-12. Sample Data in the BOOK_STOCK Table for Testing Isolation Levels

ISBN	STOCK
0001	10

The READ_UNCOMMITTED and READ_COMMITTED Isolation Levels

READ_UNCOMMITTED is the lowest isolation level that allows a transaction to read uncommitted changes made by other transactions. You can set this isolation level in the @Transaction annotation of your checkStock() method.

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}
```

You can create some threads to experiment on this transaction isolation level. In the following Main class, there are two threads you are going to create. Thread 1 increases the book stock, while thread 2 checks the book stock. Thread 1 starts 5 seconds before thread 2.

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = context.getBean(BookShop.class);

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 1");
    }
}
```

```

Thread thread2 = new Thread(new Runnable() {
    public void run() {
        bookShop.checkStock("0001");
    }
}, "Thread 2");

thread1.start();
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {}
thread2.start();
}
}

```

If you run the application, you will get the following result:

```

Thread 1—Prepare to increase book stock
Thread 1—Book stock increased by 5
Thread 1—Sleeping
Thread 2—Prepare to check book stock
Thread 2—Book stock is 15
Thread 2—Sleeping
Thread 1—Wake up
Thread 1—Book stock rolled back
Thread 2—Wake up

```

First, thread 1 increased the book stock and then went to sleep. At that time, thread 1's transaction had not yet been rolled back. While thread 1 was sleeping, thread 2 started and attempted to read the book stock. With the READ_UNCOMMITTED isolation level, thread 2 would be able to read the stock value that had been updated by an uncommitted transaction.

However, when thread 1 wakes up, its transaction will be rolled back due to a `RuntimeException`, so the value read by thread 2 is temporary and invalid. This problem is known as *dirty read*, because a transaction may read values that are “dirty.”

To avoid the dirty read problem, you should raise the isolation level of `checkStock()` to READ_COMMITTED.

```

package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_COMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}

```

If you run the application again, thread 2 won't be able to read the book stock until thread 1 has rolled back the transaction. In this way, the dirty read problem can be avoided by preventing a transaction from reading a field that has been updated by another uncommitted transaction.

```
Thread 1—Prepare to increase book stock
Thread 1—Book stock increased by 5
Thread 1—Sleeping
Thread 2—Prepare to check book stock
Thread 1—Wake up
Thread 1—Book stock rolled back
Thread 2—Book stock is 10
Thread 2—Sleeping
Thread 2—Wake up
```

In order for the underlying database to support the `READ_COMMITTED` isolation level, it may acquire an *update lock* on a row that was updated but not yet committed. Then, other transactions must wait to read that row until the update lock is released, which happens when the locking transaction commits or rolls back.

The REPEATABLE_READ Isolation Level

Now, let's restructure the threads to demonstrate another concurrency problem. Swap the tasks of the two threads so that thread 1 checks the book stock before thread 2 increases the book stock.

```
package com.apress.springrecipes.bookshop.spring;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 1");

        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                } catch (RuntimeException e) {}
            }
        }, "Thread 2");

        thread1.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {}
        thread2.start();
    }
}
```

If you run the application, you will get the following result:

```
Thread 1—Prepare to check book stock
Thread 1—Book stock is 10
Thread 1—Sleeping
Thread 2—Prepare to increase book stock
Thread 2—Book stock increased by 5
Thread 2—Sleeping
Thread 1—Wake up
Thread 2—Wake up
Thread 2—Book stock rolled back
```

First, thread 1 read the book stock and then went to sleep. At that time, thread 1's transaction had not yet been committed. While thread 1 was sleeping, thread 2 started and attempted to increase the book stock. With the READ_COMMITTED isolation level, thread 2 would be able to update the stock value that was read by an uncommitted transaction.

However, if thread 1 reads the book stock again, the value will be different from its first read. This problem is known as *nonrepeatable read* because a transaction may read different values for the same field.

To avoid the nonrepeatable read problem, you should raise the isolation level of `checkStock()` to REPEATABLE_READ.

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public int checkStock(String isbn) {
        ...
    }
}
```

If you run the application again, thread 2 won't be able to update the book stock until thread 1 has committed the transaction. In this way, the nonrepeatable read problem can be avoided by preventing a transaction from updating a value that has been read by another uncommitted transaction.

```
Thread 1—Prepare to check book stock
Thread 1—Book stock is 10
Thread 1—Sleeping
Thread 2—Prepare to increase book stock
Thread 1—Wake up
Thread 2—Book stock increased by 5
Thread 2—Sleeping
Thread 2—Wake up
Thread 2—Book stock rolled back
```

In order for the underlying database to support the REPEATABLE_READ isolation level, it may acquire a *read lock* on a row that was read but not yet committed. Then, other transactions must wait to update the row until the read lock is released, which happens when the locking transaction commits or rolls back.

The SERIALIZABLE Isolation Level

After a transaction has read several rows from a table, another transaction inserts new rows into the same table. If the first transaction reads the same table again, it will find additional rows that are different from the first read. This problem is known as *phantom read*. Actually, phantom read is very similar to nonrepeatable read but involves multiple rows.

To avoid the phantom read problem, you should raise the isolation level to the highest: SERIALIZABLE. Notice that this isolation level is the slowest because it may acquire a read lock on the full table. In practice, you should always choose the lowest isolation level that can satisfy your requirements.

Setting the Isolation Level Attribute in Transaction Advices and APIs

In a Spring transaction advice, the isolation level can be specified in the `<tx:method>` element as follows:

```
<tx:advice ...>
  <tx:attributes>
    <tx:method name="*"
      isolation="REPEATABLE_READ"/>
  </tx:attributes>
</tx:advice>
```

In Spring's transaction management API, the isolation level can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);
```

11-9. Setting the Rollback Transaction Attribute

Problem

By default, only unchecked exceptions (i.e., of type `RuntimeException` and `Error`) will cause a transaction to roll back, while checked exceptions will not. Sometimes, you may wish to break this rule and set your own exceptions for rolling back.

Solution

The exceptions that cause a transaction to roll back or not can be specified by the *rollback* transaction attribute. Any exceptions not explicitly specified in this attribute will be handled by the default rollback rule (i.e., rolling back for unchecked exceptions and not rolling back for checked exceptions).

How It Works

A transaction's rollback rule can be defined in the `@Transactional` annotation via the `rollbackFor` and `noRollbackFor` attributes. These two attributes are declared as `Class[]`, so you can specify more than one exception for each attribute.

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import java.io.IOException;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        propagation = Propagation.REQUIRES_NEW,
        rollbackFor = IOException.class,
        noRollbackFor = ArithmeticException.class)
    public void purchase(String isbn, String username) throws Exception{
        throw new ArithmeticException();
        //throw new IOException();
    }
}
```

In a Spring transaction advice, the rollback rule can be specified in the `<tx:method>` element. You can separate the exceptions with commas if there's more than one exception.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="...">
            rollback-for="java.io.IOException"
            no-rollback-for="java.lang.ArithmeticException"/>
        ...
    </tx:attributes>
</tx:advice>
```

In Spring's transaction management API, the rollback rule can be specified in a `RuleBasedTransactionAttribute` object. Because it implements the `TransactionDefinition` interface, it can be passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
RuleBasedTransactionAttribute attr = new RuleBasedTransactionAttribute();
attr.getRollbackRules().add(
    new RollbackRuleAttribute(IOException.class));
attr.getRollbackRules().add(
    new NoRollbackRuleAttribute(SendFailedException.class));
```

11-10. Setting the Timeout and Read-Only Transaction Attributes

Problem

Because a transaction may acquire locks on rows and tables, a long transaction will tie up resources and have an impact on overall performance. Besides, if a transaction only reads but does not update data, the database engine could optimize this transaction. You can specify these attributes to increase the performance of your application.

Solution

The *timeout* transaction attribute (an integer that describes seconds) indicates how long your transaction can survive before it is forced to roll back. This can prevent a long transaction from tying up resources. The *read-only* attribute indicates that this transaction will only read but not update data. The read-only flag is just a hint to enable a resource to optimize the transaction, and a resource might not necessarily cause a failure if a write is attempted.

How It Works

The timeout and read-only transaction attributes can be defined in the @Transactional annotation. Note that timeout is measured in seconds.

```
package com.apress.springrecipes.bookshop.spring;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        isolation = Isolation.REPEATABLE_READ,
        timeout = 30,
        readOnly = true)
    public int checkStock(String isbn) {
        ...
    }
}
```

In a Spring 2.0 transactional advice, the timeout and read-only transaction attributes can be specified in the `<tx:method>` element.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="checkStock"
            timeout="30"
            read-only="true"/>

```

In Spring's transaction management API, the timeout and read-only transaction attributes can be specified in a `DefaultTransactionDefinition` object and then passed to a transaction manager's `getTransaction()` method or a transaction template's constructor.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setTimeout(30);
def.setReadOnly(true);
```

11-11. Managing Transactions with Load-Time Weaving

Problem

By default, Spring's declarative transaction management is enabled via its AOP framework. However, as Spring AOP can only advise public methods of beans declared in the IoC container, you are restricted to managing transactions within this scope using Spring AOP. Sometimes, you may wish to manage transactions for nonpublic methods, or methods of objects created outside the Spring IoC container (e.g., domain objects).

Solution

Spring provides an AspectJ aspect named `AnnotationTransactionAspect` that can manage transactions for any methods of any objects, even if the methods are non-public or the objects are created outside the Spring IoC container. This aspect will manage transactions for any methods with the `@Transactional` annotation. You can choose either AspectJ's compile-time weaving or load-time weaving to enable this aspect.

How It Works

To weave this aspect into your domain classes at load time, you have to put the `@EnableLoadTimeWeaving` annotation on your configuration class. To enable Spring's `AnnotationTransactionAspect` for transaction management, you just define the `@EnableTransactionManagement` annotation and set its `mode` attribute to `aspectj`. The `@EnableTransactionManagement` annotation takes two values for the `mode` attribute: `aspectj` and `proxy`. `aspectj` stipulates that the container should use load-time or compile-time weaving to enable the transaction advice. This requires the `spring-instrument` jar to be on the classpath, as well as the appropriate configuration at load time or compile time. Alternatively, `proxy` stipulates that the container should use the Spring AOP mechanisms. It's important to note that the `aspectj` mode doesn't support configuration of the `@Transactional` annotation on interfaces. Then the transaction aspect will automatically get enabled. You also have to provide a transaction manager for this aspect. By default, it will look for a transaction manager whose name is `transactionManager`.

```
package com.apress.springrecipes.bookshop;
```

```
Configuration
@EnableTransactionManagement(mode = AdviceMode.ASPECTJ)
@EnableLoadTimeWeaving
public class BookstoreConfiguration { ... }
```

In XML you add `<context:load-time-weaver />` and specify the `mode` attribute on `<tx:annotation-driven />`.

```
<beans ... >
...
  <context:load-time-weaver />
  <tx:annotation-driven mode="aspectj"/>
</beans>
```

Note To use the Spring aspect library for AspectJ you have to include **spring-aspects** module on your CLASSPATH. To enable loadtime weaving we also have to include a javaagent, this is available in the **spring-instrument** module. If you're using Maven, add the following dependencies to your project.

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

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

For a simple Java application, you can weave this aspect into your classes at load time with the Spring agent specified as a VM argument.

```
java -javaagent:lib/spring-instrument-4.0.5.RELEASE.jar -jar Recipe_12_11_i.jar
```

Summary

This chapter discussed transactions and why you should use them. You explored the approach taken for transaction management historically in Java EE and then learned how the approach the Spring framework offers differs. You explored explicit use of transactions in your code as well as implicit use with annotation-driven aspects. You set up a database and used transactions to enforce valid state in the database.

In the next chapter, you will explore Spring Batch. Spring Batch provides infrastructure and components that can be used as the foundation for batch processing jobs.

CHAPTER 12



Spring Batch

Batch processing has been around for decades. The earliest widespread applications of technology for managing information (information technology) were applications of batch processing. These environments didn't have interactive sessions and usually didn't have the capability to load multiple applications in memory. Computers were expensive and bore no resemblance to today's servers. Typically, machines were multiuser and in use during the day (time-shared). During the evening, however, the machines would sit idle, which was a tremendous waste. Businesses invested in ways to utilize the offline time to do work aggregated through the course of the day. Out of this practice emerged batch processing.

Batch processing solutions typically run offline, indifferent to events in the system. In the past, batch processes ran offline out of necessity. Today, however, most batch processes are run offline because having work done at a predictable time and having chunks of work done is a requirement for a lot of architectures. A batch processing solution doesn't usually respond to requests, although there's no reason it couldn't be started as a consequence of a message or request. Batch processing solutions tend to be used on large datasets where the duration of the processing is a critical factor in its architecture and implementation. A process might run for minutes, hours, or days! Jobs may have unbounded durations (i.e., run until all work is finished, even if this means running for a few days), or they may be strictly bounded (jobs must proceed in constant time, with each row taking the same amount of time regardless of bound, which lets you, say, predict that a given job will finish in a certain time window.)

Batch processing has had a long history that informs even modern batch processing solutions.

Mainframe applications used batch processing, and one of the largest modern day environments for batch processing, CICS on z/OS, is still fundamentally a mainframe operating system. Customer Information Control System (CICS) is very well suited to a particular type of task: take input, process it, and write it to output. CICS is a transaction server used most in financial institutions and government that runs programs in a number of languages (COBOL, C, PLI, and so on). It can easily support thousands of transactions per second. CICS was one of the first containers, a concept familiar to Spring and Java EE users, even though CICS itself debuted in 1969! A CICS installation is very expensive, and although IBM still sells and installs CICS, many other solutions have come along since then. These solutions are usually specific to a particular environment: COBOL/CICS on mainframes, C on Unix, and, today, Java on any number of environments. The problem is that there's very little standardized infrastructure for dealing with these types of batch processing solutions. Very few people are even aware of what they're missing because there's very little native support on the Java platform for batch processing. Businesses that need a solution typically end up writing it in-house, resulting in fragile, domain-specific code.

The pieces are there, however: transaction support, fast I/O, schedulers such as Quartz, and solid threading support, as well as a very powerful concept of an application container in Java EE and Spring. It was only natural that Dave Syer and his team would come along and build Spring Batch, a batch processing solution for the Spring platform.

It's important to think about the kinds of problems this framework solves before diving into the details. A technology is defined by its solution space. A typical Spring Batch application typically reads in a lot of data and then writes it back out in a modified form. Decisions about transactional barriers, input size, concurrency, and order of steps in processing are all dimensions of a typical integration.

A common requirement is loading data from a comma-separated value (CSV) file, perhaps as a business-to-business (B2B) transaction, perhaps as an integration technique with an older legacy application. Another common application is nontrivial processing on records in a database. Perhaps the output is an update of the database record itself. An example might be resizing of images on the file system whose metadata is stored in a database or needing to trigger another process based on some condition.

Note *Fixed-width data* is a format of rows and cells, quite like a CSV file. CSV file cells are separated by commas or tabs however, and fixed-width data works by presuming certain lengths for each value. The first value might be the first nine characters, the second value the next four characters after that, and so on.

Fixed-width data which is often used with legacy or embedded systems, is a fine candidate for batch processing. Processing that deals with a resource that's fundamentally nontransactional (e.g., a web service or a file) begs for batch processing, because batch processing provides retry/skip/fail functionality that most web services will not.

It's also important to understand what Spring Batch *doesn't* do. Spring Batch is a flexible but not all-encompassing solution. Just as Spring doesn't reinvent the wheel when it can be avoided, Spring Batch leaves a few important pieces to the discretion of the implementor. Case in point: Spring Batch provides a generic mechanism by which to launch a job, be it by the command line, a Unix cron, an operating system service, Quartz (discussed in Chapter 14), or in response to an event on an enterprise service bus (for example, the Mule ESB or Spring's own ESB-like solution, Spring Integration, which is discussed in Chapter 16). Another example is the way Spring Batch manages the state of batch processes. Spring Batch requires a durable store. The only useful implementation of a JobRepository (an interface provided by Spring Batch for storing runtime data) requires a database, because a database is transactional and there's no need to reinvent it. To which database you should deploy, however, is largely unspecified, although there are useful defaults provided for you, of course.

Note The JEE7 specification includes JSR-352¹ (Batch Applications for the Java Platform) Spring Batch 3.0 is the reference implementation of this specification.

Runtime Metadata Model

Spring Batch works with a JobRepository, which is the keeper of all the knowledge and metadata for each job (including component parts such as JobInstances, JobExecution, and StepExecution). Each Job is composed of one or more Steps, one after another. With Spring Batch, a Step can conditionally follow another Step, allowing for primitive workflows. These steps can also be concurrent: two steps can run at the same time.

When a job is run, it's often coupled with JobParameters to parameterize the behavior of the Job itself. For example, a job might take a date parameter to determine which records to process. This coupling is called a JobInstance. A JobInstance is unique because of the JobParameters associated with it. Each time the same JobInstance (i.e., the same Job and JobParameters) is run, it's called a JobExecution. This is a runtime context for a version of the Job. Ideally, for every JobInstance there'd be only one JobExecution: the JobExecution that was created the first time the JobInstance ran. However, if there were any errors, the JobInstance should be restarted; the subsequent run would create another JobExecution. For every step in the original job there is a StepExecution in the JobExecution.

Thus, you can see that Spring Batch has a mirrored object graph, one reflecting the design/build time view of a job, and another reflecting the runtime view of a job. This split between the prototype and the instance is very similar to the way many workflow engines—including jBPM—work.

¹<https://jcp.org/en/jsr/detail?id=352>

For example, suppose that a daily report is generated at 2 AM. The parameter to the job would be the date (most likely the previous day's date). The job, in this case, would model a loading step, a summary step, and an output step. Each day the job is run, a new JobInstance and JobExecution would be created. If there are any retries of the same JobInstance, conceivably many JobExecutions would be created.

SPRING BATCH LIBRARIES

If you want to use Spring Batch, you need to add the appropriate libraries to the classpath. If you are using Maven, add the following dependency to your project.

```
<dependency>
  <groupId>org.springframework.batch</groupId>
  <artifactId>spring-batch-core</artifactId>
  <version>3.0.1.RELEASE</version>
</dependency>
```

12-1. Setting Up Spring Batch's Infrastructure

Problem

Spring Batch provides a lot of flexibility and guarantees to your application, but it cannot work in a vacuum. To do its work, the JobRepository requires a database. Additionally, there are several collaborators required for Spring Batch to do its work. This configuration is mostly boilerplate.

Solution

In this recipe, you'll set up the Spring Batch database and also create a Spring application configuration that can be imported by subsequent solutions. This configuration is repetitive and largely uninteresting. It will also tell Spring Batch what database to use for the metadata it stores.

How It Works

The JobRepository interface is the first thing that you'll have to deal with when setting up a Spring Batch process. You usually don't deal with it in code, but in Spring configuration it is key to getting everything else working. There's only one really useful implementation of the JobRepository interface called SimpleJobRepository, which stores information about the state of the batch processes in a database. Creation is done through a JobRepositoryFactoryBean. Another standard factory, MapJobRepositoryFactoryBean is useful mainly for testing because its state is not durable – it's an in-memory implementation. Both factories create an instance of SimpleJobRepository.

Because this JobRepository instance works on your database, you need to set up the schema for Spring Batch to work with. The schemas for different databases are in the Spring Batch distribution. The simplest way to initialize your database is to use the `<jdbc:initialize-database />` tag or a DataSourceInitializer in Java config. The files can be found in the org/springframework/batch/core directory there are several .sql files, each containing the data definition language (DDL, the subset of SQL used for defining and examining the structure of a database) for the required schema for the database of your choice. In these examples, we will use Apache Derby, so we will use the DDL for Derby: schema-derby.sql. Make sure you configure it and tell Spring Batch about it as in the following configurations.

Configure Spring Batch's Infrastructure using XML

To setup the infrastructure components mentioned in XML use the following Spring XML configuration.

```

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

    <context:component-scan base-package="com.apress.springrecipes.springbatch"/>
    <context:property-placeholder location="batch.properties" ignore-unresolvable ="true" />

    <!--Initialize the database if tables do not already exist -->
    <jdbc:initialize-database enabled="true" data-source="dataSource" ignore-failures="ALL">
        <jdbc:script location="classpath:org/springframework/batch/core/schema-derby.sql" execution="INIT"/>
    </jdbc:initialize-database>

    <beans:bean id="jobRepository"
        class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
        <beans:property name="dataSource" ref="dataSource" />
        <beans:property name="transactionManager" ref="transactionManager" />
    </beans:bean>

    <beans:bean
        id="dataSource"
        class="org.apache.commons.dbcp2.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${dataSource.driverClassName}"
        p:username="${dataSource.username}"
        p:password="${dataSource.password}"
        p:url="${dataSource.url}"/>

    <beans:bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource"/>

```

```
<beans:bean id="jobRegistry"
    class="org.springframework.batch.core.configuration.support.MapJobRegistry"/>

<beans:bean id="jobLauncher"
    class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
    p:jobRepository-ref="jobRepository"/>

<beans:bean
    id="jobRegistryBeanPostProcessor"
    class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcessor"
    p:jobRegistry-ref="jobRegistry"/>
```

Because the implementation uses a database to persist the metadata, take care to configure a `DataSource` as well as a `TransactionManager`. In this example, you're using a `PropertyPlaceholderConfigurer` to load the contents of a properties file (`batch.properties`) whose values you use to configure the data source. You need to place values for your particular database in this file. This example uses Spring's property schema ("p") to abbreviate the tedious configuration. In subsequent examples, this file will be referenced as `batch.xml`. The properties file looks like this for us:

```
dataSource.password=app
dataSource.username=app
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url= jdbc:derby://localhost:1527/batch;create=true
```

The first few beans are related strictly to configuration—nothing particularly novel or peculiar to Spring Batch: a data source, a transaction manager, and a properties resolver.

Eventually, we get to the declaration of a `MapJobRegistry` instance. This is critical—it is the central store for information regarding a given Job, and it controls the “big picture” about all Jobs in the system. Everything else works with this instance.

Next, we have a `SimpleJobLauncher`, whose sole purpose is to give you a mechanism to launch batch jobs, where a “job” in this case is our batch solution. The `jobLauncher` is used to specify the name of the batch solution to run as well as any parameters required. We'll follow up more on that in the next recipe.

Next, you define a `JobRegistryBeanPostProcessor`. This bean scans your Spring context file and associates any configured Jobs with the `MapJobRegistry`.

Finally, we get to the `SimpleJobRepository` (that is, in turn, factored by the `JobRepositoryFactoryBean`). The `JobRepository` is an implementation of “repository” (in the *Patterns of Enterprise Application Architecture* sense of the word): it handles persistence and retrieval for the domain models surrounding Steps, Jobs, and so on.

Spring Batch comes with namespace support and we can use this support to simplify our configuration a little. Spring Batch has the `job-repository` tag that makes it easier to configure the `JobRepositoryFactoryBean` and you don't have to remember the name of the bean to use. We could replace the bean definition for the `JobRepositoryFactoryBean` with the following.

```
<job-repository id="jobRepository" data-source="dataSource" transaction-manager="transactionManager" />
```

This saves a couple of lines of XML, which can be especially useful if extensive configuration is needed for the `JobRepositoryFactoryBean`.

Configure Spring Batch's Infrastructure using Java Config

As of Spring Batch 3.0, there is also the possibility to use Java based configuration. For this use the `@EnableBatchProcessing` annotation on a `@Configuration` class.

```
package com.apress.springrecipes.springbatch.config;

import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.datasource.init.DataSourceInitializer;
import org.springframework.jdbc.datasource.init.ResourceDatabasePopulator;

import javax.sql.DataSource;

@Configuration
@EnableBatchProcessing(modular = false)
@ComponentScan("com.apress.springrecipes.springbatch")
@PropertySource("classpath:/batch.properties")
public class BatchConfiguration {

    @Autowired
    private Environment env;

    @Bean
    public DataSource dataSource() {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl(env.getRequiredProperty("dataSource.url"));
        dataSource.setDriverClassName(env.getRequiredProperty("dataSource.driverClassName"));
        dataSource.setUsername(env.getProperty("dataSource.username"));
        dataSource.setPassword(env.getProperty("dataSource.password"));
        return dataSource;
    }

    @Bean
    public DataSourceInitializer databasePopulator() {
        ResourceDatabasePopulator populator = new ResourceDatabasePopulator();
        populator.addScript(new ClassPathResource("org/springframework/batch/core/schema-derby.sql"));
        populator.addScript(new ClassPathResource("sql/reset_user_registration.sql"));
        populator.setContinueOnError(true);
        populator.setIgnoreFailedDrops(true);

        DataSourceInitializer initializer = new DataSourceInitializer();
        initializer.setDatabasePopulator(populator);
        initializer.setDataSource(dataSource());
        return initializer;
    }
}
```

The `@PropertySource` annotation will instruct Spring to load our `batch.properties` file, the properties we need we are going to retrieve using the `Environment` class.

Tip we could have also used a `@Value` annotation to inject all individual properties but when needing multiple properties in a configuration class it is easier to use the `Environment` object.

This class only contains two bean definitions one for the datasource and one for initializing the database, everything else is taken care of due to the `@EnableBatchProcessing` annotation. The configuration class above will bootstrap Spring Batch with some sensible defaults. The `modular` attribute indicates if the job definitions should be loaded in the same context or in their own child contexts. For now it is enough to load the jobs in the same context and hence we set it to `false`.

The default configuration will configure a `JobRepository`, `JobRegistry`, and `JobLauncher`.

The `JobRegistry` will be an instance of the `MapJobRegistry`. This is critical—it is the central store for information regarding a given Job, and it controls the “big picture” about all Jobs in the system. Everything else works with this instance.

Next, the `JobLauncher` will be a `SimpleJobLauncher`, whose sole purpose is to give you a mechanism to launch batch jobs, where a “job” in this case is our batch solution. The `JobLauncher` is used to specify the name of the batch solution to run as well as any parameters required. We’ll follow up more on that in the next recipe.

Finally, a `JobRepository` is also created, in this case a `SimpleJobRepository` (that is, in turn, factored by the `JobRepositoryFactoryBean`). The `JobRepository` is an implementation of “repository” (in the *Patterns of Enterprise Application Architecture* sense of the word): it handles persistence and retrieval for the domain models surrounding Steps, Jobs, and so on. Due to the existence of a datasource in our configuration a JDBC version of the `JobRepository` will be created, of there are no datasources a Map based version would have been constructed.

If there are multiple datasources in your application you need to add `BatchConfigurer` to select the datasource to use for the batch part of your application.

The following Main class will use the Java based configuration for running the batch application:

```
package com.apress.springrecipes.springbatch;

import com.apress.springrecipes.springbatch.config.BatchConfiguration;
import org.springframework.batch.core.configuration.JobRegistry;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.core.repository.JobRepository;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) throws Throwable {
        ApplicationContext context = new AnnotationConfigApplicationContext(BatchConfiguration.class);

        JobRegistry jobRegistry = context.getBean("jobRegistry", JobRegistry.class);
        JobLauncher jobLauncher = context.getBean("jobLauncher", JobLauncher.class);
        JobRepository jobRepository = context.getBean("jobRepository", JobRepository.class);

        System.out.println("JobRegistry: " + jobRegistry);
        System.out.println("JobLauncher: " + jobLauncher);
        System.out.println("JobRepository: " + jobRepository);

    }
}
```

Notice the usage of the `AnnotationConfigApplicationContext`, the application is now only using Java and no more XML for configuration and running.

12-2. Reading and Writing

Problem

You want to insert data from a file into a database. This solution will be one of the simplest solutions and will give you a chance to explore the moving pieces of a typical solution.

Solution

You'll build a solution that does a minimal amount of work, while being a viable application of the technology. The solution will read in a file of arbitrary length and write out the data into a database. The end result will be almost 100 percent code free. You will rely on an existing model class and write one class (a class containing the `public static void main(String [] args())` method) to round out the example. There's no reason why the model class couldn't be a Hibernate class or something from your DAO layer, though in this case it's a brainless POJO. This solution will use the components we configured in `batch.xml`.

How It Works

This example demonstrates the simplest possible use of a Spring Batch: to provide scalability. This program will do nothing but read data from a CSV file, with fields delimited by commas and rows delimited by new lines. It then inserts the records into a table. You are exploiting the intelligent infrastructure that Spring Batch provides to avoid worrying about scaling. This application could easily be done manually. You will not exploit any of the smart transactional functionality made available to you, nor will you worry about retries for the time being.

This solution is as simple as Spring Batch solutions get. Spring Batch models solutions using XML schema. The abstractions and terms are in the spirit of classical batch processing solutions so will be portable from previous technologies and perhaps to subsequent technologies. Spring Batch provides useful default classes that you can override or selectively adjust. In the following example, you'll use a lot of the utility implementations provided by Spring Batch. Fundamentally, most solutions look about the same and feature a combination of the same set of interfaces. It's usually just a matter of picking and choosing the right ones.

When I ran this program, it worked on files with 20,000 rows, and it worked on files with 1 million rows. I experienced no increase in memory, which indicates there were no memory leaks. Naturally, it took a lot longer! (The application ran for several hours with the 1-million-row insert.)

Tip Of course, it would be catastrophic if you worked with a million rows and it failed on the penultimate record, because you'd lose all your work when the transaction rolled back! Read on for examples on chunking. Additionally, you might want to read through Chapter 11 to brush up on transactions.

```
create table USER_REGISTRATION
(
    ID BIGINT NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY (START WITH 1, INCREMENT BY 1),
    FIRST_NAME VARCHAR(255) not null,
    LAST_NAME VARCHAR(255) not null,
    COMPANY VARCHAR(255) not null,
    ADDRESS VARCHAR(255) not null,
```

```
CITY VARCHAR(255) not null,
STATE VARCHAR(255) not null,
ZIP VARCHAR(255) not null,
COUNTY VARCHAR(255) not null,
URL VARCHAR(255) not null,
PHONE_NUMBER VARCHAR(255) not null,
FAX VARCHAR(255) not null
) ;
```

DATA LOADS AND DATA WAREHOUSES

I didn't tune the table at all. For example, there are no indexes on any of the columns besides the primary key. This is to avoid complicating the example. Great care should be taken with a table like this one in a nontrivial, production-bound application.

Spring Batch applications are workhorse applications and have the potential to reveal bottlenecks in your application you didn't know you had. Imagine suddenly being able to achieve 1 million new database insertions every 10 minutes. Would your database grind to a halt? Insert speed can be a critical factor in the speed of your application. Software developers will (hopefully) think about their database schema in terms of how well it enforces the constraints of the business logic and how well it serves the overall business model. However, it's important to wear another hat, that of a DBA, when writing applications such as this one. A common solution is to create a denormalized table whose contents can be coerced into valid data once inside the database, perhaps by a trigger on inserts. This is typical in data warehousing. Later, you'll explore using Spring Batch to do processing on a record before insertion. This lets the developer verify or override the input into the database. This processing, in tandem with a conservative application of constraints that are best expressed in the database, can make for applications that are very robust *and* quick.

The Job Configuration

The configuration for the job is as follows:

```
<job
    job-repository="jobRepository"
    id="insertIntoDbFromCsvJob">
    <step id="step1">
        <tasklet transaction-manager="transactionManager">
            <chunk
                reader="csvFileReader"
                writer="jdbcItemWriter"
                commit-interval="5"
            />
        </tasklet>
    </step>
</job>
```

As described earlier, a job consists of steps, which are the real workhorse of a given job. The steps can be as complex or as simple as you like. Indeed, a step could be considered the smallest unit of work for a job. Input (what's read) is passed to the Step and potentially processed; then output (what's written) is created from the step. This processing is spelled out using a Tasklet. You can provide your own Tasklet implementation or simply use some of the preconfigured configurations for different processing scenarios. These implementations are made available in terms of subelements of the Tasklet element. One of the most important aspects of batch processing is chunk-oriented processing, which is employed here using the chunk element.

In chunk-oriented processing, input is read from a reader, optionally processed, and then aggregated. Finally, at a configurable interval—as specified by the `commit-interval` attribute to configure how many items will be processed before the transaction is committed—all the input is sent to the writer. If there is a transaction manager in play, the transaction is also committed. Right before a commit, the metadata in the database is updated to mark the progress of the job.

There are some nuances surrounding the aggregation of the input (read) values when a transaction-aware writer (or processor) rolls back. Spring Batch caches the values it reads and writes them to the writer. If the writer component is transactional, like a database, and the reader is not, there's nothing inherently wrong with caching the read values and perhaps retrying or taking some alternative approach. If the reader itself is also transactional, then the values read from the resource will be rolled back and could conceivably change, rendering the in-memory cached values stale. If this happens, you can configure the chunk to not cache the values using `reader-transactional-queue="true"` on the chunk element.

Input

The first responsibility is reading a file from the file system. You use a provided implementation for the example. Reading CSV files is a very common scenario, and Spring Batch's support does not disappoint. The `org.springframework.batch.item.file.FlatFileItemReader<T>` class delegates the task of delimiting fields and records within a file to a `LineMapper<T>`, which in turn delegates the task of identifying the fields within that record, to `LineTokenizer`. You use a `org.springframework.batch.item.file.transform.DelimitedLineTokenizer`, which is configured to delineate fields separated by a "," character.

The `FlatFileItemReader` also declares a `fieldSetMapper` attribute that requires an implementation of `FieldSetMapper`. This bean is responsible for taking the input name/value pairs and producing a type that will be given to the writer component.

In this case, you use an `BeanWrapperFieldSetMapper` that will create a JavaBean POJO of type `UserRegistration`. You name the fields so that you can reference them later in the configuration. These names don't have to be the values of some header row in the input file; they just have to correspond to the order in which the fields are found in the input file. These names are also used by the `FieldSetMapper` to match properties on a POJO. As each record is read, the values are applied to an instance of a POJO, and that POJO is returned.

```
<beans:bean
    id="csvFileReader"
    class="org.springframework.batch.item.file.FlatFileItemReader"
    p:resource="file:${user.home}/batches/registrations.csv">
    <beans:property name="lineMapper">
        <beans:bean
            class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <beans:property name="lineTokenizer">
                <beans:bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">

```

```

    p:delimiter=","
    p:names="firstName,lastName,company,address,city,state,zip,county,url,phoneNumber,fax" />
  </beans:property>
  <beans:property name="fieldSetMapper">
    <beans:bean class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper" p:targetType="com.apress.springrecipes.springbatch.UserRegistration" />
  </beans:property>
</beans:bean>
</beans:property>
</beans:bean>
```

The class returned from the reader, `UserRegistration`, is a rather plain JavaBean.

```

package com.apress.springrecipes.springbatch;

public class UserRegistration implements Serializable {

    private String firstName;
    private String lastName;
    private String company;
    private String address;
    private String city;
    private String state;
    private String zip;
    private String county;
    private String url;
    private String phoneNumber;
    private String fax;

    //... accessor / mutators omitted for brevity ...

}
```

Output

The next component to do work is the `writer`, which is responsible for taking the aggregated collection of items read from the reader. In this case, you might imagine that a new collection (`java.util.List<UserRegistration>`) is created, then written, and then reset each time the collection exceeds the commit-interval attribute on the chunk element. Because you're trying to write to a database, you use Spring Batch's `org.springframework.batch.item.database.JdbcBatchItemWriter`. This class contains support for taking input and writing it to a database. It is up to the developer to provide the input and to specify what SQL should be run for the input. It will run the SQL specified by the `sql` property, in essence reading from the database, as many times as specified by the chunk element's `commit-interval`, and then commit the whole transaction. Here, you're doing a simple insert. The names and values for the named parameters are being created by the bean configured for the `itemSqlParameterSourceProvider` property, an instance of the interface `BeanPropertyItemSqlParameterSourceProvider`, whose sole job it is to take JavaBean properties and make them available as named parameters corresponding to the property name on the JavaBean.

```

<beans:bean id="jdbcItemWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter"
p:assertUpdates="true"
    p:dataSource-ref="dataSource">
    <beans:property name="sql">
        <beans:value>
            <![CDATA[
                insert into USER_REGISTRATION(
FIRST_NAME, LAST_NAME, COMPANY, ADDRESS,
CITY, STATE, ZIP, COUNTY,
URL, PHONE_NUMBER, FAX )
values ( :firstName, :lastName, :company, :address, :city , :state, :zip, :county, :url, :phoneNumber, :fax )
]]> </beans:value>
    </beans:property>
    <beans:property name="itemSqlParameterSourceProvider">
        <beans:bean
            class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider" />
    </beans:property>
</beans:bean>

```

And that's it! A working solution. With little configuration and no custom code, you've built a solution for taking large CSV files and reading them into a database. This solution is bare bones and leaves a lot of edge cases uncared for. You might want to do processing on the item as it's read (before it's inserted), for example.

This exemplifies a simple job. It's important to remember that there are similar classes for doing the exact opposite transformation: reading from a database and writing to a CSV file.

As mentioned in Recipe 12-1 there is also a Java Config alternative available, instead of writing XML we can also create a class, annotated with `@Configuration`, to setup our job and steps. The Java Config alternative would look like the following

```

package com.apress.springrecipes.springbatch.config;

import com.apress.springrecipes.springbatch.UserRegistration;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvider;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.LineMapper;
import org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper;
import org.springframework.batch.item.file.mapping.DefaultLineMapper;
import org.springframework.batch.item.file.transform.DelimitedLineTokenizer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import org.springframework.core.io.Resource;

```

```
import javax.sql.DataSource;

@Configuration
public class UserJob {

    private static final String INSERT_REGISTRATION_QUERY =
        "insert into USER_REGISTRATION(FIRST_NAME, LAST_NAME, COMPANY, ADDRESS,CITY,STATE,ZIP,CO
UNTY,URL,PHONE_NUMBER,FAX)" +
        " values " +
        "(:firstName,:lastName,:company,:address,:city,:state,:zip,:county,:url,:phoneNumber,:fax)";

    @Autowired
    private JobBuilderFactory jobs;

    @Autowired
    private StepBuilderFactory steps;

    @Autowired
    private DataSource dataSource;

    @Value("file:${user.home}/batches/registrations.csv")
    private Resource input;

    @Bean
    public Job insertIntoDbFromCsvJob() {
        return jobs.get("insertIntoDbFromCsvJob")
            .start(step1())
            .build();
    }

    @Bean
    protected Step step1() {
        return steps.get("step1")
            .<UserRegistration,UserRegistration>chunk(5)
            .reader(csvFileReader())
            .writer(jdbcItemWriter())
            .build();
    }

    @Bean
    ItemReader<UserRegistration> csvFileReader() {
        FlatFileItemReader<UserRegistration> itemReader = new FlatFileItemReader<>();
        itemReader.setLineMapper(lineMapper());
        itemReader.setResource(input);
        return itemReader;
    }
}
```

```

@Bean
ItemWriter<UserRegistration> jdbcItemWriter() {
    JdbcBatchItemWriter itemWriter = new JdbcBatchItemWriter();
    itemWriter.setDataSource(dataSource);
    itemWriter.setSql(INSERT_REGISTRATION_QUERY);
    itemWriter.setItemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider());
    return itemWriter;
}

@Bean
LineMapper<UserRegistration> lineMapper() {
    DelimitedLineTokenizer tokenizer = new DelimitedLineTokenizer();
    tokenizer.setDelimiter(",");
    tokenizer.setNames(new String[]{"firstName", "lastName", "company", "address", "city", "state",
        "zip", "county", "url", "phoneNumber", "fax"});
}

BeanWrapperFieldSetMapper<UserRegistration> fieldSetMapper = new BeanWrapperFieldSetMapper();
fieldSetMapper.setTargetType(UserRegistration.class);

DefaultLineMapper<UserRegistration> lineMapper = new DefaultLineMapper<>();
lineMapper.setLineTokenizer(tokenizer);
lineMapper.setFieldSetMapper(fieldSetMapper);
return lineMapper;
}
}

```

Basically we configure the different components here, the ItemReader and ItemWriter with all the related beans. There are two factory classes, StepBuilderFactory and JobBuilderFactory, which help us configure our steps and jobs. Both classes follow the builder pattern and that allows us to chain the methods to configure our step and job. Let's take a closer look at our step configuration.

```

@Bean
protected Step step1() {
    return steps.get("step1")
        .chunk(5)
        .reader(csvFileReader())
        .writer(jdbcItemWriter())
        .build();
}

```

To configure the step we give it the name `step1`, we are using chunk based processing and we need to tell it that we want a chunk size of 5, next we supply it with a reader and writer and finally we tell the factory to build to the step. Effectively resulting in the same step configuration as in our XML configuration earlier on.

The configured step is finally wired as a starting point to our job, which consists only of this step.

12-3. Writing a Custom ItemWriter and ItemReader

Problem

You want to talk to a resource (you might imagine an RSS feed, or any other custom data format) that Spring Batch doesn't know how to connect to.

Solution

You can easily write your own `ItemWriter` or `ItemReader`. The interfaces are drop dead simple, and there's not a lot of responsibility placed on the implementations.

How It Works

As easy and trivial as this process is to do, it's still not better than just reusing any of the numerous provided options. If you look, you'll likely find something. There's support for writing JMS (`JmsItemWriter<T>`), JPA (`JpaItemWriter<T>`), JDBC (`JdbcBatchItemWriter<T>`), Files (`FlatFileItemWriter<T>`), iBatis (`IbatisBatchItemWriter<T>`), Hibernate (`HibernateItemWriter<T>`), and more. There's even support for writing by invoking a method on a bean (`PropertyExtractingDelegatingItemWriter<T>`) and passing to it as arguments the properties on the `Item` to be written! One of the more useful writers lets you write to a set of files that are numbered. This implementation—`MultiResourceItemWriter<T>`—delegates to other proper `ItemWriter<T>` implementation for the work, but lets you write to multiple files, not just one very large one. There's a slightly smaller but impressive set of implementations for `ItemReader` implementations. If it doesn't exist, look again. If you *still* can't find one, consider writing your own. In this recipe, we will do just that.

Writing a Custom ItemReader

The `ItemReader` example is trivial. Here, an `ItemReader` is created that knows how to retrieve `UserRegistration` objects from a remote procedure call (RPC) endpoint:

```
package com.apress.springrecipes.springbatch;

import java.util.Collection;
import java.util.Date;

import org.springframework.batch.item.ItemReader;
import org.springframework.batch.item.ParseException;
import org.springframework.batch.item.UnexpectedInputException;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.springrecipes.springbatch.UserRegistrationService;
import com.apress.springrecipes.springbatch.UserRegistration;

public class UserRegistrationItemReader
    implements ItemReader<UserRegistration> {

    @Autowired
    private UserRegistrationService userRegistrationService;
    public UserRegistration read() throws Exception,UnexpectedInputException,
    ParseException {
        Date today = new Date();
        Collection<UserRegistration> registrations =

```

```

        userRegistrationService.getOutstandingUserRegistrationBatchForDate(
            1, today);
        if (registrations!=null && registrations.size() >=1)
            return registrations.iterator().next();
        return null;
    }
}

```

As you can see, the interface is trivial. In this case, you defer most work to a remote service to provide you with the input. The interface requires that you return one record. The interface is parameterized to the type of object (the “item”) to be returned. All the read items will be aggregated and then passed to the ItemWriter.

Writing a Custom ItemWriter

The ItemWriter example is also trivial. Imagine wanting to write by invoking a remote service using any of the numerous options for remoting that Spring provides. The ItemWriter<T> interface is parameterized by the type of item you’re expecting to write. Here, you expect a UserRegistration object from the ItemReader<T>. The interface consists of one method, which expects a List of the class’s parameterized type. These are the objects read from ItemReader<T> and aggregated. If your commit-interval were ten, you might expect ten or fewer items in the List.

```

package com.apress.springrecipes.springbatch;

import java.util.List;

import org.apache.commons.lang3.builder.ToStringBuilder;
import org.slf4j.Logger;
import org.springframework.batch.item.ItemWriter;
import org.springframework.beans.factory.annotation.Autowired;

import com.apress.springrecipes.springbatch.User;
import com.apress.springrecipes.springbatch.UserRegistrationService;
import com.apress.springrecipes.springbatch.solution1.UserRegistration;

public class UserRegistrationServiceItemWriter implements
    ItemWriter<UserRegistration> {

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

    // this is the client interface to an HTTP Invoker service.
    @Autowired
    private UserRegistrationService userRegistrationService;

    /**
     * takes aggregated input from the reader and 'writes' them using a custom
     * implementation.
     */
    public void write(List<? extends UserRegistration> items)
        throws Exception {
        for (final UserRegistration userRegistration : items) {
            UserRegistration registeredUser = userRegistrationService.registerUser(userRegistration);
            logger.debug("Registered: {}",ToStringBuilder.reflectionToString(registeredUser));
        }
    }
}

```

Here, you've wired in the service's client interface. You simply loop through the `UserRegistration` objects and invoke the service, which in turn hands you back an identical instance of `UserRegistration`. If you remove the gratuitous spacing, curly brackets, and logging output, it becomes two lines of code to satisfy the requirement.

The interface for `UserRegistrationService` follows:

```
package com.apress.springrecipes.springbatch;

import java.util.Collection;
import java.util.Date;

public interface UserRegistrationService {

    Collection<UserRegistration> getOutstandingUserRegistrationBatchForDate(
        int quantity, Date date);

    UserRegistration registerUser(
        UserRegistration userRegistrationRegistration);

}
```

In our example, we have no particular implementation for the interface, as it is irrelevant: it could be any interface that Spring Batch doesn't know about already.

12-4. Processing Input Before Writing Problem

While transferring data directly from a spreadsheet or CSV dump might be useful, one can imagine having to do some sort of processing on the data before it's written. Data in a CSV file, and more generally from any source, is not usually exactly the way you expect it to be or immediately suitable for writing. Just because Spring Batch can coerce it into a POJO on your behalf, that doesn't mean the state of the data is correct. There may be additional data that you need to infer or fill in from other services before the data is suitable for writing.

Solution

Spring Batch will let you do processing on reader output. This processing can do virtually anything to the output before it gets passed to the writer, including changing the type of the data.

How It Works

Spring Batch gives the implementor a chance to perform any custom logic on the data read from reader. The `processor` attribute on the chunk element expects a reference to a bean of the interface `org.springframework.batch.item.ItemProcessor<I,O>`. Thus, the revised definition for the job from the previous recipe looks like this:

```
<job id="insertIntoDbFromCsvJob" job-repository="jobRepository">
    <step id="step1">
        <tasklet transaction-manager="transactionManager">
            <chunk
                reader="csvFileReader"
```

```

processor = "userRegistrationValidationProcessor"
writer="jdbcItemWriter"
commit-interval="5"
/>
</tasklet>
</step>
</job>

```

Or for those using Java based Job configuration

```

@Bean
protected Step step1() {
    return steps.get("step1")
        .<UserRegistration,UserRegistration>chunk(5)
        .reader(csvFileReader())
        .processor(userRegistrationValidationItemProcessor())
        .writer(jdbcItemWriter())
        .build();
}

```

The goal is to do certain validations on the data before you authorize it to be written to the database. If you determine the record is invalid, you can stop further processing by returning null from the ItemProcessor<I,O>. This is crucial and provides a necessary safeguard. One thing that you want to do is ensure that the data is the right format (for example, the schema may require a valid two-letter state name instead of the longer full state name). Telephone numbers are expected to follow a certain format, and you can use this processor to strip the telephone number of any extraneous characters, leaving only a valid (in the United States) ten-digit phone number. The same applies for U.S. zip codes, which consist of five characters and optionally a hyphen followed by a four-digit code. Finally, while a constraint guarding against duplicates is best implemented in the database, there may very well be some other eligibility criteria for a record that can be met only by querying the system before insertion.

Here's the configuration for the ItemProcessor:

```
<beans:bean id="userRegistrationValidationProcessor"
class="com.apress.springrecipes.springbatch.UserRegistrationValidationItemProcessor" />
```

The Java config equivalent

```

@Bean
ItemProcessor<UserRegistration, UserRegistration> userRegistrationValidationItemProcessor() {
    return new UserRegistrationValidationItemProcessor();
}

```

In the interest of keeping this class short, I won't reprint it in its entirety, but the salient bits should be obvious:

```

package com.apress.springrecipes.springbatch;
import java.util.Arrays;
import java.util.Collection;

import org.apache.commons.lang3.StringUtils;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.item.ItemProcessor;
import com.apress.springrecipes.springbatch.UserRegistration;

```

```

public class UserRegistrationValidationItemProcessor
    implements ItemProcessor<UserRegistration, UserRegistration> {

    private String stripNonNumbers(String input) { /* ... */ }

    private boolean isTelephoneValid(String telephone) { /* ... */ }

    private boolean isZipCodeValid(String zip) { /* ... */ }

    private boolean isValidState(String state) { /* ... */ }

    public UserRegistration process(UserRegistration input) throws Exception {
        String zipCode = stripNonNumbers(input.getZip());
        String telephone = stripNonNumbers(input.getPhoneNumber());
        String state = StringUtils.defaultString(input.getState());
        if (isTelephoneValid(telephone) && isZipCodeValid(zipCode) && isValidState(state)) {
            input.setZip(zipCode);
            input.setPhoneNumber(telephone );
            return input;
        }
        return null;
    }
}

```

The class is a parameterized type. The type information is the type of the input, as well as the type of the output. The input is what's given to the method for processing, and the output is the returned data from the method. Because you're not transforming anything in this example, the two parameterized types are the same.

Once this process has completed, there's a lot of useful information to be had in the Spring Batch metadata tables. Issue the following query on your database:

```
select * from BATCH_STEP_EXECUTION;
```

Among other things, you'll get back the exit status of the job, how many commits occurred, how many items were read, and how many items were filtered. So if the preceding job was run on a batch with a 100 rows, each item was read and passed through the processor, and it found 10 items invalid (it returned `null` 10 times), the value for the `filter_count` column would be 10. You could see that a 100 items were read from the `read_count`. The `write_count` column would reflect that 10 items didn't make it and would show 90.

Chaining Processors Together

Sometimes you might want to add extra processing that isn't congruous with the goals of the processor you've already set up. Spring Batch provides a convenience class, `CompositeItemProcessor<I,O>`, which forwards the output of the filter to the input of the successive filter. In this way, you can write many, singly focused `ItemProcessor<I,O>`s and then reuse them and chain them as necessary:

```

<beans:bean id="compositeBankCustomerProcessor"
class="org.springframework.batch.item.support.CompositeItemProcessor">
    <beans:property name="delegates">
        <beans:list>
            <bean ref="creditScoreValidationProcessor" />

```

```

<bean ref="salaryValidationProcessor" />
<bean ref="customerEligibilityProcessor" />
</beans:list>
</beans:property>
</beans:bean>
<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
<step id="step1">
    <tasklet transaction-manager="transactionManager">
        <chunk
            reader="csvFileReader"
            processor="compositeBankCustomerProcessor"
            writer="jdbcItemWriter"
            commit-interval="5"
        />
    </tasklet>
</step>
</job>

```

The example created a very simple workflow. The first `ItemProcessor<T>` will take an input of whatever's coming from the `ItemReader<T>` configured for this job, presumably a `Customer` object. It will check the credit score of the `Customer` and, if approved, forward the `Customer` to the salary and income validation processor. If everything checks out there, the `Customer` will be forwarded to the eligibility processor, where the system is checked for duplicates or any other invalid data. It will finally be forwarded to the `writer` to be added to the output. If at any point in the three processors the `Customer` fails a check, the executing `ItemProcessor` can simply return `null` and arrest processing.

12-5. Better Living through Transactions

Problem

You want your reads and writes to be robust. Ideally, they'll use transactions where appropriate and correctly react to exceptions.

Solution

Transaction capabilities are built on top of the first class support already provided by the core Spring framework. Where relevant, Spring Batch surfaces the configuration so that you can control it. Within the context of chunk-oriented processing, it also exposes a lot of control over the frequency of commits, rollback semantics, and so on.

How It Works

Transactions

Spring's core framework provides first-class support for transactions. You simply wire up a `TransactionManager` and give Spring Batch a reference, just as you would in any regular `JdbcTemplate` or `HibernateTemplate` solution. As you build your Spring Batch solutions, you'll be given opportunities to control how steps behave in a transaction. You've already seen some of the support for transactions baked right in.

The `batch.xml` file, used in all these examples, established a `BasicDataSource` and a `DataSourceTransactionManager` bean. The `TransactionManager` and `BasicDataSource` were then wired to the `JobRepository`, which was in turn wired to the `JobLauncher`, which you used to launch all jobs thus far. This enabled all the metadata your jobs create to be written to the database in a transactional way.

You might wonder why there is no explicit mention of the `TransactionManager` when you configured the `JdbcItemWriter` with a reference to `dataSource`. The transaction manager reference can be specified, but in your solutions, it wasn't required because Spring Batch will, by default, try to pluck the `PlatformTransactionManager` named `transactionManager` from the context and use it. If you want to explicitly configure this, you can specify the `transactionManager` property on the `tasklet` element. A simple `TransactionManager` for JDBC work might look like this:

```
<bean id="myCustomTransactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      p:dataSource-ref="dataSource" />

<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
    <step id="step1">
        <tasklet transaction-manager="myCustomTransactionManager" >
            <!-- ... -->
        </tasklet>
    </step>
</job>
...

```

When working with Java based configuration you can use the `transactionManager` method to set the transaction manager.

```
@Bean
protected Step step1() {
    return steps.get("step1")
        .<UserRegistration,UserRegistration>chunk(5)
        .reader(csvFileReader())
        .processor(userRegistrationValidationItemProcessor())
        .writer(jdbcItemWriter())
        .transactionManager(new DataSourceTransactionManager(dataSource))
        .build();
}
```

Items read from an `ItemReader<T>` are normally aggregated. If a commit on the `ItemWriter<T>` fails, the aggregated items are kept and then resubmitted. This process is efficient and works most of the time. One place where it breaks semantics is when reading from a transactional message queue. Reads from a message queue can and should be rolled back if the transaction they participate in (in this case, the transaction for the writer) fails:

```
<tasklet transaction-manager="customTransactionManager" >
    <chunk
        reader="jmsItemReader" is-reader-transactional-queue="true"
        processor="userRegistrationValidationProcessor"
        writer="jdbcItemWriter"
        commit-interval="5" />
</tasklet>
```

Or in Java config

```
@Bean
protected Step step1() {
    return steps.get("step1")
        .<UserRegistration,UserRegistration>chunk(5)
        .reader(csvFileReader()).readerIsTransactionalQueue()
        .processor(userRegistrationValidationItemProcessor())
        .writer(jdbcItemWriter())
        .transactionManager(new DataSourceTransactionManager(dataSource))
        .build();
}
```

Rollbacks

Handling the simple case (“read X items, and every Y items, commit a database transaction every Y items”) is easy. Spring Batch excels in the robustness it surfaces as simple configuration options for the edge and failure cases.

If a write fails on an `ItemWriter`, or some other exception occurs in processing, Spring Batch will roll back the transaction. This is valid handling for a majority of the cases. There may be some scenarios when you want to control which exceptional cases cause the transaction to roll back.

You can use the `no-rollback-exception-classes` element to configure this for the step. The value is a list of Exception classes that should not cause the transaction to roll back:

```
<step id = "step2">
    <tasklet>
        <chunk reader="reader" writer="writer" commit-interval="10" />
        <no-rollback-exception-classes>
            <include class="com.yourdomain.exceptions.YourBusinessException"/>
        </no-rollback-exception-classes>
    </tasklet>
</step>
```

When using Java based configuration to enable rollbacks first the step needs to be a fault tolerant step, which in turn can be used to specify the no-rollback exceptions. First use `faultTolerant()` to obtain a fault tolerant step, next the `skipLimit()` method can be used to specify the number of ignored rollbacks before actually stopping the job execution, finally the `noRollback()` method can be used to specify the exceptions which don’t trigger a rollback. To specify multiple exceptions you can simply chain calls to the `noRollback()` method.

```
@Bean
protected Step step1() {
    return steps.get("step1")
        .<UserRegistration,UserRegistration>chunk(10)
        .faultTolerant()
        .noRollback(com.yourdomain.exceptions.YourBusinessException.class)
        .reader(csvFileReader())
        .processor(userRegistrationValidationItemProcessor())
        .writer(jdbcItemWriter())
        .build();
}
```

12-6. Retrying

Problem

You are dealing with a requirement for functionality that may fail but is not transactional. Perhaps it is transactional but unreliable. You want to work with a resource that may fail when you try to read from or write to it. It may fail because of networking connectivity because an endpoint is down or for any other number of reasons. You know that it will likely be back up soon, though, and that it should be retried.

Solution

Use Spring Batch's retry capabilities to systematically retry the read or write.

How It Works

As you saw in the last recipe, it's easy to handle transactional resources with Spring Batch. When it comes to transient or unreliable resources, a different tack is required. Such resources tend to be distributed or manifest problems that eventually resolve themselves. Some (such as web services) cannot inherently participate in a transaction because of their distributed nature. There are products that can start a transaction on one server and propagate the transactional context to a distributed server and complete it there, although this tends to be very rare and inefficient. Alternatively, there's good support for distributed ("global" or XA) transactions if you can use it. Sometimes, however, you may be dealing with a resource that isn't either of those. A common example might be a call made to a remote service, such as an RMI service or a REST endpoint. Some invocations will fail but may be retried with some likelihood of success in a transactional scenario. For example, an update to the database resulting in `org.springframework.dao.DeadlockLoserDataAccessException` might be usefully retried.

Configuring a Step

The simplest example is in the configuration of a step. Here, you can specify exception classes on which to retry the operation. As with the rollback exceptions, you can delimit this list of exceptions with newlines or commas:

```
<step id = "step23">
<tasklet transaction-manager="transactionManager">
    <chunk reader="csvFileReader" writer="jdbcItemWriter" commit-interval="10"
        retry-limit="3" cache-capacity="10">
        <retryable-exception-classes>
            <include class="org.springframework.dao.DeadlockLoserDataAccessException"/>
        </retryable-exception-classes>
    </chunk>
</tasklet>
</step>
```

When using Java based configuration to enable retrying first the step needs to be a fault tolerant step, which in turn can be used to specify the retry limit and retryable exceptions. First use `faultTolerant()` to obtain a fault tolerant step, next the `retryLimit()` method can be used to specify the number of retry attempts, finally the `retry()` method can be used to specify the exceptions which trigger a retry. To specify multiple exceptions you can simply chain calls to the `retry()` method.

```

@Bean
protected Step step1() {
    return steps.get("step1")
        .<UserRegistration,UserRegistration>chunk(10)
        .faultTolerant()
        .retryLimit(3).retry(DeadlockLoserDataAccessException.class)
        .reader(csvFileReader())
        .writer(jdbcItemWriter())
        .transactionManager(transactionManager)
        .build();
}

```

Retry Template

Alternatively, you can leverage Spring Retry support for retries and recovery in your own code. For example, you can have a custom `ItemWriter<T>` in which retry functionality is desired or even an entire service interface for which retry support is desired.

Spring Batch supports these scenarios through the `RetryTemplate` that (much like its various other `Template` cousins) isolates your logic from the nuances of retries and instead enables you to write the code as though you were only going to attempt it once. Let Spring Batch handle everything else through declarative configuration.

The `RetryTemplate` supports many use cases, with convenient APIs to wrap otherwise tedious retry/fail/recover cycles in concise, single-method invocations.

Let's take a look at the modified version of a simple `ItemWriter<T>` from Recipe 12.4 on how to write a custom `ItemWriter<T>`. The solution was simple enough and would ideally work all the time. It fails to handle the error cases for the service, however. When dealing with RPC, always proceed as if it's almost impossible for things to go right; the service itself may surface a semantic or system violation. An example might be a duplicate database key, invalid credit card number, and so on. This is true whether the service is distributed or in-VM, of course.

Next, the RPC layer below the system may also fault. Here's the rewritten code, this time allowing for retries:

```

package com.apress.springrecipes.springbatch;

import java.util.List;

import org.apache.commons.lang3.builder.ToStringBuilder;
import org.slf4j.Logger;
import org.springframework.batch.item.ItemWriter;
import org.springframework.retry.RetryCallback;
import org.springframework.retry.RetryContext;
import org.springframework.retry.support.RetryTemplate;
import org.springframework.beans.factory.annotation.Autowired;

/**
 * This class writes the user registration by calling an RPC service (whose
 * client interface is wired in using Spring
 */
public class RetryableUserRegistrationServiceItemWriter implements ItemWriter<UserRegistration>
{

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

```

```

// this is the client interface to an HTTP Invoker service.
@Autowired
private UserRegistrationService userRegistrationService;

@Autowired
private RetryTemplate retryTemplate;

/**
 * takes aggregated input from the reader and 'writes' them using a custom
 * implementation.
 */
public void write(List<? extends UserRegistration> items)
throws Exception {
    for (final UserRegistration userRegistration : items) {
        User registeredUser = retryTemplate.execute(
            new RetryCallback<User>() {
                public User doWithRetry(RetryContext context) throws Exception {
                    return userRegistrationService.registerUser(userRegistration);
                }
            });
        logger.debug("Registered: {}", ToStringBuilder.reflectionToString(registeredUser));
    }
}
}

```

As you can see, the code hasn't changed much, and the result is much more robust. The `RetryTemplate` itself is configured in the Spring context, although it's trivial to create in code. I declare it in the Spring context only because there is some surface area for configuration when creating the object, and I try to let Spring handle the configuration.

One of the more useful settings for the `RetryTemplate` is the `BackOffPolicy` in use. The `BackOffPolicy` dictates how long the `RetryTemplate` should back off between retries. Indeed, there's even support for growing the delay between retries after each failed attempt to avoid lock stepping with other clients attempting the same invocation. This is great for situations in which there are potentially many concurrent attempts on the same resource and a race condition may ensue. There are other `BackOffPolicies`, including one that delays retries by a fixed amount called `FixedBackOffPolicy`.

```

<beans:bean id="retryTemplate" class="org.springframework.retry.support.RetryTemplate">
    <beans:property name="backOffPolicy" >
        <beans:bean class="org.springframework.batch.retry.backoff.ExponentialBackOffPolicy"
            p:initialInterval="1000" p:maxInterval="10000" p:multiplier="2" />
    </beans:property>
</beans:bean>

```

You have configured a `RetryTemplate`'s `backOffPolicy` so that the `backOffPolicy` will wait 1 second (1,000 milliseconds) before the initial retry. Subsequent attempts will double that value (the growth is influenced by the multiplier). It'll continue until the `maxInterval` is met, at which point all subsequent retry intervals will level off, retrying at a consistent interval.

AOP-Based Retries

An alternative is an AOP advisor provided by Spring Batch that will wrap invocations of methods whose success is not guaranteed in retries, as you did with the `RetryTemplate`. In the previous example, you rewrote an `ItemWriter<T>` to make use of the template. Another approach might be to merely advise the entire `UserRegistrationService` proxy with this retry logic. In this case, the code could go back to the way it was in the original example, with no `RetryTemplate`!

```
<aop:config>
    <aop:pointcut id="remote"
        expression="execution(* com..*UserRegistrationService.*(..))" />
    <aop:advisor pointcut-ref="remote" advice-ref="retryAdvice" order="-1"/>
</aop:config>

<beans:bean id="retryAdvice" class="org.springframework.retry.interceptor.
RetryOperationsInterceptor"/>
```

12-7. Controlling Step Execution

Problem

You want to control how steps are executed, perhaps to eliminate a needless waste of time by introducing concurrency or by executing steps only if a condition is true.

Solution

There are different ways to change the runtime profile of your jobs, mainly by exerting control over the way steps are executed: concurrent steps, decisions, and sequential steps.

How It Works

Thus far, you have explored running one step in a job. Typical jobs of almost any complexity will have multiple steps, however. A step provides a boundary (transactional or not) to the beans and logic it encloses. A step can have its own reader, writer, and processor. Each step helps decide what the next step will be. A step is isolated and provides focused functionality that can be assembled using the updated schema and configuration options in Spring Batch in very sophisticated workflows. In fact, some of the concepts and patterns you're about to see will be very familiar if you have an interest in business process management (BPM) systems and workflows. BPM provides many constructs for process or job control that are similar to what you're seeing here.

A step often corresponds to a bullet point when you outline the definition of a job on paper. For example, a batch job to load the daily sales and produce a report might be proposed as follows:

Daily Sales Report Job

1. Load customers from the CSV file into the database.
2. Calculate daily statistics, and write to a report file.
3. Send messages to the message queue to notify an external system of the successful registration for each of the newly loaded customers.

Sequential Steps

In the previous example, there's an implied sequence between the first two steps: the audit file can't be written until all the registrations have completed. This sort of relationship is the default relationship between two steps. One occurs after the other. Each step executes with its own execution context and shares only a parent job execution context and an order.

```
<job id="nightlyRegistrationsJob"
    job-repository="jobRepository">
    <step id="loadRegistrations" next="reportStatistics" >
        <tasklet ref = "tasklet1"/>
    </step>
    <step id="reportStatistics" next="..." >
        <tasklet ref = "tasklet2"/>
    </step>
    <!-- ... other steps ... -->
</job>
```

Notice that you specify the next attribute on the step elements to tell processing which step to go to next. When using Java based configuration the next() method can be used to specify which step to execute next.

```
@Bean
public Job nightlyRegistrationsJob () {
    return jobs.get("nightlyRegistrationsJob ")
        .start(loadRegistrations())
        .next(reportStatistics())
        .next(...)
        .build();
}
```

Concurrency

The first version of Spring Batch was oriented toward batch processing inside the same thread and, with some alteration, perhaps inside the virtual machine. There were workarounds, of course, but the situation was less than ideal.

In the outline for this example job, the first step had to come before the second two because the second two are dependent on the first. The second two, however, do not share any such dependencies. There's no reason why the audit log couldn't be written at the same time as the JMS messages are being delivered. Spring Batch provides the capability to fork processing to enable just this sort of arrangement:

```
<job job-repository="jobRepository" id="insertIntoDbFromCsvJob">
    <step id="loadRegistrations" next="finalizeRegistrations">
        <!-- ... -->
    </step>
    <split id="finalizeRegistrations" >
        <flow>
            <step id="reportStatistics" ><!-- ... --></step>
        </flow>
        <flow>
            <step id="sendJmsNotifications" ><!-- ... --></step>
        </flow>
    </split>
</job>
```

Or for those using Java based configuration can use the `split()` method on the job builder. To make a step into a flow the `flow()` method of the job builder can be used then to add more steps to the flow these can be added with the `next()` method.

The `split()` method requires a `TaskExecutor` to be set, see Recipe 3-22 for more information on scheduling and concurrency.

```
@Bean
public Job insertIntoDbFromCsvJob() {
    JobBuilder builder = jobs.get("insertIntoDbFromCsvJob");
    return builder
        .start(loadRegistrations())
        .split(taskExecutor())
        .add(
            builder.flow(reportStatistics()),
            builder.flow(sendJmsNotifications()))
        .build();
}
```

In this example, there's nothing to prevent you from having many steps within the `flow` elements, nor was there anything preventing you from having more steps after the `split` element. The `split` element, like the step elements, takes a `next` attribute as well.

Spring Batch provides a mechanism to offload processing to another process. This distribution requires some sort of durable, reliable connection. This is a perfect use of JMS because it's rock-solid and transactional, fast, and reliable. Spring Batch support is modeled at a slightly higher level, on top of the Spring Integration abstractions for Spring Integration channels. This support is not in the main Spring Batch code, it can be found in the `spring-batch-integration` project. Remote chunking lets individual steps read and aggregate items as usual in the main thread. This step is called the `Master`. Items read are sent to the `ItemProcessor<I,O>/ItemWriter<T>` running in another process (this is called the `Slave`). If the `Slave` is an aggressive consumer, you have a simple, generic mechanism to scale: work is instantly farmed out over as many JMS clients as you can throw at it. The `aggressive-consumer` pattern refers to the arrangement of multiple JMS clients all consuming the same queue's messages. If one client consumes a message and is busy processing, other idle queues will get the message instead. As long as there's a client that's idle, the message will be processed instantly.

Additionally, Spring Batch supports implicitly scaling out using a feature called *partitioning*. This feature is interesting because it's built in and generally very flexible. You replace your instance of a step with a subclass, `PartitionStep`, which knows how to coordinate distributed executors and maintains the metadata for the execution of the step, thus eliminating the need for a durable medium of communication as in the "remote chunking" technology.

The functionality here is also very generic. It could, conceivably, be used with any sort of grid fabric technology such as GridGain or Hadoop. Spring Batch ships with only a `TaskExecutorPartitionHandler`, which executes steps in multiple threads using a `TaskExecutor` strategy. This simple improvement might be enough of a justification for this feature! If you're really hurting, however, you can extend it.

Conditional Steps with Statuses

Using the `ExitStatus` of a given job or step to determine the next step is the simplest example of a conditional flow. Spring Batch facilitates this through the use of the `stop`, `next`, `fail`, and `end` elements. By default, assuming no intervention, a step will have an `ExitStatus` that matches its `BatchStatus`, which is a property whose values are defined in an enum and may be any of the following: `COMPLETED`, `STARTING`, `STARTED`, `STOPPING`, `STOPPED`, `FAILED`, `ABANDONED`, or `UNKNOWN`.

Let's look at an example that executes one of two steps based on the success of a preceding step:

```
<step id="step1" >
    <next on="COMPLETED" to="step2" > <!-- ... --></step>
    <next on="FAILED" to="failureStep" > <!-- ... --></step>
</step>
```

It's also possible to provide a wildcard. This is useful if you want to ensure a certain behavior for any number of BatchStatus, perhaps in tandem with a more specific next element that matches only one BatchStatus.

```
<step id="step1" >
    <next on="COMPLETED" to="step2" > <!-- ... --></step>
    <next on="*" to="failureStep" > <!-- ... --></step>
</step>
```

In this example, you are instructing Spring Batch to perform some step based on any unaccounted-for ExitStatus. Another option is to just stop processing altogether with a BatchStatus of FAILED. You can do this using the fail element. A less aggressive rewrite of the preceding example might be the following:

```
<step id="step1" >
    <next on="COMPLETED" to="step2" />
    <fail on="FAILED" />
    <!-- ... -->
</step>
```

In all these examples, you're reacting to the standard BatchStatuses that the Spring Batch framework provides. But it's also possible to raise your own ExitStatus. If, for example, you wanted the whole job to fail with a custom ExitStatus of "MAN DOWN", you might do something like this:

```
<step id="step1" next="step2"><!-- ... --></step>
<step id="step2" ...>
    <fail on="FAILED" exit-code="MAN DOWN" />
    <next on="*" to="step3"/>
</step>
<step id="step3"><!-- ... --></step>
```

Finally, if all you want to do is end processing with a BatchStatus of COMPLETED, you can use the end element. This is an explicit way of ending a flow as if it had run out of steps and incurred no errors.

```
<next on="COMPLETED" to="step2" />
<step id="step2" >
    <end on="COMPLETED"/>
    <next on="FAILED" to="errorStep"/>
    <!-- ... -->
</step>
```

When using Java based configuration there is the on() method to specify what needs to happen. Either one can go to the next step, stop or end the job or explicitly fail the job. For these there are several helper methods like to(), stop(), end() and fail() available on the builder.

```

@Bean
public Job insertIntoDbFromCsvJob() {
    JobBuilder builder = jobs.get("insertIntoDbFromCsvJob");
    return builder
        .start(step1())
        .on("FAILED").fail()
        .on("COMPLETED").end()
        .on("ERROR").to(errorStep())
        .on("*").to(step2())
        .build();
}

```

Conditional Steps with Decisions

If you want to vary the execution flow based on some logic more complex than a job's ExitStatuses, you may give Spring Batch a helping hand by using a decision element and providing it with an implementation of a JobExecutionDecider.

```

package com.apress.springrecipes.springbatch;

import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.StepExecution;
import org.springframework.batch.core.job.flow.FlowExecutionStatus;
import org.springframework.batch.core.job.flow.JobExecutionDecider;

public class HoroscopeDecider implements JobExecutionDecider {

    private boolean isMercuryIsInRetrograde (){ return Math.random() > .9 ; }

    public FlowExecutionStatus decide(JobExecution jobExecution,
                                       StepExecution stepExecution) {
        if (isMercuryIsInRetrograde()) {
            return new FlowExecutionStatus("MERCURY_IN_RETROGRADE");
        }
        return FlowExecutionStatus.COMPLETED;
    }
}

```

All that remains is the XML configuration:

```

<beans:bean id="horoscopeDecider" class="com.apress.springrecipes.springbatch.HoroscopeDecider"/>

<job id="job">
    <step id="step1" next="decision" ><!-- ... --></step>
    <decision id="decision" decider="horoscopeDecider">
        <next on="MERCURY_IN_RETROGRADE" to="step2" />
        <next on="COMPLETED" to="step3" />
    </decision>
    <step id="step2" next="step3"> <!-- ... --> </step>
    <step id="step3" parent="s3"> <!-- ... --> </step>
</job>

```

The equivalent in Java config

```
@Bean
public Job insertIntoDbFromCsvJob() {
    JobBuilder builder = jobs.get("insertIntoDbFromCsvJob");
    return builder
        .start(step1())
        .next((horoscopeDecider()))
        .on("MERCURY_IN_RETROGRADE").to(step2())
        .on(("COMPLETED ")).to(step3())
    .build();
}
```

12-8. Launching a Job

Problem

What deployment scenarios does Spring Batch support? How does Spring Batch launch? How does Spring Batch work with a system scheduler such as cron or autosys, or from a web application?

Solution

Spring Batch works well in all environments that Spring runs: your public static void main, OSGi, a web application—anywhere! Some use cases are uniquely challenging, though: it is rarely practical to run Spring Batch in the same thread as an HTTP response because it might end up stalling execution, for example. Spring Batch supports asynchronous execution for just this scenario. Spring Batch also provides a convenience class that can be readily used with cron or autosys to support launching jobs. Additionally, Spring's excellent scheduler namespace provides a great mechanism to schedule jobs.

How It Works

Before you get into creating a solution, it's important to know what options are available for deploying and running these solutions. All solutions require, at minimum, a job and a JobLauncher. You already configured these components in the previous recipe. The job is configured in your Spring XML application context, as you'll see later. The simplest example of launching a Spring Batch solution from Java code is about five lines of Java code, three if you've already got a handle to the ApplicationContext!

```
package com.apress.springrecipes.springbatch.*;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.Date;
```

```

public class Main {
    public static void main(String[] args) throws Throwable {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("solution2.xml");

        JobLauncher jobLauncher = ctx.getBean("jobLauncher", JobLauncher.class);
        Job job = ctx.getBean("myJobName". Job.class);
        JobExecution jobExecution = jobLauncher.run(job, new JobParameters());
    }
}

```

As you can see, the `JobLauncher` reference you configured previously is obtained and used to then launch an instance of a `Job`. The result is a `JobExecution`. You can interrogate the `JobExecution` for information on the state of the `Job`, including its exit status and runtime status.

```

JobExecution jobExecution = jobLauncher.run(job, jobParameters);
BatchStatus batchStatus = jobExecution.getStatus();
while(batchStatus.isRunning()) {
    System.out.println( "Still running..." );
    Thread.sleep( 10 * 1000 ); // 10 seconds
}

```

You can also get the `ExitStatus`:

```
System.out.println( "Exit code: "+ jobExecution.getExitStatus().getExitCode());
```

The `JobExecution` also provides a lot of other very useful information like the create time of the `Job`, the start time, the last updated date, and the end time—all as `java.util.Date` instances. If you want to correlate the job back to the database, you'll need the `JobInstance` and the ID:

```

JobInstance jobInstance = jobExecution.getJobInstance();
System.out.println( "job instance Id: "+ jobInstance.getId());

```

In our simple example, we use an empty `JobParameters` instance. In practice, this will only work once. Spring Batch builds a unique key based on the parameters and uses to keep uniquely identify one run of a given `Job` from another. You'll learn about parameterizing a `Job` in detail in the next recipe.

Launching From a Web Application

Launching a `job` from a web application requires a slightly different approach, because the client thread (presumably an HTTP request) can't usually wait for a batch job to finish. The ideal solution is to have the job execute asynchronously when launched from a controller or action in the web tier, unattended by the client thread. Spring Batch supports this scenario through the use of a Spring `TaskExecutor`. This requires a simple change to the configuration for the `JobLauncher`, although the Java code can stay the same. Here, we will use a `SimpleAsyncTaskExecutor` that will spawn a thread of execution and manage that thread without blocking.

```

<beans:bean id="jobLauncher"
class="org.springframework.batch.execution.launch.support.SimpleJobLauncher"
    p:jobRepository-ref="jobRepository" >
    <beans:property name="taskExecutor">
        <beans:bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </beans:property>
</beans:bean>

```

To do the same with Java Config requires a bit more work. As we cannot use the default settings anymore we need to add our own implementation of a BatchConfigurer to configure the TaskExecutor and add it to our SimpleJobLauncher. For this implementation we used the DefaultBatchConfigurer as a reference, we only reimplemented the createJobLauncher method to add the TaskExecutor.

```
@Component
public class CustomBatchConfigurer implements BatchConfigurer {

    private DataSource dataSource;
    private PlatformTransactionManager transactionManager;
    private JobRepository jobRepository;
    private JobLauncher jobLauncher;

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

    protected CustomBatchConfigurer() {}

    @Override
    public JobRepository getJobRepository() {
        return jobRepository;
    }

    @Override
    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }

    @Override
    public JobLauncher getJobLauncher() {
        return jobLauncher;
    }

    public TaskExecutor taskExecutor() {
        return new SimpleAsyncTaskExecutor();
    }

    @PostConstruct
    public void initialize() throws Exception {
        this.jobRepository = createJobRepository();
        this.jobLauncher = createJobLauncher();
    }

    protected JobLauncher createJobLauncher() throws Exception {
        SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
        jobLauncher.setJobRepository(jobRepository);
        jobLauncher.setTaskExecutor(taskExecutor());
        return jobLauncher;
    }
}
```

```

protected JobRepository createJobRepository() throws Exception {
    JobRepositoryFactoryBean factory = new JobRepositoryFactoryBean();
    factory.setDataSource(dataSource);
    factory.setTransactionManager(transactionManager);
    factory.afterPropertiesSet();
    return (JobRepository) factory.getObject();
}
}

```

Running from the Command Line

Another common use case is deployment of a batch process from a system scheduler such as cron or autosys, or even Window's event scheduler. Spring Batch provides a convenience class that takes as its parameters the name of the XML application context (that contains *everything* required to run a job) as well as the name of the job bean itself. Additional parameters may be provided and used to parameterize the job. These parameters must be in the form name=value. An example invocation of this class on the command line (on a Linux/Unix system), assuming that you set up the classpath, might look like this:

```
java CommandLineJobRunner jobs.xml hourlyReport date=`date +%m/%d/%Y` time=`date +%H`
```

The CommandLineJobRunner will even return system error codes (0 for success, 1 for failure, and 2 for an issue with loading the batch job) so that a shell (such as used by most system schedulers) can react or do something about the failure. More complicated return codes can be returned by creating and declaring a top-level bean that implements the interface ExitCodeMapper, in which you can specify a more useful translation of exit status messages to integer-based error codes that the shell will see on process exit.

Running On A Schedule

Spring 3.0 debuts support for a scheduling framework (see also Recipe 3-22). This framework lends itself perfectly to running Spring Batch. First, let's modify our existing application context batch.xml to use the Spring scheduling namespace. The additions consist mainly of changes to schema imports, as well as the declaration of a few beans. The application context file now starts off like so:

```

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

```

```

http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc.xsd
http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task.xsd">

<context:component-scan base-package="com.apress.springrecipes.springbatch"/>

<task:scheduler id="scheduler" pool-size="10"/>
<task:executor id="executor" pool-size="10"/>

<task:annotation-driven scheduler="scheduler" executor="executor"/>

...

```

These imports enable the simplest possible support for scheduling. The preceding declaration ensures that any bean under the package com.apress.springrecipes.springbatch will be configured and scheduled as required. Our bean is as follows:

```

package com.apress.springrecipes.springbatch.scheduler;

import org.apache.commons.lang3.StringUtils;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JobScheduler {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job job;

    public void runRegistrationsJob(Date date) throws Throwable {
        System.out.println(StringUtils.repeat("-", 100));
        System.out.println("Starting job at " + date.toString());
        JobParametersBuilder jobParametersBuilder = new JobParametersBuilder();
        jobParametersBuilder.addDate("date", date);
        jobParametersBuilder.addString("input.file", "registrations");
        JobParameters jobParameters = jobParametersBuilder.toJobParameters();
        JobExecution jobExecution = jobLauncher.run(job, jobParameters);
        System.out.println("jobExecution finished, exit code: " + jobExecution.getExitStatus().getExitCode());
    }
}

```

```

@Scheduled(fixedDelay = 1000 * 10)
public void runRegistrationsJobOnASchedule() throws Throwable {
    runRegistrationsJob(new Date());
}

}

```

There is nothing particularly novel; it's a good study of how the different components of the Spring framework work well together. The bean is recognized and becomes part of the application context because of the @Component annotation, which we enabled with the context:component-scan element in our batch.xml. There's only one Job in the user-job.xml file and only one JobLauncher, so we simply have those auto-wired into our bean. Finally, the logic for kicking off a batch run is inside the runRegistrationsJob(java.util.Date date) method. This method could be called from anywhere. Our only client for this functionality is the scheduled method runRegistrationsJobOnASchedule. The framework will invoke this method for us, according to the timeline dictated by the @Scheduled annotation.

There are other options for this sort of thing; traditionally in the Java and Spring world, this sort of problem would be a good fit for Quartz. It might still be, as the Spring scheduling support isn't designed to be as extensible as Quartz. If you are in an environment requiring more traditional, ops-friendly scheduling tools, there are of course old standbys like cron, autofs, and BMC, too.

12-9. Parameterizing a Job

Problem

The previous examples work well enough, but they leave something to be desired in terms of flexibility. To apply the batch code to some other file, you'd have to edit the configuration and hard-code the name in there. The ability to parameterize the batch solution would be very helpful.

Solution

Use JobParameters to parameterize a job, which is then available to your steps through Spring Batch's expression language or via API calls.

How It Works

Launching a Job with Parameters

A job is a prototype of a JobInstance. JobParameters are used to provide a way of identifying a unique run of a job (a JobInstance). These JobParameters allow you to give input to your batch process, just as you would with a method definition in Java. You've seen the JobParameters in previous examples but not in detail. The JobParameters object is created as you launch the job using the JobLauncher. To launch a job called dailySalesFigures, with the date for the job to work with, you would write something like this:

```

ApplicationContext ctx = new ClassPathXmlApplicationContext("solution2.xml");
JobLauncher jobLauncher = ctx.getBean("jobLauncher", JobLauncher.class);
Job job = (Job) ctx.getBean("dailySalesFigures");
jobLauncher.run(job, new JobParametersBuilder().addDate("date",
    new Date()).toJobParameters());

```

Accessing JobParameters

Technically, you can get at JobParameters via any of the ExecutionContexts (step and job). Once you have it, you can access the parameters in a type-safe way by calling `getLong()`, `getString()`, and so on. A simple way to do this is to bind to the `@BeforeStep` event, save the StepExecution, and iterate over the parameters this way. From here, you can inspect the parameters and do anything you want with them. Let's look at that in terms of the `ItemProcessor<I,O>` you wrote earlier:

```
// ...
private StepExecution stepExecution;

@BeforeStep
public void saveStepExecution(StepExecution stepExecution) {
    this.stepExecution = stepExecution;
}

public UserRegistration process(UserRegistration input) throws Exception {

    Map<String, JobParameter> params = stepExecution.getJobParameters().getParameters();

    // iterate over all of the parameters
    for (String jobParameterKey : params.keySet()) {
        System.out.println(String.format("%s=%s", jobParameterKey,
            params.get(jobParameterKey).getValue().toString()));
    }

    // access specific parameters in a type safe way
    Date date = stepExecution.getJobParameters().getDate("date");
    // etc ...
}
```

This turns out to be of limited value. The 80 percent case is that you'll need to bind parameters from the job's launch to the Spring beans in the application context. These parameters are available only at runtime, whereas the steps in the XML application context are configured at design time. This happens in many places. Previous examples demonstrated `ItemWriters<T>` and `ItemReaders <T>` with hard-coded paths. That works fine unless you want to parameterize the file name. This is hardly acceptable unless you plan on using a job just once!

The core Spring Framework features an enhanced expression language that Spring Batch uses to defer binding of the parameter until the correct time. Or, in this case, until the bean is in the correct scope. Spring Batch 2.0 introduces the "step" scope for just this purpose. Let's take a look at how you'd rework the previous example to use a parameterized file name for the `ItemReader`'s resource:

```
<beans:bean
    scope="step"
    id="csvFileReader"
    class="org.springframework.batch.item.file.FlatFileItemReader"
    p:resource="file:${user.home}/batches/#${jobParameters['input.fileName']}.csv">
    <!-- ... this is the same as before...-->
</beans:bean>
```

All you did is scope the bean (the `FlatFileItemReader<T>`) to the life cycle of a step (at which point those `JobParameters` will resolve correctly) and then used the EL syntax to parameterize the path to work off of.

Note that the earlier versions of Spring Batch (before 2.0) featured an expression language in advance of the Spring Expression Language that debuted in the core framework for 3.0 (SpEL). This expression language is by and large similar to SpEL but is not exact. In the preceding example, we reference the input variable as `#{jobParameters['input.fileName']}`, which works in Spring Batch with SpEL. In the original Spring Batch expression language, however, you could have left off the quotes.

Summary

This chapter introduced you to the concepts of batch processing, some of its history, and why it fits in a modern day architecture. You learned about Spring Batch, the batch processing from SpringSource, and how to do reading and writing with `ItemReader<T>` and `ItemWriter<T>` implementations in your batch jobs. You wrote your own `ItemReader<T>`, and `ItemWriter <T>`implementations as needed and saw how to control the execution of steps inside a job.

CHAPTER 13

NoSQL and BigData

Most applications use a relational database like Oracle, MySQL, or Postgresql; however there is more too data storage then just SQL databases. There are:

1. Relational Databases (Oracle, Mysql, Postgresql, etc.)
2. Document Stores (MongoDB, Couchbase)
3. Key-Value Stores (Redis, Volgemort)
4. Column Stores (Cassandra)
5. Graph Stores (Neo4j, Giraph)

Although each of these different technologies (and even the different implementations) have their own use, they can also be hard to use or configure. Additionally, sometimes it might feel that one writes a lot of duplicated plumbing code for handling transactions and error translation.

The Spring Data project can help make life easier; it can help configure the different technologies with the plumbing code. Each of the integration modules will have support for exception translation to Spring's consistent `DataAccessException` hierarchy and the use of Spring's templating approach.

Spring Data also provides a cross-storage solution for some technologies which means part of your model can be stored in a relational database with JPA and the other part can be stored in a graph or document store.

13-1. Using MongoDB

Problem

You want to use MongoDB to store and retrieve documents.

Solution

Download and configure MongoDB.

How It Works

Downloading and Starting MongoDB

First download MongoDB from <http://www.mongodb.org>. Select the one that is applicable for the system in use and follow the installation instructions in the manual (<http://docs.mongodb.org/manual/installation/>). When the installation is complete MongoDB can be started.

To start MongoDB execute the `mongodb` command on the command line (see Figure 13-1). This will start a MongoDB server on port 27017. If a different port is required this can be done by specifying the `--port` option on the command line when starting the server.

```
marten@imac-van-marten:~$ mongod
mongod --help for help and startup options
2014-09-22T09:34:50.026+0200 [initandlisten] MongoDB starting : pid=31541 port=27017 dbpath=/data/db 64-bit host=imac-van-marten.dynamic.ziggo.nl
2014-09-22T09:34:50.026+0200 [initandlisten] ** WARNING: soft file limits too low. Number of files is 256, should be at least 1000
2014-09-22T09:34:50.026+0200 [initandlisten] db version v2.6.4
2014-09-22T09:34:50.026+0200 [initandlisten] git version: 9014f3a
2014-09-22T09:34:50.026+0200 [initandlisten] build info: Darwin minilmountain.local 12.5.0 Darwin Kernel Version 12.5.0: Sun Sep 29 13:33:47 PDT 2013; root:xnu-2050.48.12~1/RELEASE_X86_64 BOOST_LIB_VERSION=1_49
2014-09-22T09:34:50.026+0200 [initandlisten] allocator: tcmalloc
2014-09-22T09:34:50.026+0200 [initandlisten] options: {}
2014-09-22T09:34:50.027+0200 [initandlisten] journal dir=/data/db/journal
2014-09-22T09:34:50.027+0200 [initandlisten] recover : no journal files present, no recovery needed
2014-09-22T09:34:50.184+0200 [initandlisten] preallocateIfNotFaster=true 2.78
2014-09-22T09:34:50.207+0200 [initandlisten] Waiting for connections on port 27017
```

Figure 13-1. Output after initial start of MongoDB

The default location for storing data is `\data\db` (for Windows users this is from the root of the disk where MongoDB was installed!) to change the path use the `--dbpath` option on the command line. Make sure that the directory exists and is writeable for MongoDB.

Connecting to MongoDB

For a connection to MongoDB an instance of Mongo is needed. This instance can be used to get the database to use and the actual underlying collection(s). Let's create a small system that uses MongoDB first to create an object to use for storage.

```
package com.apress.springrecipes.nosql;

public class Vehicle {

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    public Vehicle() {
    }

    public Vehicle(String vehicleNo, String color, int wheel, int seat) {
        this.vehicleNo = vehicleNo;
        this.color = color;
        this.wheel = wheel;
        this.seat = seat;
    }
    /// Getters and Setters have been omitted for brevity.
}
```

To work with this object, create a repository interface.

```
package com.apress.springrecipes.nosql;

public interface VehicleRepository {

    public long count();
    public void save(Vehicle vehicle);
    public void delete(Vehicle vehicle);
    public List<Vehicle> findAll();
    public Vehicle findByVehicleNo(String vehicleNo);

}
```

For MongoDB create the MongoDBVehicleRepository implementation of the VehicleRepository.

```
package com.apress.springrecipes.nosql;

import com.mongodb.*;
import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.List;

public class MongoDBVehicleRepository implements VehicleRepository {

    private final Mongo mongo;
    private final String collectionName;
    private final String databaseName;

    public MongoDBVehicleRepository(Mongo mongo, String databaseName, String collectionName) {
        this.mongo = mongo;
        this.databaseName=databaseName;
        this.collectionName = collectionName;
    }

    @Override
    public long count() {
        return getCollection().count();
    }

    @Override
    public void save(Vehicle vehicle) {
        BasicDBObject query = new BasicDBObject("vehicleNo", vehicle.getVehicleNo());
        DBObject dbVehicle = transform(vehicle);

        DBObject fromDB = getCollection().findAndModify(query, dbVehicle);
        if (fromDB == null) {
            getCollection().insert(dbVehicle);
        }
    }
}
```

```

@Override
public void delete(Vehicle vehicle) {
    BasicDBObject query = new BasicDBObject("vehicleNo", vehicle.getVehicleNo());
    getCollection().remove(query);
}

@Override
public List<Vehicle> findAll() {
    DBCursor cursor = getCollection().find(null);
    List<Vehicle> vehicles = new ArrayList<>(cursor.size());
    for (DBObject dbObject : cursor) {
        vehicles.add(transform(dbObject));
    }
    return vehicles;
}

@Override
public Vehicle findByVehicleNo(String vehicleNo) {
    BasicDBObject query = new BasicDBObject("vehicleNo", vehicleNo);
    DBObject dbVehicle = getCollection().findOne(query);
    return transform(dbVehicle);
}

private DBCollection getCollection() {
    return mongo.getDB(databaseName).getCollection(collectionName);
}

private Vehicle transform(DBObject dbVehicle) {
    return new Vehicle(
        (String) dbVehicle.get("vehicleNo"),
        (String) dbVehicle.get("color"),
        (int) dbVehicle.get("wheel"),
        (int) dbVehicle.get("seat"));
}

private DBObject transform(Vehicle vehicle) {
    BasicDBObject dbVehicle = new BasicDBObject("vehicleNo", vehicle.getVehicleNo())
        .append("color", vehicle.getColor())
        .append("wheel", vehicle.getWheel())
        .append("seat", vehicle.getSeat());
    return dbVehicle;
}

@PostConstruct
public void shutdown() {
    mongo.getDB(databaseName).dropDatabase();
}

```

First notice the constructor it takes three arguments. The first is the actual MongoDB client, the second is the name of the database that is going to be used, and the last is the name of the collection in which the objects are stored. Documents in mongodb are stored in collections and a collection belongs to a database.

For easy access to the DBCollection used there is the `getCollection` method which gets a connection to the DB and returns the configured DBCollection. This DBCollection can then be used to execute operations like storing, deleting, or updating documents.

The `save` method will first try to update an existing document. If this fails a new document for the given `Vehicle` will be created. To store objects, start by transforming the domain object `Vehicle` into a `DBObject` in this case a `BasicDBObject`. The `BasicDBObject` takes key/value pairs of the different properties of our `Vehicle` object.

When querying for a document the same `DBObject` is used, the key/value pairs that are present on the given object are used to lookup documents, an example can be found in the `findByVehicleNo` method in the repository.

Conversion from and to `Vehicle` objects is done through the two `transform` methods.

To use this class create the following Main class.

```
package com.apress.springrecipes.nosql;

import com.mongodb.MongoClient;
import org.apache.commons.lang3.builder.ToStringBuilder;
import org.apache.commons.lang3.builder.ToStringStyle;

import java.util.List;

public class Main {

    public static final String DB_NAME = "vehicledb";

    public static void main(String[] args) throws Exception {
        // Default mongoclient for localhost and port 27017
        MongoClient mongo = new MongoClient();

        VehicleRepository repository = new MongoDBVehicleRepository(mongo, DB_NAME, "vehicles");

        System.out.println("Number of Vehicles: " + repository.count());

        repository.save(new Vehicle("TEM0001", "RED", 4, 4));
        repository.save(new Vehicle("TEM0002", "RED", 4, 4));

        System.out.println("Number of Vehicles: " + repository.count());

        Vehicle v = repository.findByVehicleNo("TEM0001");

        System.out.println(ToStringBuilder.reflectionToString(v, ToStringStyle.SHORT_PREFIX_STYLE));

        List<Vehicle> vehicleList = repository.findAll();

        System.out.println("Number of Vehicles: " + vehicleList.size());

        for (Vehicle vehicle : vehicleList) {
            repository.delete(vehicle);
        }
    }
}
```

```

        System.out.println("Number of Vehicles: " + repository.count());

        // Cleanup and close
        mongo.getDB(DB_NAME).dropDatabase();
        mongo.close();
    }
}

```

The main class constructs an instance of the MongoClient which will try to connect to port 27017 on localhost for a MongoDB instance. If another port or host is needed there is also a constructor which takes a host and port as parameters new MongoClient("mongodb-server.local", 28018).

Next an instance of the MongoDBVehicleRepository is constructed, the earlier constructed MongoClient is passed as well as the name of the database, vehicledb, and name of the collection, vehicles.

The next lines of code will insert two vehicles into the database, try to find them, and finally delete them. The last lines in the main class will close the mongo instance and before doing so will drop the database. The latter is something you don't want to do when using a production database.

Using Spring for Configuration

The setup and configuration of the MongoClient and MongoDBVehicleRepository can easily be moved to Spring configuration.

```

package com.apress.springrecipes.nosql.config;

import com.apress.springrecipes.nosql.MongoDBVehicleRepository;
import com.apress.springrecipes.nosql.VehicleRepository;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.net.UnknownHostException;

@Configuration
public class MongoConfiguration {

    public static final String DB_NAME = "vehicledb";

    @Bean
    public Mongo mongo() throws UnknownHostException {
        return new MongoClient();
    }

    @Bean
    public VehicleRepository vehicleRepository(Mongo mongo) {
        return new MongoDBVehicleRepository(mongo, DB_NAME, " vehicles");
    }
}

```

The following method has been added to the MongoDBVehicleRepository to take care of the cleanup of the database.

```
@PreDestroy
public void cleanUp() {
    mongo.getDB(databaseName).dropDatabase();
}
```

Finally the Main program needs to be updated to reflect the changes.

```
package com.apress.springrecipes.nosql;

...
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;

import java.util.List;

public class Main {

    public static final String DB_NAME = "vehicledb";

    public static void main(String[] args) throws Exception {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(MongoConfiguration.class);
        VehicleRepository repository = ctx.getBean(VehicleRepository.class);

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

The configuration is loaded by an AnnotationConfigApplicationContext from this context the VehicleRepository bean is retrieved and used to execute the operations. When the code that has run the context is closed, it triggers the cleanUp method in the MongoDBVehicleRepository.

Using a MongoTemplate to Simplify MongoDB code

At the moment the MongoDBVehicleRepository uses the plain MongoDB API. Although not very complex it still requires knowledge about the API. Next to that there are some repetitive tasks like mapping from and to a Vehicle object. Using a MongoTemplate can simplify the repository considerably.

Note Before using Spring Data Mongo the relevant jars need to be added to the classpath. When using Maven add the following dependency:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>1.6.0.RELEASE</version>
</dependency>
```

```
package com.apress.springrecipes.nosql;

import com.mongodb.DB;
import com.mongodb.MongoException;
import org.springframework.dao.DataAccessException;
import org.springframework.data.mongodb.core.DbCallback;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Query;

import javax.annotation.PreDestroy;
import java.util.List;

import static org.springframework.data.mongodb.core.query.Criteria.where;

public class MongoDBVehicleRepository implements VehicleRepository {

    private final MongoTemplate mongo;
    private final String collectionName;

    public MongoDBVehicleRepository(MongoTemplate mongo, String collectionName) {
        this.mongo = mongo;
        this.collectionName = collectionName;
    }

    @Override
    public long count() {
        return mongo.count(null, collectionName);
    }

    @Override
    public void save(Vehicle vehicle) {
        mongo.save(vehicle, collectionName);
    }

    @Override
    public void delete(Vehicle vehicle) {
        mongo.remove(vehicle, collectionName);
    }
}
```

```

@Override
public List<Vehicle> findAll() {
    return mongo.findAll(Vehicle.class, collectionName);
}

@Override
public Vehicle findByVehicleNo(String vehicleNo) {
    return mongo.findOne(new Query(where("vehicleNo").is(vehicleNo)), Vehicle.class,
        collectionName);
}

@PreDestroy
public void cleanUp() {
    mongo.execute(new DbCallback<Object>() {
        @Override
        public Object doInDB(DB db) throws MongoException, DataAccessException {
            db.dropDatabase();
            return null;
        }
    });
}
}

```

The code looks a lot cleaner when using a `MongoTemplate`. It has convenience methods for almost every operation save, update, and delete. Additionally, it has a very nice query builder approach (see the `findByVehicleNo` method).

There are no more mappings to and from the mongo classes so there is no need to create a `DBObject` anymore. That burden is now handled by the `MongoTemplate`. To convert the `Vehicle` object to the MongoDB classes a `MongoConverter` is used. By default a `MappingMongoConverter` is used. This mapper maps properties to attribute names and vice versa and while doing so also tries to convert from and to the correct datatype. If a specific mapping is needed it is possible to write your own implementation of a `MongoConverter` and register it with the `MongoTemplate`.

Due to the use of the `MongoTemplate` the configuration needs to be modified.

```

package com.apress.springrecipes.nosql.config;

import com.apress.springrecipes.nosql.MongoDBVehicleRepository;
import com.apress.springrecipes.nosql.VehicleRepository;
import com.mongodb.Mongo;
import com.mongodb.MongoClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.core.MongoFactoryBean;
import org.springframework.data.mongodb.core.MongoTemplate;

import java.net.UnknownHostException;

```

```

@Configuration
public class MongoConfiguration {

    public static final String DB_NAME = "vehicledb";

    @Bean
    public MongoTemplate mongo() throws Exception {
        return new MongoTemplate(mongoFactoryBean().getObjectType(), DB_NAME);
    }

    @Bean
    public MongoFactoryBean mongoFactoryBean() {
        return new MongoFactoryBean();
    }

    @Bean
    public VehicleRepository vehicleRepository(MongoTemplate mongo) {
        return new MongoDBVehicleRepository(mongo, "vehicles");
    }

}

```

Notice the use of the `MongoFactoryBean`. It allows for easy setup of the `MongoClient`. It isn't a requirement for using the `MongoTemplate` but it makes it easier to configure the client. Another benefit is that there is no more `java.net.UnknownHostException` thrown that is handled internally by the `MongoFactoryBean`.

The `MongoTemplate` has various constructors. The one used here takes a `Mongo` instance and the name of the database to use. To resolve the database an instance of a `MongoDbFactory` is used; by default the `SimpleMongoDbFactory`. In most cases this is sufficient but if some special case arises, like encrypted connections, it is quite easy to extend the default implementation.

Finally the `MongoTemplate` is injected, together with the name of the collection, into the `MongoDBVehicleRepository`.

A final addition needs to be made to the `Vehicle` object. It is required that a field is available for storing the generated id. This can be either a field with the name `id` or a field with the `@Id` annotation.

```

public class Vehicle {

    private String id;

    ...
}

```

Using Annotations to Specify Mapping Information

Currently the `MongoDBVehicleRepository` needs to know the name of the collection we want to access. It would be easier and more flexible if this could be specified on the `Vehicle` object, just as with a JPA `@Table` annotation. With Spring Data Mongo this is possible using the `@Document` annotation.

```
package com.apress.springrecipes.nosql;  
  
import org.springframework.data.mongodb.core.mapping.Document;  
  
@Document(collection = "vehicles")  
public class Vehicle { ... }
```

The `@Document` annotation can take two attributes: `collection` and `language`. The `collection` property is for specifying the name of the collection to use, the `language` property is for specifying the language for this object. Now that the mapping information is on the `Vehicle` class the collection name can be removed from the `MongoDBVehicleRepository`.

```
public class MongoDBVehicleRepository implements VehicleRepository {  
  
    private final MongoTemplate mongo;  
  
    public MongoDBVehicleRepository(MongoTemplate mongo) {  
        this.mongo = mongo;  
    }  
  
    @Override  
    public long count() {  
        return mongo.count(null, Vehicle.class);  
    }  
  
    @Override  
    public void save(Vehicle vehicle) {  
        mongo.save(vehicle);  
    }  
  
    @Override  
    public void delete(Vehicle vehicle) {  
        mongo.remove(vehicle);  
    }  
  
    @Override  
    public List<Vehicle> findAll() {  
        return mongo.findAll(Vehicle.class);  
    }  
  
    @Override  
    public Vehicle findByVehicleNo(String vehicleNo) {  
        return mongo.findOne(new Query(where("vehicleNo").is(vehicleNo)), Vehicle.class);  
    }  
}
```

Of course the collection name can be removed from the configuration of the `MongoDBVehicleRepository` as well.

```
@Configuration
public class MongoConfiguration {
    ...
    @Bean
    public VehicleRepository vehicleRepository(MongoTemplate mongo) {
        return new MongoDBVehicleRepository(mongo);
    }
}
```

When running the Main class the result should still be the same as it was before.

Create a Spring Data MongoDB Repository

Although the code has been reduced a lot in that there is no more mapping from and to MongoDB classes and no more collection names passing around, it can still be reduced even further. Leveraging another feature of Spring Data Mongo the complete implementation of the MongoDBVehicleRepository could be removed.

First the configuration needs to be modified.

```
package com.apress.springrecipes.nosql.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.mongodb.core.MongoFactoryBean;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.repository.config.EnableMongoRepositories;

@Configuration
@EnableMongoRepositories(basePackages = "com.apress.springrecipes.nosql")
public class MongoConfiguration {

    public static final String DB_NAME = "vehicledb";

    @Bean
    public MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongoFactoryBean().getObjectType(), DB_NAME);
    }

    @Bean
    public MongoFactoryBean mongoFactoryBean() {
        return new MongoFactoryBean();
    }
}
```

First, notice the removal of the @Bean method that constructed the MongoDBVehicleRepository. Second, notice the addition of the @EnableMongoRepositories annotation. This enables detection of interfaces that extend the Spring Data CrudRepository and are used for domain objects annotated with @Document.

To have our VehicleRepository detected by Spring Data we need to let it extend CrudRepository or one of its sub-interfaces like MongoRepository.

```

package com.apress.springrecipes.nosql;

import org.springframework.data.mongodb.repository.MongoRepository;

public interface VehicleRepository extends MongoRepository<Vehicle, String> {

    public Vehicle findByVehicleNo(String vehicleNo);

}

```

You might wonder where all the methods have gone. They are already defined in the super interfaces and as such can be removed from this interface. The `findByVehicleNo` method is still there. This method will still be used to lookup a `Vehicle` by its `vehicleNo` property. All the `findBy` methods are converted into a MongoDB query. The part after the `findBy` is interpreted as a property name. It is also possible to write more complex queries using different operators like `and`, `or`, and `between`.

Now running the Main class again should still result in the same output, however the actual code written to work with MongoDB has been minimized.

13-2. Using Redis

Redis is a key-value cache or store and it will only hold simple datatypes like strings and hashes. When storing more complex datastructures, conversion from and to that datastructure is needed.

Downloading and Starting Redis

Redis sources can be downloaded from <http://redis.io/download> at the moment of writing version 2.8.17 was the recently released stable version. A compiled version for Windows can be found at <https://github.com/MSOpenTech/redis/tree/2.8/bin/release>. The official download site only provides unix binaries. Mac users can use homebrew (<http://brew.sh>) to install Redis.

After downloading and installing Redis, start it using the `redis-server` command from the command line. When started, the output should be similar to that in Figure 13-2, it will output the process-id (PID) and the port number (default 6379) it listens on.

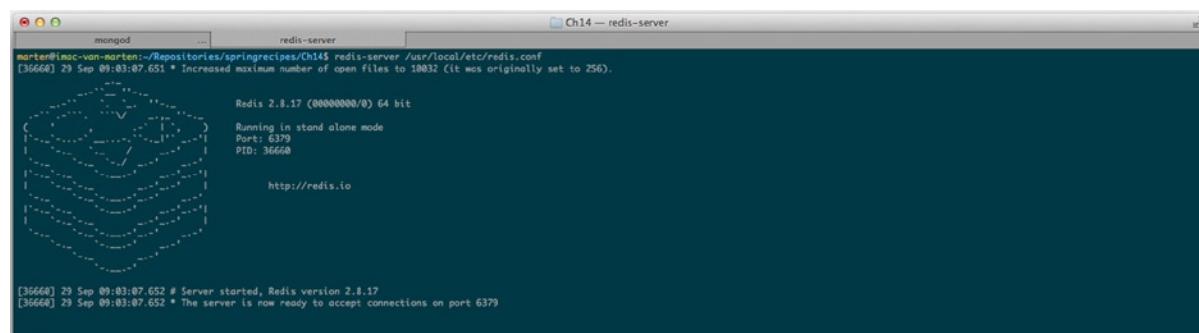


Figure 13-2. Output after starting Redis

Connecting to Redis

To be able to connect to Redis a client is needed, much like a JDBC driver to connect to a database. There are several clients available. A full list can be found on the Redis website (<http://redis.io/clients>). For this recipe the Jedis client will be used as that is quite active and recommended by the Redis team.

When using maven add the following dependency for the client.

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.6.0</version>
</dependency>
```

Let's start with a simple hello world sample to see if a connection to Redis can be made.

```
package com.apress.springrecipes.nosql;

import redis.clients.jedis.Jedis;

public class Main {

    public static void main(String[] args) {
        Jedis jedis = new Jedis("localhost");
        jedis.set("msg", "Hello World, from Redis!");
        System.out.println(jedis.get("msg"));
    }
}
```

A Jedis client is created and passed the name of the host to connect to, in this case simply localhost. The set method on the Jedis client will put a message in the store and with get the message is retrieved again.

Instead of a simple object you could also have Redis mimic a List or a Map.

```
package com.apress.springrecipes.nosql;

import redis.clients.jedis.Jedis;

public class Main {

    public static void main(String[] args) {
        Jedis jedis = new Jedis("localhost");
        jedis.rpush("authors", "Marten Deinum", "Josh Long", "Daniel Rubio", "Gary Mak");
        System.out.println("Authors: " + jedis.lrange("authors", 0, -1));

        jedis.hset("sr_3", "authors", "Gary Mak, Daniel Rubio, Josh Long, Marten Deinum");
        jedis.hset("sr_3", "published", "2014");

        System.out.println("Spring Recipes 3rd: " + jedis.hgetAll("sr_3"));
    }
}
```

With rpush and lpush one can add elements to a List, rpush adds the elements to the end of the list and lpush adds them to the start of the list. To retrieve them the lrange or rrange methods can be used, the lrange starts from the left and takes a start and end index. The sample uses -1 that indicates everything.

To add elements to a Map use hset, this takes a key and a field and value, another option is to use hmset (multi set) which takes a Map<String, String>, or Map<byte[], byte[]> as an argument.

Storing Objects with Redis

Redis is a key/value store and can only handle Strings or byte[]. The same goes for the keys. So storing an object in Redis isn't as straightforward as with other technologies. The object needs to be serialized to a String or a byte[] before storing.

Let's reuse the Vehicle class used earlier and store and retrieve that using a Jedis client.

```
package com.apress.springrecipes.nosql;

import java.io.Serializable;

public class Vehicle implements Serializable{

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    public Vehicle() {
    }

    public Vehicle(String vehicleNo, String color, int wheel, int seat) {
        this.vehicleNo = vehicleNo;
        this.color = color;
        this.wheel = wheel;
        this.seat = seat;
    }
    // getters/setters omitted
}
```

Notice the implements Serializable for the Vehicle class. This is needed to make the object serializable voor Java.

Before storing the object it needs to be converted into a byte[] in Java the ObjectOutputStream can write objects and the ByteArrayOutputStream can write to a byte[]. To transform a byte[] into an object again the ObjectInputStream and ByteArrayInputStream are of help. First create a helper class to ease the de-/serialization process.

```
package com.apress.springrecipes.nosql;

import java.io.*;

public class SerializationUtils {

    public byte[] serialize(Object obj) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        try (ObjectOutputStream out = new ObjectOutputStream(baos)) {
            out.writeObject(obj);
        }
    }
}
```

```
        return baos.toByteArray();
    } finally {
        baos.close();
    }
}

public static <T> T deserialize(byte[] bytes) throws IOException, ClassNotFoundException {
    try (ObjectInputStream in = new ObjectInputStream(new ByteArrayInputStream(bytes))) {
        return (T) in.readObject();
    }
}
```

Now in the Main class let's create a Vehicle and store it using Jedis.

```
package com.apress.springrecipes.nosql;

import redis.clients.jedis.Jedis;

public class Main {

    public static void main(String[] args) throws Exception {
        Jedis jedis = new Jedis("localhost");

        final String vehicleNo = "TEM0001";
        Vehicle vehicle = new Vehicle(vehicleNo, "RED", 4, 4);

        jedis.set(vehicleNo.getBytes(), SerializationUtils.serialize(vehicle));
        byte[] vehicleArray = jedis.get(vehicleNo.getBytes());
        System.out.println("Vehicle: " + SerializationUtils.deserialize(vehicleArray));
    }
}
```

First an instance of the `Vehicle` is created. Next the earlier created `SerializationUtils` are used to convert the object into a `byte[]`. When storing a `byte[]` the key also needs to be a `byte[]`; hence the key, here the `vehicleNo`, is converted too. Finally the same key is used to read the serialized object from the store again and converted back into an object again.

The drawback of this approach is that every object that is stored needs to implement the Serializable interface. If this isn't the case the object might be lost or an error during serialization might occur. Next to that the byte[] is a representation of the class. Also now if this class is changed there is a great chance that converting it back into an object will fail.

Another option is to use a String representation of the object. Convert the Vehicle into XML or JSON which would be more flexible than a byte[]. Let's take a look at converting the object into JSON using the excellent Jackson JSON library.

```
package com.apress.springrecipes.nosql;  
  
import com.fasterxml.jackson.databind.ObjectMapper;  
import redis.clients.jedis.Jedis;
```

```

public class Main {

    public static void main(String[] args) throws Exception {
        Jedis jedis = new Jedis("localhost");
        ObjectMapper mapper = new ObjectMapper();
        final String vehicleNo = "TEM0001";
        Vehicle vehicle = new Vehicle(vehicleNo, "RED", 4,4);

        jedis.set(vehicleNo, mapper.writeValueAsString(vehicle));

        String vehicleString = jedis.get(vehicleNo);

        System.out.println("Vehicle: " + mapper.readValue(vehicleString, Vehicle.class));
    }
}

```

First an instance of the `ObjectMapper` is needed. This object is used to convert from and to JSON. When writing the `writeValueAsString` method is used as it will transform the object into a JSON String. This String is then stored in Redis. Next the String is read again and passed to the `readValue` method of the `ObjectMapper`. Based on the type argument, `Vehicle.class` here, an object is constructed and the JSON is mapped to an instance of the given class.

To run this sample a dependency on the Jackson library is needed. Add the following dependency:

```

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.4.2</version>
</dependency>

```

Storing objects when using Redis isn't straightforward and some argue that this isn't how Redis was intended to be used (storing complex object structures).

Configuring and Using the RedisTemplate

Depending on the client library used to connect to Redis it might be harder to use the Redis API. To unify this there is the `RedisTemplate`. It can work with most Redis Java Clients out there. Next to providing a unified approach it also takes care of translating any exceptions into the Springs `DataAccessException` hierarchy. This lets it integrate nicely with any already existing data access and allows it to use Springs transactions support.

First add a dependency on the Spring Data Redis project to your list of dependencies:

```

<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>1.4.0.RELEASE</version>
</dependency>

```

The `RedisTemplate` requires a `RedisConnectionFactory` to be able to get a connection. The `RedisConnectionFactory` is an interface and several implementations are available. In this case the `JedisConnectionFactory` is needed.

```

package com.apress.springrecipes.nosql.config;

import com.apress.springrecipes.nosql.Vehicle;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;

@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String, Vehicle> redisTemplate() {
        RedisTemplate template = new RedisTemplate();
        template.setConnectionFactory(redisConnectionFactory());
        return template;
    }

    @Bean
    public RedisConnectionFactory redisConnectionFactory() {
        return new JedisConnectionFactory();
    }
}

```

Notice the return type of the `redisTemplate` bean method. The `RedisTemplate` is a generic class and requires a key and value type to be specified. In this case `String` is the type of key and `Vehicle` is the type of value, when storing and retrieving objects the `RedisTemplate` will take care of the conversion.

Conversion is done using a `RedisSerializer`, which is an interface for which several implementations exist (see Table 13-1). The default `RedisSerializer`, the `JdkSerializationRedisSerializer`, uses standard Java serialization to convert objects to a `byte[]` and back.

Table 13-1. Default `RedisSerializer` implementations

Name	Description
<code>GenericToStringSerializer</code>	<code>String</code> to <code>byte[]</code> serializer, uses the Spring <code>ConversionService</code> to convert objects to <code>String</code> before converting to a <code>byte[]</code> .
<code>Jackson2JsonRedisSerializer</code>	Read and write JSON using a Jackson 2 <code>ObjectMapper</code> .
<code>JacksonJsonRedisSerializer</code>	Read and write JSON using a Jackson <code>ObjectMapper</code> .
<code>JdkSerializationRedisSerializer</code>	Uses default Java serialization and deserialization and is the default implementation used.
<code>OxmSerializer</code>	Read and write XML using Spring's <code>Marshaller</code> and <code>Unmarshaller</code> .
<code>StringRedisSerializer</code>	Simple <code>String</code> to <code>byte[]</code> converter.

To be able to use the `RedisTemplate` the `Main` class needs to be modified. The configuration needs to be loaded and the `RedisTemplate` retrieved from it.

```

package com.apress.springrecipes.nosql;

import com.apress.springrecipes.nosql.config.RedisConfig;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.data.redis.core.RedisTemplate;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context = new AnnotationConfigApplicationContext(RedisConfig.class);
        RedisTemplate<String, Vehicle> template = context.getBean(RedisTemplate.class);

        final String vehicleNo = "TEM0001";
        Vehicle vehicle = new Vehicle(vehicleNo, "RED", 4,4);
        template.opsForValue().set(vehicleNo, vehicle);
        System.out.println("Vehicle: " + template.opsForValue().get(vehicleNo));
    }
}

```

When the RedisTemplate has been retrieved from the ApplicationContext it can be used. The biggest advantage here is that one can use objects and the template handles the hard work of converting from and to objects. Notice how the set method takes a String and Vehicle as arguments instead of only String or byte[]. This makes code more readable and easier to maintain.

By default JDK serialization is used. To use Jackson a different RedisSerializer needs to be configured.

```

package com.apress.springrecipes.nosql.config;
...
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;

@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<String, Vehicle> redisTemplate() {
        RedisTemplate template = new RedisTemplate();
        template.setConnectionFactory(redisConnectionFactory());
        template.setDefaultSerializer(new Jackson2JsonRedisSerializer(Vehicle.class));
        return template;
    }
}
...
}

```

The RedisTemplate will now use a Jackson ObjectMapper to perform the serialization and deserialization. The remainder of the code can remain the same. When running the main program again it still works and the object will be stored using JSON.

When Redis is used inside a transaction it can also participate in that same transaction. For this set the enableTransactionSupport property on the RedisTemplate to true. This will take care of executing the Redis operation inside the transaction, when the transaction commits.

13-3. Using Hadoop

Problem

You want to use Hadoop in your application for map/reduce operations.

Solution

Download and use Hadoop and add Spring Data Hadoop for easier use and configuration of jobs.

How It Works

Hadoop is a library for distributed computing, it provides a distributed file system (HDFS), a framework for job scheduling and cluster resource management (YARN), and a system for parallel processing of large datasets. Covering all of Hadoop would require another book. This recipe will cover the basics and will show how to work with Hadoop using the Spring Data Hadoop project.

Downloading and Running Hadoop

First Hadoop needs to be installed by downloading a Hadoop version from their website (<http://hadoop.apache.org/#Download+Hadoop>). This recipe will assume to operate in standalone mode and not in one of the cluster modes.

Creating and Running a Map Reduce Job - Standalone

Let's create a job that counts words in a file. For this we need a file and a Mapper and Reducer implementation. The code is in a single class, but you are free to place it in multiple source files if you like.

```
package com.apress.springrecipes.hadoop;

import org.apache.commons.lang3.StringUtils;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import java.io.IOException;
import java.util.regex.Pattern;

public class WordCount {

    private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");

    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable ONE = new IntWritable(1);

        protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            for (String word : WORD_BOUNDARY.split(value.toString())) {
                context.write(new Text(word), ONE);
            }
        }
    }

    public static class Reducer extends Reducer<Text, IntWritable, Text, IntWritable> {
        protected void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
}
```

```

public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    String line = value.toString();
    for (String word : WORD_BOUNDARY.split(line)) {
        if (StringUtils.isBlank(word)) {
            continue;
        }
        context.write(new Text(word.toLowerCase()), ONE);
    }
}

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}

public static void main(String[] args) throws Exception {
    String input = System.getProperty("user.home") + "/hadoop-recipe/input";
    String output = System.getProperty("user.home") + "/hadoop-recipe/output";
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(input));
    FileOutputFormat.setOutputPath(job, new Path(output));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

This is another example of a wordcount hadoop program. The TokenizerMapper uses a regular expression to split a string into words. The words are then transformed to lowercase. The IntSumReducer is simply counting the occurrences of the lowercase words.

The main method of the WordCount program sets up the Configuration and the Job then the system waits for the job to finish. On a successful run it will exit with 0 else with 1.

Before running the recipe, make sure there is a `hadoop-recipe/input` directory in your home folder. For the recipe the plaintext copy of *On the Origin of Species by Means of Natural Selection* by Charles Darwin was used. (More books are available on <http://www.gutenberg.org>). But any plaintext file will do. On the commandline run `hdfs dfs -copyFromLocal <local location of plaintext> input/` to copy a plaintext file to the input directory. Now everything is in place to run the program.

After running the program the output directory is created and should contain some files, including a `part-r-00000` file (depending on your filesize and configuration there might be more files). That file contains the output of running the job. To display the output run `hdfs dfs -cat output/part-r-00000`.

Before running the next wordcount job you have to remove the output directory using `hdfs -rmr output`. If you don't do this the next executions will not run.

Creating and Running a Map Reduce Job – Cluster

Running a job in standalone mode is nice but where hadoop really shines in clustering and distribution. Although you don't have a full cluster let's setup a cluster of 1 node (a so-called pseudo cluster). First make sure you can login with SSH to your localhost, try `ssh localhost` on the commandline. If you can login or get a prompt saying to accept this host you are fine; if not you need to setup keys first.

Type `ssh-keygen -t rsa -P ""` on the command line, followed by `cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys` this will create a key and add it to authorized keys. Now you should be able to login with ssh on localhost.

Next you need to modify the following configuration files `core-site.xml`, `hdfs-site.xml`, `mapred-site.xml`, `yarn-site.xml` and `hadoop-env.sh`. The instructions here are basically the same as in the tutorial on the Apache Hadoop website.¹

First modify the `core-site.xml` and set the `defaultFS` property to `localhost:9000`.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

Next up is the `hdfs-site.xml` which needs to be configured for a single node.

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

¹http://hadoop.apache.org/docs/r2.5.1/hadoop-project-dist/hadoop-common/SingleCluster.html#Pseudo-Distributed_Operation.

Next we need to configure YARN to run in a single node configuration. YARN is used to manage the jobs and resources. Let's modify the `mapred-site.xml`.

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

Finally the `yarn-site.xml` file.

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

Now that all the configuration is in place you can start the filesystem and YARN. On the command line type `start-dfs.sh` and `start-yarn.sh`. You could also run `start-all.sh` but on newer versions of Hadoop this is deprecated and probably will be removed in future versions.

When you have trouble starting Hadoop you might need to modify the `hadoop-env.sh` file. Open it and find the line that sets the `HADOOP_OPTS` variable and append `-Djava.security.krb5.realm= -Djava.security.krb5.kdc=` to it. Afterwards try to start again.

If everything is running there should be the Hadoop web console on `http://localhost:50070` and the Yarn web console on `http://localhost:8088`.

Using Spring for Hadoop to Run a Job

Spring for Hadoop makes it easier to configure and run Hadoop jobs. The biggest advantage is that you can use Spring to configure the Hadoop client and jobs. Because of the usage of Spring you can use all the nice Spring features like resource loading and property placeholders for configuration.

Spring for Hadoop at the moment of this writing only had XML support for configuring jobs so in this part of the recipe you are going to create a Spring XML based application context. Spring Hadoop has an XML namespace for easy configuration. Let's transform the job you created into a Spring configured one.

Create a `wordcount-context.xml` file for the job configuration.

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

  <hadoop:configuration file-system-uri="hdfs://localhost:9000/" />
```

```

<hadoop:job id="wordCountJob"
    mapper="com.apress.springrecipes.hadoop.TokenizerMapper"
    reducer="com.apress.springrecipes.hadoop.IntSumReducer"
    jar-by-class="com.apress.springrecipes.hadoop.WordCountSpring"
    input-path="/hadoop-recipe/input"
    output-path="/hadoop-recipe/output" />

<hadoop:job-runner job-ref="wordCountJob" run-at-startup="true" />

</beans>

```

First there is the Hadoop configuration element. This has been configured to connect to the file system on localhost on port 9000 (which is what you configured the DFS to run on). Next there is the job configuration. The id is required and needed by the job-runner further on. The job takes a mapper and reducer class, it needs the input and output path to know where to read from and store the result. The interesting part is the jar-by-class property, when submitting a job to a Hadoop cluster it needs to know the files. The mapper and reducer classes are in our generated jar and that needs to be sent to Hadoop.

Finally the job-runner element, which takes a reference to a job configuration, and should be started at startup. This takes care of creating the Hadoop job and making sure the configuration is correct and can be run on the Hadoop instance.

The last is the bootstrap class. This needs to load the `wordcount-context.xml` to kickoff the job.

```

package com.apress.springrecipes.hadoop;

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

public class WordCountSpring {

    public static void main(String[] args) throws Exception {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("wordcount-context.
            xml");
        System.out.println("Spring Hadoop WordCount Recipe Running.");
        context.registerShutdownHook();
    }
}

```

The configuration is loaded by the `ClassPathXmlApplicationContext` which immediately kicks off the job. A shutdown hook is registered so that when the job is finished the application context is cleaned up nicely.

Before running the job, remove any data from the output directory by executing `hdfs dfs -rm -r /hadoop-
recipe/output` on the command line. If there isn't any input yet, copy it to Hadoop with `hdfs dfs -copyFromLocal
<local location of plaintext> input/` on the command line.

When the jar file of these classes has been built it can be simply executed with `java -jar Recipe_13_3_i-
3.0.0.jar`. The job will be scheduled and if everything has been configured correctly will run successfully. The output should be like that in Figure 13-3.

```

2014-10-15 17:42:08,765 [main] INFO o.s.b.f.xml.XmlBeanDefinitionReader : Loading XML bean definitions from class path resource [wordcount-context.xml]
2014-10-15 17:42:08,982 [main] INFO o.o.h.conf.Configuration.deprecation : fs.default.name is deprecated. Instead, use fs.defaultFS
2014-10-15 17:42:09,040 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.used.genericoptionsparser is deprecated. Instead, use mapreduce.client.genericoptionsparser.used
2014-10-15 17:42:09,241 [main] WARN org.apache.hadoop.util.NativeCodeLoader : Unable to load native-hadoop library for your platform... using builtin classes where applicable
2014-10-15 17:42:09,250 [main] INFO org.apache.hadoop.mapreduce.Job : mapred.job.tracker: /0.0.0.0:8032
2014-10-15 17:42:09,563 [main] INFO org.apache.hadoop.yarn.client.JobRunner : Starting job WordCountJob
2014-10-15 17:42:09,676 [main] INFO org.apache.hadoop.yarn.client.RMProxy : Connecting to ResourceManager at /0.0.0.0:8032
2014-10-15 17:42:10,039 [main] INFO o.apache.hadoop.mapreduce.JobInputForm : Total input paths to process : 1
2014-10-15 17:42:10,070 [main] INFO o.apache.hadoop.mapreduce.JobSubmitter : number of splits:1
2014-10-15 17:42:10,077 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.job.name is deprecated. Instead, use mapreduce.job.name
2014-10-15 17:42:10,077 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.job.class is deprecated. Instead, use mapreduce.job.map.class
2014-10-15 17:42:10,077 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.input.dir is deprecated. Instead, use mapreduce.input.fileinputformat.inputdir
2014-10-15 17:42:10,077 [main] INFO o.o.h.conf.Configuration.deprecation : mapreduce.reduce.class is deprecated. Instead, use mapreduce.job.reduce.class
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.jar is deprecated. Instead, use mapreduce.job.jar
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.output.dir is deprecated. Instead, use mapreduce.output.fileoutputformat.outputdir
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.working.dir is deprecated. Instead, use mapreduce.job.working.dir
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.mapoutput.value.class is deprecated. Instead, use mapreduce.map.output.value.class
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : mapred.mapoutput.key.class is deprecated. Instead, use mapreduce.map.output.key.class
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : user.name is deprecated. Instead, use mapreduce.job.user.name
2014-10-15 17:42:10,078 [main] INFO o.o.h.conf.Configuration.deprecation : fs.default.name is deprecated. Instead, use fs.defaultFS
2014-10-15 17:42:10,542 [main] INFO o.apache.hadoop.mapreduce.JobSubmitter : Submitting tokens for job: job_1413382584906_0011
2014-10-15 17:42:10,732 [main] INFO o.o.o.client.api.impl.YarnClientImpl : Submitted application application_1413382584906_0011 to ResourceManager at /0.0.0.0:8032
2014-10-15 17:42:10,732 [main] INFO o.o.o.client.api.impl.YarnClientImpl : Job job_1413382584906_0011 registered at http://mac-van-marten.dynamic.ziggo.nl:8088/proxy/application_1413382584906_0011/
2014-10-15 17:42:10,752 [main] INFO org.apache.hadoop.mapreduce.Job : Running job: job_1413382584906_0011
2014-10-15 17:42:14,019 [main] INFO org.apache.hadoop.mapreduce.Job : Job job_1413382584906_0011 running in uber mode : false
2014-10-15 17:42:14,928 [main] INFO org.apache.hadoop.mapreduce.Job : map 0% reduce 0%
2014-10-15 17:42:19,274 [main] INFO org.apache.hadoop.mapreduce.Job : map 100% reduce 100%
2014-10-15 17:42:24,294 [main] INFO org.apache.hadoop.mapreduce.Job : Job job_1413382584906_0011 completed successfully
2014-10-15 17:42:25,384 [main] INFO org.apache.hadoop.mapreduce.Job : Total input paths to process : 1

```

Figure 13-3. Output of successfully running the WordCount job

The information of the job can also be found in the YARN web console. For this navigate to `http://localhost:8088` and check the status. The output should be like Figure 13-4.

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1413382584906_0011	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:42:10 GMT	Wed, 15 Oct 2014 15:42:24 GMT	FINISHED	SUCCEEDED		History
application_1413382584906_0010	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:14:00 GMT	Wed, 15 Oct 2014 15:14:13 GMT	FINISHED	SUCCEEDED		History
application_1413382584906_0009	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:12:34 GMT	Wed, 15 Oct 2014 15:12:53 GMT	FINISHED	FAILED		History
application_1413382584906_0008	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:10:44 GMT	Wed, 15 Oct 2014 15:11:03 GMT	FINISHED	FAILED		History
application_1413382584906_0007	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:09:24 GMT	Wed, 15 Oct 2014 15:09:43 GMT	FINISHED	FAILED		History
application_1413382584906_0006	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 15:07:56 GMT	Wed, 15 Oct 2014 15:08:18 GMT	FINISHED	FAILED		History
application_1413382584906_0005	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 14:59:22 GMT	Wed, 15 Oct 2014 14:59:44 GMT	FINISHED	FAILED		History
application_1413382584906_0002	marten	wordCountJob	MAPREDUCE	default	Wed, 15 Oct 2014 14:19:24 GMT	Wed, 15 Oct 2014 14:19:43 GMT	FINISHED	FAILED		History

Figure 13-4. YARN web console after running jobs

13-4. Using Neo4j

Problem

You want to use Neo4j in your application.

Solution

Use the Spring Data Neo4j library to access Neo4j.

How It Works

Downloading and Running Neo4J

Neo4J can be downloaded from the Neo4j website (<http://neo4j.com/download/>). For this recipe it is enough to download the community edition; however it should also work with the commercial version of Neo4j. Windows users can run the installer to install. Mac and Linux users can extract the archive and, from inside the directory created, start with bin/neo4j. Mac users can also use Homebrew (<http://brew.sh>) to install Neo4j with brew install neo4j, starting can then be done with neo4j start on the command line.

After starting on the command line the output should be similar to that of Figure 13-5.

```
marten@imac-van-marten:~/Repositories/springrecipes/Ch14/Recipe_14_4_i/src/main$ cd -
marten@imac-van-marten:~$ ./neo4j
Usage: neo4j { console | start | start-no-wait | stop | restart | status | info | install | remove }
marten@imac-van-marten:~$ neo4j start
Using additional JVM arguments: -server -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -Dneo4j.ext.udc.source=homebrew -Djava.awt.headless=true
properties -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled -Dneo4j.ext.udc.source=homebrew -Djava.awt.headless=true
Starting Neo4j Server...WARNING: not changing user
Process [48963] is waiting for server to be ready..... OK.
HTTP://localhost:7474/ is ready.
marten@imac-van-marten:~$
```

Figure 13-5. Output after initial start of Neo4j

Starting Neo4j

Let's start by creating a Hello World with Neo4j. Create a main class that starts an embedded server, adds some data to Neo4j and retrieves it again.

```
package com.apress.springrecipes.nosql;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.Transaction;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
```

```

public class Main {

    public static void main(String[] args) {
        final String DB_PATH = System.getProperty("user.home") + "/friends";
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);

        Transaction tx1 = db.beginTx();

        Node hello = db.createNode();
        hello.setProperty("msg", "Hello");

        Node world = db.createNode();
        world.setProperty("msg", "World");
        tx1.success();

        Iterable<Node> nodes = db.getAllNodes();
        for (Node n : nodes) {
            System.out.println("Msg: " + n.getProperty("msg"));
        }

        // Remove all nodes
    }
}

```

This main will start an embedded Neo4j server. Next it will start a transaction and create two nodes. Next all nodes are retrieved and the value of the msg property is printed to the console. Neo4j is good at traversing relations between nodes. It is especially optimized for that (just like other Graph datastores).

Let's create some nodes that have a relationship between them.

```

package com.apress.springrecipes.nosql;

import org.neo4j.cypher.javacompat.ExecutionEngine;
import org.neo4j.cypher.javacompat.ExecutionResult;
import org.neo4j.graphdb.*;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

import java.util.Map;

import static com.apress.springrecipes.nosql.Main.RelationshipTypes.*;

public class Main {

    enum RelationshipTypes implements RelationshipType {FRIENDS_WITH, MASTER_OF, SIBLING, LOCATION};

    public static void main(String[] args) {
        final String DB_PATH = System.getProperty("user.home") + "/starwars";
        GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);

        Transaction tx1 = db.beginTx();

        Label character = DynamicLabel.label("character");
        Label planet = DynamicLabel.label("planet");

```

```

Node dagobah = db.createNode(planet);
dagobah.setProperty("name", "Dagobah");

Node tatooine = db.createNode(planet);
tatooine.setProperty("name", "Tatooine");

Node alderaan = db.createNode(planet);
alderaan.setProperty("name", "Alderaan");

Node yoda = db.createNode(character);
yoda.setProperty("name", "Yoda");

Node luke = db.createNode(character);
luke.setProperty("name", "Luke Skywalker");

Node leia = db.createNode(character);
leia.setProperty("name", "Leia Organa");

Node han = db.createNode(character);
han.setProperty("name", "Han Solo");

// Relations
yoda.createRelationshipTo(luke, MASTER_OF);
yoda.createRelationshipTo(dagobah, LOCATION);
luke.createRelationshipTo(tatooine, LOCATION);
luke.createRelationshipTo(han, FRIENDS_WITH);
luke.createRelationshipTo(leia, FRIENDS_WITH);
leia.createRelationshipTo(han, FRIENDS_WITH);
leia.createRelationshipTo(alderaan, LOCATION);

tx1.success();

ExecutionEngine engine = new ExecutionEngine(db);
ExecutionResult result = engine.execute("MATCH (n) RETURN n.name as name");
String rows = "";
for ( Map<String, Object> row : result ) {
    for ( Map.Entry<String, Object> column : row.entrySet() ) {
        rows += column.getKey() + ": " + column.getValue() + "; ";
    }
    rows += "\n";
}
System.out.println(rows);
}
}

```

The code reflects a tiny part of the StarWars universe. It has characters and their locations which are actually planets. There are also relations between people (see Figure 13-6 for the relationship diagram).

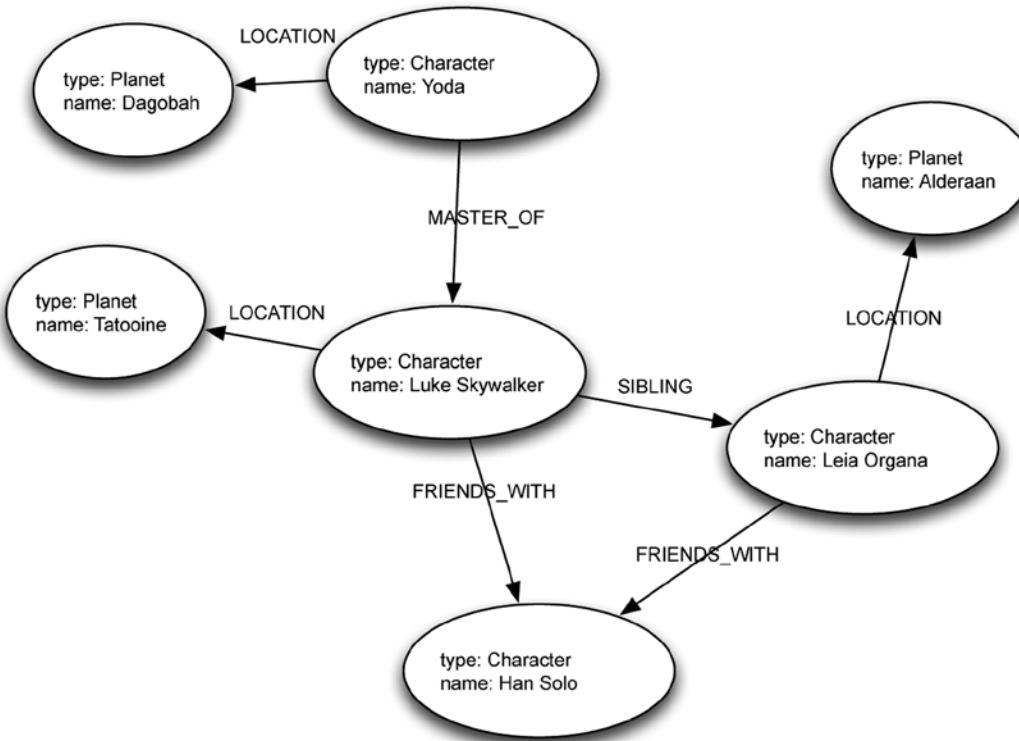


Figure 13-6. Relationships in sample

The relationships in the code are enabled by using an enum that implements a Neo4j interface `RelationshipType`. This is, as the name suggests, needed to differentiate between the different types of relationships. The type of node is differentiated by putting a `Label` on the Node. The name is set as a basic property on the node.

When running the code it will execute the cypher query `MATCH (n) RETURN n.name as name`. This selects all nodes and returns the name property of all the nodes.

Mapping objects with Neo4j

The code until now is quite low level and bound to Neo4j. Creating and manipulating Nodes is cumbersome. Ideally one would use a `Planet` and `Character` class and have that stored/retrieved from Neo4j.

First create the `Planet` and `Character` classes.

```

package com.apress.springrecipes.nosql;

public class Planet {

    private long id = -1;
    private String name;
    // Getters and Setters omitted
}
  
```

```

package com.apress.springrecipes.nosql;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Character {

    private long id = -1;
    private String name;

    private Planet location;
    private final List<Character> friends = new ArrayList<>();
    private Character apprentice;

    public void addFriend(Character friend) {
        friends.add(friend);
    }

    // Getters and Setters omitted
}

```

The `Planet` class is quite straightforward. It has an `id` and `name` property. The `Character` class is a bit more complicated. It also has the `id` and `name` properties along with some additional properties for the relationships. There is the `location` for the `LOCATION` relationship, a collection of `Characters` for the `FRIENDS_WITH` relationship and also an `apprentice` for the `MASTER_OF` relationship.

To be able to store these classes let's create a `StarwarsRepository` interface to hold the save operations.

```

package com.apress.springrecipes.nosql;

public interface StarwarsRepository {

    Planet save(Planet planet);
    Character save(Character character);

}

```

And the implementation for Neo4j.

```

package com.apress.springrecipes.nosql;

import org.neo4j.graphdb.*;
import static com.apress.springrecipes.nosql.RelationshipTypes.*;

public class Neo4jStarwarsRepository implements StarwarsRepository {

    private final GraphDatabaseService db;

    public Neo4jStarwarsRepository(GraphDatabaseService db) {
        this.db = db;
    }
}

```

```

@Override
public Planet save(Planet planet) {
    if (planet.getId() > -1) {
        return planet;
    }
    Transaction tx = db.beginTx();
    Label label = DynamicLabel.label("planet");
    Node node = db.createNode(label);
    node.setProperty("name", planet.getName());
    tx.success();
    planet.setId(node.getId());
    return planet;
}

@Override
public Character save(Character character) {
    if (character.getId() > -1) {
        return character;
    }
    Transaction tx = db.beginTx();
    Label label = DynamicLabel.label("character");
    Node node = db.createNode(label);
    node.setProperty("name", character.getName());

    if (character.getLocation() != null) {
        Planet planet = character.getLocation();
        planet = save(planet);
        node.createRelationshipTo(db.getNodeById(planet.getId()), LOCATION);
    }

    for (Character friend : character.getFriends()) {
        friend = save(friend);
        node.createRelationshipTo(db.getNodeById(friend.getId()), FRIENDS_WITH);
    }

    if (character.getApprentice() != null) {
        save(character.getApprentice());
        node.createRelationshipTo(db.getNodeById(character.getApprentice().getId()), MASTER_OF);
    }

    tx.success();
    character.setId(node.getId());
    return character;
}
}

```

There is a whole lot going on here to convert the objects into a Node object. For the Planet it is pretty easy, first check if it has already been persisted (the id is > -1 in that case); if not start a transaction, create a node, set the name property, and transfer the id to the Planet object. However for the Character class it is a bit more complicated as all the relationships need to be taken into account.

The Main class needs to be modified to reflect the changes to the classes.

```
package com.apress.springrecipes.nosql;

import org.neo4j.cypher.javacompat.ExecutionEngine;
import org.neo4j.cypher.javacompat.ExecutionResult;
import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;

import java.util.Map;

public class Main {

    public static void main(String[] args) {
        final String DB_PATH = System.getProperty("user.home") + "/friends";
        final GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);

        StarwarsRepository repository = new Neo4jStarwarsRepository(db);

        // Planets
        Planet dagobah = new Planet();
        dagobah.setName("Dagobah");

        Planet alderaan = new Planet();
        alderaan.setName("Alderaan");

        Planet tatooine = new Planet();
        tatooine.setName("Tatooine");

        dagobah = repository.save(dagobah);
        repository.save(alderaan);
        repository.save(tatooine);

        // Characters
        Character han = new Character();
        han.setName("Han Solo");

        Character leia = new Character();
        leia.setName("Leia Organa");
        leia.setLocation(alderaan);
        leia.addFriend(han);

        Character luke = new Character();
        luke.setName("Luke Skywalker");
        luke.setLocation(tatooine);
        luke.addFriend(han);
        luke.addFriend(leia);

        Character yoda = new Character();
        yoda.setName("Yoda");
        yoda.setLocation(dagobah);
        yoda.setApprentice(luke);
```

```
repository.save(han);
repository.save(luke);
repository.save(leia);
repository.save(yoda);

ExecutionEngine engine = new ExecutionEngine(db);
ExecutionResult result = engine.execute("MATCH (n) RETURN n.name as name");
String rows = "";
for ( Map<String, Object> row : result ) {
    for ( Map.Entry<String, Object> column : row.entrySet() ) {
        rows += column.getKey() + ": " + column.getValue() + "; ";
    }
    rows += "\n";
}
System.out.println(rows);
}
```

When executing the result should still be the same as before. However the main difference is now that the code is using domain objects instead of working directly with nodes. Storing the objects as Nodes in Neo4j is quite cumbersome. Luckily Spring Boot Neo4j can help to make it a lot easier.

Mapping Objects Using Spring Data Neo4j

In the conversion to nodes and relationships, properties can be quite cumbersome. Wouldn't it be nice if one could simply specify what to store where using annotations just as is done using JPA? Spring Data Neo4j offers those annotations. To make an object a Neo4j mapped entity use the `@NodeEntity` annotation on the type. Relationships can be modeled with the `@RelatedTo` annotation. To identify the field used for the id add the `@GraphId` annotation. Applying these to the `Planet` and `Character` class would make them look like the following.

```
package com.apress.springrecipes.nosql;

import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;

@NodeEntity
public class Planet {

    @GraphId
    private Long id;
    private String name;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
}
```

```

public String getName() {
    return name;
}

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

...
}

```

And the Character class.

```

package com.apress.springrecipes.nosql;

import org.springframework.data.neo4j.annotation.GraphId;
import org.springframework.data.neo4j.annotation.NodeEntity;
import org.springframework.data.neo4j.annotation.RelatedTo;

import java.util.*;

@NoArgsConstructor
public class Character {

    @GraphId
    private Long id;
    private String name;

    @RelatedTo(type="LOCATION")
    private Planet location;
    @RelatedTo(type="FRIENDS_WITH")
    private final Set<Character> friends = new HashSet<>();
    @RelatedTo(type="MASTER_OF")
    private Character apprentice;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

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

```

```

public Planet getLocation() {
    return location;
}

public void setLocation(Planet location) {
    this.location = location;
}

public Character getApprentice() {
    return apprentice;
}

public void setApprentice(Character apprentice) {
    this.apprentice = apprentice;
}

public Set<Character> getFriends() {
    return Collections.unmodifiableSet(friends);
}

public void addFriend(Character friend) {
    friends.add(friend);
}
...
}

```

Now that the entities are annotated the repository can be rewritten to use a `Neo4jTemplate` for easier access.

```

package com.apress.springrecipes.nosql;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.neo4j.conversion.Result;
import org.springframework.data.neo4j.support.Neo4jTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.PreDestroy;
import java.util.Map;

@Repository
@Transactional
public class Neo4jStarwarsRepository implements StarwarsRepository {

    private final Neo4jTemplate template;

    @Autowired
    public Neo4jStarwarsRepository(Neo4jTemplate template) {
        this.template = template;
    }
}

```

```

@Override
public Planet save(Planet planet) {
    template.save(planet);
    return planet;
}

@Override
public Character save(Character character) {
    template.save(character);
    return character;
}

@Override
public void printAll() {
    Result<Map<String, Object>> result;
    result=template.queryEngineFor().query("MATCH (n) RETURN n.name as name", null);
    String rows = "";
    for ( Map<String, Object> row : result ) {
        for ( Map.Entry<String, Object> column : row.entrySet() ) {
            rows += column.getKey() + ": " + column.getValue() + "; ";
        }
        rows += "\n";
    }
    System.out.println(rows);
}

@Override
public void cleanUp() {
    // Clean up when shutdown
    template.query("MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r", null);
}
}

```

There are a couple of things to notice: the code is a lot cleaner when using the Neo4jTemplate as a lot of the plumbing is done for you, especially mapping from and to Nodes. The repository is now also annotated with `@Transactional`. To operate on Neo4j we need a transaction and it is the easiest to let Spring manage those. The final thing to note is the addition of the `printAll` method. A transaction is required to query related objects (there is some lazy loading going on) and the repository is already transactional.

Finally the `cleanUp` method has been added to make sure that the repository cleans up the datastore when we shutdown. This is probably not something you want to use in production but for our small recipe it makes sense.

The next class to modify is the configuration class.

```

package com.apress.springrecipes.nosql.config;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.factory.GraphDatabaseFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.Neo4jConfiguration;
import org.springframework.transaction.annotation.EnableTransactionManagement;

```

```

import javax.annotation.PostConstruct;
@Configuration
@EnableTransactionManagement
@ComponentScan(basePackages = {"com.apress.springrecipes.nosql"})
public class StarwarsConfig extends Neo4jConfiguration {

    @PostConstruct
    public void init() {
        setBasePackage("com.apress.springrecipes.nosql");
    }

    @Bean(destroyMethod = "shutdown")
    public GraphDatabaseService graphDatabaseService() {
        final String DB_PATH = System.getProperty("user.home") + "/starwars";
        return new GraphDatabaseFactory().newEmbeddedDatabase(DB_PATH);
    }
}

```

The configuration class now extends the `Neo4jConfiguration` class. This class sets up the `Neo4jTemplate` and a lot of other infrastructure classes like a `transactionmanager`. The `Neo4jConfiguration` also need to know in which packages the `@NodeEntity` annotated classes are kept. This can be done in a `@PostConstruct` annotated method.

Finally the main class needs to be modified to load this configuration class.

```

package com.apress.springrecipes.nosql;

import com.apress.springrecipes.nosql.config.StarwarsConfig;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.transaction.PlatformTransactionManager;

public class Main {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(StarwarsConfig.class);
        StarwarsRepository repository = context.getBean(StarwarsRepository.class);

        Planet dagobah = new Planet();
        dagobah.setName("Dagobah");

        Planet alderaan = new Planet();
        alderaan.setName("Alderaan");

        Planet tatooine = new Planet();
        tatooine.setName("Tatooine");

        dagobah = repository.save(dagobah);
        repository.save(alderaan);
        repository.save(tatooine);

        Character han = new Character();
        han.setName("Han Solo");
    }
}

```

```

Character leia = new Character();
leia.setName("Leia Organa");
leia.setLocation(alderaan);
leia.addFriend(han);

Character luke = new Character();
luke.setName("Luke Skywalker");
luke.setLocation(tatooine);
luke.addFriend(han);
luke.addFriend(leia);

Character yoda = new Character();
yoda.setName("Yoda");
yoda.setLocation(dagobah);
yoda.setApprentice(luke);

repository.save(han);
repository.save(luke);
repository.save(leia);
repository.save(yoda);

repository.printAll();

context.close();
}

}

```

An AnnotationConfigApplicationContext is used to parse the StarwarsConfig class. This will bootstrap the application and construct all beans necessary. Next the data is being setup and the printAll method is called. The code that was there initially is now in that method. The end result should still be similar to what you got until now.

The call to context.close() will make sure the @PreDestroy method is being invoked.

Using Spring Data Neo4j Repositories

The code has been simplified considerably. The usage of the Neo4jTemplate made it a lot easier to work with entities together with Neo4j. It can be even easier. As with the JPA or Mongo version of Spring Data it can generate repositories for you. The only thing you need to do is write an interface. Let's create a PlanetRepository and CharacterRepository to operate on the entities.

```

package com.apress.springrecipes.nosql;

import org.springframework.data.neo4j.repository.GraphRepository;

public interface CharacterRepository extends GraphRepository<Character> {}

```

And the PlanetRepository.

```

package com.apress.springrecipes.nosql;

import org.springframework.data.neo4j.repository.GraphRepository;

public interface PlanetRepository extends GraphRepository<Planet> {}

```

The thing to notice here is that the repositories are interfaces that extend `GraphRepository`. This is a special interface which, eventually, also extends the Spring Data `CrudRepository`. This allows for automatic creation of repositories. The `GraphRepository` also extends `IndexRepository`, `SchemaIndexRepository`, and `TraversalRepository` which are especially useful to search on indexed properties or by traversing relationships. Instead of extending `GraphRepository` you can extend the less extensive `CRUDRepository`.

The extended interface takes a type parameter. The type is the type of entity that is supported by this repository.

Next rename the `StarwarsRepository` and its implementation, to `StarwarsService` as it isn't really a repository anymore, the implementation also needs to change to operate on the repositories instead of the `Neo4jTemplate`.

```
package com.apress.springrecipes.nosql;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import javax.annotation.PreDestroy;

@Service
@Transactional
public class Neo4jStarwarsService implements StarwarsService {

    private final PlanetRepository planetRepository;
    private final CharacterRepository characterRepository;

    @Autowired
    public Neo4jStarwarsService(PlanetRepository planetRepository,
                               CharacterRepository characterRepository) {
        this.planetRepository=planetRepository;
        this.characterRepository=characterRepository;
    }

    @Override
    public Planet save(Planet planet) {
        return planetRepository.save(planet);
    }

    @Override
    public Character save(Character character) {
        return characterRepository.save(character);
    }

    @Override
    public void printAll() {
        Result<Planet> planets = planetRepository.findAll();
        Result<Character> characters = characterRepository.findAll();
```

```

        for (Planet p : planets) {
            System.out.println(p);
        }

        for (Character c : characters) {
            System.out.println(c);
        }
    }

    @PreDestroy
    public void cleanUp() {
        // Clean up when shutdown
        characterRepository.deleteAll();
        planetRepository.deleteAll();
    }
}

```

Now all operations are done on the specific repository interfaces. Those interfaces don't create instances themselves. To enable the creation the `@EnableNeo4jRepositories` annotations needs to be added to the configuration class.

```

@Configuration
@EnableTransactionManagement
@EnableNeo4jRepositories(basePackages = {"com.apress.springrecipes.nosql"})
@ComponentScan(basePackages = {"com.apress.springrecipes.nosql"})
public class StarwarsConfig extends Neo4jConfiguration { ... }

```

Notice the `@EnableNeo4jRepositories` annotation. This annotation will scan the configured base-packages for repositories. When one is found a dynamic implementation is created, this implementation eventually delegates to the `Neo4jTemplate`.

Finally modify the Main class to use the refactored `StarwarsService`.

```

package com.apress.springrecipes.nosql;

import com.apress.springrecipes.nosql.config.StarwarsConfig;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context =
            new AnnotationConfigApplicationContext(StarwarsConfig.class);

        StarwarsService service = context.getBean(StarwarsService.class);
        ...
    }
}

```

Now all the components have been changed to use the dynamically created Spring Data Neo4j repositories.

Connecting to a Remote Neo4j Database

Until now all the coding for Neo4j has been done to an embedded Neo4j instance, however at the beginning you downloaded and installed Neo4j. Neo4j has a REST interface for remote connections. To easily connect to a remote repository Spring Data has the `spring-data-neo4j-rest` module.

After adding this dependency it is just a matter of reconfiguring the embedded repository for a remote one. The remainder of the code can be left unchanged.

```
package com.apress.springrecipes.nosql.config;

import org.neo4j.graphdb.GraphDatabaseService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.neo4j.config.EnableNeo4jRepositories;
import org.springframework.data.neo4j.config.Neo4jConfiguration;
import org.springframework.data.neo4j.rest.SpringRestGraphDatabase;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.annotation.PostConstruct;

@Configuration
@EnableTransactionManagement
@EnableNeo4jRepositories(basePackages = {"com.apress.springrecipes.nosql"})
@ComponentScan(basePackages = {"com.apress.springrecipes.nosql"})
public class StarwarsConfig extends Neo4jConfiguration {

    @PostConstruct
    public void init() {
        setBasePackage("com.apress.springrecipes.nosql");
    }

    @Bean(destroyMethod = "shutdown")
    public GraphDatabaseService graphDatabaseService() {
        return new SpringRestGraphDatabase("http://localhost:7474/db/data");
    }
}
```

The `SpringRestGraphDatabase` will connect to the Neo4j instance which by default is on port 7474 on localhost. Underneath it will construct an instance of the `RestAPIFacade` class. If more elaborate configuration is needed, like SSL or a username and password, an explicit configured `RestAPIFacade` could also be injected instead.

Summary

In this recipe you took an introductory journey into different types of datastores, including how to use them and how to make using them easier with different modules of the Spring Data family.

You started out by taking a look at document-driven stores in the form of MongoDB and the usage of the Spring Data MongoDB module. Next the journey took you to key-value stores for which Redis was used as an implementation and the usage of the Spring Data Redis module.

Then for Big Data you looked at how to do a basic processing job with plain Hadoop and how to do it with Spring Data Hadoop. As an intermezzo you also learned how to setup a single node cluster.

The final datastore was a graph based one. In this case Neo4j, for which you explored the embedded neo4j, how to use it, and build a repository for storing entities.

CHAPTER 14

Spring Java Enterprise Services and Remoting Technologies

In this chapter, you will learn about Spring's support for the most common Java enterprise services: Java Management Extensions (JMX), sending e-mail with JavaMail, and scheduling tasks with Quartz. In addition, you'll learn about Spring's support for various remoting technologies, such as RMI, Hessian, Burlap, HTTP Invoker, and SOAP web services.

JMX is part of JavaSE and is a technology for managing and monitoring system resources such as devices, applications, objects, and service-driven networks. These resources are represented as managed beans (MBeans). Spring supports JMX by exporting any Spring bean as model MBeans without programming against the JMX API. In addition, Spring can easily access remote MBeans.

JavaMail is the standard API and implementation for sending e-mail in Java. Spring further provides an abstract layer to send e-mail in an implementation-independent fashion.

There are two main options for scheduling tasks on the Java platform: JDK Timer and Quartz Scheduler (<http://quartz-scheduler.org/>). JDK Timer offers simple task scheduling features that are bundled with JDK. Compared with JDK Timer, Quartz offers more powerful job scheduling features. For both options, Spring supplies utility classes to configure scheduling tasks in a bean configuration file, without using either API directly.

Remoting is a key technology to develop distributed applications, especially multitier enterprise applications. It allows different applications or components, running in different JVMs or on different machines, to communicate with each other using a specific protocol.

Spring's remoting support is consistent across different remoting technologies. On the server side, Spring allows you to expose an arbitrary bean as a remote service through a service exporter. On the client side, Spring provides various proxy factory beans for you to create a local proxy for a remote service so that you can use the remote service as if they were local beans.

You'll learn how to use a series of remoting technologies that include: RMI, Hessian, Burlap, HTTP Invoker, and SOAP web services using Spring Web Services (Spring-WS).

14-1. Register Spring POJOs as JMX MBeans

Problem

You want to register an object in your Java application as a JMX MBean, to have the ability to look at services that are running and manipulate their state at runtime. This will allow you to run tasks like: rerun batch jobs, invoke methods, and change configuration metadata.

Solution

Spring supports JMX by allowing you to export any beans in its IoC container as model MBeans. This can be done simply by declaring an MBeanExporter instance. With Spring's JMX support, you don't need to deal with the JMX API directly. In addition, Spring enables you to declare JSR-160 (Java Management Extensions Remote API) connectors to expose MBeans for remote access over a specific protocol by using a factory bean. Spring provides factory beans for both servers and clients.

Spring's JMX support comes with other mechanisms by which you can assemble an MBean's management interface. These options include using exporting beans by method names, interfaces, and annotations. Spring can also detect and export MBeans automatically from beans declared in the IoC container and annotated with JMX-specific annotations defined by Spring.

How It Works

Suppose you're developing a utility for replicating files from one directory to another. Let's design the interface for this utility:

```
package com.apress.springrecipes.replicator;
...
public interface FileReplicator {
    public String getSrcDir();
    public void setSrcDir(String srcDir);

    public String getDestDir();
    public void setDestDir(String destDir);

    public FileCopier getFileCopier();
    public void setFileCopier(FileCopier fileCopier);

    public void replicate() throws IOException;
}
```

The source and destination directories are designed as properties of a replicator object, not method arguments. That means each file replicator instance replicates files only for a particular source and destination directory. You can create multiple replicator instances in your application.

But before you implement this replicator, let's create another interface that copies a file from one directory to another, given its name.

```
package com.apress.springrecipes.replicator;
...
public interface FileCopier {
    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException;
}
```

There are many strategies for implementing this file copier. For instance, you can make use of the `FileCopyUtils` class provided by Spring.

```

package com.apress.springrecipes.replicator;
...
import org.springframework.util.FileCopyUtils;

public class FileCopierJMXImpl implements FileCopier {

    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException {
        File srcFile = new File(srcDir, filename);
        File destFile = new File(destDir, filename);
        FileCopyUtils.copy(srcFile, destFile);
    }
}

```

With the help of a file copier, you can implement the file replicator, as shown in the following code sample.

```

package com.apress.springrecipes.replicator;

import java.io.File;
import java.io.IOException;

public class FileReplicatorJMXImpl implements FileReplicator {

    private String srcDir;
    private String destDir;
    private FileCopier fileCopier;

    // getters ommited for brevity

    // setters
    public void setSrcDir(String srcDir) {
        this.srcDir = srcDir;
    }

    public void setDestDir(String destDir) {
        this.destDir = destDir;
    }

    public void setFileCopier(FileCopier fileCopier) {
        this.fileCopier = fileCopier;
    }

    public synchronized void replicate() throws IOException {
        File[] files = new File(srcDir).listFiles();
        for (File file : files) {
            if (file.isFile()) {
                fileCopier.copyFile(srcDir, destDir, file.getName());
            }
        }
    }
}

```

Each time you call the `replicate()` method, all files in the source directory are replicated to the destination directory. To avoid unexpected problems caused by concurrent replication, you declare this method as synchronized.

Now, you can configure one or more file replicator instances in a Java Config class. The `DocumentReplicator` instance needs references to two directories: a source directory from which files are read and a target directory to which files are backed up. The code in this example attempts to read from a directory called `docs` in your operating system user's home directory and then copy to a folder called `docs_backup` in your operating system user's home directory. When this bean starts up, it creates the two directories if they don't already exist there.

Tip The “home directory” is different for each operating system, but typically on Unix it's the directory that `~` resolves to. On a Linux box, the folder might be `/home/user`. On Mac OS X, the folder might be `/Users/user`, and on Windows it might be similar to `C:\Documents and Settings\user`.

```
package com.apress.springrecipes.replicator.config;
...
@Configuration
public class FileReplicatorConfig {
    @Value("#{systemProperties['user.home']}/docs")
    private String srcDir;
    @Value("#{systemProperties['user.home']}/docs_backup")
    private String destDir;

    @Bean
    public FileCopier fileCopier() {
        FileCopier fCop = new FileCopierJMXImpl();
        return fCop;
    }

    @Bean
    public FileReplicator documentReplicator() {
        FileReplicator fRep = new FileReplicatorJMXImpl();
        fRep.setSrcDir(srcDir);
        fRep.setDestDir(destDir);
        fRep.setFileCopier(fileCopier());
        return fRep;
    }

    @PostConstruct
    public void verifyDirectoriesExist() {
        File src = new File(srcDir);
        File dest = new File(destDir);
        if (!src.exists())
            src.mkdirs();
        if (!dest.exists())
            dest.mkdirs();
    }
}
```

Initially two fields are declared using the @Value annotations to gain access to the user's home directory and define the source and destination directories. Next, two bean instances are created using the @Bean annotation. Notice the @PostConstruct annotation on the verifyDirectoriesExist(), which ensures the source and destination directories exist.

Now that we have the application's core beans, let's take a look at how to register and access the beans as an MBean.

Register MBeans without Spring's Support

First, let's see how to register a model MBean using the JMX API directly. In the following Main class, you get the FileReplicator bean from the IoC container and register it as an MBean for management and monitoring. All properties and methods are included in the MBean's management interface.

```
package com.apress.springrecipes.replicator;
...
import java.lang.management.ManagementFactory;

import javax.management.Descriptor;
import javax.management.JMException;
import javax.management.MBeanServer;
import javax.management.ObjectName;
import javax.management.modelmbean.DescriptorSupport;
import javax.management.modelmbean.InvalidTargetObjectTypeException;
import javax.management.modelmbean.ModelMBeanAttributeInfo;
import javax.management.modelmbean.ModelMBeanInfo;
import javax.management.modelmbean.ModelMBeanInfoSupport;
import javax.management.modelmbean.ModelMBeanOperationInfo;
import javax.management.modelmbean.RequiredModelMBean;

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

public class Main {

    public static void main(String[] args) throws IOException {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.
replicator.config");

        FileReplicator documentReplicator = context.getBean(FileReplicator.class);
        try {
            MBeanServer mbeanServer = ManagementFactory.getPlatformMBeanServer();
            ObjectName objectName = new ObjectName("bean:name=documentReplicator");

            RequiredModelMBean mbean = new RequiredModelMBean();
            mbean.setManagedResource(documentReplicator, "objectReference");

            Descriptor srcDirDescriptor = new DescriptorSupport(new String[] {
                "name=SrcDir", "descriptorType=attribute",
                "getMethod=getSrcDir", "setMethod=setSrcDir" });
            ModelMBeanAttributeInfo srcDirInfo = new ModelMBeanAttributeInfo(
                "SrcDir", "java.lang.String", "Source directory",
                true, true, false, srcDirDescriptor);
        }
    }
}
```

```

Descriptor destDirDescriptor = new DescriptorSupport(new String[] {
    "name=DestDir", "descriptorType=attribute",
    "getMethod=getDestDir", "setMethod=setDestDir" });
ModelMBeanAttributeInfo destDirInfo = new ModelMBeanAttributeInfo(
    "DestDir", "java.lang.String", "Destination directory",
    true, true, false, destDirDescriptor);

ModelMBeanOperationInfo getSrcDirInfo = new ModelMBeanOperationInfo(
    "Get source directory",
    FileReplicator.class.getMethod("getSrcDir"));
ModelMBeanOperationInfo setSrcDirInfo = new ModelMBeanOperationInfo(
    "Set source directory",
    FileReplicator.class.getMethod("setSrcDir", String.class));
ModelMBeanOperationInfo getDestDirInfo = new ModelMBeanOperationInfo(
    "Get destination directory",
    FileReplicator.class.getMethod("getDestDir"));
ModelMBeanOperationInfo setDestDirInfo = new ModelMBeanOperationInfo(
    "Set destination directory",
    FileReplicator.class.getMethod("setDestDir", String.class));
ModelMBeanOperationInfo replicateInfo = new ModelMBeanOperationInfo(
    "Replicate files",
    FileReplicator.class.getMethod("replicate"));

ModelMBeanInfo mbeanInfo = new ModelMBeanInfoSupport(
    "FileReplicator", "File replicator",
    new ModelMBeanAttributeInfo[] { srcDirInfo, destDirInfo },
    null,
    new ModelMBeanOperationInfo[] { getSrcDirInfo, setSrcDirInfo,
        getDestDirInfo, setDestDirInfo, replicateInfo },
    null);
mbean.setModelMBeanInfo(mbeanInfo);

mbeanServer.registerMBean(mbean, objectName);
} catch (JMException e) {
    ...
} catch (InvalidTargetObjectTypeException e) {
    ...
} catch (NoSuchMethodException e) {
    ...
}

System.in.read();
}
}

```

To register an MBean, you need an instance of the interface `javax.management.MBeanServer`. You can call the static method `ManagementFactory.getPlatformMBeanServer()` to locate a platform MBean server. It will create an MBean server if none exists and then register this server instance for future use. Each MBean requires an MBean object name that includes a domain. The preceding MBean is registered under the domain `bean` with the name `documentReplicator`.

From the preceding code, you can see that for each MBean attribute and MBean operation, you need to create a `ModelMBeanAttributeInfo` object and a `ModelMBeanOperationInfo` object for describing it. After those, you have to create a `ModelMBeanInfo` object for describing the MBean's management interface by assembling the preceding information. For details about using these classes, you can consult their Javadocs.

Moreover, you have to handle the JMX-specific exceptions when calling the JMX API. These exceptions are checked exceptions that you must handle.

Note that you must prevent your application from terminating before you look inside it with a JMX client tool. Requesting a key from the console using `System.in.read()` is a good choice.

Finally, you have to add the VM argument `-Dcom.sun.management.jmxremote` to enable local monitoring of this application. If you're using the book's source code, you can use the following:

```
java -Dcom.sun.management.jmxremote -jar Recipe_14_1_i-1.0-SNAPSHOT.jar
```

Now, you can use any JMX client tools to monitor your MBeans locally. The simplest one is JConsole, which comes with JDK. To start JConsole, just execute the `jconsole` executable file located in the bin directory of the JDK installation.

When JConsole starts, you can see a list of JMX-enabled applications on the Local tab of the connection window. Select the process that corresponds to the running Spring app (i.e., `Recipe_14_1_i-1.0-SNAPSHOT.jar`). This is illustrated in Figure 14-1.

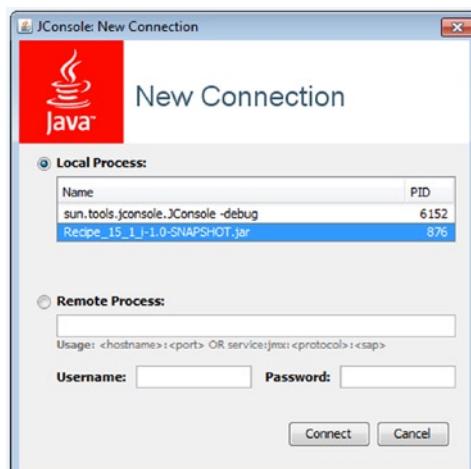


Figure 14-1. JConsole startup window

Caution If you're on Windows, you may not see any processes in JConsole. This is a known bug where JConsole isn't able to detect running Java processes. To solve this issue you'll need to ensure the user has a `hsperfdata` folder in his temp folder. This folder is used by Java and JConsole to keep track of running processes and it may not exist. For example if you're running application as user John.Doe, ensure the following path exists: `C:\Users\John.Doe\AppData\Local\Temp\hsperfdata_John.Doe\`

After connecting to the replicator application, go to the MBeans tab. Next, click on the Bean folder in the left-hand tree, followed by the operations section. In the main screen you'll see a series of buttons to invoke the bean's operations, to invoke replicate() simply click the button "replicate". This screen is illustrated in Figure 14-2.

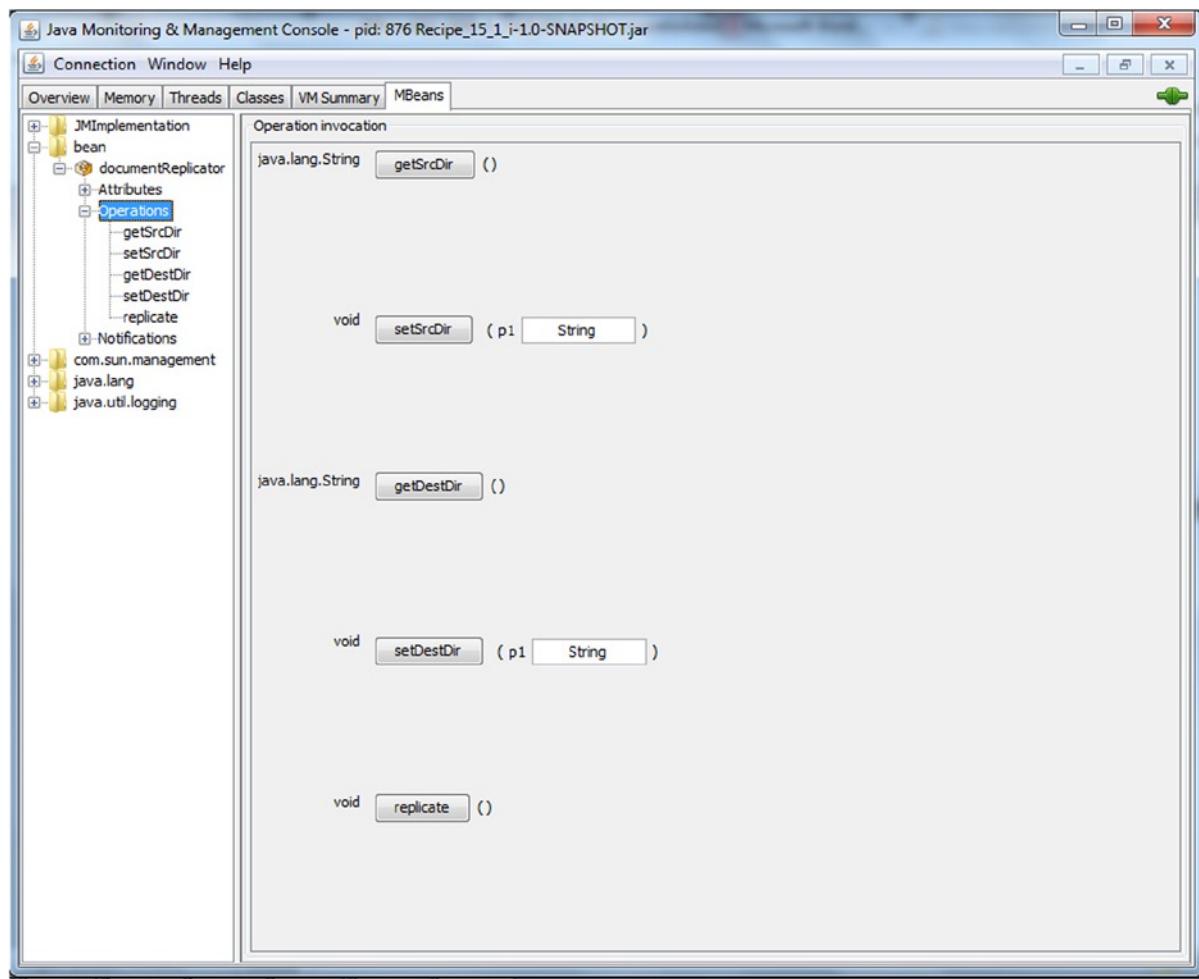


Figure 14-2. JConsole simulate Spring bean operation

You'll see a 'Method successfully invoked' pop-up window. With this action all the filters in the source folder are copied/synchronized with those in the destination folder.

Register MBean with Spring support

The prior application relied on the use of the JMX API directly. As you saw in the Main application class, there's a lot of code that can be difficult to write, manage, and sometimes understand. To export beans configured in the Spring IoC container as MBeans, you simply create an `MBeanExporter` instance and specify the beans to export, with their MBean object names as the keys. This can be done by adding the following configuration class. Note that the key entry in the `beansToExport` Map is used as the `ObjectName` for the bean referenced by the corresponding entry value.

```

package com.apress.springrecipes.replicator.config;

import com.apress.springrecipes.replicator.FileReplicator;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jmx.export.MBeanExporter;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class JmxConfig {

    @Autowired
    private FileReplicator fileReplicator;

    @Bean
    public MBeanExporter mbeanExporter() {
        MBeanExporter mbeanExporter = new MBeanExporter();
        mbeanExporter.setBeans(beansToExport());
        return mbeanExporter;
    }

    private Map<String, Object> beansToExport() {
        Map<String, Object> beansToExport = new HashMap<>();
        beansToExport.put("bean:name=documentReplicator", fileReplicator);
        return beansToExport;
    }
}

```

The preceding configuration exports the `FileReplicator` bean as an MBean, under the domain `bean` and with the name `documentReplicator`. By default, all public properties are included as attributes and all public methods (with the exception of those from `java.lang.Object`) are included as operations in the MBean's management interface.

And with the help of this Spring JMX exported the main class in the application can be cut down to the following lines:

```

package com.apress.springrecipes.replicator;
...
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {

    public static void main(String[] args) throws IOException {
        new AnnotationConfigApplicationContext("com.apress.springrecipes.replicator.config");
        System.in.read();
    }
}

```

Work with multiple MBean server instances

The Spring MBeanExporter approach can locate an MBean server instance and register MBeans with it implicitly. The JDK creates an MBean server the first time when you locate it, so there's no need to create an MBean server explicitly. The same case applies if an application is running in an environment that provides an MBean server (e.g., a Java application server).

However, if you have multiple MBean servers running, you need to tell the mbeanServer bean to which server it should bind. You do this by specifying the agentId of the server. To figure out the agentId of a given server in JConsole, for example, go to the MBeans tab and in the left-hand tree go to JMImplementation /MBeanServerDelegate/Attributes/MBeanServerId. There, you'll see the string value. On our local machine, the value is workstation_1253860476443. To enable it, configure the agentId property of the MBeanServer.

```
@Bean
public MBeanServerFactoryBean mbeanServer() {
    MBeanServerFactoryBean mbeanServer = new MBeanServerFactoryBean();
    mbeanServer.setLocateExistingServerIfPossible(true);
    mbeanServer.setAgentId("workstation_1253860476443");
    return mbeanServer;
}
```

If you have multiple MBean server instances in your context, you can explicitly specify a specific MBean server for MBeanExporter to export your MBeans to. In this case, MBeanExporter will not locate an MBean server; it will use the specified MBean server instance. This property is for you to specify a particular MBean server when more than one is available.

```
@Bean
public MBeanExporter mbeanExporter() {
    MBeanExporter mbeanExporter = new MBeanExporter();
    mbeanExporter.setBeans(beansToExport());
    mbeanExporter.setServer(mbeanServer().getobject());
    return mbeanExporter;
}

@Bean
public MBeanServerFactoryBean mbeanServer() {
    MBeanServerFactoryBean mbeanServer = new MBeanServerFactoryBean();
    mbeanServer.setLocateExistingServerIfPossible(true);
    return mbeanServer;
}
```

Register MBeans for Remote Access with RMI

If you want your MBeans to be accessed remotely, you need to enable a remoting protocol for JMX. JSR-160 defines a standard for JMX remoting through a JMX connector. Spring allows you to create a JMX connector server through ConnectorServerFactoryBean.

By default, ConnectorServerFactoryBean creates and starts a JMX connector server bound to the service URL `service:jmx:jmxmp://localhost:9875`, which exposes the JMX connector through the JMX Messaging Protocol (JMXMP). However, most JMX implementations don't support JMXMP. Therefore, you should choose a widely supported remoting protocol for your JMX connector, such as RMI. To expose your JMX connector through a specific protocol, you just provide the service URL for it.

```

@Bean
public FactoryBean<Registry> rmiRegistry() {
    return new RmiRegistryFactoryBean();
}

@Bean
@DependsOn("rmiRegistry")
public FactoryBean<JMXConnectorServer> connectorServer() {
    ConnectorServerFactoryBean connectorServerFactoryBean = new ConnectorServerFactoryBean();
    connectorServerFactoryBean
        .setServiceUrl("service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator");
    return connectorServerFactoryBean;
}

```

You specify the preceding URL to bind your JMX connector to an RMI registry listening on port 1099 of localhost. If no RMI registry has been created externally, you should create one by using `RmiRegistryFactoryBean`. The default port for this registry is 1099, but you can specify another one in its port property. Note that `ConnectorServerFactoryBean` must create the connector server after the RMI registry is created and ready. You can set the depends-on attribute for this purpose.

Now, your MBeans can be accessed remotely via RMI. Note there's no need to start an RMI enabled app with the `JMX-Dcom.sun.management.jmxremote` flag, as you did in previous apps.

When JConsole starts, you can enter `service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator` in the service URL on the 'Remote processes' section of the connection window. This is illustrated in Figure 14-3.

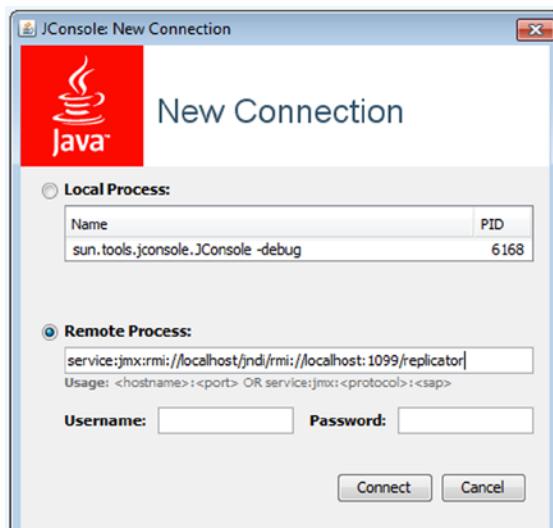


Figure 14-3. JConsole connection to MBean through RMI

Once the connection is established, you can invoke bean methods just as you did with previous examples.

Assemble the Management Interface of MBeans

By default, the Spring MBeanExporter exports all public properties of a bean as MBean attributes and all public methods as MBean operations. But you can assemble the management interface of MBeans using an MBean assembler. The simplest MBean assembler in Spring is `MethodNameBasedMBeanInfoAssembler`, which allows you to specify the names of the methods to export.

```
@Configuration
public class JmxConfig {
    ...
    @Bean
    public MBeanExporter mbeanExporter() {
        MBeanExporter mbeanExporter = new MBeanExporter();
        mbeanExporter.setBeans(beansToExport());
        mbeanExporter.setAssembler(assembler());
        return mbeanExporter;
    }

    @Bean
    public MBeanInfoAssembler assembler() {
        MethodNameBasedMBeanInfoAssembler assembler;
        assembler = new MethodNameBasedMBeanInfoAssembler();
        assembler.setManagedMethods(new String[] {"getSrcDir", "setSrcDir", "getDestDir",
            "setDestDir", "replicate"});
        return assembler;
    }
}
```

Another MBean assembler is `InterfaceBasedMBeanInfoAssembler`, which exports all methods defined in the interfaces you specified.

```
@Bean
public MBeanInfoAssembler assembler() {
    InterfaceBasedMBeanInfoAssembler assembler = new InterfaceBasedMBeanInfoAssembler();
    assembler.setManagedInterfaces(new Class[] {FileReplicator.class});
    return assembler;
}
```

Spring also provides `MetadataMBeanInfoAssembler` to assemble an MBean's management interface based on the metadata in the bean class. It supports two types of metadata: JDK annotations and Apache Commons Attributes (behind the scenes, this is accomplished using a strategy interface `JmxAttributeSource`). For a bean class annotated with JDK annotations, you specify an `AnnotationJmxAttributeSource` instance as the attribute source of `MetadataMBeanInfoAssembler`.

```
@Bean
public MBeanInfoAssembler assembler() {
    MetadataMBeanInfoAssembler assembler = new MetadataMBeanInfoAssembler();
    assembler.setAttributeSource(new AnnotationJmxAttributeSource());
    return assembler;
}
```

Then, you annotate your bean class and methods with the annotations `@ManagedResource`, `@ManagedAttribute`, and `@ManagedOperation` for `MetadataMBeanInfoAssembler` to assemble the management interface for this bean. The annotations are easily interpreted. They expose the element that they annotate. If you have a JavaBeans-compliant property, JMX will use the term attribute. Classes themselves are referred to as resources. In JMX, methods will be called operations. Knowing that, it's easy to see what the following code does:

```
package com.apress.springrecipes.replicator;  
...  
import org.springframework.jmx.export.annotation.ManagedAttribute;  
import org.springframework.jmx.export.annotation.ManagedOperation;  
import org.springframework.jmx.export.annotation.ManagedResource;  
  
@ManagedResource(description = "File replicator")  
public class FileReplicatorJMXImpl implements FileReplicator {  
    ...  
    @ManagedAttribute(description = "Get source directory")  
    public String getSrcDir() {  
        ...  
    }  
  
    @ManagedAttribute(description = "Set source directory")  
    public void setSrcDir(String srcDir) {  
        ...  
    }  
  
    @ManagedAttribute(description = "Get destination directory")  
    public String getDestDir() {  
        ...  
    }  
  
    @ManagedAttribute(description = "Set destination directory")  
    public void setDestDir(String destDir) {  
        ...  
    }  
    ...  
    @ManagedOperation(description = "Replicate files")  
    public synchronized void replicate() throws IOException {  
        ...  
    }  
}
```

Register MBeans with Annotations

In addition to exporting a bean explicitly with `MBeanExporter`, you can simply configure its subclass `AnnotationMBeanExporter` to auto-detect MBeans from beans declared in the IoC container. You don't need to configure an MBean assembler for this exporter, because it uses `MetadataMBeanInfoAssembler` with `AnnotationJmxAttributeSource` by default. You can delete the previous beans and assembler properties for this registration and simply leave the following:

```
@Configuration
public class JmxConfig {

    @Bean
    public MBeanExporter mbeanExporter() {
        AnnotationMBeanExporter mbeanExporter = new AnnotationMBeanExporter();
        return mbeanExporter;
    }
}
```

`AnnotationMBeanExporter` detects any beans configured in the IoC container with the `@ManagedResource` annotation and exports them as MBeans. By default, this exporter exports a bean to the domain whose name is the same as its package name. Also, it uses the bean's name in the IoC container as its MBean name, and the bean's short class name as its type. So the `documentReplicator` bean will be exported under the following MBean object name:

```
com.apress.springrecipes.replicator:name=documentReplicator, type=FileReplicatorJMXImpl
```

If you don't want to use the package name as the domain name, you can set the default domain for the exporter by adding the `defaultDomain` property.

```
@Bean
public MBeanExporter mbeanExporter() {
    AnnotationMBeanExporter mbeanExporter = new AnnotationMBeanExporter();
    mbeanExporter.setDefaultDomain("bean");
    return mbeanExporter;
}
```

After setting the default domain to `bean`, the `documentReplicator` bean is exported under the following MBean object name:

```
bean:name=documentReplicator,type=FileReplicatorJMXImpl
```

In addition, you can specify a bean's MBean object name in the `objectName` attribute of the `@ManagedResource` annotation. For example, you can export the file copier as an MBean by annotating it with the following annotations:

```
package com.apress.springrecipes.replicator;
...
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedOperationParameter;
import org.springframework.jmx.export.annotation.ManagedOperationParameters;
import org.springframework.jmx.export.annotation.ManagedResource;
```

```

@ManagedResource(
    objectName = "bean:name=fileCopier,type=FileCopierJMXImpl",
    description = "File Copier")
public class FileCopierImpl implements FileCopier {

    @ManagedOperation(
        description = "Copy file from source directory to destination directory")
    @ManagedOperationParameters( {
        @ManagedOperationParameter(
            name = "srcDir", description = "Source directory"),
        @ManagedOperationParameter(
            name = "destDir", description = "Destination directory"),
        @ManagedOperationParameter(
            name = "filename", description = "File to copy") })
    public void copyFile(String srcDir, String destDir, String filename)
        throws IOException {
        ...
    }
}

```

However, specifying the object name in this way works only for classes that you're going to create a single instance of in the IoC container (e.g., file copier), not for classes that you may create multiple instances of (e.g., file replicator). This is because you can only specify a single object name for a class. As a result, you shouldn't try to run the same server multiple times without changing the names.

Finally, another possibility is to rely on Spring class scanning to export MBeans decorated with `@ManagedResource`. If the beans are initialized in a Java Config class, you can decorate the configuration class with the `@EnableMBeanExport`. This tells Spring to export any beans created with the `@Bean` annotation, which are decorated with the `@EnableMBeanSupport`.

```

package com.apress.springrecipes.replicator.config;

...
import org.springframework.context.annotation.EnableMBeanExport;

@Configuration
@EnableMBeanExport
public class FileReplicatorConfig {
    ...
    @Bean
    public FileReplicatorJMXImpl documentReplicator() {
        FileReplicatorJMXImpl fRep = new FileReplicatorJMXImpl();
        fRep.setSrcDir(srcDir);
        fRep.setDestDir(destDir);
        fRep.setFileCopier(fileCopier());
        return fRep;
    }
    ...
}

```

Due to the presence of the `@EnableMBeanExport`, the bean `documentReplicator` of the type `FileReplicatorJMXImpl` gets exported as an MBean because its source is decorated with the `@ManagedResource` annotation.

Caution The use of the `@EnableMBeanExport` is done on `@Bean` instances with concrete classes, not interfaces like previous examples. Interface-based beans 'hide' the target class, which also hide the JMX managed resource annotations and the MBean is not exported.

14-2. Publish and Listen to JMX Notifications

Problem

You want to publish JMX notifications from your MBeans and listen to them with JMX notification listeners.

Solution

Spring allows your beans to publish JMX notifications through the `NotificationPublisher` interface. You can also register standard JMX notification listeners in the IoC container to listen to JMX notifications.

How It Works

Publish JMX Notifications

The Spring IoC container supports the beans that are going to be exported as MBeans to publish JMX notifications. These beans must implement the `NotificationPublisherAware` interface, to get access to `NotificationPublisher` so that they can publish notifications.

```
package com.apress.springrecipes.replicator;
...
import javax.management.Notification;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;

@ManagedResource(description = "File replicator")
public class FileReplicatorImpl implements FileReplicator,
    NotificationPublisherAware {
    ...
    private int sequenceNumber;
    private NotificationPublisher notificationPublisher;

    public void setNotificationPublisher(
        NotificationPublisher notificationPublisher) {
        this.notificationPublisher = notificationPublisher;
    }
    @ManagedOperation(description = "Replicate files")
    public void replicate() throws IOException {
        notificationPublisher.sendNotification(
            new Notification("replication.start", this, sequenceNumber));
    ...
}
```

```

        notificationPublisher.sendNotification(
            new Notification("replication.complete", this, sequenceNumber));
    sequenceNumber++;
}
}

```

In this file replicator, you send a JMX notification whenever a replication starts or completes. The notification is visible both in the standard output in the console and can also be seen in the JConsole Notifications menu in the MBeans tab, as illustrated in Figure 14-4.

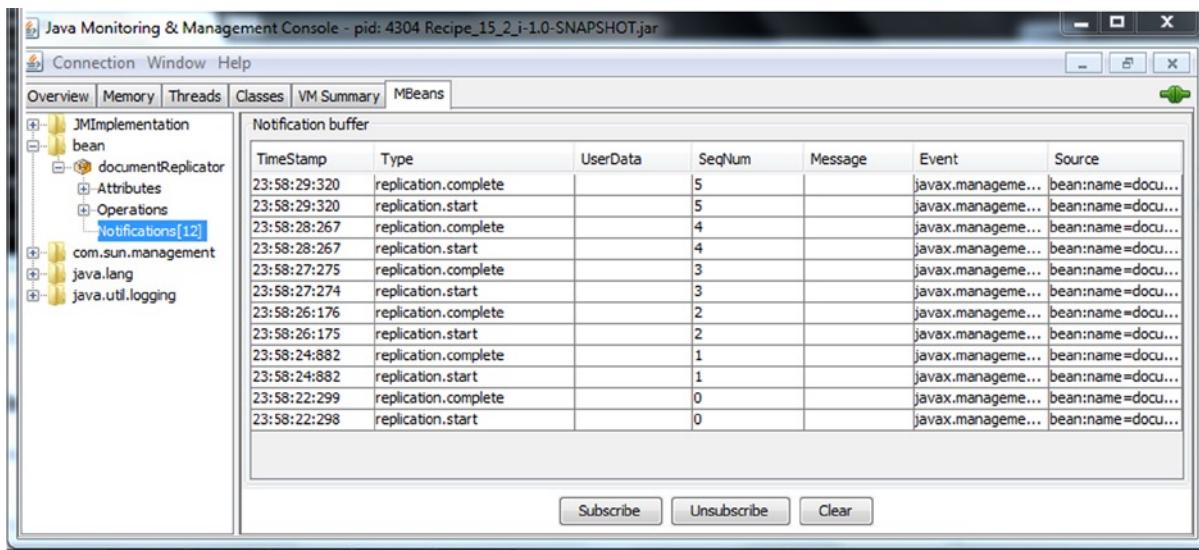


Figure 14-4. MBean events reported in JConsole

To see notifications in JConsole you must first click the ‘Subscribe’ button that appears toward the bottom, as illustrated in Figure 14-4. Then, when invoke the `replicate()` method using the JConsole button in the MBean ‘operations’ section, you’ll see two new notifications arrive. The first argument in the `Notification` constructor is the notification type, while the second is the notification source.

Listen to JMX Notifications

Now, let’s create a notification listener to listen to JMX notifications. Because a listener will be notified of many different types of notifications, such as `javax.management.AttributeChangeNotification` when an MBean’s attribute has changed, you have to filter those notifications that you are interested in handling.

```

package com.apress.springrecipes.replicator;

import javax.management.Notification;
import javax.management.NotificationListener;

public class ReplicationNotificationListener implements NotificationListener {

```

```

public void handleNotification(Notification notification, Object handback) {
    if (notification.getType().startsWith("replication")) {
        System.out.println(
            notification.getSource() + " " +
            notification.getType() + "#" +
            notification.getSequenceNumber());
    }
}
}

```

Then, you can register this notification listener with your MBean exporter to listen to notifications emitted from certain MBeans.

```

@Bean
public AnnotationMBeanExporter mbeanExporter() {
    AnnotationMBeanExporter mbeanExporter = new AnnotationMBeanExporter();
    mbeanExporter.setDefaultDomain("bean");
    mbeanExporter.setNotificationListenerMappings(notificationMappings());
    return mbeanExporter;
}

public Map<String, NotificationListener> notificationMappings() {
    Map<String, NotificationListener> mappings = new HashMap<>();
    mappings.put("bean:name=documentReplicator,type=FileReplicatorJMXImpl",
                new ReplicationNotificationListener());
    return mappings;
}

```

14-3. Access Remote JMX MBeans in Spring Problem

You want to access JMX MBeans running on a remote MBean server exposed by a JMX connector. When accessing remote MBeans directly with the JMX API, you have to write complex JMX-specific code.

Solution

Spring offers two approaches to simplify remote MBean access. First, it provides a factory bean to create an MBean server connection declaratively. With this server connection, you can query and update an MBean's attributes, as well as invoke its operations. Second, Spring provides another factory bean that allows you to create a proxy for a remote MBean. With this proxy, you can operate a remote MBean as if it were a local bean.

How It Works

Access Remote MBeans through an MBean Server Connection

A JMX client requires an MBean server connection to access MBeans running on a remote MBean server. Spring provides `org.springframework.jmx.support.MBeanServerConnectionFactoryBean` for you to create a connection to a remote JSR-160-enabled MBean server declaratively. You only have to provide the service URL for it to locate the MBean server. Now let's declare this factory bean in your client bean configuration file (e.g., `beans-jmx-client.xml`).

```

package com.apress.springrecipes.replicator.config;

import org.springframework.beans.factory.FactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jmx.support.MBeanServerConnectionFactoryBean;

import javax.management.MBeanServerConnection;
import java.net.MalformedURLException;

@Configuration
public class JmxClientConfiguration {

    @Bean
    public FactoryBean<MBeanServerConnection> mbeanServerConnection()
        throws MalformedURLException {
        MBeanServerConnectionFactoryBean mBeanServerConnectionFactoryBean =
            new MBeanServerConnectionFactoryBean();
        mBeanServerConnectionFactoryBean
            .setServiceUrl("service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator");
        return mBeanServerConnectionFactoryBean;
    }
}

```

With the MBean server connection created by this factory bean, you can access and operate the MBeans running on the RMI server running on port 1099.

Tip You can use the RMI server presented in Recipe 14-1 which exposes MBeans. If you're using the book's source code, after you build the application with Gradle, you can start the server with the command: `java -jar Recipe_14_1_iii-1.0-SNAPSHOT.jar`.

With the connection established between both points, you can query and update an MBean's attributes through the `getAttribute()` and `setAttribute()` methods, giving the MBean's object name and attribute name. You can also invoke an MBean's operations by using the `invoke()` method.

```

package com.apress.springrecipes.replicator;

import javax.management.Attribute;
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;

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

public class Client {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.replicator.config");
    }
}

```

```

MBeanServerConnection mbeanServerConnection =
    context.getBean(MBeanServerConnection.class);

ObjectName mbeanName = new ObjectName(
    "bean:name=documentReplicator");

String srcDir = (String) mbeanServerConnection.getAttribute(
    mbeanName, "SrcDir");

mbeanServerConnection.setAttribute(
    mbeanName, new Attribute("DestDir", srcDir + "_backup"));

mbeanServerConnection.invoke(
    mbeanName, "replicate", new Object[] {}, new String[] {});
}

}

```

In addition, let's create a JMX notification listener, so we can listen in on file replication notifications:

```

package com.apress.springrecipes.replicator;

import javax.management.Notification;
import javax.management.NotificationListener;

public class ReplicationNotificationListener implements NotificationListener {

    public void handleNotification(Notification notification, Object handback) {
        if (notification.getType().startsWith("replication")) {
            System.out.println(
                notification.getSource() + " " +
                notification.getType() + " #" +
                notification.getSequenceNumber());
        }
    }
}

```

You can register this notification listener to the MBean server connection to listen to notifications emitted from this MBean server.

```

package com.apress.springrecipes.replicator;
...
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;

public class Client {

    public static void main(String[] args) throws Exception {
        ...
        MBeanServerConnection mbeanServerConnection =
            (MBeanServerConnection) context.getBean("mbeanServerConnection");
    }
}

```

```

ObjectName mbeanName = new ObjectName(
    "bean:name=documentReplicator");

mbeanServerConnection.addNotificationListener(
    mbeanName, new ReplicationNotificationListener(), null, null);
...
}

}

```

After you run this application client, check JConsole for the RMI server application — using ‘Remote process’ at service:jmx:rmi://localhost/jndi/rmi://localhost:1099/replicator. Under the ‘Notifications’ menu of the MBeans tab, you’ll see new notification of type jmx.attribute.change as illustrated in Figure 14-5.

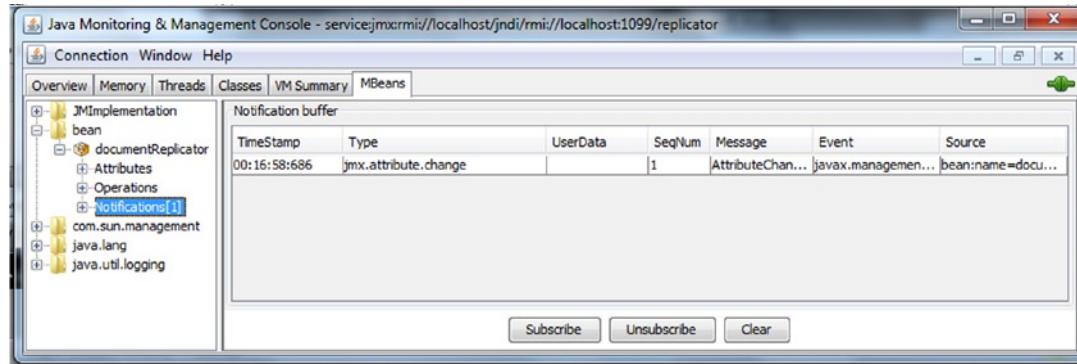


Figure 14-5. JConsole notification event invoked through RMI

Access Remote MBeans Through an MBean Proxy

Another approach that Spring offers for remote MBean access is through MBeanProxy, which can be created by MBeanProxyFactoryBean.

```

package com.apress.springrecipes.replicator.config;

...
import org.springframework.beans.factory.FactoryBean;
import org.springframework.jmx.access.MBeanProxyFactoryBean;
import org.springframework.jmx.support.MBeanServerConnectionFactoryBean;

import javax.management.MBeanServerConnection;
import java.net.MalformedURLException;

@Configuration
public class JmxClientConfiguration {
    ...
}

```

```

@Bean
public MBeanProxyFactoryBean fileReplicatorProxy() throws Exception {
    MBeanProxyFactoryBean fileReplicatorProxy = new MBeanProxyFactoryBean();
    fileReplicatorProxy.setServer(mbeanServerConnection().getObject());
    fileReplicatorProxy.setObjectName("bean:name=documentReplicator");
    fileReplicatorProxy.setProxyInterface(FileReplicator.class);
    return fileReplicatorProxy;
}
}

```

You need to specify the object name and the server connection for the MBean you are going to proxy. The most important is the proxy interface, whose local method calls will be translated into remote MBean calls behind the scenes.

Now, you can operate the remote MBean through this proxy as if it were a local bean. The preceding MBean operations invoked on the MBean server connection directly can be simplified as follows:

```

package com.apress.springrecipes.replicator;
...
public class Client {

    public static void main(String[] args) throws Exception {
        ...
        FileReplicator fileReplicatorProxy = context.getBean(FileReplicator.class);
        String srcDir = fileReplicatorProxy.getSrcDir();
        fileReplicatorProxy.setDestDir(srcDir + "_backup");
        fileReplicatorProxy.replicate();
    }
}

```

14-4. Send E-mail with Spring's E-mail Support

Problem

Many applications need to send e-mail. In a Java application, you can send e-mail with the JavaMail API. However, when using JavaMail, you have to handle JavaMail-specific mail sessions and exceptions. As a result, an application becomes JavaMail dependent and hard to switch to another e-mail API.

Solution

Spring's e-mail support makes it easier to send e-mail by providing an abstract and implementation-independent API for sending e-mail. The core interface of Spring's e-mail support is `MailSender`.

The `JavaMailSender` interface is a subinterface of `MailSender` that includes specialized JavaMail features such as Multipurpose Internet Mail Extensions (MIME message support). To send an e-mail message with HTML content, inline images, or attachments, you have to send it as a MIME message.

How It Works

Suppose you want the file replicator application from the previous recipes to notify the administrator of any error. First, you create the following ErrorNotifier interface, which includes a method for notifying of a file copy error:

```
package com.apress.springrecipes.replicator;  
  
public interface ErrorNotifier {  
  
    public void notifyCopyError(String srcDir, String destDir, String filename);  
}
```

Note Invoking this notifier in case of error is left for you to accomplish. As you can consider error handling a crosscutting concern, AOP would be an ideal solution to this problem. You can write an after throwing advice to invoke this notifier.

Next, you can implement this interface to send a notification in a way of your choice. The most common way is to send e-mail. Before you implement the interface in this way, you may need a local e-mail server that supports the Simple Mail Transfer Protocol (SMTP) for testing purposes. We recommend installing Apache James Server (<http://james.apache.org/server/index.html>), which is very easy to install and configure.

Note You can download Apache James Server (e.g., version 2.3.2) from the Apache James web site and extract it to a directory of your choice to complete the installation. To start it, just execute the run script (located in the bin directory).

Let's create two user accounts for sending and receiving e-mail with this server. By default, the remote manager service of James listens on port 4555. You can telnet, using a console, to this port and run the following commands to add the users system and admin, whose passwords are 12345:

```
> telnet 127.0.0.1 4555  
JAMES Remote Administration Tool 2.3.2  
Please enter your login and password  
Login id:  
root  
Password:  
itroot  
Welcome root. HELP for a list of commands  
adduser system 12345  
User system added  
adduser admin 12345  
User admin added  
listusers  
Existing accounts 2  
user: admin  
user: system  
quit  
Bye
```

Send E-mail Using the JavaMail API

Now, let's take a look at how to send e-mail using the JavaMail API. You can implement the `ErrorNotifier` interface to send e-mail notifications in case of errors.

```
package com.apress.springrecipes.replicator;

import java.util.Properties;

import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

public class EmailErrorNotifier implements ErrorNotifier {

    public void notifyCopyError(String srcDir, String destDir, String filename) {
        Properties props = new Properties();
        props.put("mail.smtp.host", "localhost");
        props.put("mail.smtp.port", "25");
        props.put("mail.smtp.username", "system");
        props.put("mail.smtp.password", "12345");
        Session session = Session.getDefaultInstance(props, null);
        try {
            Message message = new MimeMessage(session);
            message.setFrom(new InternetAddress("system@localhost"));
            message.setRecipients(Message.RecipientType.TO,
                InternetAddress.parse("admin@localhost"));
            message.setSubject("File Copy Error");
            message.setText(
                "Dear Administrator,\n\n" +
                "An error occurred when copying the following file :\n" +
                "Source directory : " + srcDir + "\n" +
                "Destination directory : " + destDir + "\n" +
                "Filename : " + filename);
            Transport.send(message);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

You first open a mail session connecting to an SMTP server by defining the properties. Then, you create a message from this session for constructing your e-mail. After that, you send the e-mail by making a call to `Transport.send()`. When dealing with the JavaMail API, you have to handle the checked exception `MessagingException`. Note that all of these classes, interfaces, and exceptions are defined by JavaMail.

Next, declare an instance of `EmailErrorNotifier` in the Spring IoC container for sending e-mail notifications in case of file replication errors.

```
package com.apress.springrecipes.replicator.config;

import com.apress.springrecipes.replicator.EmailErrorNotifier;
import com.apress.springrecipes.replicator.ErrorNotifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MailConfiguration {

    @Bean
    public ErrorNotifier errorNotifier() {
        return new EmailErrorNotifier();
    }
}
```

You can write the following Main class to test `EmailErrorNotifier`. After running it, you can configure your e-mail application to receive the e-mail from your James Server via POP3.

```
package com.apress.springrecipes.replicator;

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

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.replicator.config");

        ErrorNotifier errorNotifier = context.getBean(ErrorNotifier.class);
        errorNotifier.notifyCopyError("c:/documents", "d:/documents", "spring.doc");
    }
}
```

To verify the email was sent, you can login to the POP server included with Apache James. You can telnet, using a console, to port 110 and run the following commands to view the email for user admin, whose password is the same you set on creation:

```
> telnet 127.0.0.1 110
OK workstation POP3 server <JAMES POP3 Server 2.3.2> ready
USER admin
+OK
PASS 12345
+OK Welcome admin
LIST
+ OK 1 698
RETR 1
+OK Message follows
...

```

Send E-mail with Spring's MailSender

Now, let's look at how to send e-mail with the help of Spring's `MailSender` interface, which can send `SimpleMailMessage` in its `send()` method. With this interface, your code is no longer JavaMail specific, and now it's easier to test.

```
package com.apress.springrecipes.replicator;

import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;

public class EmailErrorNotifier implements ErrorNotifier {

    private MailSender mailSender;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void notifyCopyError(String srcDir, String destDir, String filename) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom("system@localhost");
        message.setTo("admin@localhost");
        message.setSubject("File Copy Error");
        message.setText(
            "Dear Administrator,\n\n" +
            "An error occurred when copying the following file :\n" +
            "Source directory : " + srcDir + "\n" +
            "Destination directory : " + destDir + "\n" +
            "Filename : " + filename);
        mailSender.send(message);
    }
}
```

Next, you have to configure a `MailSender` implementation in the bean configuration file and inject it into `EmailErrorNotifier`. In Spring, the unique implementation of this interface is `JavaMailSenderImpl`, which uses JavaMail to send e-mail.

```
@Configuration
public class MailConfiguration {

    @Bean
    public ErrorNotifier errorNotifier() {
        EmailErrorNotifier errorNotifier = new EmailErrorNotifier();
        errorNotifier.setMailSender(mailSender());
        return errorNotifier;
    }

    @Bean
    public JavaMailSenderImpl mailSender() {
        JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
        mailSender.setHost("localhost");
        mailSender.setPort(25);
    }
}
```

```

        mailSender.setUsername("system");
        mailSender.setPassword("12345");
        return mailSender;
    }
}

```

The default port used by `JavaMailSenderImpl` is the standard SMTP port 25, so if your e-mail server listens on this port for SMTP, you can simply omit this property. Also, if your SMTP server doesn't require user authentication, you needn't set the username and password.

If you have a JavaMail session configured in your Java app server, you can first look it up with the help of `JndiLocatorDelegate`.

```

@Bean
public Session mailSession() throws NamingException {
    return JndiLocatorDelegate
        .createDefaultResourceRefLocator()
        .lookup("mail/Session", Session.class);
}

```

You can inject the JavaMail session into `JavaMailSenderImpl` for its use. In this case, you no longer need to set the host, port, username, or password.

```

@Bean
public JavaMailSenderImpl mailSender() {
    JavaMailSenderImpl mailSender = new JavaMailSenderImpl();
    mailSender.setSession(mailSession());
    return mailSender;
}

```

Define an E-mail Template

Constructing an e-mail message from scratch in the method body is not efficient, because you have to hard-code the e-mail properties. Also, you may have difficulty in writing the e-mail text in terms of Java strings. You can consider defining an e-mail message template in the bean configuration file and construct a new e-mail message from it.

```

@Configuration
public class MailConfiguration {
    ...
    @Bean
    public ErrorNotifier errorNotifier() {
        EmailErrorNotifier errorNotifier = new EmailErrorNotifier();
        errorNotifier.setMailSender(mailSender());
        errorNotifier.setCopyErrorMailMessage(copyErrorMailMessage());
        return errorNotifier;
    }

    @Bean
    public SimpleMailMessage copyErrorMailMessage() {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom("system@localhost");
        message.setTo("admin@localhost");
        message.setSubject("File Copy Error");
    }
}

```

```

        message.setText("Dear Administrator,\n" +
            "\n" +
            " An error occurred when copying the following file :\n" +
            "\t\t Source directory : %s\n" +
            "\t\t Destination directory : %s\n" +
            "\t\t Filename : %s");
    return message;
}
}

```

Note that in the preceding message text, you include the placeholders %s, which will be replaced by message parameters through `String.format()`. Of course, you can also use a powerful templating language such as Velocity or FreeMarker to generate the message text according to a template. It's also a good practice to separate mail message templates from bean configuration files.

Each time you send e-mail, you can construct a new `SimpleMailMessage` instance from this injected template. Then you can generate the message text using `String.format()` to replace the %s placeholders with your message parameters.

```

package com.apress.springrecipes.replicator;
...
import org.springframework.mail.SimpleMailMessage;

public class EmailErrorNotifier implements ErrorNotifier {
    ...
    private SimpleMailMessage copyErrorMailMessage;

    public void setCopyErrorMailMessage(SimpleMailMessage copyErrorMailMessage) {
        this.copyErrorMailMessage = copyErrorMailMessage;
    }

    public void notifyCopyError(String srcDir, String destDir, String filename) {
        SimpleMailMessage message = new SimpleMailMessage(copyErrorMailMessage);
        message.setText(String.format(
            copyErrorMailMessage.getText(), srcDir, destDir, filename));
        mailSender.send(message);
    }
}

```

Send e-mail with attachments (MIME Messages)

So far, the `SimpleMailMessage` class you used can send only a simple plain text e-mail message. To send e-mail that contains HTML content, inline images, or attachments, you have to construct and send a MIME message instead. MIME is supported by JavaMail through the `javax.mail.internet.MimeMessage` class.

First of all, you have to use the `JavaMailSender` interface instead of its parent interface `MailSender`. The `JavaMailSenderImpl` instance you injected does implement this interface, so you needn't modify your bean configurations. The following notifier sends Spring's bean configuration file as an e-mail attachment to the administrator:

```

package com.apress.springrecipes.replicator;

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

```

```

import org.springframework.core.io.ClassPathResource;
import org.springframework.mail.MailParseException;
import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

public class EmailErrorNotifier implements ErrorNotifier {

    private JavaMailSender mailSender;
    private SimpleMailMessage copyErrorMailMessage;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setCopyErrorMailMessage(SimpleMailMessage copyErrorMailMessage) {
        this.copyErrorMailMessage = copyErrorMailMessage;
    }

    public void notifyCopyError(String srcDir, String destDir, String filename) {
        MimeMessage message = mailSender.createMimeMessage();
        try {
            MimeMessageHelper helper = new MimeMessageHelper(message, true);
            helper.setFrom(copyErrorMailMessage.getFrom());
            helper.setTo(copyErrorMailMessage.getTo());

            helper.setSubject(copyErrorMailMessage.getSubject());
            helper.setText(String.format(
                copyErrorMailMessage.getText(), srcDir, destDir, filename));

            ClassPathResource config = new ClassPathResource("beans.xml");
            helper.addAttachment("beans.xml", config);
        } catch (MessagingException e) {
            throw new MailParseException(e);
        }
        mailSender.send(message);
    }
}

```

Unlike `SimpleMailMessage`, the `MimeMessage` class is defined by JavaMail, so you can only instantiate it by calling `mailSender.createMimeMessage()`. Spring provides the helper class `MimeMessageHelper` to simplify the operations of `MimeMessage`. It allows you to add an attachment from a Spring Resource object. However, the operations of this helper class can still throw JavaMail's `MessagingException`. You have to convert this exception into Spring's mail runtime exception for consistency.

Spring offers another method for you to construct a MIME message, which is through implementing the `MimeMessagePreparator` interface.

```

package com.apress.springrecipes.replicator;
...
import javax.mail.internet.MimeMessage;

import org.springframework.mail.javamail.MimeMessagePreparator;

```

```

public class EmailErrorNotifier implements ErrorNotifier {
    ...
    public void notifyCopyError(
        final String srcDir, final String destDir, final String filename) {
        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws Exception {
                MimeMessageHelper helper =
                    new MimeMessageHelper(mimeMessage, true);
                helper.setFrom(copyErrorMailMessage.getFrom());
                helper.setTo(copyErrorMailMessage.getTo());
                helper.setSubject(copyErrorMailMessage.getSubject());
                helper.setText(String.format(
                    copyErrorMailMessage.getText(), srcDir, destDir, filename));

                ClassPathResource config = new ClassPathResource("beans.xml");
                helper.addAttachment("beans.xml", config);
            }
        };
        mailSender.send(preparator);
    }
}

```

In the `prepare()` method, you can prepare the `MimeMessage` object, which is precreated for `JavaMailSender`. If there's any exception thrown, it will be converted into Spring's mail runtime exception automatically.

14-5. Schedule tasks with Spring's Quartz Support Problem

Your application has an advanced scheduling requirement that you want to fulfill using Quartz Scheduler. Such a requirement might be something seemingly complex like the ability to run at arbitrary times, or at strange intervals ("every other Thursday, but only after 10 am and before 2 pm"). Moreover, you want to configure scheduling jobs in a declarative way.

Solution

Spring provides utility classes for Quartz to enable scheduling jobs without programming against the Quartz API.

How It Works

Use Quartz Without Spring's Support

To use Quartz for scheduling, you first need to create a job by implementing the `Job` interface. For example, the following job executes the `replicate()` method of the file replicator designed in the previous recipes. It retrieves a job data map — which is a Quartz concept to define jobs — through the `JobExecutionContext` object.

```

package com.apress.springrecipes.replicator;
...
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

```

```

public class FileReplicationJob implements Job {

    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        Map dataMap = context.getJobDetail().getJobDataMap();
        FileReplicator fileReplicator =
            (FileReplicator) dataMap.get("fileReplicator");
        try {
            fileReplicator.replicate();
        } catch (IOException e) {
            throw new JobExecutionException(e);
        }
    }
}

```

After creating the job, you configure and schedule it with the Quartz API. For instance, the following scheduler runs your file replication job every 60 seconds with a 5-second delay for the first time of execution:

```

package com.apress.springrecipes.replicator;
...
import org.quartz.JobDetail;
import org.quartz.JobDataMap;
import org.quartz.JobBuilder;
import org.quartz.Trigger;
import org.quartz.TriggerBuilder;
import org.quartz.SimpleScheduleBuilder;
import org.quartz.DateBuilder.IntervalUnit.*;
import org.quartz.Scheduler;
import org.quartz.impl.StdSchedulerFactory;

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

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.replicator.config");

        FileReplicator documentReplicator = context.getBean(FileReplicator.class);

        JobDataMap jobDataMap = new JobDataMap();
        jobDataMap.put("fileReplicator", documentReplicator);

        JobDetail job = JobBuilder.newJob(FileReplicationJob.class)
            .withIdentity("documentReplicationJob")
            .storeDurably()
            .usingJobData(jobDataMap)
            .build();

        Trigger trigger = TriggerBuilder.newTrigger()
            .withIdentity("documentReplicationTrigger")

```

```

.startAt(new Date(System.currentTimeMillis() + 5000))
.forJob(job)
.withSchedule(SimpleScheduleBuilder.simpleSchedule()
    .withIntervalInSeconds(60)
    .repeatForever())
.build();

Scheduler scheduler = new StdSchedulerFactory().getScheduler();
scheduler.start();
scheduler.scheduleJob(job, trigger);
}
}

```

In the Main class, you first create a job map. In this case it's a single job, where the key is a descriptive name and the value is an object reference for the job. Next, you define the job details for the file replication job in a Job Detail object and prepare job data in its jobDataMap property. Next, you create a SimpleTrigger object to configure the scheduling properties. Finally, you create a scheduler to run your job using this trigger.

Quartz supports various types of schedules to run jobs at different intervals. Schedules are defined as part of triggers. In the most recent release Quartz schedules are: SimpleScheduleBuilder, CronScheduleBuilder, CalendarIntervalScheduleBuilder and DailyTimeIntervalScheduleBuilder. SimpleScheduleBuilder allows you to schedule jobs setting properties such as start time, end time, repeat interval, and repeat count. CronScheduleBuilder accepts a Unix cron expression for you to specify the times to run your job. For example, you can replace the preceding SimpleScheduleBuilder with the following CronScheduleBuilder to run a job at 17:30 every day:

```
.withSchedule(CronScheduleBuilder.cronSchedule(" 0 30 17 * * ?"))
```

A cron expression is made up of seven fields (the last field is optional), separated by spaces. Table 14-1 shows the field description for a cron expression.

Table 14-1. Field Description for a Cron Expression

Position	Field Name	Range
1	Second	0–59
2	Minute	0–59
3	Hour	0–23
4	Day of month	1–31
5	Month	1–12 or JAN–DEC
6	Day of week	1–7 or SUN–SAT
7	Year (optional)	1970–2099

Each part of a cron expression can be assigned a specific value (e.g., 3), a range (e.g., 1–5), a list (e.g., 1,3,5), a wildcard (* matches all values), or a question mark (? is used in either of the “Day of month” and “Day of week” fields for matching one of these fields but not both).

CalendarIntervalScheduleBuilder allows you to schedule jobs based on calendar times (day, week, month, year), whereas the DailyTimeIntervalScheduleBuilder provides convenience utilities to set a job's end time (e.g., methods like endingDailyAt() & endingDailyAfterCount())

Use Quartz with Spring's Support

When using Quartz, you can create a job by implementing the Job interface and retrieve job data from the job data map through JobExecutionContext. To decouple your job class from the Quartz API, Spring provides QuartzJobBean, which you can extend to retrieve job data through setter methods. QuartzJobBean converts the job data map into properties and injects them via the setter methods.

```
package com.apress.springrecipes.replicator;
...
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.springframework.scheduling.quartz.QuartzJobBean;

public class FileReplicationJob extends QuartzJobBean {

    private FileReplicator fileReplicator;

    public void setFileReplicator(FileReplicator fileReplicator) {
        this.fileReplicator = fileReplicator;
    }

    protected void executeInternal(JobExecutionContext context)
        throws JobExecutionException {
        try {
            fileReplicator.replicate();
        } catch (IOException e) {
            throw new JobExecutionException(e);
        }
    }
}
```

Then, you can configure a Quartz JobDetail object in Spring's bean configuration file through JobDetailBean. By default, Spring uses this bean's name as the job name. You can modify it by setting the name property.

```
@Bean
@.Autowired
public JobDetailFactoryBean documentReplicationJob(FileReplicator fileReplicator) {
    JobDetailFactoryBean documentReplicationJob = new JobDetailFactoryBean();
    documentReplicationJob.setJobClass(FileReplicationJob.class);
    documentReplicationJob.setDurability(true);
    documentReplicationJob.setJobDataAsMap(Collections.singletonMap("fileReplicator", fileReplicator));
    return documentReplicationJob;
}
```

Spring also offers MethodInvokingJobDetailFactoryBean for you to define a job that executes a single method of a particular object. This saves you the trouble of creating a job class. You can use the following job detail to replace the previous:

```
@Bean
@.Autowired
public MethodInvokingJobDetailFactoryBean documentReplicationJob(FileReplicator fileReplicator) {
```

```
MethodInvokingJobDetailFactoryBean documentReplicationJob =
new MethodInvokingJobDetailFactoryBean();
documentReplicationJob.setTargetObject(fileReplicator);
documentReplicationJob.setTargetMethod("replicatie");
return documentReplicationJob;
}
```

Once you define a job, you can configure a Quartz Trigger. Spring supports the `SimpleTriggerFactoryBean` and the `CronTriggerFactoryBean`. The `SimpleTriggerFactoryBean`, requires a reference to a `JobDetail` object and provides common values for schedule properties, such as start time and repeat count.

```
@Bean
@Autowired
public SimpleTriggerFactoryBean documentReplicationTrigger(JobDetail documentReplicationJob) {
    SimpleTriggerFactoryBean documentReplicationTrigger = new SimpleTriggerFactoryBean();
    documentReplicationTrigger.setJobDetail(documentReplicationJob);
    documentReplicationTrigger.setStartDelay(5000);
    documentReplicationTrigger.setRepeatInterval(60000);
    return documentReplicationTrigger;
}
```

You can also use the `CronTriggerFactoryBean` to configure a cron like schedule.

```
@Bean
@Autowired
public CronTriggerFactoryBean documentReplicationTrigger(JobDetail documentReplicationJob) {
    CronTriggerFactoryBean documentReplicationTrigger = new CronTriggerFactoryBean();
    documentReplicationTrigger.setJobDetail(documentReplicationJob);
    documentReplicationTrigger.setStartDelay(5000);
    documentReplicationTrigger.setCronExpression("0/60 * * * * ?");
    return documentReplicationTrigger;
}
```

Finally, you once you have the Quartz job and trigger, you can configure a `SchedulerFactoryBean` instance to create a `Scheduler` object for running your trigger. You can specify multiple triggers in this factory bean.

```
@Bean
@Autowired
public SchedulerFactoryBean scheduler(Trigger[] triggers) {
    SchedulerFactoryBean scheduler = new SchedulerFactoryBean();
    scheduler.setTriggers(triggers);
    return scheduler;
}
```

Now, you can simply start your scheduler with the following Main class. In this way, you don't require a single line of code for scheduling jobs.

```
package com.apress.springrecipes.replicator;

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

```

public class Main {

    public static void main(String[] args) throws Exception {
        new AnnotationConfigApplicationContext("com.apress.springrecipes.replicator.config");
    }
}

```

14-6. Schedule tasks with Spring's Scheduling

Problem

You want to schedule a method invocation in a consistent manner, using either a cron expression, an interval, or a rate, and you don't want to have to go through Quartz just to do it.

Solution

Spring has support to configure TaskExecutors and TaskSchedulers. This capability, coupled with the ability to schedule method execution using the @Scheduled annotation, makes Spring scheduling support work with a minimum of fuss: all you need are a method, an annotation, and to have switched on the scanner for annotations.

How It Works

Let's revisit the example in the last recipe: we want to schedule a call to the replication method on the bean using a cron expression. Our configuration class looks like the following:

```

package com.apress.springrecipes.replicator.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.SchedulingConfigurer;
import org.springframework.scheduling.config.ScheduledTaskRegistrar;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

@Configuration
@EnableScheduling
public class SchedulingConfiguration implements SchedulingConfigurer {

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.setScheduler(scheduler());
    }

    @Bean(destroyMethod = "shutdown")
    public Executor scheduler() {
        return Executors.newScheduledThreadPool(10);
    }
}

```

We enable the annotation driven scheduling support by specifying `@EnableScheduling`, this will register a bean which scans the beans in the application context for the `@Scheduled` annotation. We also implemented the interface `SchedulingConfigurer` as we wanted to do some additional configuration of our scheduler. We want to give it a pool of 10 threads to execute our scheduled tasks.

```
package com.apress.springrecipes.replicator;

import org.springframework.scheduling.annotation.Scheduled;
import java.io.File;
import java.io.IOException;

public class FileReplicatorImpl implements FileReplicator {

    ...
    @Scheduled(fixedDelay = 60 * 1000)
    public synchronized void replicate() throws IOException {
        File[] files = new File(srcDir).listFiles();

        for (File file : files) {
            if (file.isFile()) {
                fileCopier.copyFile(srcDir, destDir, file.getName());
            }
        }
    }
}
```

Note that we've annotated the `replicate()` method with a `@Scheduled` annotation. Here, we've told the scheduler to execute the method every 60 seconds. Alternatively, we might specify a `fixedRate` value for the `@Scheduled` annotation, which would measure the time between successive starts and then trigger another run.

```
@Scheduled(fixedRate = 60 * 1000)
public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();

    for (File file : files) {
        if (file.isFile()) {
            fileCopier.copyFile(srcDir, destDir, file.getName());
        }
    }
}
```

Finally, we might want more complex control over the execution of the method. In this case, we can use a cron expression, just as we did in the Quartz example.

```
@Scheduled( cron = "0/60 * * * ? " )
public synchronized void replicate() throws IOException {
    File[] files = new File(srcDir).listFiles();
```

```

for (File file : files) {
    if (file.isFile()) {
        fileCopier.copyFile(srcDir, destDir, file.getName());
    }
}
}

```

There is support for configuring all of this in the Java too. This might be useful if you didn't want to, or couldn't, add an annotation to an existing bean method. Here's a look at how we might re-create the preceding annotation-centric examples using the Spring `ScheduledTaskRegistrar`.

```

@Configuration
@EnableScheduling
public class SchedulingConfiguration implements SchedulingConfigurer {

    @Autowired
    private FileReplicator fileReplicator;

    @Override
    public void configureTasks(ScheduledTaskRegistrar taskRegistrar) {
        taskRegistrar.setScheduler(scheduler());
        taskRegistrar.addFixedDelayTask(new Runnable(){
            @Override
            public void run() {
                try {
                    fileReplicator.replicate();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }, 60000);
    }

    @Bean(destroyMethod = "shutdown")
    public Executor scheduler() {
        return Executors.newScheduledThreadPool(10);
    }
}

```

14-7. Expose and Invoke Services through RMI

Problem

You want to expose a service from your Java application for other Java-based clients to invoke remotely. Because both parties are running on the Java platform, you can choose a pure Java-based solution without considering cross-platform portability.

Solution

Remote Method Invocation (RMI) is a Java-based remoting technology that allows two Java applications running in different JVMs to communicate with each other. With RMI, an object can invoke the methods of a remote object. RMI relies on object serialization to marshal and unmarshal method arguments and return values.

To expose a service through RMI, you have to create the service interface that extends `java.rmi.Remote` and whose methods declare throwing `java.rmi.RemoteException`. Then, you create the service implementation for this interface. After that, you start an RMI registry and register your service to it. So there are quite a lot of steps required for exposing a simple service.

To invoke a service through RMI, you first look up the remote service reference in an RMI registry, and then, you can call the methods on it. However, to call the methods on a remote service, you must handle `java.rmi.RemoteException` in case any exception is thrown by the remote service.

Spring's remoting facilities can significantly simplify the RMI usage on both the server and client sides. On the server side, you can use `RmiServiceExporter` to export a Spring POJO as an RMI service whose methods can be invoked remotely. It's just several lines of bean configuration without any programming. Beans exported in this way don't need to implement `java.rmi.Remote` or throw `java.rmi.RemoteException`. On the client side, you can simply use `RmiProxyFactoryBean` to create a proxy for the remote service. It allows you to use the remote service as if it were a local bean. Again, it requires no additional programming at all.

How It Works

Suppose you're going to build a weather web service for clients running on different platforms. This service includes an operation for querying a city's temperatures on multiple dates. First, let's create the `TemperatureInfo` class representing the minimum, maximum, and average temperatures of a particular city and date.

```
package com.apress.springrecipes.weather;
...
public class TemperatureInfo implements Serializable {

    private String city;
    private Date date;
    private double min;
    private double max;
    private double average;

    // Constructors, Getters and Setters
    ...
}
```

Next, let's define the service interface that includes the `getTemperatures()` operation, which returns a city's temperatures on multiple dates.

```
package com.apress.springrecipes.weather;
...
public interface WeatherService {

    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates);
}
```

You have to provide an implementation for this interface. In a production application, you would implement this service interface by querying the database. Here, we'll hard-code the temperatures for testing purposes.

```

package com.apress.springrecipes.weather;
...
public class WeatherServiceImpl implements WeatherService {

    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
        List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();
        for (Date date : dates) {
            temperatures.add(new TemperatureInfo(city, date, 5.0, 10.0, 8.0));
        }
        return temperatures;
    }
}

```

Expose an RMI Service

Next, let's expose the weather service as an RMI service. To use Spring's remoting facilities, we'll create a Java Config class to create the necessary beans and export the weather service as an RMI service by using `RmiServiceExporter`.

```

package com.apress.springrecipes.weather.config;

...
import com.apress.springrecipes.weather.WeatherService;
import com.apress.springrecipes.weather.WeatherServiceImpl;

import org.springframework.remoting.rmi.RmiServiceExporter;

@Configuration
public class WeatherConfig {

    @Bean
    public WeatherService weatherService() {
        WeatherService wService = new WeatherServiceImpl();
        return wService;
    }

    @Bean
    public RmiServiceExporter rmiService() {
        RmiServiceExporter rmiService = new RmiServiceExporter();
        rmiService.setServiceName("WeatherService");
        rmiService.setServiceInterface(com.apress.springrecipes.weather.WeatherService.class);
        rmiService.setService(weatherService());
        return rmiService;
    }
}

```

There are several properties you must configure for an `RmiServiceExporter` instance, including the service name, the service interface, and the service object to export. You can export any bean configured in the IoC container as an RMI service. `RmiServiceExporter` will create an RMI proxy to wrap this bean and bind it to the RMI registry. When the proxy receives an invocation request from the RMI registry, it will invoke the corresponding method on the bean.

By default, `RmiServiceExporter` attempts to look up an RMI registry at localhost port 1099. If it can't find the RMI registry, it will start a new one. However, if you want to bind your service to another running RMI registry, you can specify the host and port of that registry in the `registryHost` and `registryPort` properties. Note that once you specify the registry host, `RmiServiceExporter` will not start a new registry, even if the specified registry doesn't exist.

Run the following `RmiServer` class to create an application context:

```
package com.apress.springrecipes.weather;

import org.springframework.context.support.GenericXmlApplicationContext;

public class RmiServer {

    public static void main(String[] args) {
        new AnnotationConfigApplicationContext("com.apress.springrecipes.weather.config");
    }
}
```

In this configuration, the server will launch; among the output, you should see a message indicating that an existing RMI registry could not be found.

Invoke an RMI Service

By using Spring's remoting facilities, you can invoke a remote service just like a local bean. For example, you can create a client that refers to the weather service by its interface.

```
package com.apress.springrecipes.weather;
...
public class WeatherServiceClient {

    @Autowired
    private WeatherService weatherService;

    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }

    public TemperatureInfo getTodayTemperature(String city) {
        List<Date> dates = Arrays.asList(new Date[] { new Date() });
        List<TemperatureInfo> temperatures =
            weatherService.getTemperatures(city, dates);
        return temperatures.get(0);
    }
}
```

Notice the `weatherService` field is marked as `@Autowired`, so we'll need to create an instance of this bean. The `weatherService` will use `RmiProxyFactoryBean` to create a proxy for the remote service. Then, you can use this service as if it were a local bean. The following Java Config class illustrates the necessary beans for the RMI client:

```
package com.apress.springrecipes.weather.config;

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

```

import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.rmi.RmiProxyFactoryBean;
import com.apress.springrecipes.weather.WeatherServiceClient;
import com.apress.springrecipes.weather.WeatherService;

@Configuration
public class WeatherConfigClient {

    @Bean
    public RmiProxyFactoryBean weatherService() {
        RmiProxyFactoryBean rmiProxy = new RmiProxyFactoryBean();
        rmiProxy.setServiceUrl("rmi://localhost:1099/WeatherService");
        rmiProxy.setServiceInterface(com.apress.springrecipes.weather.WeatherService.class);
        return rmiProxy;
    }

    @Bean
    public WeatherServiceClient weatherClient() {
        WeatherServiceClient wServiceClient = new WeatherServiceClient();
        return wServiceClient;
    }
}

```

There are two properties you must configure for an `RmiProxyFactoryBean` instance. The `serviceUrl` property specifies the host and port of the RMI registry, as well as the service name. The `serviceInterface` allows this factory bean to create a proxy for the remote service against a known, shared Java interface. The proxy will transfer the invocation requests to the remote service transparently. In addition to the `RmiProxyFactoryBean` instance, we also create an instance of the `WeatherServiceClient` called `weatherClient`.

To start the client, simply create a configuration file to scan the Java Config class.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.2.xsd">

    <context:component-scan base-package="com.apress.springrecipes.weather.config" />
</beans>

```

And next, run the following `RmiClient` main class:

```

package com.apress.springrecipes.weather;

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

```

```

public class RmiClient {

    public static void main(String[] args) {
        ApplicationContext context =
            new GenericXmlApplicationContext("appContext.xml");
        WeatherServiceClient client =
            (WeatherServiceClient) context.getBean("weatherClient");

        TemperatureInfo temperature = client.getTodayTemperature("Houston");
        System.out.println("Min temperature : " + temperature.getMin());
        System.out.println("Max temperature : " + temperature.getMax());
        System.out.println("Average temperature : " + temperature.getAverage());
    }
}

```

14-8. Expose and Invoke Services through HTTP

Problem

RMI communicates through its own protocol, which may not pass through firewalls. Ideally, you'd like to communicate over HTTP.

Solution

Hessian and Burlap are two simple lightweight remoting technologies developed by Caucho Technology (<http://www.caucho.com/>). They both communicate using proprietary messages over HTTP and have their own serialization mechanism, but they are much simpler than RMI. The only difference between Hessian and Burlap is that Hessian communicates using binary messages, and Burlap communicates using XML messages. The message formats of both Hessian and Burlap are also supported on other platforms besides Java, such as PHP, Python, C#, and Ruby. This allows your Java applications to communicate with applications running on the other platforms.

In addition to the preceding two technologies, the Spring framework itself also offers a remoting technology called HTTP Invoker. It also communicates over HTTP, but uses Java's object serialization mechanism to serialize objects. Unlike Hessian and Burlap, HTTP Invoker requires both sides of a service to be running on the Java platform and using the Spring framework. However, it can serialize all kinds of Java objects, some of which may not be serialized by Hessian/Burlap's proprietary mechanism.

Spring's remoting facilities are consistent in exposing and invoking remote services with these technologies. On the server side, you can create a service exporter such as `HessianServiceExporter`, `BurlapServiceExporter`, or `HttpInvokerServiceExporter` to export a Spring bean as a remote service whose methods can be invoked remotely. It's just a few lines of bean configurations without any programming. On the client side, you can also configure a proxy factory bean such as `HessianProxyFactoryBean`, `BurlapProxyFactoryBean`, or `HttpInvokerProxyFactoryBean` to create a proxy for a remote service. It allows you to use the remote service as if it were a local bean. Again, it requires no additional programming at all.

How It Works

Expose a Hessian Service

We'll use the same weather service from the previous RMI recipe and expose it as a Hessian service with Spring. We'll create a simple web application using Spring MVC to deploy the service. First, let's create an Initializer class to bootstrap the web application and Spring application context.

```
package com.apress.springrecipes.weather.config;
```

```

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

public class Initializer implements WebApplicationInitializer {

    public void onStartup(ServletContext container)
        throws ServletException {
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.scan("com.apress.springrecipes.weather.config");

        ServletRegistration.Dynamic dispatcher =
            container.addServlet("dispatcher", new DispatcherServlet(context));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/*");
    }
}

```

The Initializer class defines a DispatcherServlet to map all URLs under the root path (*). Note the servlet instance is initialized with the Spring application context. The Spring application context scans the com.apress.springrecipes.weather.config package for @Configuration classes.

```

package com.apress.springrecipes.weather.config;

import com.apress.springrecipes.weather.WeatherService;
import com.apress.springrecipes.weather.WeatherServiceImpl;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.remoting.cauchy.HessianServiceExporter;

@Configuration
public class WeatherConfigHessianServer {

    @Bean
    public WeatherService weatherService() {
        WeatherService wService = new WeatherServiceImpl();
        return wService;
    }

    @Bean(name = "/weather")
    public HessianServiceExporter exporter() {
        HessianServiceExporter exporter = new HessianServiceExporter();
        exporter.setService(weatherService());
        exporter.setServiceInterface(WeatherService.class);
        return exporter;
    }
}

```

The scanning component allows Spring to inspect a Java Config class that instantiates the `weatherService` bean, which contains the operations that are to be exposed by the `HessianServiceExporter`. The `weatherService` bean in this case is identical to the one used in the previous RMI recipe. You can also consult the book's source code to get the pre-built application.

For a `HessianServiceExporter` instance, you have to configure a service object to export and its service interface. You can export any Spring bean as a Hessian service, and `HessianServiceExporter` creates a proxy to wrap this bean. When the proxy receives an invocation request, it invokes the corresponding method on that bean. By default, `BeanNameUrlHandlerMapping` is preconfigured for Spring MVC applications, so this means beans are mapped to URL patterns specified as bean names. The preceding configuration maps the URL pattern `/weather` to this exporter.

Next, you can deploy this web application to a web container (e.g., Apache Tomcat 7.0). By default, Tomcat listens on port 8080, so if you deploy your application to the `hessian` context path, you can access this service with the following URL:

```
http://localhost:8080/hessian/weather
```

Invoke a Hessian Service

By using Spring's remoting facilities, you can invoke a remote service just like a local bean. In a client application you can create a `HessianProxyFactoryBean` instance in a Java config class to create a proxy for the remote Hessian service. Then you can use this service as if it were a local bean.

```
@Bean
public HessianProxyFactoryBean weatherService() {
    HessianProxyFactoryBean factory = new HessianProxyFactoryBean();
    factory.setServiceUrl("http://localhost:8080/hessian/weather");
    factory.setServiceInterface(WeatherService.class);
    return factory;
}
```

For a `HessianProxyFactoryBean` instance you have to configure two properties. The `service URL` property specifies the URL for the target service. The `service interface` property is for this factory bean to create a local proxy for the remote service. The proxy will send the invocation requests to the remote service transparently.

Expose a Burlap Service

The configuration for exposing a Burlap service is similar to that for Hessian, except you should use `BurlapServiceExporter` instead.

```
@Bean(name = "/weather")
public BurlapServiceExporter exporter() {
    BurlapServiceExporter exporter = new BurlapServiceExporter();
    exporter.setService(weatherService());
    exporter.setServiceInterface(WeatherService.class);
    return exporter;
}
```

Invoke a Burlap Service

Invoking a Burlap service is very similar to Hessian. The only difference is that you should use BurlapProxyFactoryBean.

```
@Bean
public BurlapProxyFactoryBean weatherService() {
    BurlapProxyFactoryBean factory = new BurlapProxyFactoryBean();
    factory.setServiceUrl("http://localhost:8080/burlap/weather");
    factory.setServiceInterface(WeatherService.class);
    return factory;
}
```

Expose an HTTP Invoker Service

Similarly, the configuration for exposing a service using HTTP Invoker is similar to that of Hessian and Burlap, except you have to use HttpInvokerServiceExporter instead.

```
@Bean(name = "/weather")
public HttpInvokerServiceExporter exporter() {
    HttpInvokerServiceExporter exporter = new HttpInvokerServiceExporter();
    exporter.setService(weatherService());
    exporter.setServiceInterface(WeatherService.class);
    return exporter;
}
```

Invoke an HTTP Invoker Service

Invoking a service exposed by HTTP Invoker is also similar to Hessian and Burlap. This time, you have to use HttpInvokerProxyFactoryBean.

```
@Bean
public HttpInvokerProxyFactoryBean weatherService() {
    HttpInvokerProxyFactoryBean factory = new HttpInvokerProxyFactoryBean();
    factory.setServiceUrl("http://localhost:8080/httpinvoker/weather");
    factory.setServiceInterface(WeatherService.class);
    return factory;
}
```

14-9. Expose and invoke SOAP Web Services with JAX-WS

Problem

SOAP is an enterprise standard and cross-platform application communication technology. Most modern and mission critical software remoting tasks (e.g., banking services, inventory applications) typically use this standard. You want to be able to invoke third-party SOAP web services from your Java applications, as well as expose web services from your Java applications so third parties on different platforms can invoke them via SOAP.

Solution

Use JAX-WS @WebService and @WebMethod annotations, as well as Spring's SimpleJaxWsServiceExporter to allow access to bean business logic via SOAP. You can also leverage Apache CXF with Spring to expose SOAP services in a Java server like Tomcat. To access SOAP services you can use Apache CXF with Spring or leverage Spring's JaxWsPortProxyFactoryBean.

How It Works

JAX-WS 2.0 is the successor of JAX-RPC 1.1 — the Java API for XML-based Web services. So if you're going to use SOAP in the context of Java, JAX-WS is the most recent standard which enjoys support in both Java EE and the standard JDK.

Expose a Web Service Using the JAX-WS Endpoint Support in the JDK

You can rely on Java's JDK JAX-WS runtime support to expose JAX-WS services. This means you don't necessarily need to deploy JAX-WS services as part of a Java web application. By default, the JAX-WS support in the JDK is used if you have no other runtime. Let's implement the weather service application from previous recipes with JAX-WS using the JDK.

We need to annotate the weather service to indicate that it should be exposed to clients. The revised WeatherServiceImpl needs to be decorated with @WebService and @WebMethod annotations. Where the main class is decorated with @WebService and the method that will be exposed by the service is decorated with @WebMethod.

```
package com.apress.springrecipes.weather;

import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@WebService(serviceName = "weather", endpointInterface =
" com.apress.springrecipes.weather.WeatherService ")
public class WeatherServiceImpl implements WeatherService {

    @WebMethod(operationName = "getTemperatures")
    public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
        List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();

        for (Date date : dates) {
            temperatures.add(new TemperatureInfo(city, date, 5.0, 10.0, 8.0));
        }

        return temperatures;
    }
}
```

Note you don't need to provide any parameters to the annotations, like endpointInterface or serviceName, but we do here for to make the resulting SOAP contract more readable. Similarly, you don't need to provide an operationName on the @WebMethod annotation. This is generally good practice anyway, because it insulates clients of the SOAP endpoint from any refactoring you may do on the Java implementation.

Next, so Spring is able to detect beans with `@WebService` annotations we rely on Spring's `SimpleJaxWsServiceExporter`. This is an `@Bean` definition of this class in a Java Config class.

```
@Bean
public SimpleJaxWsServiceExporter jaxWsService() {
    SimpleJaxWsServiceExporter simpleJaxWs = new SimpleJaxWsServiceExporter();
    simpleJaxWs.setBaseAddress("http://localhost:8888/jaxws/");
    return simpleJaxWs;
}
```

Notice the bean definition calls `setBaseAddress` and sets it to `http://localhost:8888/jaxws/`. This is the endpoint for the application's JAX-WS service. Under this address — which is a standalone server spun by the JDK — is where all the beans defined with `@WebService` annotations will reside. So if there's a `@WebService` name called `weather` it will become accessible under `http://localhost:8888/jaxws/weather`.

If you launch a browser and inspect the results at `http://localhost:8888/jaxws/weather?wsdl`, you'll see the generated SOAP WSDL contract, as follows:

```
<!--
Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.4-b01.
-->
<!--
Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is JAX-WS RI 2.2.4-b01.
-->
<definitions xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/soap/" xmlns:tns="http://weather.springrecipes.apress.com/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns=""http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://weather.springrecipes.apress.com/" name="weather">
<types>
<xsd:schema>
<xsd:import namespace="http://weather.springrecipes.apress.com/" schemaLocation="http://localhost:8888/jaxws/weather?xsd=1"/>
</xsd:schema>
</types>
<message name="getTemperatures">
<part name="parameters" element="tns:getTemperatures"/>
</message>
<message name="getTemperaturesResponse">
<part name="parameters" element="tns:getTemperaturesResponse"/>
</message>
<portType name="WeatherService">
<operation name="getTemperatures">
<input wsam:Action="http://weather.springrecipes.apress.com/WeatherService/getTemperaturesRequest" message="tns:getTemperatures"/>
<output wsam:Action="http://weather.springrecipes.apress.com/WeatherService/getTemperaturesResponse" message="tns:getTemperaturesResponse"/>
</operation>
</portType>
<binding name="WeatherServiceImplPortBinding" type="tns:WeatherService">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
<operation name="getTemperatures">
```

```

<soap:operation soapAction="" />
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
</binding>
<service name="weather">
<port name="WeatherServiceImplPort" binding="tns:WeatherServiceImplPortBinding">
<soap:address location="http://localhost:8888/jaxws/weather" />
</port>
</service>
</definitions>

```

The SOAP WSDL contract is used by clients to access the service. If you inspect the generated WSDL you'll see it's pretty basic — describing the weather service method — but more importantly it's programming language neutral. This neutrality is the whole purpose of SOAP, to be able to access services across diverse platforms that can interpret SOAP.

Expose a Web Service Using CXF

Exposing a stand-alone SOAP endpoint using the `SimpleJaxWsServiceExporter` and the JAX-WS JDK support is simple. However, this solution ignores the fact that most Java applications in real-world environments operate on Java app runtimes, such as Tomcat. Tomcat by itself doesn't support JAX-WS, so we'll need to equip the application with a JAX-WS runtime.

There are many choices, and you're free to take your pick. Two popular choices are Axis2 and CXF, both of which are Apache projects. For our example, we'll embed CXF since it's robust, fairly well tested, and provides support for other important standards like JAX-RS, the API for REST-ful endpoints.

First let's take a look at the `Initializer` class to bootstrap an application under a Servlet 3.0 compliant server like Apache Tomcat 7.0.

```

package com.apress.springrecipes.weather.config;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.servlet.DispatcherServlet;
import org.springframework.web.context.ContextLoaderListener;

import org.springframework.web.context.support.XmlWebApplicationContext;

import org.apache.cxf.transport.servlet.CXFServlet;

import javax.servlet.ServletRegistration;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;

public class Initializer implements WebApplicationInitializer {
    public void onStartup(ServletContext container) throws ServletException {
        XmlWebApplicationContext context = new XmlWebApplicationContext();

```

```

        context.setConfigLocation("/WEB-INF/appContext.xml");

        container.addListener(new ContextLoaderListener(context));

        ServletRegistration.Dynamic cxf = container.addServlet("cxf", new CXFServlet());
        cxf.setLoadOnStartup(1);
        cxf.addMapping("/*");
    }
}

```

This Initializer class will look pretty much as all Spring MVC applications do. The only exception here is that we've configured a CXFServlet, which handles a lot of the heavy lifting required to expose our service. In the Spring MVC configuration file, we'll be using the Spring namespace support that CXF provides for configuring services.

The Spring context file is simple, most of it is boilerplate XML namespace and Spring context file imports. The only two salient stanzas are below, where we first configure the service itself as usual. Finally, we use the CXF jaxws:endpoint namespace to configure our endpoint.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxrs="http://cxfrs.apache.org/jaxrs"
       xmlns:simple="http://cxfrs.apache.org/simple"
       xmlns:soap="http://cxfrs.apache.org/bindings/soap"
       xmlns:jaxws="http://cxfrs.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://cxfrs.apache.org/jaxrs http://cxfrs.apache.org/schemas/jaxrs.xsd
                           http://cxfrs.apache.org/simple http://cxfrs.apache.org/schemas/simple.xsd
                           http://cxfrs.apache.org/bindings/soap
                           http://cxfrs.apache.org/schemas/configuration/soap.xsd
                           http://cxfrs.apache.org/jaxws http://cxfrs.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>
    <import resource="classpath:META-INF/cxf/cxf.xml"/>

    <jaxws:endpoint implementor="#weatherService" address="/weather">
        <jaxws:binding>
            <soap:soapBinding version="1.2"/>
        </jaxws:binding>
    </jaxws:endpoint>

</beans>

```

We tell the jaxws:endpoint factory to use the weatherService Spring bean as the implementation. We tell it at what address to publish the service using the address element. In this case, because the Initializer mounts the CXF servlet under the root directory (/), the CXF weatherService endpoint becomes accessible under /weather.

Note the Java code in `weatherServiceImpl` stays the same as before, with the `@WebService` and `@WebMethod` annotations in place. Launch the application and your web container, and then bring up the application in your browser. In the book's source code, the application is built in a WAR called `jaxws.war` and given CXF is deployed at the root context (/), the SOAP WSDL contract is available at `http://localhost:8080/weather`. If you bring up the page at `http://localhost:8080/`, you'll see a directory of the available services and their operations.

Click the link for the service's WSDL—or simply append ?wsdl to the service endpoint—to see the WSDL for the service. The WSDL contract is pretty similar to the ones described in the previous section using JAX-WS JDK support. The only difference is the WSDL contract is generated with the help of CXF.

Invoke a Web Service Using Spring's JaxWsPortProxyFactoryBean

Spring provides the functionality to access a SOAP WSDL contract and communicate with the underlying services as if it were a regular Spring bean. This functionality is provided by `JaxWsPortProxyFactoryBean`. The following is a sample bean definition to access the SOAP weather service with `JaxWsPortProxyFactoryBean`.

```
@Bean
public JaxWsPortProxyFactoryBean weatherService() throws MalformedURLException {
    JaxWsPortProxyFactoryBean weatherService = new JaxWsPortProxyFactoryBean();
    weatherService.setServiceInterface(WeatherService.class);
    weatherService.setWsdlDocumentUrl(new URL("http://localhost:8888/jaxws/weather?WSDL"));
    weatherService.setNamespaceUri("http://weather.springrecipes.apress.com/");
    weatherService.setServiceName("weather");
    weatherService.setPortName("WeatherServiceImplPort");
    return weatherService;
}
```

The bean instance is given the `weatherService` name. It's through this reference that you'll be able to invoke the underlying SOAP service methods, as if they were running locally (e.g., `weatherService.getTemperatures(city, dates)`). The `JaxWsPortProxyFactoryBean` requires several properties which are described next.

The `serviceInterface` property defines the service interface for the SOAP call. In the case of the weather service, you can use the server-side implementation code which is the same. In case you're accessing a SOAP service for which you don't have the server-side code, you can always create this Java interface parting from the WSDL contract with a tool like `java2wsdl`. Note the `serviceInterface` used by the client side needs to use the same JAX-WS annotation used by the server-side implementation (i.e., `@WebService`).

The `wsdlDocumentUrl` property represents the location of the WSDL contract. In this case, it's pointing toward the CXF SOAP endpoint from this recipe, but you can equally define this property to access the JAX-WS JDK endpoint from this recipe or any WSDL contract for that matter.

The `namesapceUrl`, `serviceName`, and `portName` are properties that pertain to the WSDL contract itself. Since there can be various namespaces, services, and ports in a WSDL contract, you need to tell Spring which values to use for the purpose of accessing the service. The values presented here can easily be verified by doing a manual inspection to the weather WSDL contract.

Invoke a Web Service Using CXF

Let's now use CXF to define a web service client. Our client is the same as in previous recipes, and there is no special Java configuration or coding to be done. We simply need the interface of the service on the classpath. Once that's done, you can use CXF's namespace support to create a client.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/context"/>
```

```

http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.2.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

<import resource="classpath:META-INF/cxf/cxf.xml"/>
<import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>
<jaxws:client serviceClass="com.apress.springrecipes.weather.WeatherService"
    address="http://localhost:8080/weather" id="weatherService"/>
</beans>

```

We use the `jaxws:client` namespace support to define to which interface the proxy should be bound, and the endpoint of the service itself. That is all that's required. Our examples from previous recipes works otherwise unchanged: here we inject the client into the `WeatherServiceClient` and invoke it using the `weatherService` reference (e.g., `weatherService.getTemperatures(city, dates)`).

14-10. Introduction to contract first SOAP Web Services

Problem

You want to develop a contract first SOAP Web Service instead of a code first SOAP Web service as you did in the previous recipe.

Solution

There are two ways to develop SOAP web services. One is called 'code first' which means you start with a Java class and then build out toward a WSDL contract. The other method is called 'contract first' which means you start with an XML data contract — something simpler than WSDL — and build in toward a Java class to implement the service.

To create a data contract for a 'contract first' SOAP web service, you'll need an XSD file or XML Schema file that describes the operations and data supported by the service. The requirement for an XSD file is because 'under the hood' the communication between a SOAP service client and server takes place as XML defined in an XSD file.

However, because an XSD file can be difficult to write correctly, it's preferable to start by creating sample XML messages and then generating the XSD file from them. Then with the XSD file, you can leverage something like Spring-WS to build the SOAP web service parting from the XSD file.

How It Works

Create Sample XML Messages

Let's do the same weather service presented in previous recipes, but this time using the SOAP 'contract first' approach. So you're asked to write a SOAP service that is able to communicate weather information based on a city and date, returning the minimum, maximum, and average temperatures.

Instead of going in head first and writing code to support the previous functionality as you did in the previous recipe, let's describe the temperature of a particular city and date using a 'contract first' approach with an XML message like the following:

```

<TemperatureInfo city="Houston" date="2014-12-01">
    <min>5.0</min>
    <max>10.0</max>
    <average>8.0</average>
</TemperatureInfo>

```

This is the first step toward having a data contract in a SOAP ‘contract first’ way for the weather service. Now let’s define some operations. You want to allow clients to query the temperatures of a particular city for multiple dates. Each request consists of a city element and multiple date elements. We’ll also specify the namespace for this request to avoid naming conflicts with other XML documents. Let’s create this XML message and save it into a file called `request.xml`.

```
<GetTemperaturesRequest
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <city>Houston</city>
  <date>2014-12-01</date>
  <date>2014-12-08</date>
  <date>2014-12-15</date>
</GetTemperaturesRequest>
```

The response for a request of the previous type would consist of multiple `TemperatureInfo` elements, each of which represents the temperature of a particular city and date, in accordance with the requested dates. Let’s create this XML message and save it to a file called `response.xml`.

```
<GetTemperaturesResponse
  xmlns="http://springrecipes.apress.com/weather/schemas">
  <TemperatureInfo city="Houston" date="2014-12-01">
    <min>5.0</min>
    <max>10.0</max>
    <average>8.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-08">
    <min>4.0</min>
    <max>13.0</max>
    <average>7.0</average>
  </TemperatureInfo>
  <TemperatureInfo city="Houston" date="2007-12-15">
    <min>10.0</min>
    <max>18.0</max>
    <average>15.0</average>
  </TemperatureInfo>
</GetTemperaturesResponse>
```

Generate an XSD File from sample XML Messages

Now, you can generate the XSD file from the preceding sample XML messages. Most XML tools and enterprise Java IDEs can generate an XSD file from a couple of XML files. Here, we’ll use Apache XMLBeans (<http://xmlbeans.apache.org/>) to generate the XSD file.

Note You can download Apache XMLBeans (e.g., v2.6.0) from the Apache XMLBeans web site and extract it to a directory of your choice to complete the installation.

Apache XMLBeans provides a tool called `inst2xsd` to generate XSD files from XML files. The simplest is called Russian doll design, which generates local elements and local types for the target XSD file. Because there's no enumeration type used in your XML messages, you can disable the enumeration generation feature. You can execute the following command to generate the XSD file from the previous XML files:

```
inst2xsd -design rd -enumerations never request.xml response.xml
```

The generated XSD file will have the default name `schema0.xsd`, located in the same directory. Let's rename it to `temperature.xsd`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    targetNamespace="http://springrecipes.apress.com/weather/schemas"
    xmlns:xss="http://www.w3.org/2001/XMLSchema">

    <xss:element name="GetTemperaturesRequest">
        <xss:complexType>
            <xss:sequence>
                <xss:element type="xss:string" name="city" />
                <xss:element type="xss:date" name="date"
                    maxOccurs="unbounded" minOccurs="0" />
            </xss:sequence>
        </xss:complexType>
    </xss:element>

    <xss:element name="GetTemperaturesResponse">
        <xss:complexType>
            <xss:sequence>
                <xss:element name="TemperatureInfo"
                    maxOccurs="unbounded" minOccurs="0">
                    <xss:complexType>
                        <xss:sequence>
                            <xss:element type="xss:float" name="min" />
                            <xss:element type="xss:float" name="max" />
                            <xss:element type="xss:float" name="average" />
                        </xss:sequence>
                        <xss:attribute type="xss:string" name="city"
                            use="optional" />
                        <xss:attribute type="xss:date" name="date"
                            use="optional" />
                    </xss:complexType>
                </xss:element>
            </xss:sequence>
        </xss:complexType>
    </xss:element>
</xss:schema>
```

Optimizing the Generated XSD File

As you can see, the generated XSD file allows clients to query temperatures for unlimited dates. If you want to add a constraint on the maximum and minimum query dates, you can modify the maxOccurs and minOccurs attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema attributeFormDefault="unqualified"
    elementFormDefault="qualified"
    targetNamespace="http://springrecipes.apress.com/weather/schemas"
    xmlns:xss="http://www.w3.org/2001/XMLSchema

```

Previewing the Generated WSDL File

As you will learn shortly and in full detail, Spring-WS is equipped to automatically generate a WSDL contract parting from an XSD file. The following snippet illustrates the Spring bean configuration for this purpose — we'll add context on how to use this snippet in the next recipe which describes how to build SOAP web services with Spring-WS.

```
<swws:dynamic-wsdl id="temperature" portType="Weather" locationUri="/">
    <swws:xsd location="/WEB-INF/temperature.xsd"/>
</swws:dynamic-wsdl>
```

Here, we'll preview the generated WSDL file to better understand the service contract. For simplicity's sake, the less important parts are omitted.

```
<?xml version="1.0" encoding="UTF-8" ?>
<wsdl:definitions ...
    targetNamespace="http://springrecipes.apress.com/weather/schemas">
    <wsdl:types>
        <!-- Copied from the XSD file -->
        ...
    </wsdl:types>
    <wsdl:message name="GetTemperaturesResponse">
        <wsdl:part element="schema:GetTemperaturesResponse"
            name="GetTemperaturesResponse">
        </wsdl:part>
    </wsdl:message>
    <wsdl:message name="GetTemperaturesRequest">
        <wsdl:part element="schema:GetTemperaturesRequest"
            name="GetTemperaturesRequest">
        </wsdl:part>
    </wsdl:message>
    <wsdl:portType name="Weather">
        <wsdl:operation name="GetTemperatures">
            <wsdl:input message="schema:GetTemperaturesRequest"
                name="GetTemperaturesRequest">
            </wsdl:input>
            <wsdl:output message="schema:GetTemperaturesResponse"
                name="GetTemperaturesResponse">
            </wsdl:output>
        </wsdl:operation>
    </wsdl:portType>
    ...
    <wsdl:service name="WeatherService">
        <wsdl:port binding="schema:WeatherBinding" name="WeatherPort">
            <soap:address
                location="http://localhost:8080/weather/services" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

In the Weather port type, a GetTemperatures operation is defined whose name is derived from the prefix of the input and output messages (i.e., <GetTemperaturesRequest> and <GetTemperaturesResponse>). The definitions of these two elements are included in the <wsdl:types> part, as defined in the data contract.

Now with the WSDL contract in hand, you can generate the necessary Java interfaces and then write the backing code for each of the operations that started out as XML messages. This full technique is explored in the next recipe which uses Spring-WS for the process.

14-11. Expose and invoke SOAP Web Services with Spring-WS

Problem

You've have an XSD file to develop a 'contract first' SOAP web service and don't know how or what to use to implement the 'contract first' SOAP service.

SPRING-WS AND JAX-WS 'CONTRACT FIRST' SUPPORT

Spring-WS was designed from the outset to support 'contract first' SOAP web services. However, this does not mean Spring-WS is the only way to create SOAP web services in Java. JAX-WS implementations like CXF & Axis also support this technique.

Nevertheless, Spring-WS is the more mature and natural approach to do 'contract first' SOAP web services in the context of Spring applications. Describing other 'contract first' SOAP Java techniques, would lead us outside the scope of the Spring framework.

Solution

Spring-WS provides a set of facilities to develop 'contract first' SOAP web services. The essential tasks for building a Spring-WS web service include the following:

- Set up and configure a Spring MVC application for Spring-WS
- Map web service requests to endpoints
- Create service endpoints to handle the request messages and return the response messages
- Publish the WSDL file for the web service

How It Works

Set Up a Spring-WS Application

To implement a web service using Spring-WS, let's first create a web application initializer class to bootstrap a web application with a SOAP web service. You need to configure the `MessageDispatcherServlet` servlet which is part of Spring-WS. This servlet specializes in dispatching web service messages to appropriate endpoints and detecting the framework facilities of Spring-WS.

```
package com.apress.springrecipes.weather.config;

import org.springframework.ws.transport.http.support.
AbstractAnnotationConfigMessageDispatcherServletInitializer;

public class Initializer extends AbstractAnnotationConfigMessageDispatcherServletInitializer {

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

```

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

}

```

To make configuration easier there is the `AbstractAnnotationConfigMessageDispatcher` `ServletInitializer` base class we can extend. We need to supply it with the configuration classes which make up the `rootConfig` and the `servletConfig`, the first can be null the latter is required.

The configuration above will bootstrap a `MessageDispatcherServlet` using the `SpringWsConfiguration` class and register it for the `/services/*` and `*.wsdl` urls.

```

package com.apress.springrecipes.weather.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;
import org.springframework.ws.config.annotation.EnableWs;
import org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition;
import org.springframework.xml.xsd.SimpleXsdSchema;
import org.springframework.xml.xsd.XsdSchema;

@Configuration
@EnableWs
@ComponentScan("com.apress.springrecipes.weather")
public class SpringWsConfiguration {

    ...
}

```

The `SpringWsConfiguration` class is annotated with `@EnableWs` that registers necessary beans to make the `MessageDispatcherServlet` work. We also have a `@ComponentScan` that scans for our `@Service` and `@Endpoint` bean.

Create Service Endpoints

Spring-WS supports annotating an arbitrary class as a service endpoint by the `@Endpoint` annotation so it becomes accessible as a service. Besides the `@Endpoint` annotation, you also need to annotate handler methods with the `@PayloadRoot` annotation to map service requests. And each handler method also relies on the `@ResponsePayload` and `@RequestPayload` annotations to handle the incoming and outgoing service data.

```

package com.apress.springrecipes.weather;

import org.springframework.ws.server.endpoint.annotation.Endpoint;
import org.springframework.ws.server.endpoint.annotation.PayloadRoot;
import org.springframework.ws.server.endpoint.annotation.ResponsePayload;
import org.springframework.ws.server.endpoint.annotation.RequestPayload;
import org.springframework.beans.factory.annotation.Autowired;

```

```

import org.dom4j.DocumentHelper;
import org.dom4j.Document;
import org.dom4j.Element;
import org.dom4j.XPath;
import org.dom4j.xpath.DefaultXPath;
...
 
@Endpoint
public class TemperatureEndpoint {
 
    private static final String namespaceUri = "http://springrecipes.apress.com/weather/schemas";
    private XPath cityPath;
    private XPath datePath;
    private DateFormat dateFormat;
    @Autowired
    private WeatherService weatherService;
 
    public TemperatureEndpoint() {
        // Create the XPath objects, including the namespace
        Map<String, String> namespaceUris = new HashMap<String, String>();
        namespaceUris.put("weather", namespaceUri);
        cityPath = new DefaultXPath("/weather:GetTemperaturesRequest/weather:city");
        cityPath.setNamespaceURIs(namespaceUris);
        datePath = new DefaultXPath("/weather:GetTemperaturesRequest/weather:date");
        datePath.setNamespaceURIs(namespaceUris);
        dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    }
 
    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }
 
    @PayloadRoot(localPart="GetTemperaturesRequest",namespace=namespaceUri)
    @ResponsePayload
    public Element getTemperature(@RequestPayload Element requestElement) throws Exception {
        // Extract the service parameters from the request message
        String city = cityPath.valueOf(requestElement);
        List<Date> dates = new ArrayList<Date>();
        for (Object node : datePath.selectNodes(requestElement)) {
            Element element = (Element) node;
            dates.add(dateFormat.parse(element.getText()));
        }
 
        // Invoke the back-end service to handle the request
        List<TemperatureInfo> temperatures =
            weatherService.getTemperatures(city, dates);
 
        // Build the response message from the result of back-end service
        Document responseDocument = DocumentHelper.createDocument();
        Element responseElement = responseDocument.addElement(
            "GetTemperaturesResponse", namespaceUri);
    }
}

```

```

        for (TemperatureInfo temperature : temperatures) {
            Element temperatureElement = responseElement.addElement(
                "TemperatureInfo");
            temperatureElement.addAttribute("city", temperature.getCity());
            temperatureElement.addAttribute(
                "date", dateFormat.format(temperature.getDate()));
            temperatureElement.addElement("min").setText(
                Double.toString(temperature.getMin()));
            temperatureElement.addElement("max").setText(
                Double.toString(temperature.getMax()));
            temperatureElement.addElement("average").setText(
                Double.toString(temperature.getAverage()));
        }
        return responseElement;
    }
}

```

In the `@PayloadRoot` annotation, you specify the local name (`getTemperaturesRequest`) and namespace (<http://springrecipes.apress.com/weather/schemas>) of the payload root element to be handled. Next, the method is decorated with the `@ResponsePayload` indicating the method's return value is the service response data. In addition, the method's input parameter is decorated with the `@RequestPayload` annotation to indicate it's the service input value.

Then inside the handler method, you first extract the service parameters from the request message. Here, you use XPath to help locate the elements. The XPath objects are created in the constructor so that they can be reused for subsequent request handling. Note that you must also include the namespace in the XPath expressions, or else they will not be able to locate the elements correctly.

After extracting the service parameters, you invoke the back-end service to handle the request. Because this endpoint is configured in the Spring IoC container, it can easily refer to other beans through dependency injection.

Finally, you build the response message from the back-end service's result. In this case we used the dom4j library that provides a rich set of APIs for you to build an XML message. But you can use any other XML processing API or Java parser you wish (e.g., DOM).

Because you already defined a `@ComponentScan` in the `SpringWsConfiguration` class, Spring automatically picks up all the Spring-WS annotations and deploys the endpoint to the servlet.

Publish the WSDL File

The last step to complete the SOAP web service is to publish the WSDL file. In Spring-WS, it's not necessary for you to write the WSDL file manually, you only need to add a bean to the `SpringWsConfiguration` class.

```

@Bean
public DefaultWsdl11Definition temperature() {
    DefaultWsdl11Definition temperature = new DefaultWsdl11Definition();
    temperature.setPortTypeName("Weather");
    temperature.setLocationUri("/");
    temperature.setSchema(temperatureSchema());
    return temperature;
}

```

```
@Bean
public XsdSchema temperatureSchema() {
    return new SimpleXsdSchema(new ClassPathResource("/META-INF/xsd/temperature.xsd"));
}
```

The DefaultWsdl11Definition class requires that you specify two properties: a portTypeName for the service, as well as a locationUri on which to deploy the final WSDL. In addition, it also requires that you specify the location of the XSD file from which to create the WSDL — see the previous recipe for details on how to create an XSD file. In this case, the XSD file will be located inside the application's META-INF directory.

Because you have defined <GetTemperaturesRequest> and <GetTemperaturesResponse> in your XSD file, and you have specified the port type name as Weather, the WSDL builder will generate the following WSDL port type and operation for you. The following snippet is taken from the generated WSDL file:

```
<wsdl:portType name="Weather">
    <wsdl:operation name="GetTemperatures">
        <wsdl:input message="schema:GetTemperaturesRequest"
                    name="GetTemperaturesRequest" />
        <wsdl:output message="schema:GetTemperaturesResponse"
                    name="GetTemperaturesResponse" />
    </wsdl:operation>
</wsdl:portType>
```

Finally, you can access this WSDL file by joining its definition's bean name and the .wsdl suffix. Assuming the web application is packaged in a WAR file named springws, then the service is deployed in <http://localhost:8080/springws> — because the Spring-WS servlet in the initializer is deployed on the /services directory — and the WSDL file's URL would be <http://localhost:8080/springws/services/weather/temperature.wsdl>, given that the bean name of the WSDL definition is temperature.

Invoke SOAP Web Services with Spring-WS

Now, let's create a Spring-WS client to invoke the weather service according to the contract it publishes. You can create a Spring-WS client by parsing the request and response XML messages. As an example, we will use dom4j to implement it. But you are free to choose any other XML parsing APIs for it.

To shield the client from the low-level invocation details, we'll create a local proxy to call the SOAP web service. This proxy also implements the WeatherService interface, and it translates local method calls into remote SOAP web service calls.

```
package com.apress.springrecipes.weather;
...
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.DocumentResult;
import org.dom4j.io.DocumentSource;
import org.springframework.ws.client.core.WebServiceTemplate;

public class WeatherServiceProxy implements WeatherService {

    private static final String namespaceUri =
        "http://springrecipes.apress.com/weather/schemas";
```

```

private DateFormat dateFormat;
private WebServiceTemplate webServiceTemplate;

public WeatherServiceProxy() throws Exception {
    dateFormat = new SimpleDateFormat("yyyy-MM-dd");
}

public void setWebServiceTemplate(WebServiceTemplate webServiceTemplate) {
    this.webServiceTemplate = webServiceTemplate;
}

public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {

    // Build the request document from the method arguments
    Document requestDocument = DocumentHelper.createDocument();
    Element requestElement = requestDocument.addElement(
        "GetTemperaturesRequest", namespaceUri);
    requestElement.addElement("city").setText(city);
    for (Date date : dates) {
        requestElement.addElement("date").setText(dateFormat.format(date));
    }

    // Invoke the remote web service
    DocumentSource source = new DocumentSource(requestDocument);
    DocumentResult result = new DocumentResult();
    webServiceTemplate.sendSourceAndReceiveToResult(source, result);

    // Extract the result from the response document
    Document responseDocument = result.getDocument();
    Element responseElement = responseDocument.getRootElement();
    List<TemperatureInfo> temperatures = new ArrayList<TemperatureInfo>();
    for (Object node : responseElement.elements("TemperatureInfo")) {
        Element element = (Element) node;
        try {
            Date date = dateFormat.parse(element.attributeValue("date"));
            double min = Double.parseDouble(element.elementText("min"));
            double max = Double.parseDouble(element.elementText("max"));
            double average = Double.parseDouble(
                element.elementText("average"));
            temperatures.add(
                new TemperatureInfo(city, date, min, max, average));
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
    return temperatures;
}
}

```

In the `getTemperatures()` method, you first build the request message using the dom4j API. `WebServiceTemplate` provides a `sendSourceAndReceiveToResult()` method that accepts a `java.xml.transform.Source` and a `java.xml.transform.Result` object as arguments. You have to build a `dom4j.DocumentSource` object to wrap your request document and create a new `dom4j.DocumentResult` object for the method to write the response document to it. Finally, you get the response message and extract the results from it.

With the service proxy written, we can declare it in a configuration class and later call it using a standalone class.

```
package com.apress.springrecipes.weather.config;

import com.apress.springrecipes.weather.WeatherServiceClient;
import com.apress.springrecipes.weather.WeatherServiceProxy;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.ws.client.core.WebServiceTemplate;

@Configuration
public class SpringWsClientConfiguration {

    @Bean
    public WeatherServiceClient weatherServiceClient() throws Exception {
        WeatherServiceClient weatherServiceClient = new WeatherServiceClient();
        weatherServiceClient.setWeatherService(weatherServiceProxy());
        return weatherServiceClient;
    }

    @Bean
    public WeatherServiceProxy weatherServiceProxy() throws Exception {
        WeatherServiceProxy weatherServiceProxy = new WeatherServiceProxy();
        weatherServiceProxy.setWebServiceTemplate(webServiceTemplate());
        return weatherServiceProxy;
    }

    @Bean
    public WebServiceTemplate webServiceTemplate() {
        WebServiceTemplate webServiceTemplate = new WebServiceTemplate();
        webServiceTemplate.setDefaultUri("http://localhost:8080/springws/services");
        return webServiceTemplate;
    }
}
```

Note the `webServiceTemplate` has its `defaultUri` value set to the endpoint defined for the Spring-WS endpoint in the previous sections. Once the configuration is loaded by an application, you can call the SOAP service using the following class.

```
package com.apress.springrecipes.weather;

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

```

public class SpringWSInvokerClient {

    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.weather.config");

        WeatherServiceClient client = context.getBean(WeatherServiceClient.class);
        TemperatureInfo temperature = client.getTodayTemperature("Houston");
        System.out.println("Min temperature : " + temperature.getMin());
        System.out.println("Max temperature : " + temperature.getMax());
        System.out.println("Average temperature : " + temperature.getAverage());
    }
}

```

14-12. Develop SOAP Web Services with Spring-WS and XML Marshalling

Problem

To develop web services with the contract-first approach, you have to process request and response XML messages. If you parse the XML messages with XML parsing APIs directly, you'll have to deal with the XML elements one by one with low-level APIs, which is a cumbersome and inefficient task.

Solution

Spring-WS supports using XML marshalling technology to marshal and unmarshal objects to and from XML documents. In this way, you can deal with object properties instead of XML elements. This technology is also known as object/XML mapping (OXM), because you are actually mapping objects to and from XML documents.

To implement endpoints with an XML marshalling technology, you can configure an XML marshaller for it. Table 14-2 lists the marshallers provided by Spring for different XML marshalling APIs.

Table 14-2. Marshallers for Different XML Marshalling APIs

API	Marshaller
JAXB 2.0	org.springframework.oxm.jaxb.Jaxb2Marshaller
Castor	org.springframework.oxm.castor.CastorMarshaller
XMLBeans	org.springframework.oxm.xmlbeans.XmlBeansMarshaller
JiBX	org.springframework.oxm.jibx.JibxMarshaller
XStream	org.springframework.oxm.xstream.XStreamMarshaller

Similarly, Spring WS clients can also use this same marshalling and unmarshalling technique to simplify XML data processing.

How It Works

Create Service Endpoints with XML Marshalling

Spring-WS supports various XML marshalling APIs, including JAXB 2.0, Castor, XMLBeans, JiBX, and XStream. As an example, we'll create a service endpoint using Castor (www.castor.org) as the marshaller. Using other XML marshalling APIs is very similar.

The first step in using XML marshalling is creating the object model according to the XML message formats. This model can usually be generated by the marshalling API. For some marshalling APIs, the object model must be generated by them so that they can insert marshalling-specific information. Because Castor supports marshalling between XML messages and arbitrary Java objects, you can start creating the following classes by yourself.

```
package com.apress.springrecipes.weather;
...
public class GetTemperaturesRequest {

    private String city;
    private List<Date> dates;

    // Constructors, Getters and Setters
    ...
}

package com.apress.springrecipes.weather;
...
public class GetTemperaturesResponse {

    private List<TemperatureInfo> temperatures;

    // Constructors, Getters and Setters
    ...
}
```

With the object model created, you can easily integrate marshalling on any endpoint. Let's apply this technique to the endpoint presented in the previous recipe.

```
package com.apress.springrecipes.weather;
...
@Endpoint
public class TemperatureMarshallingEndpoint {

    private static final String namespaceUri = "http://springrecipes.apress.com/weather/schemas";

    @Autowired
    private WeatherService weatherService;

    public void setWeatherService(WeatherService weatherService) {
        this.weatherService = weatherService;
    }
}
```

```

@PayloadRoot(localPart="GetTemperaturesRequest",namespace=namespaceUri)
public GetTemperaturesResponse getTemperature(GetTemperaturesRequest request) {
    List<TemperatureInfo> temperatures = weatherService.getTemperatures(request.getCity(),
        request.getDates());
    return new GetTemperaturesResponse(temperatures);
}
}

```

Notice that all you have to do in this new method endpoint is handle the request object and return the response object. Then, it will be marshalled to the response XML message.

In addition to this endpoint modification, a marshalling endpoint also requires that both the marshaller and unmarshaller properties be set. Usually, you can specify a single marshaller for both properties. For Castor, you declare a `CastorMarshaller` bean as the marshaller. Next to the marshaller we also need to register a `MethodArgumentResolver` and `MethodReturnValueHandler` to actually handle the marshalling of the method argument and return type. For this we extend the `WsConfigurerAdapter` and override the `addArgumentResolvers` and `addReturnValueHandlers` methods and add the `MarshallingPayloadMethodProcessor` to both the lists.

```

@Configuration
@EnableWs
@ComponentScan("com.apress.springrecipes.weather")
public class SpringWsConfiguration extends WsConfigurerAdapter {

    @Bean
    public MarshallingPayloadMethodProcessor marshallingPayloadMethodProcessor() {
        return new MarshallingPayloadMethodProcessor(marshaller());
    }

    @Bean
    public Marshaller marshaller() {
        CastorMarshaller marshaller = new CastorMarshaller();
        marshaller.setMappingLocation(new ClassPathResource("/mapping.xml"));
        return marshaller;
    }

    @Override
    public void addArgumentResolvers(List<MethodArgumentResolver> argumentResolvers) {
        argumentResolvers.add(marshallingPayloadMethodProcessor());
    }

    @Override
    public void addReturnValueHandlers(List<MethodReturnValueHandler> returnValueHandlers) {
        returnValueHandlers.add(marshallingPayloadMethodProcessor());
    }
}

```

Note that Castor requires a mapping configuration file to know how to map objects to and from XML documents. You can create this file in the classpath root and specify it in the `mappingLocation` property (e.g., `mapping.xml`). The following Castor mapping file defines the mappings for the `GetTemperaturesRequest`, `GetTemperaturesResponse`, and `TemperatureInfo` classes:

```
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Mapping DTD Version 1.0//EN"
 "http://castor.org/mapping.dtd">

<mapping>
  <class name="com.apress.springrecipes.weather.GetTemperaturesRequest">
    <map-to xml="GetTemperaturesRequest"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="city" type="string">
      <bind-xml name="city" node="element" />
    </field>
    <field name="dates" collection="arraylist" type="string"
      handler="com.apress.springrecipes.weather.DateFieldHandler">
      <bind-xml name="date" node="element" />
    </field>
  </class>

  <class name="com.apress.springrecipes.weather.
GetTemperaturesResponse">
    <map-to xml="GetTemperaturesResponse"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="temperatures" collection="arraylist"
      type="com.apress.springrecipes.weather.TemperatureInfo">
      <bind-xml name="TemperatureInfo" node="element" />
    </field>
  </class>

  <class name="com.apress.springrecipes.weather.TemperatureInfo">
    <map-to xml="TemperatureInfo"
      ns-uri="http://springrecipes.apress.com/weather/schemas" />
    <field name="city" type="string">
      <bind-xml name="city" node="attribute" />
    </field>
    <field name="date" type="string"
      handler="com.apress.springrecipes.weather.DateFieldHandler">
      <bind-xml name="date" node="attribute" />
    </field>
    <field name="min" type="double">
      <bind-xml name="min" node="element" />
    </field>
    <field name="max" type="double">
      <bind-xml name="max" node="element" />
    </field>
    <field name="average" type="double">
      <bind-xml name="average" node="element" />
    </field>
  </class>
</mapping>
```

In addition, in all the date fields you have to specify a handler to convert the dates with a particular date format. This handler is shown next:

```
package com.apress.springrecipes.weather;
...
import org.exolab.castor.mapping.GeneralizedFieldHandler;

public class DateFieldHandler extends GeneralizedFieldHandler {

    private DateFormat format = new SimpleDateFormat("yyyy-MM-dd");

    public Object convertUponGet(Object value) {
        return format.format((Date) value);
    }

    public Object convertUponSet(Object value) {
        try {
            return format.parse((String) value);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    public Class getField Type() {
        return Date.class;
    }
}
```

Invoking Web Services with XML Marshalling

A Spring-WS client can also marshal and unmarshal the request and response objects to and from XML messages. As an example, we will create a client using Castor as the marshaller so that you can reuse the object models `GetTemperaturesRequest`, `GetTemperaturesResponse`, and `TemperatureInfo`, and the mapping configuration file, `mapping.xml`, from the service endpoint.

Let's implement the service proxy with XML marshalling. `WebServiceTemplate` provides a `marshalSendAndReceive()` method that accepts a request object as the method argument, which will be marshalled to the request message. This method has to return a response object that will be unmarshalled from the response message.

```
package com.apress.springrecipes.weather;
...
public class WeatherServiceProxy implements WeatherService {

    ...
    public void setWebServiceTemplate(WebServiceTemplate webServiceTemplate) {
        this.webServiceTemplate = webServiceTemplate;
    }

    public WebServiceTemplate getWebServiceTemplate() {
        return webServiceTemplate;
    }
}
```

```

public List<TemperatureInfo> getTemperatures(String city, List<Date> dates) {
    GetTemperaturesRequest request = new GetTemperaturesRequest(city, dates);
    GetTemperaturesResponse response = (GetTemperaturesResponse)
        getWebServiceTemplate().marshalSendAndReceive(request);
    return response.getTemperatures();
}
}

```

When you are using XML marshalling, `WebServiceTemplate` requires both the `marshaller` and `unmarshaller` properties to be set. You can also set them to `WebServiceGatewaySupport` if you extend this class to have `WebServiceTemplate` auto-created. Usually, you can specify a single marshaller for both properties. For Castor, you declare a `CastorMarshaller` bean as the marshaller.

```

@Configuration
public class SpringWsClientConfiguration {
    ...
    @Bean
    public WebServiceTemplate webServiceTemplate() {
        WebServiceTemplate webServiceTemplate = new WebServiceTemplate(marshaller());
        webServiceTemplate.setDefaultUri("http://localhost:8080/springws/services");
        return webServiceTemplate;
    }
    @Bean
    public Marshaller marshaller() {
        CastorMarshaller marshaller = new CastorMarshaller();
        marshaller.setMappingLocation(new ClassPathResource("/mapping.xml"));
        return marshaller;
    }
}

```

Summary

This chapter discussed JMX and a few of the surrounding specifications. You learned how to export Spring beans as JMX MBeans and how to use those MBeans from a client, both remotely and locally by using Spring's proxies. You published and listened to notification events on a JMX server from Spring.

You also learned how to do e-mail tasks with the aid of Spring, including how to create e-mail templates and send emails with attachments (MIME messages). You also learned how to schedule tasks using the Quartz Scheduler, as well as Spring's task namespace.

This chapter also introduced you to the various remoting technologies supported by Spring. You learned how to both publish and consume an RMI service. We also discussed how to build services that operate through HTTP, using three different techniques/protocols: Burlap, Hessian, and HTTPInovker. Next, we discussed SOAP web services and how to use JAX-WS, as well as the Apache CXF framework to build and consume these types of services. Finally, we discussed contract first SOAP web services and how to leverage Spring-WS to create and consume these types of services.

CHAPTER 15

Spring Messaging

In this chapter, you will learn about Spring's support for Java Message Service (JMS). JMS defines a set of standard APIs for message-oriented communication (using message-oriented middleware, a.k.a. MOM).

With JMS, different applications can communicate in a loosely coupled way compared with other remoting technologies such as RMI. However, when using the JMS API to send and receive messages, you have to manage the JMS resources yourself and handle the JMS API's exceptions, which results in many lines of JMS-specific code. Spring simplifies JMS's usage with a template-based approach. Moreover, Spring enables beans declared in its IoC container to listen for JMS messages and react to them.

Messaging is a very powerful technique for scaling applications. It allows work that would otherwise overwhelm a service to be queued up. It also encourages a decoupled architecture. A component, for example, might only consume messages with a single `java.util.Map`-based key/value pair. This loose contract makes it a viable hub of communication for multiple, disparate systems.

In this chapter, we'll refer quite a bit to "topics" and "queues". Messaging solutions are designed to solve two types of architecture requirements: messaging from one point in an application to another known point, and messaging from one point in an application to many other unknown points. These patterns are the middleware equivalents of telling somebody something face to face and saying something over a loud speaker to a room of people, respectively.

If you want messages sent on a message queue to be broadcast to an unknown set of clients who are "listening" for the message (as in the loud speaker analogy), send the message on a "topic". If you want the message sent to a single, known client, then you send it over a "queue".

By the end of this chapter, you will be able to create and access message-based middleware using Spring and JMS. This chapter will also provide you with a working knowledge of messaging in general, which will help you when we discuss Spring Integration. You will also know how to use Spring's JMS support to simplify sending, receiving, and listening for JMS messages.

15-1. Send and Receive JMS Messages with Spring

Problem

To send or receive a JMS message, you have to perform the following tasks:

1. Create a JMS connection factory on a message broker.
2. Create a JMS destination, which can be either a queue or a topic.
3. Open a JMS connection from the connection factory.
4. Obtain a JMS session from the connection.
5. Send or receive the JMS message with a message producer or consumer.

6. Handle `JMSException`, which is a checked exception that must be handled.
7. Close the JMS session and connection.

As you can see, a lot of coding is required to send or receive a simple JMS message. In fact, most of these tasks are boilerplate and require you to repeat them each time when dealing with JMS.

Solution

Spring offers a template-based solution to simplify JMS code. With a JMS template (Spring framework class `JmsTemplate`), you can send and receive JMS messages with much less code. The template handles the boilerplate tasks and also converts the JMS API's `JMSException` hierarchy into Spring's runtime exception `org.springframework.jms.JmsException` hierarchy.

This chapter focuses on Spring support for JMS 1.1, which is the most stable JMS release and is available through the Spring `JmsTemplate` class. JMS 1.0.2 is deprecated -- although it's supported in Spring through the legacy `JmsTemplate102` class. The JMS 2.0 specification is being finalized at the time of this writing, so there's no stable implementation for it.

How It Works

Suppose that you are developing a post office system that includes two subsystems: the front desk subsystem and the back office subsystem. When the front desk receives mail, it passes the mail to the back office for categorizing and delivering. At the same time, the front desk subsystem sends a JMS message to the back office subsystem, notifying it of new mail. The mail information is represented by the following class:

```
package com.apress.springrecipes.post;
public class Mail {

    private String mailId;
    private String country;
    private double weight;

    // Constructors, Getters and Setters
    ...
}
```

The methods for sending and receiving mail information are defined in the `FrontDesk` and `BackOffice` interfaces as follows:

```
package com.apress.springrecipes.post;

public interface FrontDesk {

    public void sendMail(Mail mail);
}

package com.apress.springrecipes.post;

public interface BackOffice {

    public Mail receiveMail();
}
```

Before you can send and receive JMS messages, you need to install a JMS message broker. For simplicity's sake, we have chosen Apache ActiveMQ (<http://activemq.apache.org/>) as our message broker, which is very easy to install and configure. ActiveMQ is an open source message broker that fully supports JMS 1.1.

Note You can download ActiveMQ (e.g., v5.9.1) from the ActiveMQ web site and extract it to a directory of your choice to complete the installation.

Send and Receive Messages without Spring's JMS template Support

First, let's look at how to send and receive JMS messages without Spring's JMS template support. The following `FrontDeskImpl` class sends JMS messages with the JMS API directly.

```
package com.apress.springrecipes.post;

import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;

public class FrontDeskImpl implements FrontDesk {

    public void sendMail(Mail mail) {
        ConnectionFactory cf =
            new ActiveMQConnectionFactory("tcp://localhost:61616");
        Destination destination = new ActiveMQQueue("mail.queue");

        Connection conn = null;
        try {
            conn = cf.createConnection();
            Session session =
                conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer = session.createProducer(destination);

            MapMessage message = session.createMapMessage();
            message.setString("mailId", mail.getMailId());
            message.setString("country", mail.getCountry());
            message.setDouble("weight", mail.getWeight());
            producer.send(message);

            session.close();
        } catch (JMSEException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (JMSException e) {
                }
            }
        }
    }
}
```

In the preceding `sendMail()` method, you first create JMS-specific `ConnectionFactory` and `Destination` objects with the classes provided by ActiveMQ. The message broker URL is the default for ActiveMQ if you run it on localhost. In JMS, there are two types of destinations: queue and topic.

As explained at the start of the chapter, a queue is for the point-to-point communication model, while topic is for the publish-subscribe communication model. Because you are sending JMS messages point to point from front desk to back office, you should use a message queue. You can easily create a topic as a destination using the `ActiveMQTopic` class.

Next, you have to create a connection, session, and message producer before you can send your message.

There are several types of messages defined in the JMS API, including `TextMessage`, `MapMessage`, `BytesMessage`, `ObjectMessage`, and `StreamMessage`. `MapMessage` contains message content in key/value pairs like a map. All of them are interfaces, whose super class is simply `Message`. In the meantime, you have to handle `JMSEException`, which may be thrown by the JMS API. Finally, you must remember to close the session and connection to release system resources. Every time a JMS connection is closed, all its opened sessions will be closed automatically. So you only have to ensure that the JMS connection is closed properly in the `finally` block.

On the other hand, the following BackOfficeImpl class receives JMS messages with the JMS API directly.

```
package com.apress.springrecipes.post;
```

```
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.MessageConsumer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;

public class BackOfficeImpl implements BackOffice {

    public Mail receiveMail() {
        ConnectionFactory cf =
            new ActiveMQConnectionFactory("tcp://localhost:61616");
        Destination destination = new ActiveMQQueue("mail.queue");

        Connection conn = null;
        try {
            conn = cf.createConnection();
            Session session =
                conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer consumer = session.createConsumer(destination);

```

```
conn.start();
MapMessage message = (MapMessage) consumer.receive();
Mail mail = new Mail();
mail.setMailId(message.getString("mailId"));
mail.setCountry(message.getString("country"));
mail.setWeight(message.getDouble("weight"));
session.close();
return mail;
} catch (JMSEException e) {
    throw new RuntimeException(e);
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch (JMSEException e) {
        }
    }
}
}
```

Most of the code in this method is similar to that for sending JMS messages, except that you create a message consumer and receive a JMS message from it. Note that we used the connection's `start()` method here, although we didn't in the `FrontDeskImpl` example before.

When using a Connection to receive messages, you can add listeners to the connection that are invoked on receipt of a message, or you can block synchronously, waiting for a message to arrive. The container has no way of knowing which approach you will take and so it doesn't start polling for messages until you've explicitly called `start()`. If you add listeners or do any kind of configuration, you do so before you invoke `start()`.

Finally, let's create two POJO configuration files—one for the front desk subsystem (e.g., beans-front.xml), and one for the back office subsystem (e.g., beans-back.xml)—in the root of the classpath.

```
<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-3.2.xsd">

    <bean id="frontDesk"
          class="com.apress.springrecipes.post.FrontDeskImpl" />
    </beans>

<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-3.2.xsd">

    <bean id="backOffice"
          class="com.apress.springrecipes.post.BackOfficeImpl" />
    </beans>
```

Now, the front desk and back office subsystems are almost ready to send and receive JMS messages. But before moving on to the final step, start up the ActiveMQ message broker.

To start ActiveMQ, just execute one of the ActiveMQ startup scripts for your operating system. The script itself is called `activemq.sh` or `activemq.bat` for Unix variants or Windows, respectively, and is located in the `bin` directory.

You can easily monitor the ActiveMQ messaging broker's activity. In a default installation, you can open <http://localhost:8161/admin/queueGraph.jsp> to see what's happening with `mail.queue`, the queue used in these examples. Alternatively, ActiveMQ exposes very useful beans and statistics from JMX. Simply run `jconsole`, and drill down to `org.apache.activemq` in the MBeans tab.

Next, let's create a couple of main classes to run the message system. One for the FrontDesk subsystem -- `FrontDeskMain` class -- and another for the BackOffice subsystem -- `BackOfficeMain` --class.

```
package com.apress.springrecipes.post;

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

public class FrontDeskMain {

    public static void main(String[] args) {
        ApplicationContext context = new GenericXmlApplicationContext("beans-front.xml");

        FrontDesk frontDesk = context.getBean("FrontDesk.class");
        frontDesk.sendMail(new Mail("1234", "US", 1.5));
    }
}

package com.apress.springrecipes.post;

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

public class BackOfficeMain {

    public static void main(String[] args) {
        ApplicationContext context = new GenericXmlApplicationContext("beans-back.xml");

        BackOffice backOffice = context.getBean(BackOffice.class);
        Mail mail = backOffice.receiveMail();
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

Every time you run the `FrontDesk` application with the previous `FrontDeskMain` class a message is sent to the broker and every time you run the `BackOffice` application with the previous `BackOfficeMain` class an attempt is made pick a message from the broker.

Send and Receive Messages with Spring's JMS Template

Spring offers a JMS template that can significantly simplify your JMS code. To send a JMS message with this template, you simply call the `send()` method and provide a message destination, as well as a `MessageCreator` object, which creates the JMS message you are going to send. The `MessageCreator` object is usually implemented as an anonymous inner class.

```

package com.apress.springrecipes.post;

import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class FrontDeskImpl implements FrontDesk {

    private JmsTemplate jmsTemplate;
    private Destination destination;

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void setDestination(Destination destination) {
        this.destination = destination;
    }

    public void sendMail(final Mail mail) {
        jmsTemplate.send(destination, new MessageCreator() {
            public Message createMessage(Session session) throws JMSEException {
                MapMessage message = session.createMapMessage();
                message.setString("mailId", mail.getMailId());
                message.setString("country", mail.getCountry());
                message.setDouble("weight", mail.getWeight());
                return message;
            }
        });
    }
}

```

Note that an inner class can only access arguments or variables of the enclosing method that are declared as final. The MessageCreator interface declares only a `createMessage()` method for you to implement. In this method, you create and return your JMS message with the provided JMS session.

A JMS template helps you to obtain and release the JMS connection and session, and it sends the JMS message created by your MessageCreator object. Moreover, it converts the JMS API's JMSEException hierarchy into Spring's JMS runtime exception hierarchy, whose base exception class is `org.springframework.jms.JmsException`. You can catch the `JmsException` thrown from `send` and the other `send` variants and then take action in the catch block if you want.

In the front desk subsystem's bean configuration file, you declare a JMS template that refers to the JMS connection factory for opening connections. Then, you inject this template as well as the message destination into your front desk bean.

```

<beans ...>
    <bean id="connectionFactory"
          class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

```

```

<bean id="mailDestination"
      class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="mail.queue" />
</bean>

<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>

<bean id="frontDesk"
      class="com.apress.springrecipes.post.FrontDeskImpl">
    <property name="destination" ref="mailDestination" />
    <property name="jmsTemplate" ref="jmsTemplate" />
</bean>
</beans>

```

We could also use Java based configuration to express the same configuration.

```

package com.apress.springrecipes.post.config;

import com.apress.springrecipes.post.FrontDeskImpl;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.core.JmsTemplate;

import javax.jms.ConnectionFactory;
import javax.jms.Queue;

@Configuration
public class FrontOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        return new ActiveMQConnectionFactory("tcp://localhost:61616");
    }

    @Bean
    public Queue destination() {
        return new ActiveMQQueue("mail.queue");
    }

    @Bean
    public JmsTemplate jmsTemplate() {
        JmsTemplate jmsTemplate = new JmsTemplate();
        jmsTemplate.setConnectionFactory(connectionFactory());
        return jmsTemplate;
    }
}

```

```

@Bean
public FrontDeskImpl frontDesk() {
    FrontDeskImpl frontDesk = new FrontDeskImpl();
    frontDesk.setJmsTemplate(jmsTemplate());
    frontDesk.setDestination(destination());
    return frontDesk;
}
}

```

To receive a JMS message with a JMS template, you call the `receive()` method by providing a message destination. This method returns a JMS message, `javax.jms.Message`, whose type is the base JMS message type (that is, an interface), so you have to cast it into proper type before further processing.

```

package com.apress.springrecipes.post;

import javax.jms.Destination;
import javax.jms.JMSEException;
import javax.jms.MapMessage;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.JmsUtils;

public class BackOfficeImpl implements BackOffice {

    private JmsTemplate jmsTemplate;
    private Destination destination;

    public void setJmsTemplate(JmsTemplate jmsTemplate) {
        this.jmsTemplate = jmsTemplate;
    }

    public void setDestination(Destination destination) {
        this.destination = destination;
    }

    public Mail receiveMail() {
        MapMessage message = (MapMessage) jmsTemplate.receive(destination);
        try {
            if (message == null) {
                return null;
            }
            Mail mail = new Mail();
            mail.setMailId(message.getString("mailId"));
            mail.setCountry(message.getString("country"));
            mail.setWeight(message.getDouble("weight"));
            return mail;
        } catch (JMSEException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }
}

```

However, when extracting information from the received `MapMessage` object, you still have to handle the JMS API's `JMSEException`. This is in stark contrast to the default behavior of the framework, where it automatically maps exceptions for you when invoking methods on the `JmsTemplate`. To make the type of the exception thrown by this method consistent, you have to make a call to `JmsUtils.convertJmsAccessException()` to convert the JMS API's `JMSEException` into Spring's `JmsException`.

In the back office subsystem's bean configuration file, you declare a JMS template and inject it together with the message destination into your back office bean.

```
<beans ...>
    <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>

    <bean id="mailDestination"
        class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="mail.queue" />
    </bean>

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="receiveTimeout" value="10000" />
    </bean>
    <bean id="backOffice"
        class="com.apress.springrecipes.post.BackOfficeImpl">
        <property name="destination" ref="mailDestination" />
        <property name="jmsTemplate" ref="jmsTemplate" />
    </bean>
</beans>
```

Pay special attention to the JMS template `receiveTimeout` property, it specifies how long to wait in milliseconds. By default, this template waits for a JMS message at the destination forever, and the calling thread is blocked in the meantime. To avoid waiting for a message so long, you should specify a receive timeout for this template. If there's no message available at the destination in the duration, the JMS template's `receive()` method will return a null message.

Of course we can express this as Java based configuration

```
package com.apress.springrecipes.post.config;

import com.apress.springrecipes.post.BackOfficeImpl;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQQueue;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.core.JmsTemplate;

import javax.jms.ConnectionFactory;
import javax.jms.Queue;
```

```

@Configuration
public class BackOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        return new ActiveMQConnectionFactory("tcp://localhost:61616");
    }

    @Bean
    public Queue destination() {
        return new ActiveMQQueue("mail.queue");
    }

    @Bean
    public JmsTemplate jmsTemplate() {
        JmsTemplate jmsTemplate = new JmsTemplate();
        jmsTemplate.setConnectionFactory(connectionFactory());
        jmsTemplate.setReceiveTimeout(10000);
        return jmsTemplate;
    }

    @Bean
    public BackOfficeImpl backOffice() {
        BackOfficeImpl backOffice = new BackOfficeImpl();
        backOffice.setDestination(destination());
        backOffice.setJmsTemplate(jmsTemplate());
        return backOffice;
    }
}

```

In your applications, the main use of receiving a message might be because you're expecting a response to something or want to check for messages at an interval, handling the messages and then spinning down until the next interval. If you intend to receive messages and respond to them as a service, you're likely going to want to use the message-driven POJO functionality described later in this chapter. There, we discuss a mechanism that will constantly sit and wait for messages, handling them by calling back into your application as the messages arrive.

Send and Receive Messages to and from a Default Destination

Instead of specifying a message destination for each JMS template's `send()` and `receive()` method call, you can specify a default destination for a JMS template. Then, you no longer need to inject it into your message sender and receiver beans again.

```

<beans ...>
    ...
    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="defaultDestination" ref="mailDestination" />
    </bean>

```

```
<bean id="frontDesk"
      class="com.apress.springrecipes.post.FrontDeskImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
  </bean>
</beans>
```

In Java config:

```
@Configuration
public class FrontOfficeConfiguration {
  ...
  @Bean
  public JmsTemplate jmsTemplate() {
    JmsTemplate jmsTemplate = new JmsTemplate();
    jmsTemplate.setConnectionFactory(connectionFactory());
    jmsTemplate.setDefaultDestination(mailDestination());
    return jmsTemplate;
  }

  @Bean
  public FrontDeskImpl frontDesk() {
    FrontDeskImpl frontDesk = new FrontDeskImpl();
    frontDesk.setJmsTemplate(jmsTemplate());
    return frontDesk;
  }
}
```

For the back office the configuration would look like this.

```
<beans ...>
  ...
  <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
    <property name="receiveTimeout" value ="10000"/>
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestination" ref="mailDestination" />
  </bean>
  <bean id="backOffice"
        class="com.apress.springrecipes.post.BackOfficeImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
  </bean>
</beans>
```

In Java based configuration

```
@Configuration
public class BackOfficeConfiguration {
  ...
  @Bean
  public JmsTemplate jmsTemplate() {
    JmsTemplate jmsTemplate = new JmsTemplate();
    jmsTemplate.setConnectionFactory(connectionFactory());
```

```
jmsTemplate.setDefaultDestination(mailDestination());
jmsTemplate.setReceiveTimeout(10000);
return jmsTemplate;
}

@Bean
public BackOfficeImpl backOffice() {
    BackOfficeImpl backOffice = new BackOfficeImpl();
    backOffice.setJmsTemplate(jmsTemplate());
    return backOffice;
}
}
```

With the default destination specified for a JMS template, you can delete the setter method for a message destination from your message sender and receiver classes. Now, when you call the `send()` and `receive()` methods, you no longer need to specify a message destination.

```
package com.apress.springrecipes.post;
...
import org.springframework.jms.core.MessageCreator;

public class FrontDeskImpl implements FrontDesk {
    ...
    public void sendMail(final Mail mail) {
        jmsTemplate.send(new MessageCreator() {
            ...
        });
    }
}

package com.apress.springrecipes.post;
...
import javax.jms.MapMessage;
...

public class BackOfficeImpl implements BackOffice {
    ...
    public Mail receiveMail() {
        MapMessage message = (MapMessage) jmsTemplate.receive();
        ...
    }
}
```

In addition, instead of specifying an instance of the `Destination` interface for a JMS template, you can specify the destination name to let the JMS template resolve it for you, so you can delete the `destination` property declaration from both bean configuration files. This is done by adding the `defaultDestinationName` property.

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    ...
    <property name="defaultDestinationName" value="mail.queue" />
</bean>
```

And in Java based configuration style.

```
@Bean
public JmsTemplate jmsTemplate() {
    JmsTemplate jmsTemplate = new JmsTemplate();
    ...
    jmsTemplate.setDefaultDestinationName("mail.queue");
    return jmsTemplate;
}
```

Extend the JmsGatewaySupport Class

JMS sender and receiver classes can also extend `JmsGatewaySupport` to retrieve a JMS template. You have the following two options for classes that extend `JmsGatewaySupport` to create their JMS template:

- Inject a JMS connection factory for `JmsGatewaySupport` to create a JMS template on it automatically. However, if you do it this way, you won't be able to configure the details of the JMS template.
- Inject a JMS template for `JmsGatewaySupport` that is created and configured by you.

Of them, the second approach is more suitable if you have to configure the JMS template yourself. You can delete the private field `jmsTemplate` and its setter method from both your sender and receiver classes. When you need access to the JMS template, you just make a call to `getJmsTemplate()`.

```
package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
...

public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    ...
    public void sendMail(final Mail mail) {
        getJmsTemplate().send(new MessageCreator() {
            ...
        });
    }
}

package com.apress.springrecipes.post;
...

import org.springframework.jms.core.support.JmsGatewaySupport;

public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        MapMessage message = (MapMessage) getJmsTemplate().receive();
        ...
    }
}
```

15-2. Convert JMS Messages

Problem

Your application receives messages from your message queue but needs to transform those messages from the JMS-specific type to a business-specific class.

Solution

Spring provides an implementation of `SimpleMessageConverter` to handle the translation of a JMS message received to a business object and the translation of a business object to a JMS message. You can leverage the default or provide your own.

How It Works

The previous recipes handled the raw JMS messages. Spring's JMS template can help you convert JMS messages to and from Java objects using a message converter. By default, the JMS template uses `SimpleMessageConverter` for converting `TextMessage` to or from a string, `BytesMessage` to or from a byte array, `MapMessage` to or from a map, and `ObjectMessage` to or from a serializable object.

For the front desk and back office classes of the past recipe, you can send and receive a map using the `convertAndSend()` and `receiveAndConvert()` methods, where the map is converted to/from `MapMessage`.

```
package com.apress.springrecipes.post;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    public void sendMail(Mail mail) {
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("mailId", mail.getMailId());
        map.put("country", mail.getCountry());
        map.put("weight", mail.getWeight());
        getJmsTemplate().convertAndSend(map);
    }
}

package com.apress.springrecipes.post;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        Map map = (Map) getJmsTemplate().receiveAndConvert();
        Mail mail = new Mail();
        mail.setMailId((String) map.get("mailId"));
        mail.setCountry((String) map.get("country"));
        mail.setWeight((Double) map.get("weight"));
        return mail;
    }
}
```

You can also create a custom message converter by implementing the `MessageConverter` interface for converting mail objects.

```
package com.apress.springrecipes.post;

import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;

public class MailMessageConverter implements MessageConverter {

    public Object fromMessage(Message message) throws JMSEException,
        MessageConversionException {
        MapMessage mapMessage = (MapMessage) message;
        Mail mail = new Mail();
        mail.setMailId(mapMessage.getString("mailId"));
        mail.setCountry(mapMessage.getString("country"));
        mail.setWeight(mapMessage.getDouble("weight"));
        return mail;
    }

    public Message toMessage(Object object, Session session) throws JMSEException,
        MessageConversionException {
        Mail mail = (Mail) object;
        MapMessage message = session.createMapMessage();
        message.setString("mailId", mail.getMailId());
        message.setString("country", mail.getCountry());
        message.setDouble("weight", mail.getWeight());
        return message;
    }
}
```

To apply this message converter, you have to declare it in both bean configuration files and inject it into the JMS template.

```
<beans ...>
    ...
    <bean id="mailMessageConverter"
        class="com.apress.springrecipes.post.MailMessageConverter" />

    <bean id="jmsTemplate"
        class="org.springframework.jms.core.JmsTemplate">
        ...
        <property name="messageConverter" ref="mailMessageConverter" />
    </bean>
</beans>
```

Of course this can be expressed in Java configuration.

```
@Configuration
public class BackOfficeConfiguration {
    ...
    @Bean
    public JmsTemplate jmsTemplate() {
        JmsTemplate jmsTemplate = new JmsTemplate();
        jmsTemplate.setMessageConverter(mailMessageConverter());
        ...
        return jmsTemplate;
    }

    @Bean
    public MailMessageConverter mailMessageConverter() {
        return new MailMessageConverter();
    }
}
```

When you set a message converter for a JMS template explicitly, it will override the default `SimpleMessageConverter`. Now, you can call the JMS template's `convertAndSend()` and `receiveAndConvert()` methods to send and receive mail objects.

```
package com.apress.springrecipes.post;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {
    public void sendMail(Mail mail) {
        getJmsTemplate().convertAndSend(mail);
    }
}

package com.apress.springrecipes.post;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {
    public Mail receiveMail() {
        return (Mail) getJmsTemplate().receiveAndConvert();
    }
}
```

15-3. Manage JMS Transactions

Problem

You want to participate in transactions with JMS so that the receipt and sending of messages are transactional.

Approach

You can use the same transactions strategy as you would for any Spring component. Leverage Spring's `TransactionManager` implementations as needed and wire the behavior into beans.

Solution

When producing or consuming multiple JMS messages in a single method, if an error occurs in the middle, the JMS messages produced or consumed at the destination may be left in an inconsistent state. You have to surround the method with a transaction to avoid this problem.

In Spring, JMS transaction management is consistent with other data access strategies. For example, you can annotate the methods that require transaction management with the `@Transactional` annotation.

```
package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
import org.springframework.transaction.annotation.Transactional;
...
public class FrontDeskImpl extends JmsGatewaySupport implements FrontDesk {

    @Transactional
    public void sendMail(Mail mail) {
        ...
    }
}

package com.apress.springrecipes.post;

import org.springframework.jms.core.support.JmsGatewaySupport;
import org.springframework.transaction.annotation.Transactional;
...
public class BackOfficeImpl extends JmsGatewaySupport implements BackOffice {

    @Transactional
    public Mail receiveMail() {
        ...
    }
}
```

Then, in both bean configuration files, you add the `<tx:annotation-driven />` element (for Java based configuration add the `@EnableTransactionManagement` annotation) and declare a transaction manager. The corresponding transaction manager for local JMS transactions is `JmsTransactionManager`, which requires a reference to the JMS connection factory.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">
    ...
    <tx:annotation-driven />
    <bean id="transactionManager"
        class="org.springframework.jms.connection.JmsTransactionManager">
```

```

<property name="connectionFactory">
    <ref bean="connectionFactory" />
</property>
</bean>
</beans>

```

In the situation where one is using Java based configuration use the `@EnableTransactionManagement` annotation and add the `JmsTransactionManager` as a `@Bean` annotated method.

```

package com.apress.springrecipes.post.config;
...
import org.springframework.jms.connection.JmsTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

import javax.jms.ConnectionFactory;

@Configuration
@EnableTransactionManagement
public class BackOfficeConfiguration {
...
    @Bean
    public ConnectionFactory connectionFactory() { ... }

    @Bean
    public PlatformTransactionManager transactionManager() {
        return new JmsTransactionManager(connectionFactory());
    }
}

```

If you require transaction management across multiple resources, such as a data source and an ORM resource factory, or if you need distributed transaction management, you have to configure JTA transaction in your app server and use `JtaTransactionManager`. Note that for multiple resource transactions support, the JMS connection factory must be XA compliant (i.e., it must support distributed transactions).

15-4. Create Message-Driven POJOs in Spring Problem

When you call the `receive()` method on a JMS message consumer to receive a message, the calling thread is blocked until a message is available. The thread can do nothing but wait. This type of message reception is called synchronous reception, because an application must wait for the message to arrive before it can finish its work.

You can create a message-driven POJO (MDP) to support asynchronous reception of JMS messages. A MDP is decorated with the `@MessageDriven` annotation.

Note A message driven POJO or MDP in the context of this recipe refers to a POJO that can listen for JMS messages without any particular run-time requirements. It does not refer to message driven beans (MDBs) aligned to the EJB specification that require an EJB container.

Solution

Spring allows beans declared in its IoC container to listen for JMS messages in the same way as MDBs which are based on the EJB spec. Because Spring adds message-listening capabilities to POJOs, they are called message-driven POJOs (MDPs).

How It Works

Suppose you want to add an electronic board to the post office's back office to display mail information in real time as it arrives from the front desk. As the front desk sends a JMS message along with mail, the back office subsystem can listen for these messages and display them on the electronic board. For better system performance, you should apply the asynchronous JMS reception approach to avoid blocking the thread that receives these JMS messages.

Listen for JMS Messages with Message Listeners

First, you create a message listener to listen for JMS messages. The message listener provides an alternative to the approach taken in `BackOfficeImpl` in previous recipes with the `JMSTemplate`. A listener can also consume messages from a broker. For example, the following `MailListener` listens for JMS messages that contain mail information:

```
package com.apress.springrecipes.post;

import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;

import org.springframework.jms.support.JmsUtils;

public class MailListener implements MessageListener {

    public void onMessage(Message message) {
        MapMessage mapMessage = (MapMessage) message;
        try {
            Mail mail = new Mail();
            mail.setMailId(mapMessage.getString("mailId"));
            mail.setCountry(mapMessage.getString("country"));
            mail.setWeight(mapMessage.getDouble("weight"));
            displayMail(mail);
        } catch (JMSException e) {
            throw JmsUtils.convertJmsAccessException(e);
        }
    }

    private void displayMail(Mail mail) {
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

A message listener must implement the `javax.jms.MessageListener` interface. When a JMS message arrives, the `onMessage()` method will be called with the message as the method argument. In this sample, you simply display the mail information to the console. Note that when extracting message information from a `MapMessage` object, you need to handle the JMS API's `JMSEException`. You can make a call to `JmsUtils.convertJmsAccessException()` to convert it into Spring's runtime exception `JmsException`.

Next, you have to configure this listener in the back office's bean configuration file. Declaring this listener alone is not enough to listen for JMS messages. You need a message listener container to monitor JMS messages at a message destination and trigger your message listener on message arrival.

```
<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-3.2.xsd">

    <bean id="connectionFactory"
          class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
    <bean id="mailListener"
          class="com.apress.springrecipes.post.MailListener" />

    <bean
          class="org.springframework.jms.listener.SimpleMessageListenerContainer">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="destinationName" value="mail.queue" />
        <property name="messageListener" ref="mailListener" />
    </bean>
</beans>
```

Spring provides several types of message listener containers for you to choose from in the `org.springframework.jms.listener` package, of which `SimpleMessageListenerContainer` and `DefaultMessageListenerContainer` are the most commonly used. `SimpleMessageListenerContainer` is the simplest one that doesn't support transactions. If you have a transaction requirement in receiving messages, you have to use `DefaultMessageListenerContainer`.

Now, you can start the message listener. Since we won't need to invoke a bean to trigger message consumption -- the listener will do it for us -- the following main class which only starts the Spring IoC container is enough:

```
package com.apress.springrecipes.post;

import org.springframework.context.support.GenericXmlApplicationContext;

public class BackOfficeMain {

    public static void main(String[] args) {
        new GenericXmlApplicationContext("beans-back.xml");
    }
}
```

When you start this `BackOffice` application it will listen for messages on the message broker (i.e., ActiveMQ). As soon as the `FrontDesk` application sends a message to the broker, the `BackOffice` application will react and display the message to the console.

You can use Java based configuration to configure a `SimpleMessageListenerContainer` it looks quite similar to the xml based version:

```
package com.apress.springrecipes.post.config;

import com.apress.springrecipes.post.MailListener;
import org.apache.activemq.ActiveMQConnectionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.listener.SimpleMessageListenerContainer;

import javax.jms.ConnectionFactory;

@Configuration
public class BackOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() { ... }

    @Bean
    public MailListener mailListener() {
        return new MailListener();
    }

    @Bean
    public Object container() {
        SimpleMessageListenerContainer smlc = new SimpleMessageListenerContainer();
        smlc.setConnectionFactory(connectionFactory());
        smlc.setDestinationName("mail.queue");
        smlc.setMessageListener(mailListener());
        return smlc;
    }
}
```

Listen for JMS Messages with POJOs

While a listener that implements the `MessageListener` interface can listen for messages, so can an arbitrary bean declared in the Spring IoC container. Doing so means that beans are decoupled from the Spring framework interfaces as well as the JMS `MessageListener` interface. For a method of this bean to be triggered on message arrival, it must accept one of the following types as its sole method argument:

- Raw JMS message type: For `TextMessage`, `MapMessage`, `BytesMessage`, and `ObjectMessage`
- `String`: For `TextMessage` only
- `Map`: For `MapMessage` only
- `byte[]`: For `BytesMessage` only
- `Serializable`: For `ObjectMessage` only

For example, to listen for `MapMessage`, you declare a method that accepts a map as its argument. This listener no longer needs to implement the `MessageListener` interface.

```

package com.apress.springrecipes.post;
...
public class MailListener {

    public void displayMail(Map map) {
        Mail mail = new Mail();
        mail.setMailId((String) map.get("mailId"));
        mail.setCountry((String) map.get("country"));
        mail.setWeight((Double) map.get("weight"));
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}

```

A POJO is registered to a listener container through a `MessageListenerAdapter` instance. This adapter implements the `MessageListener` interface and will delegate message handling to the target bean's method via reflection.

```

<beans ...>
    ...
    <bean id="mailListener"
          class="com.apress.springrecipes.post.MailListener" />

    <bean id="mailListenerAdapter"
          class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
        <property name="delegate" ref="mailListener" />
        <property name="defaultListenerMethod" value="displayMail" />
    </bean>

    <bean
          class="org.springframework.jms.listener.SimpleMessageListenerContainer">
        <property name="connectionFactory" ref="connectionFactory" />
        <property name="destinationName" value="mail.queue" />
        <property name="messageListener" ref="mailListenerAdapter" />
    </bean>
</beans>

```

You have to set the `delegate` property of `MessageListenerAdapter` to your target bean. By default, this adapter will call the method whose name is `handleMessage` on that bean. If you want to call another method, you can specify it in the `defaultListenerMethod` property. Finally, notice that you have to register the listener adapter, not the target bean, with the listener container.

Convert JMS Messages

You can also create a message converter for converting mail objects from JMS messages that contain mail information. Because message listeners receive messages only, the method `toMessage()` will not be called, so you can simply return null for it. However, if you use this message converter for sending messages too, you have to implement this method. The following example reprints the `MailMessageConvertor` class written earlier:

```

package com.apress.springrecipes.post;

import javax.jms.JMSEException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;

```

```

import org.springframework.jms.support.converter.MessageConversionException;
import org.springframework.jms.support.converter.MessageConverter;

public class MailMessageConverter implements MessageConverter {

    public Object fromMessage(Message message) throws JMSException,
        MessageConversionException {
        MapMessage mapMessage = (MapMessage) message;
        Mail mail = new Mail();
        mail.setMailId(mapMessage.getString("mailId"));
        mail.setCountry(mapMessage.getString("country"));
        mail.setWeight(mapMessage.getDouble("weight"));
        return mail;
    }

    public Message toMessage(Object object, Session session) throws JMSException,
        MessageConversionException {
        ...
    }
}

```

A message converter should be applied to a listener adapter for it to convert messages into objects before calling your POJO's methods.

```

<beans ...>
    ...
    <bean id="mailMessageConverter"
        class="com.apress.springrecipes.post.MailMessageConverter" />

    <bean id="mailListenerAdapter"
        class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
        <property name="delegate" ref="mailListener" />
        <property name="defaultListenerMethod" value="displayMail" />
        <property name="messageConverter" ref="mailMessageConverter" />
    </bean>
</beans>

```

With this message converter, the listener method of your POJO can accept a mail object as the method argument.

```

package com.apress.springrecipes.post;

public class MailListener {

    public void displayMail(Mail mail) {
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}

```

When using Java based configuration the configuration looks like the following:

```
package com.apress.springrecipes.post.config;
...
import com.apress.springrecipes.post.MailListener;
import com.apress.springrecipes.post.MailMessageConverter;
import org.springframework.jms.listener.adapter.MessageListenerAdapter;
import org.springframework.jms.support.converter.MessageConverter;

@Configuration
public class BackOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() { ... }

    @Bean
    public MailListener mailListener() {
        return new MailListener();
    }

    @Bean
    public Object container() {
        SimpleMessageListenerContainer smlc = new SimpleMessageListenerContainer();
        smlc.setConnectionFactory(connectionFactory());
        smlc.setDestinationName("mail.queue");
        smlc.setMessageListener(mailListenerAdapter());
        return smlc;
    }

    @Bean
    public MessageConverter mailMessageConverter() {
        return new MailMessageConverter();
    }

    @Bean
    public MessageListenerAdapter mailListenerAdapter() {
        MessageListenerAdapter mailListenerAdapter = new MessageListenerAdapter();
        mailListenerAdapter.setDelegate(mailListener());
        mailListenerAdapter.setDefaultListenerMethod("displayMail");
        mailListenerAdapter.setMessageConverter(mailMessageConverter());
        return mailListenerAdapter;
    }
}
```

Manage JMS Transactions

As mentioned before, `SimpleMessageListenerContainer` doesn't support transactions. So, if you need transaction management for your message listener method, you have to use `DefaultMessageListenerContainer` instead. For local JMS transactions, you can simply enable its `sessionTransacted` property, and your listener method will run within a local JMS transaction (as opposed to XA transactions).

```
<bean
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="destinationName" value="mail.queue" />
    <property name="messageListener" ref="mailListenerAdapter" />
    <property name="sessionTransacted" value="true" />
</bean>
```

The equivalent in Java based configuration:

```
@Bean
public DefaultMessageListenerContainer container() {
    DefaultMessageListenerContainer dmlc = new DefaultMessageListenerContainer();
    dmlc.setConnectionFactory(connectionFactory());
    dmlc.setDestinationName("mail.queue");
    dmlc.setMessageListener(mailListenerAdapter());
dmlc.setSessionTransacted(true);
    return dmlc;
}
```

However, if you want your listener to participate in a JTA transaction, you need to declare a `JtaTransactionManager` instance and inject it into your listener container.

Use Spring's JMS Schema

Spring offers a JMS schema to simplify the JMS listener configurations. You must add the `jms` schema definition to the `<beans>` root element to achieve this.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
                           http://www.springframework.org/schema/jms
                           http://www.springframework.org/schema/jms/spring-jms-3.2.xsd">

    <bean id="connectionFactory"
          class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616" />
    </bean>
    <bean id="transactionManager"
          class="org.springframework.jms.connection.JmsTransactionManager">
        <property name="connectionFactory">
            <ref bean="connectionFactory" />
        </property>
    </bean>

    <bean id="mailMessageConverter"
          class="com.apress.springrecipes.post.MailMessageConverter" />

    <bean id="mailListener"
          class="com.apress.springrecipes.post.MailListener" />

```

```

<jms:listener-container
    connection-factory="connectionFactory"
    transaction-manager="transactionManager"
    message-converter="mailMessageConverter">
    <jms:listener
        destination="mail.queue"
        ref="mailListener" method="displayMail" />
</jms:listener-container>
</beans>

```

Note, you don't need to specify the `connection-factory` attribute for a listener container explicitly if your JMS connection factory's name is `connectionFactory`, in which case it's located by default.

15-5. Cache and pool JMS connections

Problem

Throughout this chapter, for the sake of simplicity, we've explored Spring's JMS support with a very simple instance of `org.apache.activemq.ActiveMQConnectionFactory` as the connection factory. This isn't the best choice in practice. As with all things, there are performance considerations.

The crux of the issue is that `JmsTemplate` closes sessions and consumers on each invocation. This means that it tears down all those objects and restores frees the memory. This is "safe," but not performant, as some of the objects created—like Consumers—are meant to be long lived. This behavior stems from the use of the `JmsTemplate` in application server environments, where typically the application server's Connection Factory is used, and it, internally, provides connection pooling. In this environment, restoring all the objects simply returns it to the pool, which is the desirable behavior.

Solution

There's no "one size fits all" solution to this. You need to weigh the qualities you're looking for and react appropriately.

How It Works

Generally, you want a connection factory that provides pooling and caching of some sort when publishing messages using `JmsTemplate`. The first place to look for a pooled connection factory might be your application server. It may very well provide one by default.

In the examples in this chapter, we use ActiveMQ in a stand-alone configuration. ActiveMQ, like many vendors, provides a pooled connection factory class alternative (ActiveMQ provides two, actually: one for use consuming messages with a JCA connector and another one for use outside of a JCA container), we can use these instead to handle caching producers and sessions when sending messages. The following configuration pools a connection factory in a stand-alone configuration. It's a drop-in replacement for the previous examples when publishing messages.

```

<bean id="connectionFactory" class="org.apache.activemq.pool.PooledConnectionFactory"
    destroy-method="stop">
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL">
                <value>tcp://localhost:61616</value>
            </property>
        </bean>
    </property>
</bean>

```

The equivalent in Java based configuration:

```
@Bean(destroyMethod = "stop")
public ConnectionFactory connectionFactory() {
    ActiveMQConnectionFactory connectionFactoryToUse =
        new ActiveMQConnectionFactory("tcp://localhost:61616");
PooledConnectionFactory connectionFactory = new PooledConnectionFactory();
connectionFactory.setConnectionFactory(connectionFactoryToUse);
return connectionFactory;
}
```

If you are receiving messages, you could still stand some more efficiency, because the `JmsTemplate` constructs a new `MessageConsumer` each time as well. In this situation, you have a few alternatives: use Spring's various `*MessageListenerContainer` implementations mechanism (MDPs), because it caches consumers correctly, or use Spring's `ConnectionFactory` implementations. The first implementation, `org.springframework.jms.connection.SingleConnectionFactory`, returns the same underlying JMS connection each time (which is thread-safe according to the JMS API) and ignores calls to the `close()` method. Generally, this implementation works well with the JMS 1.1 API. A newer alternative is the `org.springframework.jms.connection.CachingConnectionFactory`. First, the obvious advantage is that it provides the ability to cache multiple instances. And second, it caches sessions, `MessageProducers`, and `MessageConsumers`.

15-6 Send and Receive AMQP Messages with Spring

Problem

You want to use RabbitMQ to send and receive messages.

Solution

The Spring AMQP project (<http://projects.spring.io/spring-amqp/>) which provides easy access and use of the AMQP protocol. It has support similar to that of Spring JMS. It comes with a `RabbitTemplate` which provides basic send and receive options, it also comes with a `MessageListenerContainer` option which mimics Spring JMS.

How It Works

First look at how we can send a message using the `RabbitTemplate`, to get access to the `RabbitTemplate` it is the simplest to extend `RabbitGatewaySupport`. Below is the `FrontDeskImpl` which uses the `RabbitTemplate`.

Send Messages with Spring's template Support

The `FrontDeskImpl` class extends `RabbitGatewaySupport`, this class configures a `RabbitTemplate` based on the configuration we pass in. To send a message we use the `getRabbitTemplate` method to get the template and next to convert and send the message, for this you use the `convertAndSend` method. This method will first use a `MessageConverter` to convert the message into JSON and then send it to the queue we have configured.

```
package com.apress.springrecipes.post;

import org.springframework.amqp.rabbit.core.support.RabbitGatewaySupport;
```

```
public class FrontDeskImpl extends RabbitGatewaySupport implements FrontDesk {
    public void sendMail(final Mail mail) {
        getRabbitTemplate().convertAndSend(mail);
    }
}
```

Let's take a look at the configuration:

```
package com.apress.springrecipes.post.config;

import com.apress.springrecipes.post.FrontDeskImpl;
import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class FrontOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory connectionFactory = new CachingConnectionFactory("127.0.0.1");
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        connectionFactory.setPort(5672);
        return connectionFactory;
    }

    @Bean
    public RabbitTemplate rabbitTemplate() {
        RabbitTemplate rabbitTemplate = new RabbitTemplate();
        rabbitTemplate.setConnectionFactory(connectionFactory());
        rabbitTemplate.setMessageConverter(new Jackson2JsonMessageConverter());
        rabbitTemplate.setRoutingKey("mail.queue");
        return rabbitTemplate;
    }

    @Bean
    public FrontDeskImpl frontDesk() {
        FrontDeskImpl frontDesk = new FrontDeskImpl();
        frontDesk.setRabbitTemplate(rabbitTemplate());
        return frontDesk;
    }
}
```

The configuration is quite similar to the JMS configuration. We need a ConnectionFactory to connect to our RabbitMQ broker. We use a `CachingConnectionFactory` so that we can reuse our connections. Next there is the `RabbitTemplate` which uses the connection and has a `MessageConverter` to convert the message. The message is being converted into JSON using the `Jackson2` library hence the configuration of the `Jackson2JsonMessageConverter`.

Finally the RabbitTemplate is passed into the FrontDeskImpl so that it is available for usage.

Before you can send and receive AMQP messages, you need to install a AMQP message broker. For simplicity's sake, we have chosen RabbitMQ (<http://www.rabbitmq.org/>) as our broker, which is very easy to install and configure. RabbitMQ is an open source broker that fully supports AMQP 0.9.1.

Note You can download RabbitMQ (e.g., v3.3.3) from the RabbitMQ web site and extract it to a directory of your choice to complete the installation.

When your AMQP broker is setup you can use the following main class to run the frontdesk application.

```
package com.apress.springrecipes.post;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class FrontDeskMain {

    public static void main(String[] args) throws Exception {
        ConfigurableApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.post.config");
        context.registerShutdownHook();

        FrontDesk frontDesk = context.getBean(FrontDesk.class);
        frontDesk.sendMail(new Mail("1234", "US", 1.5));

        System.in.read();
    }
}
```

Listen for AMQP Messages with Message Listeners

Spring AMQP supports MessageListenerContainers for retrieving messages in the same way as it Spring JMS does for JMS. Spring AMQP has a SimpleMessageListenerContainer which can be used to retrieve messages. To be able to receive messages we need a MessageListener which we can pass into SimpleMessageListenerContainer, this listener will be invoked for each incoming message.

Let's take a look at the MessageListener which is used.

```
package com.apress.springrecipes.post;

public class Maillistener {

    public void displayMail(Mail mail) {
        System.out.println("Mail #" + mail.getMailId() + " received");
    }
}
```

The MailListener is exactly the same as the one created in Recipe 15-4 for receiving JMS messages. The difference is in the configuration.

```
package com.apress.springrecipes.post.config;

import com.apress.springrecipes.post.MailListener;
import org.springframework.amqp.core.AmqpAdmin;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.CachingConnectionFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitAdmin;
import org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer;
import org.springframework.amqp.rabbit.listener.adapter.MessageListenerAdapter;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class BackOfficeConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory connectionFactory = new CachingConnectionFactory("127.0.0.1");
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        connectionFactory.setPort(5672);
        return connectionFactory;
    }

    @Bean
    public AmqpAdmin amqpAdmin() {
        return new RabbitAdmin(connectionFactory());
    }

    @Bean
    public Queue mailQueue() {
        return new Queue("mail.queue");
    }

    @Bean
    public SimpleMessageListenerContainer messageListenerContainer() {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(connectionFactory());
        container.setMessageListener(mailListenerAdapter());
        container.setQueues(mailQueue());
        return container;
    }
}
```

```

@Bean
public MessageListenerAdapter mailListenerAdapter() {
    MessageListenerAdapter adapter = new MessageListenerAdapter();
    adapter.setDelegate(mailListener());
    adapter.setDefaultListenerMethod("displayMail");
    adapter.setMessageConverter(new Jackson2JsonMessageConverter());
    return adapter;
}

@Bean
public MailListener mailListener() {
    return new MailListener();
}
}

```

The SimpleMessageListenerContainer needs a ConnectionFactory for this we are using the CachingConnectionFactory. We also need to pass in the Queue that is used to listen on. The MailListener is invoked by a MessageListenerAdapter which will invoke the displayMail method after converting the incoming JSON message into a Mail object using the Jackson2JsonMessageConverter.

Finally there is the RabbitAdmin bean this will auto register the configured queue(s) with our message broker.

To listen to messages create a class with a main method that only needs to construct the application context. The SimpleMessageListenerContainer will be automatically started and starts to listen for messages.

```

package com.apress.springrecipes.post;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class BackOfficeMain {

    public static void main(String[] args) {
        ApplicationContext context =
            new AnnotationConfigApplicationContext("com.apress.springrecipes.post.config");
    }
}

```

Summary

This chapter explored Spring's support for JMS: how JMS fits in an architecture and how to use Spring to build message-oriented architectures. You learned how to both produce and consume messages using a message queue. You worked with Active MQ, a reliable open source message queue. You learned how to build message-driven POJOs using a MessageListenerContainer.

Finally you explored Spring AMQP to send messages using the AMQP protocol using the Spring AMQP project.

The next chapter will explore Spring Integration, which is an ESB-like framework for building application integration solutions, similar to Mule ESB and ServiceMix. You will be able to leverage the knowledge gained in this chapter to take your message-oriented applications to new heights with Spring integration.

CHAPTER 16



Spring Integration

In this chapter, you will learn the principles behind enterprise application integration (EAI), used by many modern applications to decouple dependencies between components. The Spring framework provides a powerful and extensible framework called Spring Integration. Spring Integration provides the same level of decoupling for disparate systems and data that the core Spring framework provides for components within an application.

This chapter aims to give you all the required knowledge to understand the patterns involved in *EAI*, to understand what an *enterprise service bus (ESB)* is, and - ultimately - how to build solutions using Spring Integration. If you've used an EAI server or an ESB, you'll find that Spring Integration is markedly simpler than anything you're likely to have used before.

After finishing this chapter, you will be able to write fairly sophisticated Spring Integration solutions to integrate applications, to let them to share services and data. You will learn Spring Integration's many options for configuration, too. Spring Integration can be configured entirely in a standard XML namespace, if you like, but you'll probably find that a hybrid approach, using annotations and XML, is more natural. You will also learn why Spring Integration is a very attractive alternative for people coming from a classic enterprise application integration background. If you've used an ESB before, such as Mule or ServiceMix, or a classical EAI server such as Axway's Integrator or TIBCO's ActiveMatrix, the idioms explained here should be familiar, and the configuration refreshingly straightforward.

Note To use Spring Integration, you need to have the requisite framework libraries and adapters on the classpath. If you are using Apache Maven, you should add - at a minimum for this chapter - the following dependencies to your project.

```
<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-core</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-httpinvoker</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-file</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>
```

```

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-jms</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.springframework.integration</groupId>
  <artifactId>spring-integration-adapter</artifactId>
  <version>4.0.0.RELEASE</version>
</dependency>

```

16-1. Integrating One System with Another Using EAI

Problem

You have two applications that need to talk to each other through external interfaces. You need to establish a connection between the applications' services and/or their data.

Solution

You need to employ *EAI*, which is the discipline of integrating applications and data using a set of well-known patterns. These patterns are usefully summarized and embodied in a landmark book called *Enterprise Integration Patterns*, by Gregor Hohpe, Bobby Woolf, and colleagues. Today the patterns are canonical and are the lingua franca of the modern-day ESB.

How It Works

Picking an Integration Style

There are multiple integration styles, each best suited for certain types of applications and requirements. The basic premise is simple: your application can't speak directly to the other system using the native mechanism in one system. So you can devise a bridging connection, something to build on top of, abstract, or work around some characteristic about the other system in a way that's advantageous to the invoking system. What you abstract is different for each application. Sometimes it's the location, sometimes it's the synchronous or asynchronous nature of the call, and sometimes it's the messaging protocol. There are many criteria for choosing an integration style, related to how tightly coupled you want your application to be, to server affinity, to the demands of the messaging formats, and so on. In a way, TCP/IP is the most famous of all integration techniques because it decouples one application from another's server.

You have probably built applications that use some or all of the following integration styles (using Spring, no less!). Shared Database, for example, is easily achieved using Spring's JDBC support; Remote Procedure Invocation is easily achieved using Spring's exporter functionality.

The four integration styles are as follows:

- *File transfer*: Have each application produce files of shared data for others to consume and consume files that others have produced.
- *Shared database*: Have the applications store the data they want to share in a common database. This usually takes the form of a database to which different applications have access. This is not usually a favored approach because it means exposing your data to different clients who might not respect the constraints you have in place (but not codified). Using views and stored procedures can often make this option possible, but it's not ideal. There's no particular support for talking to a database, per se, but you can build an endpoint that deals with new results in a SQL database as message payloads. Integration with databases doesn't tend to be granular or message-oriented, but batch-oriented instead. After all, a million new rows in a database isn't an event so much as a batch! It's no surprise then that Spring Batch (discussed in Chapter 21) included terrific support for JDBC-oriented input and output.
- *Remote Procedure Invocation*: Have each application expose some of its procedures so that they can be invoked remotely and have applications invoke them to initiate behavior and exchange data. There is specific support for optimizing RPC (remote procedure calls such as SOAP, RMI, and HTTP Invoker) exchanges using Spring Integration.
- *Messaging*: Have each application connect to a common messaging system and exchange data and invoke behavior using messages. This style, most enabled by JMS in the JEE world, also describes other asynchronous or multicast publish/subscribe architectures. In a way, an ESB or an EAI container such as Spring Integration lets you handle most of the other styles as though you were dealing with a messaging queue: a request comes in on a queue and is managed, responded to, or forwarded onward on another queue.

Building on an ESB Solution

Now that you know how you want to approach the integration, it's all about actually implementing it. You have many choices in today's world. If the requirement is common enough, most middleware or frameworks will accommodate it in some way. JEE, .NET, and others handle common cases very well: SOAP, XMLRPC, a binary layer such as EJB or binary remoting, JMS, or a MQ abstraction. If, however, the requirement is somewhat exotic, or you have a lot of configuration to do, then perhaps an ESB is required. An ESB is middleware that provides a high-level approach to modeling integrations, in the spirit of the patterns described by EAI. The ESB provides a manageable configuration format for orchestrating the different pieces of an integration in a simple high-level format.

Spring Integration, an API in the SpringSource Portfolio, provides a robust mechanism for modeling a lot of these integration scenarios that work well with Spring. Spring Integration has many advantages over a lot of other ESBs, especially the lightweight nature of the framework. The nascent ESB market is filled with choices. Some are former EAI servers, reworked to address the ESB-centric architectures. Some are genuine ESBs, built with that in mind. Some are little more than message queues with adapters.

Indeed, if you're looking for an extraordinarily powerful EAI server (with almost integration with the JEE platform and a very hefty price tag), you might consider Axway Integrator. There's very little it can't do. Vendors such as TIBCO and WebMethods made their marks (and were subsequently acquired) because they provided excellent tools for dealing with integration in the enterprise. These options, although powerful, are usually very expensive and middleware-centric: your integrations are deployed to the middleware.

Standardization attempts, such as Java Business Integration (JBI), have proven successful to an extent, and there are good compliant ESBs based on these standards (OpenESB and ServiceMix for example). One of the thought leaders in the ESB market is the Mule ESB, which has a good reputation; it is free/open source friendly, community friendly, and lightweight. These characteristics also make Spring Integration attractive. Often, you simply need to talk to another open system, and you don't want to requisition a purchase approval for middleware that's more expensive than some houses!

Each Spring Integration application is completely embedded and needs no server infrastructure. In fact, you could deploy an integration inside another application, perhaps in your web application endpoint. Spring Integration flips the deployment paradigms of most ESBs on their head: you deploy Spring Integration into your application; you don't deploy your application into Spring Integration. There are no start and stop scripts and no ports to guard.

The simplest possible working Spring Integration application is a simple Java `public static void main()` method to bootstrap a Spring context:

```
package com.apress.springrecipes.springintegration;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String [] args){
        String nameOfSpringIntegrationXmlConfigurationFile = args[0];
        ClassPathXmlApplicationContext applicationContext = new ClassPathXmlApplicationContext(
            nameOfSpringIntegrationXmlConfigurationFile) ;
        applicationContext.start();
    }
}
```

You created a standard Spring application context and started it. The contents of the Spring application context will be discussed in subsequent recipes, but it's helpful to see how simple it is. You might decide to hoist the context up in a web application, an EJB container, or anything else you want. Indeed, you can use Spring Integration to power the e-mail polling functionality in a Swing/JavaFX application! It's as lightweight as you want it to be.

In subsequent examples, the configuration shown should be put in an XML file and that XML file referenced as the first parameter when running this class. When the main method runs to completion, your context will start up the Spring Integration bus and start responding to requests on the components configured in the application context's XML.

16-2. Integrating Two Systems Using JMS

Problem

You want to build an integration to connect one application to another using JMS, which provides locational and temporal decoupling on modern middleware for Java applications. You're interested in applying more sophisticated routing and want to isolate your code from the specifics of the origin of the message (in this case, the JMS queue or topic).

Solution

While you can do this by using regular JMS code or EJB's support for message-driven beans (MDBs), or using core Spring's message-driven POJO (MDP) support, all are necessarily coded for handling messages coming specifically from JMS. Your code is tied to JMS. Using an ESB lets you hide the origin of the message from the code that's handling it. You'll use this solution as an easy way to see how a Spring Integration solution can be built. Spring Integration provides an easy way to work with JMS, just as you might use MDPs in the core Spring container. Here, however, you could conceivably replace the JMS middleware with an e-mail, and the code that reacts to the message could stay the same.

How It Works

Building an Message Driven Pojo (MDP) Using Spring Integration

As you recall from Chapter 15, Spring can replace EJB's MDB functionality by using MDPs. This is a powerful solution for anyone wanting to build something that handles messages on a message queue. You'll build an MDP, but you will configure it using Spring Integration's more concise configuration and provide an example of a very rudimentary integration. All this integration will do is take an inbound JMS message (whose payload is of type `Map<String, Object>`).

As with a standard MDP, configuration for the `ConnectionFactory` exists. There's also a lot of other schema required for using the configuration elements available in Spring Integration. Shown following is a configuration file. You can store in on the classpath and pass it in as a parameter to the `Spring ApplicationContext` on creation (as you did in the previous recipe, in the `Main` class).

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

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

    <context:annotation-config/>

    <beans:bean id="connectionFactory"
        class="org.springframework.jms.connection.CachingConnectionFactory">
        <beans:property name="targetConnectionFactory">
            <beans:bean class="org.apache.activemq.ActiveMQConnectionFactory">
                <beans:property name="brokerURL" value="tcp://localhost:61616"/>
            </beans:bean>
        </beans:property>
        <beans:property name="sessionCacheSize" value="10"/>
        <beans:property name="cacheProducers" value="false"/>
    </beans:bean>

    <beans:bean id="inboundHelloWorldJMSPingServiceActivator"
        class="com.apress.springrecipes.springintegration.InboundHelloWorldJMSMessageProcessor"/>

    <channel id="inboundHelloJMSMessageChannel"/>
```

```

<jms:message-driven-channel-adapter
    channel="inboundHelloJMSMessageChannel"
    extract-payload="true"
    connection-factory="connectionFactory"
    destination-name="solution011"/>

<service-activator input-channel="inboundHelloJMSMessageChannel"
ref="inboundHelloWorldJMSPingServiceActivator"/>
</beans:beans>
```

As you can see, the most intimidating part is the schema import! The rest of the code is standard boilerplate. You define a `connectionFactory` exactly as if you were configuring a standard MDP.

Then, you define any beans specific to this solution: in this case a bean that responds to messages coming in to the bus from the message queue, `inboundHelloWorldJMSPingServiceActivator`. A `service-activator` is a generic endpoint in Spring Integration that's used to invoke functionality—whether it be an operation in a service, or some routine in a regular POJO, or anything you want instead—in response to a message sent in on an input channel. Although this will be covered in some detail, it's interesting here only because you are using it to respond to messages. These beans taken together are the collaborators in the solution, and this example is fairly representative of how most integrations look: you define your collaborating components; then you define the configuration using Spring Integration schema that configures the solution itself.

The configuration starts with the `inboundHelloJMSMessageChannel` channel, which tells Spring Integration what to name the point-to-point connection from the message queue to the `service-activator`. You typically define a new channel for every point-to-point connection.

Next is a `jms:message-driven-channel-adapter` configuration element that instructs Spring Integration to send messages coming from the message queue destination `solution011` to Spring Integration `inboundHelloJMSMessageChannel`. An *adapter* is a component that knows how to speak to a specific type of subsystem and translate messages on that subsystem into something that can be used in the Spring Integration bus. Adapters also do the same in reverse, taking messages on the Spring Integration bus and translating them into something a specific subsystem will understand. This is different from a `service-activator` (covered next) in that it's meant to be a general connection between the bus and the foreign endpoint. A `service-activator`, however, only helps you invoke your application's business logic on receipt of a message. What you do in the business logic, connecting to another system or not, is up to you.

The next component, a `service-activator`, listens for messages coming into that channel and invokes the bean referenced by the `ref` attribute, which in this case is the bean defined previously: `inboundHelloWorldJMSPingServiceActivator`.

As you can see, there's quite a bit of configuration, but the only custom Java code needed was the `inboundHelloWorldJMSPingServiceActivator`, which is the part of the solution that Spring can't infer by itself.

```

package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;

import java.util.Map;

public class InboundHelloWorldJMSMessageProcessor {

    private final Logger logger = LoggerFactory.getLogger(InboundHelloWorldJMSMessageProcessor.class);
```

```

@ServiceActivator
public void handleIncomingJmsMessage(Message<Map<String, Object>> inboundJmsMessage)
    throws Throwable {
    Map<String, Object> msg = inboundJmsMessage.getPayload();
    logger.debug("firstName: {}, lastName: {}, id: {}", msg.get("firstName"),
        msg.get("lastName"),
        msg.get("id"));

    // you can imagine what we could do here: put
    // the record into the database, call a webservice,
    // write it to a file, etc, etc
}

}

```

Notice that there is an annotation, `@ServiceActivator`, that tells Spring to configure this component, and this method as the recipient of the message payload from the channel, which is passed to the method as `Message<Map<String, Object>> inboundJmsMessage`. In the previous configuration, `extract-payload="true"`, which tells Spring Integration to take the payload of the message from the JMS queue (in this case, a `Map<String, Object>`) and extract it and pass *that* as the payload of the message that's being moved through Spring Integration's channels as a `org.springframework.messaging.Message<T>`. The Spring Integration Message is not to be confused with the JMS Message interface, although they have some similarities. Had you not specified the `extract-payload` option, the type of payload on the Spring Integration Message interface would have been `javax.jms.Message`. The onus of extracting the payload would have been on you, the developer, but sometimes getting access to that information is useful.

Rewritten to handle unwrapping the `javax.jms.Message`, the example would look a little different:

```

package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;

import javax.jms.MapMessage;

public class InboundHelloWorldJMSMessageProcessor {

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

    @ServiceActivator
    public void handleIncomingJmsMessageWithPayloadNotExtracted(
        Message<javax.jms.Message> msgWithJmsMessageAsPayload) throws Throwable {
        javax.jms.MapMessage jmsMessage = (MapMessage) msgWithJmsMessageAsPayload.getPayload();
        logger.debug("firstName: {}, lastName: {}, id: {}", jmsMessage.getString("firstName"),
            jmsMessage.getString("lastName"),
            jmsMessage.getLong("id"));

        // you can imagine what we could do here: put
        // the record into the database, call a websrvice,
        // write it to a file, etc, etc
    }

}

```

You could have specified the payload type as the type of the parameter passed into the method. If the payload of the message coming from JMS was of type `Cat`, for example, the method prototype could just as well have been `public void handleIncomingJmsMessageWithPayloadNotExtracted(Cat inboundJmsMessage) throws Throwable`. Spring Integration will figure out the right thing to do. In this case, we prefer access to the Spring Integration `Message<T>`, which has header values that can be useful to interrogate.

Also note that you don't need to specify `throws Throwable`. Error handling can be as generic or as specific as you want in Spring Integration.

In the example, you use the `@ServiceActivator` to invoke the functionality where the integration ends. However, you can forward the response from the activation on to the next channel by returning a value from the method. The type of the return value is what will be used to determine the next message sent in the system. If you return a `Message<T>`, that will be sent directly. If you return something other than `Message<T>`, that value will be wrapped as a payload in a `Message<T>` instance, and that will become the next Message that is ultimately sent to the next component in the processing pipeline. This `Message<T>` will be sent on the output channel that's configured on the service-activator. There is no requirement to send a message on the output channel with the same type as the message that came on in the input channel; this is an effective way to *transform* the message type. A service-activator is a very flexible component in which to put hooks to your system and to help mold the integration.

This solution is pretty straightforward, and in terms of configuration for one JMS queue, it's not really a win over straight MDPs because there's an extra level of indirection to overcome. The Spring Integration facilities make building complex integrations easier than Spring Core or EJB3 could because the configuration is centralized. You have a birds-eye view of the entire integration, with routing and processing centralized, so you can better reposition the components in your integration. However, as you'll see, Spring Integration wasn't meant to compete with EJB and Spring Core; it shines at solutions that couldn't naturally be built using EJB3 or Spring Core.

16-3. Interrogating Spring Integration Messages for Context Information

Problem

You want more information about the message coming into the Spring Integration processing pipeline than the type of the message implicitly can give you.

Solution

Interrogate the Spring Integration `Message<T>` for header information specific to the message. These values are enumerated as header values in a map (of type `Map<String, Object>`).

How It Works

Using `MessageHeaders` for Fun and Profit

The Spring Integration `Message<T>` interface is a generic wrapper that contains a pointer to the actual payload of the message as well as to headers that provide contextual message metadata. You can manipulate or augment this metadata to enable/enhance the functionality of components that are downstream, too; for example, when sending a message through e-mail it's useful to specify the `TO/FROM` headers.

Any time you expose a class to the framework to handle some requirement (such as the logic you provide for the service-activator component or a transformer component), there will be some chance to interact with the `Message<T>` and with the message headers. Remember that Spring Integration pushes a `Message<T>` through a processing pipeline. Each component that interfaces with the `Message<T>` instance has to act on it, do something with it, or forward it on. One way of providing information to those components, and of getting information about what's happened in the components up until that point, is to interrogate the `MessageHeaders`.

There are several values that you should be aware of when working with Spring Integration (see Tables 16-1 and 16-2). These constants are exposed on the `org.springframework.messaging.MessageHeaders` interface and `org.springframework.integration.IntegrationMessageHeaderAccessor`.

Table 16-1. Common Headers Found in core Spring Messaging

Constant	Description
ID	This is a unique value assigned to the message by the Spring Integration engine.
TIMESTAMP	Timestamp assigned to the message.
REPLY_CHANNEL	The String name of the channel to which the output of the current component should be sent. This can be overridden.
ERROR_CHANNEL	The String name of the channel to which the output of the current component should be sent if an exception bubbles up into the runtime. This can be overridden.
CONTENT_TYPE	The content type (mimetype) of the message, mainly used for web sockets messages.

Table 16-2. Common Headers Found in Spring Integration

Constant	Description
CORRELATION_ID	This is optional. It is used by some components (such as aggregators) to group messages together in some sort of processing pipeline.
EXPIRATION_DATE	Used by some components as a threshold for processing after which a component can wait no longer in processing.
PRIORITY	The priority of the message, higher numbers indicate a higher priority.
SEQUENCE_NUMBER	The order in which the message is to be sequenced; typically used with a sequencer.
SEQUENCE_SIZE	The size of the sequence so that an aggregator can know when to stop waiting for more messages and move forward. This is useful in implementing “join” functionality.

Next to the headers defined by Spring Messaging there are also some commonly used headers in Spring Integration, these are defined in the `org.springframework.integration.IntegrationMessageHeaderAccessor` class (see Table 16-2).

Some header values are specific to the type of the source message’s payload; for example, payloads sourced from a file on the file system are different from those coming in from a JMS queue, which are different from messages coming from an e-mail system. These different components are typically packaged in their own JARs, and there’s usually some class that provides constants for accessing these headers. An example of component-specific headers are the constants defined for files on `org.springframework.integration.file.FileHeaders`: `FILENAME` and `PREFIX`. Naturally, when in doubt, you can just enumerate the values manually because the headers are just a `java.util.Map` instance.

```

package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageHeaders;

import java.io.File;
import java.util.Map;

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

    @ServiceActivator
    public void interrogateMessage(Message<File> message) {
        MessageHeaders headers = message.getHeaders();
        for (Map.Entry<String, Object> header : headers.entrySet()) {
            logger.debug("{} : {}", header.getKey(), header.getValue());
        }
    }
}

```

These headers let you interrogate the specific features of these messages without surfacing them as a concrete interface dependency if you don't want them. They can also be used to help processing and allow you to specify custom metadata to downstream components. The act of providing extra data for the benefit of a downstream component is called *message enrichment*. Message enrichment is when you take the headers of a given Message and add to them, usually to the benefit of components in the processing pipeline downstream. You might imagine processing a message to add a customer to a customer relationship management (CRM) system that makes a call to a third-party web site to establish credit ratings. This credit is added to the headers so the component downstream is tasked with either adding the customer or rejecting it can make its decisions.

Another way to get access to header metadata is to simply have it passed as parameters to your component's method. You simply annotate the parameter with the @Header annotation, and Spring Integration will take care of the rest.

```

package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.Header;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.file.FileHeaders;
import org.springframework.messaging.MessageHeaders;

import java.io.File;

```

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

    @ServiceActivator
    public void interrogateMessage(
        @Header(MessageHeaders.ID) String uuid,
        @Header(FileHeaders.FILENAME) String fileName, File file) {
        logger.debug("the id of the message is {}, and name of the file payload is {}", uuid,
        fileName);
    }
}
```

You can also have Spring Integration simply pass the `Map<String, Object>`:

```
package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.Header;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.file.FileHeaders;
import org.springframework.messaging.MessageHeaders;

import java.io.File;
import java.util.Map;

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

    @ServiceActivator
    public void interrogateMessage(
        @Header(MessageHeaders.ID) Map<String, Object> headers, File file) {
        logger.debug("the id of the message is {}, and name of the file payload is {}", headers.get(MessageHeaders.ID), headers.get(FileHeaders.FILENAME));
    }
}
```

16-4. Integrating Two Systems Using a File System

Problem

You want to build a solution that takes files on a well-known, shared file system and uses them as the conduit for integration with another system. An example might be that your application produces a comma-separated value (CSV) dump of all the customers added to a system every hour. The company's third-party financial system is updated with these sales by a process that checks a shared folder, mounted over a network file system, and processes the CSV records. What's required is a way to treat the presence of a new file as an event on the bus.

Solution

You have an idea of how this could be built by using standard techniques, but you want something more elegant. Let Spring Integration isolate you from the event-driven nature of the file system and from the file input/output requirements and instead let's use it to focus on writing the code that deals with the `java.io.File` payload itself. With this approach, you can write unit-testable code that accepts an input and responds by adding the customers to the financial system. When the functionality is finished, you configure it in the Spring Integration pipeline and let Spring Integration invoke your functionality whenever a new file is recognized on the file system. This is an example of an event-driven architecture (EDA). EDAs let you ignore how an event was generated and focus instead on reacting to them, in much the same way that event-driven GUIs let you change the focus of your code from controlling how a user triggers an action to actually reacting to the invocation itself. Spring Integration makes it a natural approach for loosely coupled solutions. In fact, this code should look very similar to the solution you built for the JMS queue because it's just another class that takes a parameter (a Spring Integration Message<T>, and a parameter of the same type as the payload of the message, and so on).

How It Works

Concerns in Dealing with a File System

Building a solution to talk to JMS is old hat. Instead, let's consider what building a solution using a shared file system might look like. Imagine how to build it without an ESB solution. You need some mechanism by which to poll the file system periodically and detect new files. Perhaps Quartz and some sort of cache? You need something to read in these files quickly and then pass the payload to your processing logic efficiently. Finally, your system needs to work with that payload.

Spring Integration frees you from all that infrastructure code; all you need to do is configure it. There are some issues with dealing with file-system-based processing, however, that are up to you to resolve. Behind the scenes, Spring Integration is still dealing with polling the file system and detecting new files. It can't possibly have a semantically correct idea for your application of when a file is "completely" written, and thus providing a way around that is up to you.

Several approaches exist. You might write out a file and then write another zero-byte file. The presence of that file would mean it's safe to assume that the real payload is present. Configure Spring Integration to look for that file. If it finds it, it knows that there's another file (perhaps with the same name and a different file extension?) and that it can start reading it/working with it. Another solution along the same line is to have the client ("producer") write the file to the directory using a name that the glob pattern Spring Integration is using to poll the directory won't detect. Then, when it's finished writing, issue an `mv` command if you trust your file system to do the right thing there.

Let's revisit the first solution, but this time with a file-based adapter. The configuration looks conceptually the same as before, except the configuration for the adapter has changed, and with that has gone a lot of the configuration for the JMS adapter, like the connection factory. Instead, you tell Spring Integration about a different source from whence messages will come: the file system.

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

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

```

http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://www.springframework.org/schema/integration/file
http://www.springframework.org/schema/integration/file/spring-integration-file.xsd">

<context:annotation-config>

<beans:bean id="inboundHelloWorldFileMessageProcessor"
    class="com.apress.springrecipes.springintegration.InboundHelloWorldFileMessageProcessor"/>

<channel id="inboundFileChannel"/>

<file:inbound-channel-adapter directory="${user.home}/inboundFiles/new/"
    channel="inboundFileChannel"
    filename-pattern="^new.*csv"
    <poller default="true" fixed-rate="10" time-unit="SECONDS" />
</file:inbound-channel-adapter>
<service-activator input-channel="inboundFileChannel" ~
ref="inboundHelloWorldFileMessageProcessor"/>

</beans:beans>

```

This is nothing you haven't already seen, really. The code for `file:inbound-channel-adapter` is the only new element, and it comes with its own schema, which is in the prologue for the XML itself.

The code for the `service-activator` has changed to reflect the fact that you're expecting a message containing a message of type `Message<java.io.File>`.

```

package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;

import java.io.File;

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

    @ServiceActivator
    public void handleIncomingFileMessage(Message<File> inboundJmsMessage)
        throws Throwable {
        File filePayload = inboundJmsMessage.getPayload();
        logger.debug("absolute path: {}, size: {}", filePayload.getAbsolutePath(), filePayload.
length());
    }
}

```

16-5. Transforming a Message from One Type to Another

Problem

You want to send a message into the bus and transform it before working with it further. Usually, this is done to adapt the message to the requirements of a component downstream. You might also want to transform a message by enriching it—adding extra headers or augmenting the payload so that components downstream in the processing pipeline can benefit from it.

Solution

Use a `transformer` component to take a `Message<T>` of a payload and send the `Message<T>` out with a payload of a different type. You can also use the transformer to add extra headers or update the values of headers for the benefit of components downstream in the processing pipeline.

How It Works

Spring Integration provides a `transformer` message endpoint to permit the augmentation of the message headers or the transformation of the message itself. In Spring Integration, components are chained together, and output from one component is returned by way of the method invoked for that component. The return value of the method is passed out on the “reply channel” for the component to the next component, which receives it as an input parameter.

A `transformer` component lets you change the type of the object being returned or add extra headers and that updated object is what is passed to the next component in the chain.

Modifying a Message's Payload

The configuration of a transformer component is very much in keeping with everything you've seen so far:

```
package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Transformer;
import org.springframework.messaging.Message;

import java.util.Map;

public class InboundJMSMessageToCustomerTransformer {

    @Transformer
    public Customer transformJMSMapToCustomer(
        Message<Map<String, Object>> inboundSpringIntegrationMessage) {
        Map<String, Object> jmsMessagePayload = inboundSpringIntegrationMessage.getPayload();
        Customer customer = new Customer();
        customer.setFirstName((String) jmsMessagePayload.get("firstName"));
        customer.setLastName((String) jmsMessagePayload.get("lastName"));
        customer.setId((Long) jmsMessagePayload.get("id"));
        return customer;
    }
}
```

Nothing terribly complex is happening here: a `Message<T>` of type `Map<String, Object>` is passed in. The values are manually extracted and used to build an object of type `Customer`. The `Customer` object is returned, which has the effect of passing it out on the reply channel for this component. The next component in the configuration will receive this object as its input `Message<T>`.

The solution is mostly the same as you've seen, but there is a new `transformer` element:

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  ...
>

<context:annotation-config/>

<beans:bean id="connectionFactory"
  class="org.springframework.jms.connection.CachingConnectionFactory">
  <beans:property name="targetConnectionFactory">
    <beans:bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <beans:property name="brokerURL" value="tcp://localhost:61616"/>
    </beans:bean>
  </beans:property>
  <beans:property name="sessionCacheSize" value="10"/>
  <beans:property name="cacheProducers" value="false"/>
</beans:bean>
<beans:bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <beans:property name="connectionFactory" ref="connectionFactory"/>
</beans:bean>

<beans:bean id="inboundJMSMessageToCustomerTransformer"
  class="com.apress.springrecipes.springintegration.InboundJMSMessageToCustomerTransformer"/>

  <beans:bean id="inboundCustomerServiceActivator" -->
    class="com.apress.springrecipes.springintegration.InboundCustomerServiceActivator"/>
    <channel id="inboundHelloJMSMessageChannel"/>
    <channel id="inboundCustomerChannel"/>
    <jms:message-driven-channel-adapter channel="inboundHelloJMSMessageChannel"
      extract-payload="true" connection-factory="connectionFactory" destination-name="solution015"/>

<transformer input-channel="inboundHelloJMSMessageChannel"
  ref="inboundJMSMessageToCustomerTransformer"
  output-channel="inboundCustomerChannel"/>

  <service-activator input-channel="inboundCustomerChannel" ref="inboundCustomerServiceActivator" />

</beans:beans>
```

Here, you're also specifying an `output-channel` attribute on the component, which tells a component on what channel to send the component's response output, in this case, the `Customer`.

The code in the next component can now declare a dependency on the `Customer` interface with impunity. You can, with transformers, receive messages from any number of sources and transform into a `Customer` so that you can reuse the `InboundCustomerServiceActivator`:

```
package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;

public class InboundCustomerServiceActivator {
    private static final Logger logger = LoggerFactory.getLogger(InboundCustomerServiceActivator.class);

    @ServiceActivator
    public void doSomethingWithCustomer(Message<Customer> customerMessage) {
        Customer customer = customerMessage.getPayload();
        logger.debug("id={}, firstName: {}, lastName: {}",
                    customer.getId(), customer.getFirstName(), customer.getLastName());
    }
}
```

Modifying a Message's Headers

Sometimes changing a message's payload isn't enough. Sometimes you want to update the payload as well as the headers. Doing this is slightly more interesting because it involves using the `MessageBuilder<T>` class, which allows you to create new `Message<T>` objects with any specified payload and any specified header data. The XML configuration is identical in this case.

```
package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.core.Message;
import org.springframework.integration.message.MessageBuilder;

import java.util.Map;

public class InboundJMSMessageToCustomerTransformer {
    @Transformer
    public Message<Customer> transformJMSMapToCustomer(
        Message<Map<String, Object>> inboundSpringIntegrationMessage) {
        Map<String, Object> jmsMessagePayload =
            inboundSpringIntegrationMessage.getPayload();
        Customer customer = new Customer();
        customer.setFirstName((String) jmsMessagePayload.get("firstName"));
        customer.setLastName((String) jmsMessagePayload.get("lastName"));
        customer.setId((Long) jmsMessagePayload.get("id"));
        return MessageBuilder.withPayload(customer)
            .copyHeadersIfAbsent( inboundSpringIntegrationMessage.getHeaders())
            .setHeaderIfAbsent("randomlySelectedForSurvey", Math.random() > .5)
            .build();
    }
}
```

As before, this code is simply a method with an input and an output. The output is constructed dynamically using `MessageBuilder<T>` to create a message that has the same payload as the input message as well as copy the existing headers and adds an extra header: `randomlySelectedForSurvey`.

16-6. Error Handling Using Spring Integration

Problem

Spring Integration brings together systems distributed across different nodes; computers; and services, protocol, and language stacks. Indeed, a Spring Integration solution might not even finish in remotely the same time period as when it started. Exception handling, then, can never be as simple as a language-level try/catch block in a single thread for any component with asynchronous behavior. This implies that many of the kinds of solutions you're likely to build, with channels and queues of any kind, need a way of signaling an error that is distributed and natural to the component that created the error. Thus, an error might get sent over a JMS queue on a different continent, or in process, on a queue in a different thread.

Solution

Use Spring Integration's support for an error channel, both implicit and explicitly via code.

How It Works

Spring Integration provides the ability to catch exceptions and send them to an error channel of your choosing. By default, it's a global channel called `errorChannel`. By default Spring Integration registers a `LoggingHandler` to this channel, which does nothing more than log the exception and stacktrace. To make this work we have to tell the message-driven-channel-adapter that we want the error to be sent to the `errorChannel`, we can do this by configuring the `error-channel` attribute.

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  ...
  >
  <context:annotation-config/>

  <jms:message-driven-channel-adapter channel="inboundHelloJMSMessageChannel"
    extract-payload="true"
    connection-factory="connectionFactory"
    destination-name="solution015" error-channel="errorChannel"
  />

</beans:beans>
```

Custom Handler to handle Exception

Of course you can also have components subscribe to messages from this channel to override the exception handling behavior. You can create a class that will be invoked whenever a message comes in on the `errorChannel` channel:

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
...
>
<context:annotation-config/>

<beans:bean id="defaultErrorHandlerServiceActivator" class="com.apress.springrecipes.springintegration.DefaultErrorHandlerServiceActivator"/>
<service-activator input-channel="errorChannel" ref="defaultErrorHandlerServiceActivator"/>
</beans:beans>
```

The Java code is exactly as you'd expect it to be. Of course, the component that receives the error message from the `errorChannel` doesn't need to be a service-activator. We just use it for convenience here. The code for the following service-activator depicts some of the machinations you might go through to build a handler for the `errorChannel`:

```
package com.apress.springrecipes.springintegration;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.messaging.Message;
import org.springframework.messaging.MessagingException;

public class DefaultErrorHandlerServiceActivator {
    private static final Logger logger =
        LoggerFactory.getLogger(DefaultErrorHandlerServiceActivator.class);

    @ServiceActivator
    public void handleThrowable(Message<Throwable> errorMessage)
        throws Throwable {
        Throwable throwable = errorMessage.getPayload();
        logger.debug("Message: {}", throwable.getMessage(), throwable);

        if (throwable instanceof MessagingException) {
            Message<?> failedMessage = ((MessagingException) throwable).getFailedMessage();

            if (failedMessage != null) {
                // do something with the original message
            }
        } else {
            // it's something that was thrown in the execution of code in some component you created
        }
    }
}
```

All errors thrown from Spring Integration components will be a subclass of `MessagingException`. `MessagingException` carries a pointer to the original `Message` that caused an error, which you can dissect for more context information. In the example, you're doing a nasty `instanceof`. Clearly, being able to delegate to custom exception handlers based on the type of exception would be useful.

Routing to Custom Handlers Based on the Type of Exception

Sometimes, more specific error handling is required. One way to discriminate by `Exception` type is to use the `org.springframework.integration.router.ErrorMessageExceptionRouter` class. In the following code, this router is configured as an exception-type-router, which in turn listens to `errorChannel`. It then splinters off, using the type of the exception as the predicate in determining which channel should get the results.

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

<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  ...
>
  <context:annotation-config/>
  <channel id="customErrorChannelForMyCustomException"/>
  <exception-type-router input-channel="errorChannel">
    <mapping exception-type="com.apress.springrecipes.springintegration.MyCustomException"
      channel="customErrorChannelForMyCustomException"/>
  </exception-type-router>
</beans:beans>
```

Building a Solution with Multiple Error Channels

The preceding example might work fine for simple cases, but often different integrations require different error-handling approaches, which implies that sending all the errors to the same channel can eventually lead to a large switch-laden class that's too complex to maintain. Instead, it's better to selectively route error messages to the error channel most appropriate to each integration. This avoids centralizing all error handling. One way to do that is to explicitly specify on what channel errors for a given integration should go. The following example shows a component (`service-activator`) that upon receiving a message, adds a header indicating the name of the error channel. Spring Integration will use that header and forward errors encountered in the processing of this message to that channel.

```
package com.apress.springrecipes.springintegration;

import org.apache.log4j.Logger;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.core.Message;
import org.springframework.integration.core.MessageHeaders;
import org.springframework.integration.message.MessageBuilder;

public class ServiceActivatorThatSpecifiesErrorChannel {
    private static final Logger logger = Logger.getLogger(
        ServiceActivatorThatSpecifiesErrorChannel.class);
```

```

@ServiceActivator
public Message<?> startIntegrationFlow(Message<?> firstMessage)
    throws Throwable {
    return MessageBuilder.fromMessage(firstMessage).
        setHeaderIfAbsent( MessageHeaders.ERROR_CHANNEL,
        "errorChannelForMySolution").build();
}
}

```

Thus, all errors that come from the integration in which this component is used will be directed to `customErrorChannel`, to which you can subscribe any component you like.

16-7. Forking Integration Control: Splitters and Aggregators

Problem

You want to fork the process flow from one component to many, either all at once or to a single one based on a predicate condition.

Solution

You can use a splitter component (and maybe its cohort, the aggregator component) to fork and join (respectively) control of processing.

How It Works

One of the fundamental cornerstones of an ESB is routing. You've seen how components can be chained together to create sequences in which progression is mostly linear. Some solutions require the capability to split a message into many constituent parts. One reason this might be is that some problems are parallel in nature and don't depend on each other in order to complete. You should strive to achieve the efficiencies of parallelism wherever possible.

Using a Splitter

It's often useful to divide large payloads into separate messages with separate processing flows. In Spring Integration, this is accomplished by using a splitter component. A splitter takes an input message and asks you, the user of the component, on what basis it should split the `Message<T>`: you're responsible for providing the split functionality. Once you've told Spring Integration how to split a `Message<T>`, it forwards each result out on the `output-channel` of the splitter component. In a few cases, Spring Integration ships with useful splitters that require no customization. One example is the splitter provided to partition an XML payload along an XPath query, `XPathMessageSplitter`.

One example of a useful application of a splitter might be a text file with rows of data, each of which must be processed. Your goal is to be able to submit each row to a service that will handle the processing. What's required is a way to extract each row and forward each row as a new `Message<T>`.

The configuration for such a solution looks like this:

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

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

    <context:annotation-config/>

    <poller id="poller" default="true">
        <interval-trigger interval="1000"/>
    </poller>
    <beans:bean id="fileSplitter"
        class="com.apress.springrecipes.springintegration.CustomerBatchFileSplitter"/>
    <beans:bean id="customerDeletionServiceActivator"
        class="com.apress.springrecipes.springintegration.CustomerDeletionServiceActivator"/>
    <channel id="customerBatchChannel"/>
    <channel id="customerIdChannel"/>
    <file:inbound-channel-adapter
        directory="file:${user.home}/customerstoremove/new/"
        channel="customerBatchChannel"  filename-pattern="customerstoremove-*.*txt"/>

    <splitter input-channel="customerBatchChannel"
        ref="fileSplitter" output-channel="customerIdChannel" />

    <service-activator input-channel="customerIdChannel"
        ref="customerDeletionServiceActivator"/>

</beans:beans>
```

The configuration for this is not terribly different from the previous solutions. The Java code is just about the same as well, except that the return type of the method annotated by the `@Splitter` annotation is of type `java.util.Collection`.

```

package com.apress.springrecipes.springintegration;

import org.apache.commons.io.IOUtils;
import org.springframework.integration.annotation.Splitter;
import java.io.File;
import java.io.FileReader;
import java.io.Reader;
import java.util.Collection;

public class CustomerBatchFileSplitter {
    @Splitter
    public Collection<String> splitAFile(File file) throws Throwable {
        Reader reader = new FileReader(file);
        Collection<String> lines = IOUtils.readLines(reader);
        IOUtils.closeQuietly(reader);
        return lines;
    }
}

```

A message payload is passed in as a `java.io.File` and the contents are read. The result (a collection or array value; in this case, a `Collection<String>`) is returned. Spring Integration executes a kind of for each on the results, sending each value in the collection out on the output-channel configured for the splitter. Often, you split messages so that the individual pieces can be forwarded to processing that's more focused. Because the message is more manageable, the processing requirements are damped. This is true in many different architectures: in map/reduce solutions tasks are split and then processed in parallel, and the fork/join constructs in a BPM system let control flow proceed in parallel so that the total work product can be achieved quicker.

Using Aggregators

At some point you'll need to do the reverse: combine many messages into one, and create a single result that can be returned on the output-channel. An `@Aggregator` collects a series of messages (based on some correlation that you help Spring Integration make between the messages) and publishes a single message to the components downstream. Suppose that you know that you're expecting 22 different messages from 22 actors in the system, but you don't know when. This is similar to a company that auctions off a contract and collects all the bids from different vendors before choosing the ultimate vendor. The company can't accept a bid until all bids have been received from all companies. Otherwise, there's the risk of prematurely signing a contract that would not be in the best interest of the company. An aggregator is perfect for building this type of logic.

There are many ways for Spring Integration to correlate incoming messages. To determine how many messages to read until it can stop, it uses the class `SequenceSizeCompletionStrategy`, which reads a well-known header value (aggregators are often used after a splitter). Thus, the default header value is provided by the `splitter`, though there's nothing stopping you from creating the header parameters yourself) to calculate how many it should look for and to note the index of the message relative to the expected total count (e.g., 3/22).

For correlation when you might not have a size but know that you're expecting messages that share a common header value within a known time, Spring Integration provides the `HeaderAttributeCorrelationStrategy`. In this way, it knows that all messages with that value are from the same group, in the same way that your last name identifies you as being part of a larger group.

Let's revisit the last example. Suppose that the file was split (by lines, each belonging to a new customer) and subsequently processed. You now want to reunite the customers and do some cleanup with everyone at the same time. In this example, you use the default completion-strategy and correlation-strategy. The only custom logic is

a POJO with an `@Aggregator` annotation on a method expecting a collection of `Message<T>` objects. It could, of course, be a collection of `Customer` objects, because they are what you're expecting as output from the previous splitter. You return on the reply channel a `Message<T>` that has the entire collection as its payload:

```
<beans:bean id="customAggregator" class="com.apress.springrecipes.<-->
springintegration.MessagePayloadAggregator"/>
...
<channel id="deletedCustomerChannel"/>
<channel id="summaryChannel"/>
...
<aggregator input-channel="deletedCustomerChannel"
ref="customAggregator"
output-channel="summaryChannel" />

<service-activator input-channel=" summaryChannel "
ref="summaryServiceActivator"/>
```

The Java code is quite simple:

```
package com.apress.springrecipes.springintegration;

import org.springframework.integration.annotation.Aggregator;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.messaging.Message;

import java.util.ArrayList;
import java.util.List;

public class MessagePayloadAggregator {

    @Aggregator
    public Message<?> joinMessages(List<Message<Customer>>customers) {
        if (customers.size() > 0) {
            List<Customer> payload = new ArrayList<>(customers.size());
            for(Message<Customer> customer : customers) {
                payload.add(customer.getPayload());
            }
            return MessageBuilder.withPayload(payload).copyHeadersIfAbsent(customers.get(0).
getHeaders()).build();
        }
        return null;
    }
}
```

16-8. Conditional Routing with Routers

Problem

You want to conditionally move a message through different processes based on some criteria. This is the EAI equivalent to an if/else branch.

Solution

You can use a `router` component to alter the processing flow based on some predicate. You can also use a router to multicast a message to many subscribers (as you did with the `splitter`).

How It Works

With a router you can specify a known list of channels on which the incoming `Message` should be passed. This has some powerful implications. It means you can change the flow of a process conditionally, and it also means that you can forward a `Message` to as many (or as few) channels as you want. There are some convenient default routers available to fill common needs, such as payload-type-based routing (`PayloadTypeRouter`) and routing to a group or list of channels (`RecipientListRouter`).

Imagine, for example, a processing pipeline that routes customers with high credit scores to one service and customers with lower credit scores to another process in which the information is queued up for a human audit and verification cycle. The configuration is, as usual, very straightforward. In the following example, you show the configuration. One `router` element, which in turn delegates the routing logic to a class, is `CustomerCreditScoreRouter`.

```
<beans:bean id="customerCreditScoreRouter"-
    class="com.apress.springrecipes.springintegration.CustomerCreditScoreRouter"/>
...
<channel id="safeCustomerChannel"/>
<channel id="riskyCustomerChannel"/>
...
<router input-channel="customerIdChannel" ref="customerCreditScoreRouter"/>
```

The Java code is similarly approachable. It feels a lot like a workflow engine's conditional element, or even a JSF backing-bean method, in that it extricates the routing logic into the XML configuration, away from code, delaying the decision until runtime. In the example, the `Strings` returned are the names of the channels on which the `Message` should pass.

```
package com.apress.springrecipes.springintegration;
import org.springframework.integration.annotation.Router;

public class CustomerCreditScoreRouter {
    @Router
    public String routeByCustomerCreditScore(Customer customer) {
        if (customer.getCreditScore() > 770) {
            return "safeCustomerChannel";
        } else {
            return "riskyCustomerChannel";
        }
    }
}
```

If you decide that you'd rather not let the `Message<T>` pass and want to arrest processing, you can return `null` instead of a `String`.

16-9. Staging Events Using Spring Batch

Problem

You have a file with a million records in it. This file's too big to handle as one event; it's far more natural to react to each row as an event.

Solution

Spring Batch works very well with these types of solutions. It allows you to take an input file or a payload and reliably, and systematically, decompose it into events that an ESB can work with.

How It Works

Spring Integration does support reading files into the bus, and Spring Batch does support providing custom, unique endpoints for data. However, just like Mom always says, “just because you can, it doesn’t mean you *should*.”

Although it seems as if there’s a lot of overlap here, it turns out that there is a distinction (albeit a fine one). While both systems will work with files and message queues, or anything else you could conceivably write code to talk to, Spring Integration doesn’t do well with large payloads because it’s hard to deal with something as large as a file with a million rows that might require hours of work as an *event*. That’s simply too big a burden for an ESB. At that point, the term *event* has no meaning. A million records in a CSV file isn’t an event on a bus, it’s a file with a million records, each of which might in turn *be* events. It’s a subtle distinction.

A file with a million rows needs to be decomposed into smaller events. Spring Batch can help here: it allows you to systematically read through, apply validations, and optionally skip and retry invalid records. The processing can begin on an ESB such as Spring Integration. Spring Batch and Spring Integration can be used together to build truly scalable decoupled systems.

Staged event-driven architecture (SEDA) is an architecture style that deals with this sort of processing situation. In SEDA, you dampen the load on components of the architecture by staging it in queues, and let advance only what the components downstream can handle. Put another way, imagine video processing. If you ran a site with a million users uploading video that in turn needed to be transcoded and you only had ten servers, your system would fail if your system attempted to process each video as soon as it received the uploaded video. Transcoding can take hours and pegs a CPU (or multiple CPUs!) while it works. The most sensible thing to do would be to store the file and then, as capacity permits, process each one. In this way, the load on the nodes that handle transcoding is managed. There’s always only enough work to keep the machine humming, but not overrun.

Similarly, no processing system (such as an ESB) can deal with a million records at once efficiently. Strive to decompose bigger events and messages into smaller ones. Let’s imagine a hypothetical solution designed to accommodate a drop of batch files representing hourly sales destined for fulfillment. The batch files are dropped onto a mount that Spring Integration is monitoring. Spring Integration kicks off processing as soon as it sees a new file. Spring Integration tells Spring Batch about the file and launches a Spring Batch job asynchronously.

Spring Batch reads the file, transforms the records into objects, and writes the output to a JMS topic with a key correlating the original batch to the JMS message. Naturally, this takes half a day to get done, but it does get done. Spring Integration, completely unaware that the job it started half a day ago is now finished, begins popping messages off the topic, one by one. Processing to fulfill the records would begin. Simple processing involving multiple components might begin on the ESB.

If fulfillment is a long-lived process with a long-lived, conversational state involving many actors, perhaps the fulfillment for each record could be farmed to a BPM engine. The BPM engine would thread together the different actors and work lists, allow work to continue over the course of days instead of the small millisecond timeframes Spring Integration is more geared to. In this example, we talked about using Spring Batch as a springboard to dampen the load for components downstream. In this case, the component downstream was again a Spring Integration process that took the work and set it up to be funneled into a BPM engine where final processing could begin.

Spring Integration could use directory polling as a trigger to start a batch job and for the start supply the name of the file to process. To launch a job from Spring Integration, Spring Batch provides the `JobLaunchingMessageHandler` (this is part of the `spring-batch-integration` module). This class takes a `JobLaunchRequest` to determine which job with which parameters to start. You have to create a transformer to change the incoming `Message<File>` to a `JobLaunchRequest`.

The transformer could look like the following

```
package com.apress.springrecipes.springintegration;
```

```

import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobParametersBuilder;
import org.springframework.batch.integration.launch.JobLaunchRequest;
import org.springframework.integration.annotation.Transformer;
import org.springframework.messaging.Message;

import java.io.File;

public class FileToJobLaunchRequestTransformer {

    private final Job job;
    private final String fileParameterName;

    public FileToJobLaunchRequestTransformer(Job job, String fileParameterName) {
        this.job=job;
        this.fileParameterName=fileParameterName;
    }

    @Transformer
    public JobLaunchRequest transform(Message<File> message) throws Exception {
        JobParametersBuilder builder = new JobParametersBuilder();
        builder.addString(fileParameterName, message.getPayload().getAbsolutePath());
        return new JobLaunchRequest(job, builder.toJobParameters());
    }
}

```

The transformer needs a Job and a filename parameter to be constructed, this parameter is used in the Spring Batch job to determine which file needs to be loaded. The incoming message is transformed in a JobLaunchRequest using the full name of the file as a parameter value. This request can be used to launch a batch job.

To wire everything together the following configuration could be used (note the Spring Batch setup is missing here see chapter 13 for information on setting up Spring Batch).

```

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

    <channel id="customerBatchChannel"/>

    <file:inbound-channel-adapter
        directory="file:${user.home}/customerstoremove/new/"
        channel="customerBatchChannel"  filename-pattern="newCustomers-*.*txt"/>

    <beans:bean id="transformer"
        class="com.apress.springrecipes.springintegration.FileToJobLaunchRequestTransformer">
        <beans:constructor-arg index="0" value="importCustomersJob" />
    
```

```

<beans:constructor-arg index="1" value="filename" />
</beans:bean>

<beans:bean id="jobLaunchingMessageHandler" class="org.springframework.batch.integration.launch.JobLaunchingMessageHandler">
    <beans:constructor-arg ref="jobLauncher" />
</beans:bean>

<chain input-channel="customerBatchChannel">
    <transformer ref="transformer" />
    <service-activator ref="jobLaunchingMessageHandler" />
</chain>
</beans:beans>

```

The `FileToJobLaunchRequestTransformer` is configured as well as the `JobLaunchingMessageHandler`. A `file-inbound-channel-adapter` is used to poll for files, when a file is detected a message is placed on the `customerBatchChannel`. A chain is configured to listen to the `customerBatchChannel` when a message is received it is first transformed and next passed on to the `JobLaunchingMessageHandler`.

Now a batch job will be launched to process the file. A typical job would probably use a `FlatFileItemReader` to actually read the file, `JmsItemWriter` could be used to write messages per read row on a topic. In Spring integration a `jms-inbound-channel-adapter` could be used to receive to messages and process them.

16-10. Using Gateways

Problem

You want to expose an interface to clients of your service, without betraying the fact that your service is implemented in terms of messaging middleware.

Solution

Use a gateway—a pattern from the classic book *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley, 2004) that enjoys rich support in Spring Integration.

How It Works

A *gateway* is a distinct animal, similar to a lot of other patterns but ultimately different enough to warrant its own consideration. You used adapters in previous examples to enable two systems to speak in terms of foreign, loosely coupled, middleware components. This foreign component can be anything: the file system, JMS queues/topics, Twitter, and so on.

You also know what a *façade* is, serving to abstract away the functionality of other components in an abbreviated interface to provide courser functionality. You might use a façade to build an interface oriented around vacation planning that in turn abstracts away the minutiae of using a car rental, hotel reservation, and airline reservation system.

You build a gateway, on the other hand, to provide an interface for your system that insulates clients from the middleware or messaging in your system, so that they're not dependent on JMS or Spring Integration APIs, for example. A gateway allows you to express compile time constraints on the inputs and outputs of your system.

There are several reasons why you might want to do this. First, it's cleaner. If you have the latitude to insist that clients comply with an interface, this is a good way to provide that interface. Your use of middleware can be an implementation detail. Perhaps your architectures messaging middleware can be to exploit the performance increases had by leveraging asynchronous messaging, but you didn't intend for those performance gains to come at the cost of a precise, explicit external facing interface.

This feature—the capability to hide messaging behind a POJO interface—is very interesting and has been the focus of several other projects. Lingo, a project from Codehaus.org that is no longer under active development, had

such a feature that was specific to JMS and the Java EE Connector Architecture (JCA—it was originally used to talk about the Java Cryptography Architecture, but is more commonly used for The Java EE Connector Architecture now). Since then, the developers have moved on to work on Apache Camel.

In this recipe, you'll explore Spring Integration's core support for messaging gateways and explore its support for message exchange patterns. Then, you'll see how to completely remove implementation details from the client-facing interface.

SimpleMessagingGateway

The most fundamental support for gateways comes from the Spring Integration class `SimpleMessagingGateway`. The class provides the ability to specify a channel on which requests should be sent and a channel on which responses are expected. Finally, the channel on which replies are sent can be specified. This gives you the ability to express in-out and in-only patterns on top of your existing messaging systems. This class supports working in terms of payloads, isolating you from the gory details of the messages being sent and received. This is already one level of abstraction. You could, conceivably, use the `SimpleMessagingGateway` and Spring Integration's concept of channels to interface with file systems, JMS, e-mail, or any other system and deal simply with payloads and channels. There are implementations already provided for you to support some of these common endpoints such as web services and JMS.

Let's look at using a generic messaging gateway. In this example, you'll send messages to a service-activator and then receive the response. You manually interface with the `SimpleMessageGateway` so that you can see how convenient it is.

```
package com.apress.springrecipes.springintegration;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.core.MessageChannel;
import org.springframework.integration.gateway.SimpleMessagingGateway;

public class SimpleMessagingGatewayExample {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx =
            new ClassPathXmlApplicationContext("solution042.xml");
        MessageChannel request = (MessageChannel) ctx.getBean("request");
        MessageChannel response = (MessageChannel) ctx.getBean("response");
        SimpleMessagingGateway msgGateway = new SimpleMessagingGateway();
        msgGateway.setRequestChannel(request);
        msgGateway.setReplyChannel(response);
        Number result = (Number) msgGateway.sendAndReceive(new Operands(22, 4));
        System.out.println("Result: " + result.floatValue());
    }
}
```

The interface is very straightforward. The `SimpleMessagingGateway` needs a request and a response channel, and it coordinates the rest. In this case, you're doing nothing but forwarding the request to a service-activator, which in turn adds the operands and sends them out on the reply channel. The configuration XML is sparse because most of the work is done in those five lines of Java code.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans ... >
    <beans:bean id="additionService" class="com.apress.springrecipes.springintegration.
AdditionService" />
    <channel id="request" />
```

```

<channel id="response" />
<service-activator ref="additionService"
    method="add"
    input-channel="request"
    output-channel="response" />
</beans>
```

Breaking the Interface Dependency

The previous example demonstrates what's happening behind the scenes. You're dealing only with Spring Integration interfaces and are isolated from the nuances of the endpoints. However, there are still plenty of inferred constraints that a client might easily fail to comply with. The simplest solution is to hide the messaging behind an interface. Let's look at building a fictional hotel reservation search engine. Searching for a hotel might take a long time, and ideally processing should be offloaded to a separate server. An ideal solution is JMS because you could implement the aggressive consumer pattern and scale simply by adding more consumers. The client would still block waiting for the result, in this example, but the server(s) would not be overloaded or in a blocking state.

You'll build two Spring Integration solutions. One for the client (which will in turn contain the gateway) and one for the service itself, which, presumably, is on a separate host connected to the client only by way of well-known message queues.

Let's look at the client configuration first. The first thing that the client configuration does is import a shared application context (to save typing if nothing else) that declares a JMS connection factory that you reference in the client and service application contexts. (We won't repeat all of that here because it's not relevant or noteworthy.)

Then, you declare two channels, imaginatively named `requests` and `responses`. Messages sent on the `requests` channel are forwarded to the `jms:outbound-gateway` that you've declared. The `jms:outbound-gateway` is the component that does most of the work. It takes the message you created and sends it to the request JMS destination, setting up the reply headers and so on. Finally, you declare a generic `gateway` element, which does most of the magic. The `gateway` element simply exists to identify the component and the interface, to which the proxy is cast and made available to clients.

```

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

    <beans:import resource="shared-context.xml" />
    <context:annotation-config />

    <channel id="requests" />
    <channel id="responses" />
```

```

<jms:outbound-gateway
    request-destination-name="inboundHotelReservationSearchDestination"
    request-channel="requests"
    reply-destination-name="outboundHotelReservationSearchResultsDestination"
    reply-channel="responses"
    connection-factory="connectionFactory" />

<gateway id="vacationService"
    service-interface="com.apress.springrecipes.springintegration.myholiday.VacationService" />

</beans:beans>

```

One thing that's conspicuously absent is any mention of an output or input channel from and to the gateway element. While it is possible to declare default request/reply message queues in the configuration for the gateway, realistically, most methods on an interface will require their own request/reply queues. So, you configure the channels on the interface itself.

```

package com.apress.springrecipes.springintegration.myholiday;

import java.util.List;
import org.springframework.integration.annotation.Gateway;

public interface VacationService {

    @Gateway(requestChannel = "requests", replyChannel = "responses")
    List<HotelReservation> findHotels(HotelReservationSearch hotelReservationSearch);

}

```

This is the client-facing interface. There is no coupling between the client-facing interface exposed via the gateway component and the interface of the service that ultimately handles the messages. We use the interface for the service and the client to simplify the names needed to understand everything that's going on. This is not like traditional, synchronous remoting in which the service interface and the client interface match.

In this example, you're using two very simple objects for demonstration: HotelReservationSearch and HotelReservation. There is nothing interesting about these objects in the slightest; they are simple POJOs that implement Serializable and contain a few accessor/mutators to flesh out the example domain.

The client Java code demonstrates how all of this comes together:

```

package com.apress.springrecipes.springintegration;

import com.apress.springrecipes.springintegration.myholiday.HotelReservation;
import com.apress.springrecipes.springintegration.myholiday.HotelReservationSearch;
import com.apress.springrecipes.springintegration.myholiday.VacationService;
import org.apache.commons.lang3.time.DateUtils;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.Date;
import java.util.List;

public class Main {
    public static void main(String[] args) throws Throwable {

```

```

// Start server
ApplicationContext ctx = new ClassPathXmlApplicationContext("server-integration-context.xml");

// Start client and do a search
ApplicationContext clientCtx =
    new ClassPathXmlApplicationContext("client-integration-context.xml");
VacationService vacationService = clientCtx.getBean(VacationService.class);

Date now = new Date();
HotelReservationSearch hotelReservationSearch =
    new HotelReservationSearch(200f, 2, DateUtils.addDays(now, 1), DateUtils.addDays(now, 8));
List<HotelReservation> results = vacationService.findHotels(hotelReservationSearch);

System.out.printf("Found %s results.", results.size());
System.out.println();

for (HotelReservation reservation : results) {
    System.out.printf("\t%s", reservation.toString());
    System.out.println();
}
}
}
}

```

It just doesn't get any cleaner than that! No Spring Integration interfaces whatsoever. You make a request, searching is done, and you get the result back when the processing is done.

The service implementation for this setup is interesting, not because of what you've added, but because of what's not there:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/integration"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jms="http://www.springframework.org/schema/integration/jms"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/jms
    http://www.springframework.org/schema/integration/jms/spring-integration-jms.xsd">
    <beans:import resource="shared-context.xml" />
    <context:annotation-config />

    <channel id="inboundHotelReservationSearchChannel" />
    <channel id="outboundHotelReservationSearchResultsChannel" />

    <beans:bean id="vacationServiceImpl"
        class="com.apress.springrecipes.springintegration.myholiday.VacationServiceImpl" />

```

```

<jms:inbound-gateway
    request-channel="inboundHotelReservationSearchChannel"
    request-destination-name="inboundHotelReservationSearchDestination"
    connection-factory="connectionFactory" />

<service-activator
    input-channel="inboundHotelReservationSearchChannel"
    ref="vacationServiceImpl"
    method="findHotels" />

</beans:beans>

```

Here, you've defined an inbound JMS gateway element. The messages from the inbound JMS gateway are put on a channel, `inboundHotelReservationSearchChannel`, whose messages are forwarded to a service-activator, as you would expect. The service-activator is what handles actual processing. What's interesting here is that there's no mention of a response channel, for either the service-activator, or for the inbound JMS gateway. The service-activator looks, and fails to find, a reply channel and so uses the reply channel created by the inbound JMS gateway component, which in turn has created the reply channel based on the header metadata in the inbound JMS message. Thus, everything just works without specification.

The implementation is a simple useless implementation of the interface:

```

package com.apress.springrecipes.springintegration.myholiday;
import java.util.Arrays;
import java.util.List;
import org.springframework.beans.factory.InitializingBean;

public class VacationServiceImpl implements VacationService, InitializingBean {
    private List<HotelReservation> hotelReservations;
    public void afterPropertiesSet() throws Exception {
        hotelReservations = Arrays.asList(
            new HotelReservation("Bilton", 243.200F),
            new HotelReservation("West Western", 75.0F),
            new HotelReservation("Theirfield Inn", 70F),
            new HotelReservation("Park Inn", 200.00F));
    }

    public List<HotelReservation> findHotels(HotelReservationSearch searchMsg) {
        try {
            Thread.sleep(1000);
        } catch (Throwable th) {
            // eat the exception
        }
        return hotelReservations;
    }
}

```

Summary

This chapter discussed building an integration solution using Spring Integration, an ESB-like framework built on top of the Spring framework. You were introduced to the core concepts of enterprise application integration (EAI). You learned how to handle a few integration scenarios, including JMS and file polling.

In the next chapter, you will explore the capabilities of Spring in the field of testing.

CHAPTER 17



Spring Testing

In this chapter, you will learn about basic techniques you can use to test Java applications, and the testing support features offered by the Spring framework. These features can make your testing tasks easier and lead you to better application design. In general, applications developed with the Spring framework and the dependency injection pattern are easy to test.

Testing is a key activity for ensuring quality in software development. There are many types of testing, including unit testing, integration testing, functional testing, system testing, performance testing, and acceptance testing. Spring's testing support focuses on unit and integration testing, but it can also help with other types of testing. Testing can be performed either manually or automatically. However, since automated tests can be run repeatedly and continuously at different phases of a development process, they are highly recommended, especially in agile development processes. The Spring framework is an agile framework that fits these kinds of processes.

Many testing frameworks are available on the Java platform. Currently, JUnit and TestNG are the most popular. JUnit has a long history and a large user group in the Java community. TestNG is another popular Java testing framework. Compared to JUnit, TestNG offers additional powerful features such as test grouping, dependent test methods, and data-driven tests.

Spring's testing support features have been offered by the Spring TestContext framework, which requires Java 1.5 or higher. This framework abstracts the underlying testing framework with the following concepts:

- *Test context*: This encapsulates the context of a test's execution, including the application context, test class, current test instance, current test method, and current test exception.
- *Test context manager*: This manages a test context for a test and triggers test execution listeners at predefined test execution points, including when preparing a test instance, before executing a test method (before any framework-specific initialization methods), and after executing a test method (after any framework-specific cleanup methods).
- *Test execution listener*: This defines a listener interface; by implementing this, you can listen to test execution events. The TestContext framework provides several test execution listeners for common testing features, but you are free to create your own.

Spring provides convenient TestContext support classes for JUnit 4 and TestNG, with particular test execution listeners preregistered. You can simply extend these support classes to use the TestContext framework without having to know much about the framework details.

After finishing this chapter, you will understand the basic concepts and techniques of testing and the popular Java testing frameworks JUnit and TestNG. You will also be able to create unit tests and integration tests using the Spring TestContext framework.

17-1. Creating Tests with JUnit and TestNG

Problem

You would like to create automated tests for your Java application so that they can be run repeatedly to ensure the correctness of your application.

Solution

The most popular testing frameworks on the Java platform are JUnit and TestNG. JUnit 4 incorporates several major improvements over JUnit 3, which relies on the base class (i.e., `TestCase`) and the method signature (i.e., methods whose names begin with `test`) to identify test cases—an approach that lacks flexibility. JUnit 4 allows you to annotate your test methods with JUnit's `@Test` annotation, so an arbitrary public method can be run as a test case. TestNG is another powerful testing framework that makes use of annotations. It also provides a `@Test` annotation type for you to identify test cases.

How It Works

Suppose you are going to develop a system for a bank. To ensure the system's quality, you have to test every part of it. First, let's consider an interest calculator, whose interface is defined as follows:

```
package com.apress.springrecipes.bank;

public interface InterestCalculator {
    public void setRate(double rate);
    public double calculate(double amount, double year);
}
```

Each interest calculator requires a fixed interest rate to be set. Now, you can implement this calculator with a simple interest formula:

```
package com.apress.springrecipes.bank;

public class SimpleInterestCalculator implements InterestCalculator {

    private double rate;

    public void setRate(double rate) {
        this.rate = rate;
    }

    public double calculate(double amount, double year) {
        if (amount < 0 || year < 0) {
            throw new IllegalArgumentException("Amount or year must be positive");
        }
        return amount * year * rate;
    }
}
```

Next, you will test this simple interest calculator with the popular testing frameworks JUnit and TestNG (version 5).

Tip Usually, a test and its target class are located in the same package, but the source files of tests are stored in a separate directory (e.g., `test`) from the source files of other classes (e.g., `src`).

Testing with JUnit 4

In JUnit 4, a class that contains test cases no longer needs to extend the `TestCase` class. It can be an arbitrary class. A test case is simply a public method with the `@Test` annotation. Similarly, you no longer need to override the `setUp()` and `tearDown()` methods, but rather annotate a public method with the `@Before` or `@After` annotation. You can also annotate a public static method with `@BeforeClass` or `@AfterClass` to have it run once before or after all test cases in the class.

Since your class doesn't extend `TestCase`, it doesn't inherit the assert methods. So, you have to call the static assert methods declared in the `org.junit.Assert` class directly. However, you can import all assert methods via a static import statement in Java 1.5. You can create the following JUnit 4 test cases to test your simple interest calculator.

Note To compile and run test cases created for JUnit 4, you have to include JUnit 4 on your CLASSPATH. If you are using Maven, add the following dependency to your project:

```
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.11</version>
</dependency>
```

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class SimpleInterestCalculatorJUnit4Tests {
    private InterestCalculator interestCalculator;

    @Before
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }
}
```

```

@Test
public void calculate() {
    double interest = interestCalculator.calculate(10000, 2);
    assertEquals(interest, 1000.0, 0);
}

@Test(expected = IllegalArgumentException.class)
public void illegalCalculate() {
    interestCalculator.calculate(-10000, 2);
}
}

```

JUnit 4 offers a powerful feature that allows you to expect an exception to be thrown in a test case. You can simply specify the exception type in the expected attribute of the @Test annotation.

Testing with TestNG

A TestNG test looks very similar to a JUnit 4 one, except that you have to use the classes and annotation types defined by the TestNG framework.

Note To compile and run test cases created for TestNG, you have to add TestNG to your CLASSPATH. If you are using Maven, add the following dependency to your project.

```

<dependency>
<groupId>org.testng</groupId>
<artifactId>testng</artifactId>
<version>6.6.8</version>
</dependency>

```

```

package com.apress.springrecipes.bank;

import static org.testng.Assert.*;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SimpleInterestCalculatorTestNGTests {
    private InterestCalculator interestCalculator;

    @BeforeMethod
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }
}

```

```

@Test
public void calculate() {
    double interest = interestCalculator.calculate(10000, 2);
    assertEquals(interest, 1000.0);
}

@Test(expectedExceptions = IllegalArgumentException.class)
public void illegalCalculate() {
    interestCalculator.calculate(-10000, 2);
}
}

```

Note If you are using Eclipse for development, you can download and install the TestNG Eclipse plug-in from <http://testng.org/doc/eclipse.html> to run TestNG tests in Eclipse. Again, you will see a green bar if all your tests pass and a red bar otherwise.

One of the powerful features of TestNG is its built-in support for data-driven testing. TestNG cleanly separates test data from test logic so that you can run a test method multiple times for different data sets. In TestNG, test data sets are provided by data providers, which are methods with the `@DataProvider` annotation.

```

package com.apress.springrecipes.bank;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class SimpleInterestCalculatorTestNGTests {

    private InterestCalculator interestCalculator;

    @BeforeMethod
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    @DataProvider(name = "legal")
    public Object[][] createLegalInterestParameters() {
        return new Object[][] { new Object[] { 10000, 2, 1000.0 } };
    }

    @DataProvider(name = "illegal")
    public Object[][] createIllegalInterestParameters() {
        return new Object[][] { new Object[] { -10000, 2 },
            new Object[] { 10000, -2 }, new Object[] { -10000, -2 } };
    }
}

```

```

@Test(dataProvider = "legal")
public void calculate(double amount, double year, double result) {
    double interest = interestCalculator.calculate(amount, year);
    assertEquals(interest, result);
}

@Test(
    dataProvider = "illegal",
    expectedExceptions = IllegalArgumentException.class)
public void illegalCalculate(double amount, double year) {
    interestCalculator.calculate(amount, year);
}
}

```

If you run the preceding test with TestNG, the `calculate()` method will be executed once, while the `illegalCalculate()` method will be executed three times, as there are three data sets returned by the `illegal` data provider.

17-2. Creating Unit Tests and Integration Tests

Problem

A common testing technique is to test each module of your application in isolation and then test them in combination. You would like to apply this skill in testing your Java applications.

Solution

Unit tests are used to test a single programming unit. In object-oriented languages, a *unit* is usually a class or a method. The scope of a unit test is a single unit, but in the real world, most units won't work in isolation. They often need to cooperate with others to complete their tasks. When testing a unit that depends on other units, a common technique you can apply is to simulate the unit's dependencies with stubs and mock objects, both of which can reduce complexity of your unit tests caused by dependencies.

A *stub* is an object that simulates a dependent object with the minimum number of methods required for a test. The methods are implemented in a predetermined way, usually with hard-coded data. A stub also exposes methods for a test to verify the stub's internal states. In contrast to a stub, a *mock object* usually knows how its methods are expected to be called in a test. The mock object then verifies the methods actually called against the expected ones. In Java, there are several libraries that can help create mock objects, including EasyMock and jMock. The main difference between a stub and a mock object is that a stub is usually used for *state verification*, while a mock object is used for *behavior verification*.

Integration tests, in contrast, are used to test several units in combination as a whole. They test if the integration and interaction between units are correct. Each of these units should already have been tested with unit tests, so integration testing is usually performed after unit testing.

Finally, note that applications developed using the principle of “separating interface from implementation” and the dependency injection pattern are easy to test, both for unit testing and integration testing. This is because that principle and pattern can reduce coupling between different units of your application.

How It Works

Creating Unit Tests for Isolated Classes

The core functions of your bank system should be designed around customer accounts. First, you create the following domain class, Account, with a custom equals() method:

```
package com.apress.springrecipes.bank;

public class Account {

    private String accountNo;
    private double balance;

    // Constructors, Getters and Setters
    ...

    public boolean equals(Object obj) {
        if (!(obj instanceof Account)) {
            return false;
        }
        Account account = (Account) obj;
        return account.accountNo.equals(accountNo) && account.balance == balance;
    }
}
```

Next, you define the following DAO interface for persisting account objects in your bank system's persistence layer:

```
package com.apress.springrecipes.bank;

public interface AccountDao {

    public void createAccount(Account account);
    public void updateAccount(Account account);
    public void removeAccount(Account account);
    public Account findAccount(String accountNo);
}
```

To demonstrate the unit testing concept, let's implement this interface by using a map to store account objects. The `AccountNotFoundException` and `DuplicateAccountException` classes are subclasses of `RuntimeException` that you should be able to create yourself.

```
package com.apress.springrecipes.bank;
...
public class InMemoryAccountDao implements AccountDao {

    private Map<String, Account> accounts;

    public InMemoryAccountDao() {
        accounts = Collections.synchronizedMap(new HashMap<String, Account>());
    }
}
```

```

public boolean accountExists(String accountNo) {
    return accounts.containsKey(accountNo);
}

public void createAccount(Account account) {
    if (accountExists(account.getAccountNo())) {
        throw new DuplicateAccountException();
    }
    accounts.put(account.getAccountNo(), account);
}

public void updateAccount(Account account) {
    if (!accountExists(account.getAccountNo())) {
        throw new AccountNotFoundException();
    }
    accounts.put(account.getAccountNo(), account);
}

public void removeAccount(Account account) {
    if (!accountExists(account.getAccountNo())) {
        throw new AccountNotFoundException();
    }
    accounts.remove(account.getAccountNo());
}

public Account findAccount(String accountNo) {
    Account account = accounts.get(accountNo);
    if (account == null) {
        throw new AccountNotFoundException();
    }
    return account;
}
}

```

Obviously, this simple DAO implementation doesn't support transactions. However, to make it thread-safe, you can wrap the map storing accounts with a synchronized map so that it will be accessed serially.

Now, let's create unit tests for this DAO implementation with JUnit 4. As this class doesn't depend directly on other classes, it's easy to test. To ensure that this class works properly for exceptional cases as well as normal cases, you should also create exceptional test cases for it. Typically, exceptional test cases expect an exception to be thrown.

```

package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

```

```
public class InMemoryAccountDaoTests {  
  
    private static final String EXISTING_ACCOUNT_NO = "1234";  
    private static final String NEW_ACCOUNT_NO = "5678";  
  
    private Account existingAccount;  
    private Account newAccount;  
    private InMemoryAccountDao accountDao;  
  
    @Before  
    public void init() {  
        existingAccount = new Account(EXISTING_ACCOUNT_NO, 100);  
        newAccount = new Account(NEW_ACCOUNT_NO, 200);  
        accountDao = new InMemoryAccountDao();  
        accountDao.createAccount(existingAccount);  
    }  
  
    @Test  
    public void accountExists() {  
        assertTrue(accountDao.accountExists(EXISTING_ACCOUNT_NO));  
        assertFalse(accountDao.accountExists(NEW_ACCOUNT_NO));  
    }  
  
    @Test  
    public void createNewAccount() {  
        accountDao.createAccount(newAccount);  
        assertEquals(accountDao.findAccount(NEW_ACCOUNT_NO), newAccount);  
    }  
  
    @Test(expected = DuplicateAccountException.class)  
    public void createDuplicateAccount() {  
        accountDao.createAccount(existingAccount);  
    }  
  
    @Test  
    public void updateExistedAccount() {  
        existingAccount.setBalance(150);  
        accountDao.updateAccount(existingAccount);  
        assertEquals(accountDao.findAccount(EXISTING_ACCOUNT_NO), existingAccount);  
    }  
  
    @Test(expected = AccountNotFoundException.class)  
    public void updateNotExistedAccount() {  
        accountDao.updateAccount(newAccount);  
    }  
  
    @Test  
    public void removeExistedAccount() {  
        accountDao.removeAccount(existingAccount);  
        assertFalse(accountDao.accountExists(EXISTING_ACCOUNT_NO));  
    }  
}
```

```

@Test(expected = AccountNotFoundException.class)
public void removeNotExistedAccount() {
    accountDao.removeAccount(newAccount);
}

@Test
public void findExistedAccount() {
    Account account = accountDao.findAccount(EXISTING_ACCOUNT_NO);
    assertEquals(account, existingAccount);
}

@Test(expected = AccountNotFoundException.class)
public void findNotExistedAccount() {
    accountDao.findAccount(NEW_ACCOUNT_NO);
}
}

```

Creating Unit Tests for Dependent Classes Using Stubs and Mock Objects

Testing an independent class is easy, because you needn't consider how its dependencies work and how to set them up properly. However, testing a class that depends on results of other classes or services (e.g., database services and network services) would be a little bit difficult. For example, let's consider the following `AccountService` interface in the service layer:

```

package com.apress.springrecipes.bank;

public interface AccountService {

    public void createAccount(String accountNo);
    public void removeAccount(String accountNo);
    public void deposit(String accountNo, double amount);
    public void withdraw(String accountNo, double amount);
    public double getBalance(String accountNo);
}

```

The implementation of this service interface has to depend on an `AccountDao` object in the persistence layer to persist account objects. The `InsufficientBalanceException` class is also a subclass of `RuntimeException` that you have to create.

```

package com.apress.springrecipes.bank;

public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }
}

```

```

public void createAccount(String accountNo) {
    accountDao.createAccount(new Account(accountNo, 0));
}

public void removeAccount(String accountNo) {
    Account account = accountDao.findAccount(accountNo);
    accountDao.removeAccount(account);
}

public void deposit(String accountNo, double amount) {
    Account account = accountDao.findAccount(accountNo);
    account.setBalance(account.getBalance() + amount);
    accountDao.updateAccount(account);
}

public void withdraw(String accountNo, double amount) {
    Account account = accountDao.findAccount(accountNo);
    if (account.getBalance() < amount) {
        throw new InsufficientBalanceException();
    }
    account.setBalance(account.getBalance() - amount);
    accountDao.updateAccount(account);
}

public double getBalance(String accountNo) {
    return accountDao.findAccount(accountNo).getBalance();
}
}

```

A common technique used in unit testing to reduce complexity caused by dependencies is using stubs. A stub must implement the same interface as the target object so that it can substitute for the target object. For example, you can create a stub for AccountDao that stores a single customer account and implements only the `findAccount()` and `updateAccount()` methods, as they are required for `deposit()` and `withdraw()`:

```

package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceImplStubTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountDaoStub accountDaoStub;
    private AccountService accountService;

    private class AccountDaoStub implements AccountDao {

        private String accountNo;
        private double balance;

```

```

public void createAccount(Account account) {}
public void removeAccount(Account account) {}

public Account findAccount(String accountNo) {
    return new Account(this.accountNo, this.balance);
}

public void updateAccount(Account account) {
    this.accountNo = account.getAccountNo();
    this.balance = account.getBalance();
}
}

@Before
public void init() {
    accountDaoStub = new AccountDaoStub();
    accountDaoStub.accountNo = TEST_ACCOUNT_NO;
    accountDaoStub.balance = 100;
    accountService = new AccountServiceImpl(accountDaoStub);
}

@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
    assertEquals(accountDaoStub.balance, 150, 0);
}

@Test
public void withdrawWithSufficientBalance() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
    assertEquals(accountDaoStub.balance, 50, 0);
}

@Test(expected = InsufficientBalanceException.class)
public void withdrawWithInsufficientBalance() {
    accountService.withdraw(TEST_ACCOUNT_NO, 150);
}
}

```

However, writing stubs yourself requires a lot of coding. A more efficient technique is to use mock objects. The Mockito library is able to dynamically create mock objects that work in a record/playback mechanism.

Note To use Mockito for testing, you have to add it to your CLASSPATH. If you are using Maven, add the following dependency to your project.

```
<dependency>
<groupId>org.mockito</groupId>
<artifactId>mockito-core</artifactId>
<version>1.9.5</version>
</dependency>
```

```
package com.apress.springrecipes.bank;

import org.junit.Before;
import org.junit.Test;

import static org.mockito.Mockito.*;

public class AccountServiceImplMockTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    private AccountDao accountDao;
    private AccountService accountService;

    @Before
    public void init() {
        accountDao = mock(AccountDao.class);
        accountService = new AccountServiceImpl(accountDao);
    }

    @Test
    public void deposit() {
        // Setup
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        when(accountDao.findAccount(TEST_ACCOUNT_NO)).thenReturn(account);

        // Execute
        accountService.deposit(TEST_ACCOUNT_NO, 50);

        // Verify
        verify(accountDao, times(1)).findAccount(any(String.class));
        verify(accountDao, times(1)).updateAccount(account);
    }

    @Test
    public void withdrawWithSufficientBalance() {
        // Setup
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        when(accountDao.findAccount(TEST_ACCOUNT_NO)).thenReturn(account);
```

```

    // Execute
    accountService.withdraw(TEST_ACCOUNT_NO, 50);

    // Verify
    verify(accountDao, times(1)).findAccount(any(String.class));
    verify(accountDao, times(1)).updateAccount(account);

}

@Test(expected = InsufficientBalanceException.class)
public void testWithdrawWithInsufficientBalance() {
    // Setup
    Account account = new Account(TEST_ACCOUNT_NO, 100);
    when(accountDao.findAccount(TEST_ACCOUNT_NO)).thenReturn(account);

    // Execute
    accountService.withdraw(TEST_ACCOUNT_NO, 150);
}
}

```

With Mockito, you can create a mock object dynamically for an arbitrary interface or class. This mock can be instructed to have certain behavior for method calls and you can use it to selectively verify if something has happened. In your test you want that in the `findAccount` method that a certain `Account` object is returned. You use the `Mockito.when` method for this and you can then either return a value, throw an exception, or do more elaborate things with an `Answer`. The default behavior for the mock is to return null.

You use the `Mockito.verify` method to do selective verification of things that have happened. You want to make sure that our `findAccount` method is called and that the account gets updated.

Creating Integration Tests

Integration tests are used to test several units in combination to ensure that the units are properly integrated and can interact correctly. For example, you can create an integration test to test `AccountServiceImpl` using `InMemoryAccountDao` as the DAO implementation:

```

package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

```

```

@Before
public void init() {
    accountService = new AccountServiceImpl(new InMemoryAccountDao());
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}

@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
}

@Test
public void withdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
}

@After
public void cleanup() {
    accountService.removeAccount(TEST_ACCOUNT_NO);
}
}

```

17-3. Unit Testing Spring MVC Controllers

Problem

In a web application, you would like to test the web controllers developed with the Spring MVC framework.

Solution

A Spring MVC controller is invoked by `DispatcherServlet` with an HTTP request object and an HTTP response object. After processing a request, the controller returns it to `DispatcherServlet` for rendering the view. The main challenge of unit testing Spring MVC controllers, as well as web controllers in other web application frameworks, is simulating HTTP request objects and response objects in a unit testing environment. Fortunately, Spring supports web controller testing by providing a set of mock objects for the Servlet API (including `MockHttpServletRequest`, `MockHttpServletResponse`, and `MockHttpSession`).

To test a Spring MVC controller's output, you need to check if the object returned to `DispatcherServlet` is correct. Spring also provides a set of assertion utilities for checking the contents of an object.

How It Works

In your bank system, suppose you are going to develop a web interface for bank staff to input the account number and amount of a deposit. You create a controller named `DepositController` using the techniques you already know from Spring MVC:

```
package com.apress.springrecipes.bank.web;
```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DepositController {

    private AccountService accountService;

    @Autowired
    public DepositController(AccountService accountService) {
        this.accountService = accountService;
    }

    @RequestMapping("/deposit.do")
    public String deposit(
        @RequestParam("accountNo") String accountNo,
        @RequestParam("amount") double amount,
        ModelMap model) {
        accountService.deposit(accountNo, amount);
        model.addAttribute("accountNo", accountNo);
        model.addAttribute("balance", accountService.getBalance(accountNo));
        return "success";
    }
}

```

Because this controller doesn't deal with the Servlet API, testing it is very easy. You can test it just like a simple Java class:

```

package com.apress.springrecipes.bank.web;

import static org.junit.Assert.*;

import com.apress.springrecipes.bank.AccountService;
import org.junit.Before;
import org.junit.Test;
import org.mockito.Mockito;
import org.springframework.ui.ModelMap;

public class DepositControllerTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private static final double TEST_AMOUNT = 50;
    private AccountService accountService;
    private DepositController depositController;
}

```

```

@Before
public void init() {
    accountService = Mockito.mock(AccountService.class);
    depositController = new DepositController(accountService);
}

@Test
public void deposit() {
    //Setup
    Mockito.when(accountService.getBalance(TEST_ACCOUNT_NO)).thenReturn(150.0);
    ModelMap model = new ModelMap();

    //Execute
    String viewName =
        depositController.deposit(TEST_ACCOUNT_NO, TEST_AMOUNT, model);

    assertEquals(viewName, "success");
    assertEquals(model.get("accountNo"), TEST_ACCOUNT_NO);
    assertEquals(model.get("balance"), 150.0);
}
}

```

17-4. Managing Application Contexts in Integration Tests

Problem

When creating integration tests for a Spring application, you have to access beans declared in the application context. Without Spring's testing support, you have to load the application context manually in an initialization method of your tests, a method with `@Before` or `@BeforeClass` in JUnit 4. However, as an initialization method is called before each test method or test class, the same application context may be reloaded many times. In a large application with many beans, loading an application context may require a lot of time, which causes your tests to run slowly.

Solution

Spring's testing support facilities can help you manage the application context for your tests, including loading it from one or more bean configuration files and caching it across multiple test executions. An application context will be cached across all tests within a single JVM, using the configuration file locations as the key. As a result, your tests can run much faster without reloading the same application context many times.

Starting from Spring 2.5, the `TestContext` framework provides two test execution listeners related to context management. They will be registered with a test context manager by default if you don't specify your own explicitly.

- `DependencyInjectionTestExecutionListener`: This injects dependencies, including the managed application context, into your tests.
- `DirtiesContextTestExecutionListener`: This handles the `@DirtiesContext` annotation and reloads the application context when necessary.

To have the `TestContext` framework manage the application context, your test class has to integrate with a test context manager internally. For your convenience, the `TestContext` framework provides support classes that do this, as shown in Table 17-1. These classes integrate with a test context manager and implement the `ApplicationContextAware` interface, so they can provide access to the managed application context through the protected field `applicationContext`. Your test class can simply extend the corresponding `TestContext` support class for your testing framework.

Table 17-1. TestContext Support Classes for Context Management

Testing Framework	TestContext Support Class*
JUnit 4	AbstractJUnit4SpringContextTests
TestNG	AbstractTestNGSpringContextTests

*These TestContext support classes have only `DependencyInjectionTestExecutionListener` and `DirtiesContextTestExecutionListener` enabled.

If you are using JUnit 4 or TestNG, you can integrate your test class with a test context manager by yourself and implement the `ApplicationContextAware` interface directly, without extending a TestContext support class. In this way, your test class doesn't bind to the TestContext framework class hierarchy, so you can extend your own base class. In JUnit 4, you can simply run your test with the test runner `SpringJUnit4ClassRunner` to have a test context manager integrated. However, in TestNG, you have to integrate with a test context manager manually.

How It Works

First, let's declare an `AccountService` instance and an `AccountDao` instance in the bean configuration class as follows. Later, you will create integration tests for them.

```
package com.apress.springrecipes.bank.config;

import com.apress.springrecipes.bank.AccountDao;
import com.apress.springrecipes.bank.AccountService;
import com.apress.springrecipes.bank.AccountServiceImpl;
import com.apress.springrecipes.bank.InMemoryAccountDao;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class BankConfiguration {

    @Bean
    public AccountDao accountDao() {
        return new InMemoryAccountDao();
    }

    @Bean
    public AccountService accountService() {
        return new AccountServiceImpl(accountDao());
    }
}
```

Accessing the Context with the TestContext Framework in JUnit 4

If you are using JUnit 4 to create tests with the TestContext framework, you will have two options to access the managed application context. The first option is by implementing the `ApplicationContextAware` interface. For this option, you have to explicitly specify a Spring-specific test runner for running your test—`SpringJUnit4ClassRunner`. You can specify this in the `@RunWith` annotation at the class level.

```
package com.apress.springrecipes.bank;

import com.apress.springrecipes.bank.config.BankConfiguration;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import static org.junit.Assert.assertEquals;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests implements ApplicationContextAware {

    private static final String TEST_ACCOUNT_NO = "1234";
    private ApplicationContext applicationContext;
    private AccountService accountService;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        this.applicationContext=applicationContext;
    }

    @Before
    public void init() {
        accountService = applicationContext.getBean(AccountService.class);
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @After
    public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }

}
```

You can specify the configuration classes in the `classes` attribute of the `@ContextConfiguration` annotation at the class level, when using xml based configuration you can use the `locations` attribute instead. If you don't specify any test configuration the `TestContext` will try to detect one. It will first try to load a file by joining the test class name with `-context.xml` as the suffix (i.e., `AccountServiceJUnit4Tests-context.xml`) from the same package as the test class. Next it will scan the test class for any public static inner classes that are annotated with `@Configuration`. If a file or classes are detected those will be used to load the test configuration.

By default, the application context will be cached and reused for each test method, but if you want it to be reloaded after a particular test method, you can annotate the test method with the `@DirtiesContext` annotation so that the application context will be reloaded for the next test method.

The second option to access the managed application context is by extending the `TestContext` support class specific to JUnit 4: `AbstractJUnit4SpringContextTests`. This class implements the `ApplicationContextAware` interface, so you can extend it to get access to the managed application context via the protected field `applicationContext`. However, you first have to delete the private field `applicationContext` and its setter method. Note that if you extend this support class, you don't need to specify `SpringJUnit4ClassRunner` in the `@RunWith` annotation, because this annotation is inherited from the parent.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests extends AbstractJUnit4SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @Before
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

Accessing the Context with the `TestContext` Framework in TestNG

To access the managed application context with the `TestContext` framework in TestNG, you can extend the `TestContext` support class `AbstractTestNGSpringContextTests`. This class also implements the `ApplicationContextAware` interface.

```
package com.apress.springrecipes.bank;

import com.apress.springrecipes.bank.config.BankConfiguration;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
```

```

import static org.testng.Assert.assertEquals;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceTestNGContextTests extends AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @AfterMethod
    public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }

}

```

If you don't want your TestNG test class to extend a TestContext support class, you can implement the `ApplicationContextAware` interface just as you did for JUnit 4. However, you have to integrate with a test context manager by yourself. Please refer to the source code of `AbstractTestNGSpringContextTests` for details.

17-5. Injecting Test Fixtures into Integration Tests

Problem

The test fixtures of an integration test for a Spring application are mostly beans declared in the application context. You might wish to have the test fixtures automatically injected by Spring via dependency injection, which saves you the trouble of retrieving them from the application context manually.

Solution

Spring's testing support facilities can inject beans automatically from the managed application context into your tests as test fixtures.

Starting in Spring 2.5's TestContext framework, `DependencyInjectionTestExecutionListener` can automatically inject dependencies into your tests. If you have this listener registered, you can simply annotate a setter method or field of your test with Spring's `@Autowired` annotation or JSR-250's `@Resource` annotation to have a fixture injected automatically. For `@Autowired`, the fixture will be injected by type, and for `@Resource`, it will be injected by name.

How It Works

Injecting Test Fixtures with the TestContext Framework in JUnit 4

When using the TestContext framework to create tests, you can have their test fixtures injected from the managed application context by annotating a field or setter method with the `@Autowired` or `@Resource` annotations. In JUnit 4, you can specify `SpringJUnit4ClassRunner` as your test runner without extending a support class.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

If you annotate a field or setter method of a test with `@Autowired`, it will be injected using auto-wiring by type. You can further specify a candidate bean for auto-wiring by providing its name in the `@Qualifier` annotation. However, if you want a field or setter method to be auto-wired by name, you can annotate it with `@Resource`.

By extending the TestContext support class `AbstractJUnit4SpringContextTests`, you can also have test fixtures injected from the managed application context. In this case, you don't need to specify `SpringJUnit4ClassRunner` for your test, as it is inherited from the parent.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractJUnit4SpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests extends AbstractJUnit4SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
```

```
@Autowired
private AccountService accountService;
...
}
```

Injecting Test Fixtures with the TestContext Framework in TestNG

In TestNG, you can extend the TestContext support class `AbstractTestNGSpringContextTests` to have test fixtures injected from the managed application context:

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceTestNGContextTests extends AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

17-6. Managing Transactions in Integration Tests

Problem

When creating integration tests for an application that accesses a database, you usually prepare the test data in the initialization method. After each test method runs, it may have modified the data in the database. So, you have to clean up the database to ensure that the next test method will run from a consistent state. As a result, you have to develop many database cleanup tasks.

Solution

Spring's testing support facilities can create and roll back a transaction for each test method, so the changes you make in a test method won't affect the next one. This can also save you the trouble of developing cleanup tasks to clean up the database.

Starting from Spring 2.5, the TestContext framework provides a test execution listener related to transaction management. It will be registered with a test context manager by default if you don't specify your own explicitly.

- `TransactionalTestExecutionListener`: This handles the `@Transactional` annotation at the class or method level and has the methods run within transactions automatically.

Your test class can extend the corresponding `TestContext` support class for your testing framework, as shown in Table 17-2, to have its test methods run within transactions. These classes integrate with a test context manager and have `@Transactional` enabled at the class level. Note that a transaction manager is also required in the bean configuration file.

Table 17-2. *TestContext Support Classes for Transaction Management*

Testing Framework	TestContext Support Class*
JUnit 4	<code>AbstractTransactionalJUnit4SpringContextTests</code>
TestNG	<code>AbstractTransactionalTestNGSpringContextTests</code>

* These `TestContext` support classes have `TransactionalTestExecutionListener` enabled in addition to `DependencyInjectionTestExecutionListener` and `DirtiesContextTestExecutionListener`.

In JUnit 4 and TestNG, you can simply annotate `@Transactional` at the class level or the method level to have the test methods run within transactions, without extending a `TestContext` support class. However, to integrate with a test context manager, you have to run the JUnit 4 test with the test runner `SpringJUnit4ClassRunner`, and you have to do it manually for a TestNG test.

How It Works

Let's consider storing your bank system's accounts in a relational database. You can choose any JDBC-compliant database engine that supports transactions and then execute the following SQL statement on it to create the `ACCOUNT` table. Here, we have chosen Apache Derby as our database engine and created the table in the bank instance.

```
CREATE TABLE ACCOUNT (
    ACCOUNT_NO      VARCHAR(10)      NOT NULL,
    BALANCE         DOUBLE          NOT NULL,
    PRIMARY KEY (ACCOUNT_NO)
);
```

Next, you create a new DAO implementation that uses JDBC to access the database. You can take advantage of `SimpleJdbcTemplate` to simplify your operations.

```
package com.apress.springrecipes.bank;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcAccountDao extends JdbcDaoSupport implements AccountDao {

    public void createAccount(Account account) {
        String sql = "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)";
        getJdbcTemplate().update(
            sql, account.getAccountNo(), account.getBalance());
    }

    public void updateAccount(Account account) {
        String sql = "UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_NO = ?";
        getJdbcTemplate().update(
            sql, account.getBalance(), account.getAccountNo());
    }
}
```

```

public void removeAccount(Account account) {
    String sql = "DELETE FROM ACCOUNT WHERE ACCOUNT_NO = ?";
    getJdbcTemplate().update(sql, account.getAccountNo());
}

public Account findAccount(String accountNo) {
    String sql = "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?";
    double balance = getJdbcTemplate().queryForObject(
        sql, Double.class, accountNo);
    return new Account(accountNo, balance);
}
}

```

Before you create integration tests to test the `AccountService` instance that uses this DAO to persist account objects, you have to replace `InMemoryAccountDao` with this DAO in the configuration class, and configure the target data source as well.

Note To access a database running on the Derby server, you have to add the client library to the CLASSPATH of your application. If you are using Maven, add the following dependency to your project.

```

<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derbyclient</artifactId>
<version>10.10.2.0</version>
</dependency>

```

```

@Configuration
public class BankConfiguration {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(ClientDriver.class.getName());
        dataSource.setUrl("jdbc:derby://localhost:1527/bank;create=true");
        dataSource.setUsername("app");
        dataSource.setPassword("app");
        return dataSource;
    }

    @Bean
    public AccountDao accountDao() {
        JdbcAccountDao accountDao = new JdbcAccountDao();
        accountDao.setDataSource(dataSource());
        return accountDao;
    }
}

```

```

@Bean
public AccountService accountService() {
    return new AccountServiceImpl(accountDao());
}
}

```

Managing Transactions with the TestContext Framework in JUnit 4

When using the TestContext framework to create tests, you can have the tests' methods run within transactions by annotating `@Transactional` at the class or method level. In JUnit 4, you can specify `SpringJUnit4ClassRunner` for your test class so that it doesn't need to extend a support class.

```

package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = BankConfiguration.class)
@Transactional
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // Don't need cleanup() anymore
    ...
}

```

If you annotate a test class with `@Transactional`, all of its test methods will run within transactions. If you would like a particular method not to run within a transaction, you can annotate it with `@NotTransactional`. An alternative is to annotate individual methods with `@Transactional`, not the entire class.

By default, transactions for test methods will be rolled back at the end. You can alter this behavior by disabling the `defaultRollback` attribute of `@TransactionConfiguration`, which should be applied to the class level. Also, you can override this class-level rollback behavior at the method level with the `@Rollback` annotation, which requires a Boolean value.

Note that methods with the `@Before` or `@After` annotation will be executed within the same transactions as test methods. If you have methods that need to perform initialization or cleanup tasks before or after a transaction, you have to annotate them with `@BeforeTransaction` or `@AfterTransaction`. Notice that these methods will not be executed for test methods annotated with `@NotTransactional`.

Finally, you also need a transaction manager configured in the bean configuration file. By default, a bean whose type is `PlatformTransactionManager` will be used, but you can specify another one in the `transactionManager` attribute of the `@TransactionConfiguration` annotation by giving its name.

```
@Bean
public PlatformTransactionManager transactionManager() {
    return new DataSourceTransactionManager(dataSource());
}
```

In JUnit 4, an alternative to managing transactions for test methods is to extend the transactional `TestContext` support class `AbstractTransactionalJUnit4SpringContextTests`, which has `@Transactional` enabled at the class level so that you don't need to enable it again. By extending this support class, you don't need to specify `SpringJUnit4ClassRunner` for your test, as it is inherited from the parent.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests extends AbstractTransactionalJUnit4SpringContextTests {
    ...
}
```

Managing Transactions with the `TestContext` Framework in TestNG

To create TestNG tests that run within transactions, your test class can extend the `TestContext` support class `AbstractTransactionalTestNGSpringContextTests` to have its methods run within transactions:

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceTestNGContextTests extends
    AbstractTransactionalTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;
```

```

@BeforeMethod
public void init() {
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}

// Don't need cleanup() anymore
...
}

```

17-7. Accessing a Database in Integration Tests

Problem

When creating integration tests for an application that accesses a database, especially one developed with an ORM framework, you might wish to access the database directly to prepare test data and validate the data after a test method runs.

Solution

Spring's testing support facilities can create and provide a JDBC template for you to perform database-related tasks in your tests.

Starting in Spring 2.5's TestContext framework, your test class can extend one of the transactional TestContext support classes to access the precreated SimpleJdbcTemplate instance. These classes also require a data source and a transaction manager in the bean configuration file.

How It Works

Accessing a Database with the TestContext Framework

When using the TestContext framework to create tests, you can extend the corresponding TestContext support class to use a JdbcTemplate instance via a protected field. For JUnit 4, this class is AbstractTransactionalJUnit4SpringContextTests, which provides similar convenient methods for you to count the number of rows in a table, delete rows from a table, and execute a SQL script:

```

package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests extends AbstractTransactionalJUnit4SpringContextTests
{
    ...
    @Before
    public void init() {
        executeSqlScript("classpath:/bank.sql",true);
        jdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }
}

```

```

@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    double balance = jdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 150.0, 0);
}

@Test
public void withdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = jdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 50.0, 0);
}
}

```

In TestNG, you can extend `AbstractTransactionalTestNGSpringContextTests` to use a `JdbcTemplate` instance:

```

package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests;

@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceTestNGContextTests extends AbstractTransactionalTestNGSpringContextTests
{
    ...
    @BeforeMethod
    public void init() {
        executeSqlScript("classpath:/bank.sql",true);
        jdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = jdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 150, 0);
    }
}

```

```

@Test
public void withdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = jdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 50, 0);
}
}

```

17-8. Using Spring's Common Testing Annotations

Problem

You often have to manually implement common testing tasks, such as expecting an exception to be thrown, repeating a test method multiple times, ensuring that a test method will complete in a particular time period, and so on.

Solution

Spring's testing support provides a common set of testing annotations to simplify your test creation. These annotations are Spring-specific but independent of the underlying testing framework. Of these, the following annotations are very useful for common testing tasks. However, they are only supported for use with JUnit:

- **@Repeat**: This indicates that a test method has to run multiple times. The number of times it will run is specified as the annotation value.
- **@Timed**: This indicates that a test method must complete in a specified time period (in milliseconds). Otherwise, the test fails. Note that the time period includes the repetitions of the test method and any initialization and cleanup methods.
- **@IfProfileValue**: This indicates that a test method can only run in a specific testing environment. This test method will run only when the actual profile value matches the specified one. You can also specify multiple values so that the test method will run if any of the values is matched. By default, `SystemPropertyValueSource` is used to retrieve system properties as profile values, but you can create your own `PropertyValueSource` implementation and specify it in the `@PropertyValueSourceConfiguration` annotation.

Starting from Spring 2.5's `TestContext` framework, you can use Spring's testing annotations by extending one of the `TestContext` support classes. If you don't extend a support class but run your JUnit 4 test with the test runner `SpringJUnit4ClassRunner`, you can also use these annotations.

How It Works

Using Common Testing Annotations with the `TestContext` Framework

When using the `TestContext` framework to create tests for JUnit 4, you can use Spring's testing annotations if you run your test with `SpringJUnit4ClassRunner` or extend a JUnit 4 `TestContext` support class:

```

package com.apress.springrecipes.bank;
...

```

```

import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests extends AbstractTransactionalJUnit4SpringContextTests
{
    ...
    @Test
    @Timed(milliseconds = 1000)
    public void deposit() {
        ...
    }

    @Test
    @Repeat(5)
    public void withdraw() {
        ...
    }
}

```

17-9. Integration Testing Spring MVC Controllers

Problem

In a web application, you would like to integration test the web controllers developed with the Spring MVC framework.

Solution

A Spring MVC controller is invoked by `DispatcherServlet` with an HTTP request object and an HTTP response object. After processing a request, the controller returns it to `DispatcherServlet` for rendering the view. The main challenge of integration testing Spring MVC controllers, as well as web controllers in other web application frameworks, is simulating HTTP request objects and response objects in a unit testing environment as well as setting up the mocked environment for a unit test. Fortunately, Spring has the mock MVC part of the Spring Test support. This allows for easy setup of a mocked servlet environment.

Spring Test Mock MVC will setup a `WebApplicationContext` according to your configuration. Next you can use the `MockMvc` API to simulate HTTP requests and verifying the result.

How It Works

In our banking application we want to integration test our `DepositController`. Before we can start testing we need to create a configuration class to configure our web-related beans.

```

package com.apress.springrecipes.bank.web.config;

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

```

```

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.ViewResolver;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "com.apress.springrecipes.bank.web")
public class BankWebConfiguration {

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}

```

The configuration enables annotation based controllers by using the `@EnableWebMvc` annotation, next you want the `@Controller` annotated beans to be picked up automatically using the `@ComponentScan` annotation. Finally there is an `InternalResourceViewResolver` which turns the name of the view into a URL which normally would be rendered by the browser, which you will now validate in the controller.

Now that the web based configuration is in place we can start to create our integration test. This unit test has to load our `BankWebConfiguration` class and also has to be annotated with `@WebAppConfiguration` to inform the `TestContext` framework we want a `WebApplicationContext` instead of a plain `ApplicationContext`.

Integration Testing Spring MVC Controllers with JUnit

In JUnit it is the easiest to extend one of the base classes, in this case `AbstractTransactionalJUnit4SpringContextTests` as we want to insert some test data and rollback after the tests complete.

```

package com.apress.springrecipes.bank.web;

import com.apress.springrecipes.bank.config.BankConfiguration;
import com.apress.springrecipes.bank.web.config.BankWebConfiguration;
import org.junit.Before;
import org.junit.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

```

```

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.forwardedUrl;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@ContextConfiguration(classes= { BankWebConfiguration.class, BankConfiguration.class})
@WebAppConfiguration
public class DepositControllerJUnit4ContextTests extends
AbstractTransactionalJUnit4SpringContextTests {

    private static final String ACCOUNT_PARAM = "accountNo";
    private static final String AMOUNT_PARAM = "amount";

    private static final String TEST_ACCOUNT_NO = "1234";
    private static final String TEST_AMOUNT = "50.0";

    @Autowired
    private WebApplicationContext webApplicationContext;

    private MockMvc mockMvc;

    @Before
    public void init() {
        executeSqlScript("classpath:/bank.sql", true);
        jdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @Test
    public void deposit() throws Exception {
        mockMvc.perform(
            get("/deposit.do")
                .param(ACCOUNT_PARAM, TEST_ACCOUNT_NO)
                .param(AMOUNT_PARAM, TEST_AMOUNT))
            .andDo(print())
            .andExpect(forwardedUrl("/WEB-INF/views/success.jsp"))
            .andExpect(status().isOk());
    }
}

```

In the init method you prepare the MockMvc object by using the convenient MockMvcBuilders. Using the factory method webAppContextSetup we can use the already loaded WebApplicationContext to initialize the MockMvc object. The MockMvc object basically mimics the behavior of the DispatcherServlet which you would use in a Spring MVC based application. It will use the passed in WebApplicationContext to configure the handler mappings, view resolution strategies and will also apply any interceptors that are configured.

There is also some setup of a test account so that we have something to work with.

In the deposit test method the initialized MockMvc object is used to simulate an incoming request to the /deposit.do url with 2 request parameters accountNo and amount. The MockMvcRequestBuilders.get factory method results in a RequestBuilder instance that is passed to the MockMvc.perform method.

The `perform` method returns a `ResultActions` object which can be used to do assertions and certain actions on the return result. The test method prints the information for the created request and returned response using `andDo(print())`, this can be useful while debugging your test. Finally there are two assertions to verify that everything works as expected. The `DepositController` returns success as the viewname which should lead to a forward to `/WEB-INF/views/success.jsp` due to the configuration of the `ViewResolver`. The return code of the request should be `200 (OK)` which can be tested with `status().isOk()` or `status().is(200)`.

Integration Testing Spring MVC Controllers with TestNG

Spring Mock MVC can also be used with TestNG extend the appropriate base class `AbstractTransactionalTestNGSpringContextTests` and add the `@WebAppConfiguration` annotation.

```
@ContextConfiguration(classes= { BankWebConfiguration.class, BankConfiguration.class })
@WebAppConfiguration
public class DepositControllerTestNGContextTests
    extends AbstractTransactionalTestNGSpringContextTests {
    ...
    @BeforeMethod
    public void init() {
        executeSqlScript("classpath:/bank.sql", true);
        jdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }
    ...
}
```

Summary

In this chapter, you learned about the basic concepts and techniques used in testing Java applications. JUnit and TestNG are the most popular testing frameworks on the Java platform. JUnit 4 introduces several major improvements over JUnit 3. The most important of these is annotation support. TestNG is also a powerful annotation-based testing framework.

Unit tests are used for testing a single programming unit, which is typically a class or a method in object-oriented languages. When testing a unit that depends on other units, you can use stubs and mock objects to simulate its dependencies, thus making the tests simpler. In contrast, integration tests are used to test several units as a whole.

In the web layer, controllers are usually hard to test. Spring offers mock objects for the Servlet API so that you can easily simulate web request and response objects to test a web controller. As of Spring 3.2 there is also Spring Mock MVC for easy integration testing of your controllers.

Spring's testing support facilities can manage application contexts for your tests by loading them from bean configuration files and caching them across multiple test executions. In releases prior to 4.0, Spring offered testing support specific to JUnit 3, as of Spring 4.0 this support has been removed.

You can access the managed application context in your tests, as well as have your test fixtures injected from the application context automatically. In addition, if your tests involve database updates, Spring can manage transactions for them so that changes made in one test method will be rolled back and thus won't affect the next test method. Spring can also create a JDBC template for you to prepare and validate your test data in the database.

Spring provides a common set of testing annotations to simplify your test creation. These annotations are Spring-specific but independent of the underlying testing framework. However, some of these are only supported for use with JUnit.

CHAPTER 18



Grails

When you embark on the creation of a Java web application, you need to put together a series of Java classes, create configuration files, and establish a particular layout, all of which have little to do with the problems an application solves. Such pieces are often called “scaffolding code” or “scaffolding steps,” since they are just the means to an end—the end being what an application actually accomplishes.

Grails is a framework designed to limit the amount of scaffolding steps you need to take in Java applications. Based on the Groovy language, which is a Java Virtual Machine-compatible language, Grails automates many steps that need to be undertaken in a Java application on the basis of conventions.

For example, when you create application controllers, they are eventually accompanied by a series of views (e.g., JavaServer Pages [JSP] pages), in addition to requiring some type of configuration file to make them work. If you generate a controller using Grails, Grails automates numerous steps using conventions (e.g., creating views and configuration files). You can later modify whatever Grails generates to more specific scenarios, but Grails undoubtedly shortens your development time since you won’t need to write everything from scratch (e.g., write XML configuration files and prepare a project directory structure).

Grails is fully integrated with Spring 4.0, so you can use it to kick-start your Spring applications and thus reduce your development efforts.

18-1. Getting and Installing Grails

Problem

You want to start creating a Grails application but don’t know where to get Grails and how to set it up.

Solution

You can download Grails at <http://www.grails.org/>. Ensure that you download Grails version 2.4 or higher, since only those versions support Spring 4.0. Grails is a self-contained framework that comes with various scripts to automate the creation of Java applications. In this sense, you simply need to unpack the distribution and perform a few installation steps in order to create Java applications on your workstation.

How It Works

After you unpack Grails on your workstation, define two environment variables on your operating system: `GRAILS_HOME` and `PATH`. This allows you to invoke Grails operations from anywhere on your workstation.

If you use a Linux workstation, you can edit the global `bashrc` file—located under the `/etc/` directory, or a user's `.bashrc` file, located under a user's home directory. Note that, depending on the Linux distribution, these last file names can vary (e.g., `bash.bashrc`). Both files use identical syntax to define environment variables, with one file used to define variables for all users and another for a single user. Place the following contents in either one:

```
GRAILS_HOME=/<installation_directory>/grails
export GRAILS_HOME
export PATH=$PATH:$GRAILS_HOME/bin
```

If you use a Windows workstation, go to the Control Panel, and click the System icon. On the window that emerges, click the Advanced Options tab. Next, click the “Environment variables” box to bring up the environment variable editor. From there, you can add or modify environment variables for either a single user or all users, using the following steps:

1. Click the New box.
2. Create an environment variable with the name `GRAILS_HOME` and a value corresponding to the Grails installation directory (e.g., `/<installation_directory>/grails`).
3. Select the `PATH` environment variable, and click the Modify box.
4. Add the `;%GRAILS_HOME%\bin` value to the end of the `PATH` environment variable.

Caution Be sure to *add* this last value and not modify the `PATH` environment variable in any other way, as this may cause certain applications to stop working.

Once you perform these steps in either a Windows or Linux workstation, you can start creating Grails applications. If you execute the command `grails help` from any directory on your workstation, you should see Grails's numerous commands.

18-2. Creating a Grails Application Problem

You want to create a Grails application.

Solution

To create a Grails application, invoke the following command wherever you wish to create an application: `grails create-app <grailsappname>`. This creates a Grails application directory, with a project structure in accordance to the framework's design.

If this last command fails, consult Recipe 18-1, on “Getting and Installing Grails.” The `grails` command should be available from any console or terminal if Grails was installed correctly.

How It Works

For example, typing `grails create-app court` creates a Grails application under a directory named `court`. Inside this directory, you will find a series of files and directories generated by Grails on the basis of conventions. The initial project structure for a Grails application is the following:

```
application.properties  
grails-app/  
grailsw  
grailsw.bat  
lib/  
scripts/  
src/  
test/  
web-app/  
wrapper/
```

Note In addition to this last layout, Grails also creates a series of working directories and files (i.e., not intended to be modified directly) for an application. These working directories and files are placed under a user's home directory under the name `.grails/<grails_version>/`.

As you can note from this last listing, Grails generates a series of files and directories that are common in most Java applications. Directories like `src` for placing source code files and a `web-app` directory that includes the common layout for Java web application(e.g., `/WEB-INF/`, `/META-INF/`, `css`, `images`, and `js`).

Right out of the box, Grails saves you time by putting these common Java application constructs together using a single command.

A Grails Application's File and Directory Structure

Since some of these files and directories are Grails specific, we will describe the purpose behind each one:

- **application.properties:** Used to define an application's properties, including the Grails version, servlet version, and an application's name
- **grails-app:** A directory containing the core of an application, which further contains the following folders:
 1. **assets:** A directory containing an application's static resources (i.e. `.css` and `.js` files)
 2. **conf:** A directory containing an application's configuration sources
 3. **controllers:** A directory containing an application's controllers files
 4. **domain:** A directory containing an application's domain files
 5. **i18n:** A directory containing an application's internationalization (i18n) files

6. `migrations`:
 7. `services`: A directory containing an application's service files
 8. `taglib`: A directory containing an application's tag libraries
 9. `utils`: A directory containing an application's utility files
 10. `views`: A directory containing an application's view files
- `lib`: Directory used for libraries (i.e., JARs)
 - `scripts`: Directory used for scripts
 - `src`: Directory used for an application's source code files; contains two subfolders named `groovy` and `java`, for sources written in these respective languages
 - `test`: Directory used for an application's test files. Contains two subfolders named `integration` and `unit`, for tests belonging to these respective types
 - `web-app`: Directory used for an application's deployment structure; contains the standard web archive (WAR) files and directory structure (e.g., `/WEB-INF/`, `/META-INF/`, `css`, `images`, and `js`)

Note Grails does not support Apache Maven out of the box. However, if you prefer to use Maven as your build tool, there is support for using Maven with Grails. Consult <http://www.grails.org/Maven+Integration>.

Running an Application

Grails comes preconfigured to run applications on an Apache Tomcat web container. Note the files pertaining to the Apache Tomcat container are one of those placed under a user's home directory (i.e., `.grails/<grails_version>/`), they are not visible. Similar to the creation of creating a Grails application, the process of running Grails applications is highly automated.

Placed under the root directory of a Grails application, invoke `grails run-app`. This command will trigger the build process for an application if it's needed, as well as start the Apache Tomcat web container and deploy the application.

Since Grails operates on conventions, an application is deployed under a context named after the project name. So for example, the application named `court` is deployed to the URL `http://localhost:8080/court/`. Figure 18-1 illustrates the default main screen for Grails applications.

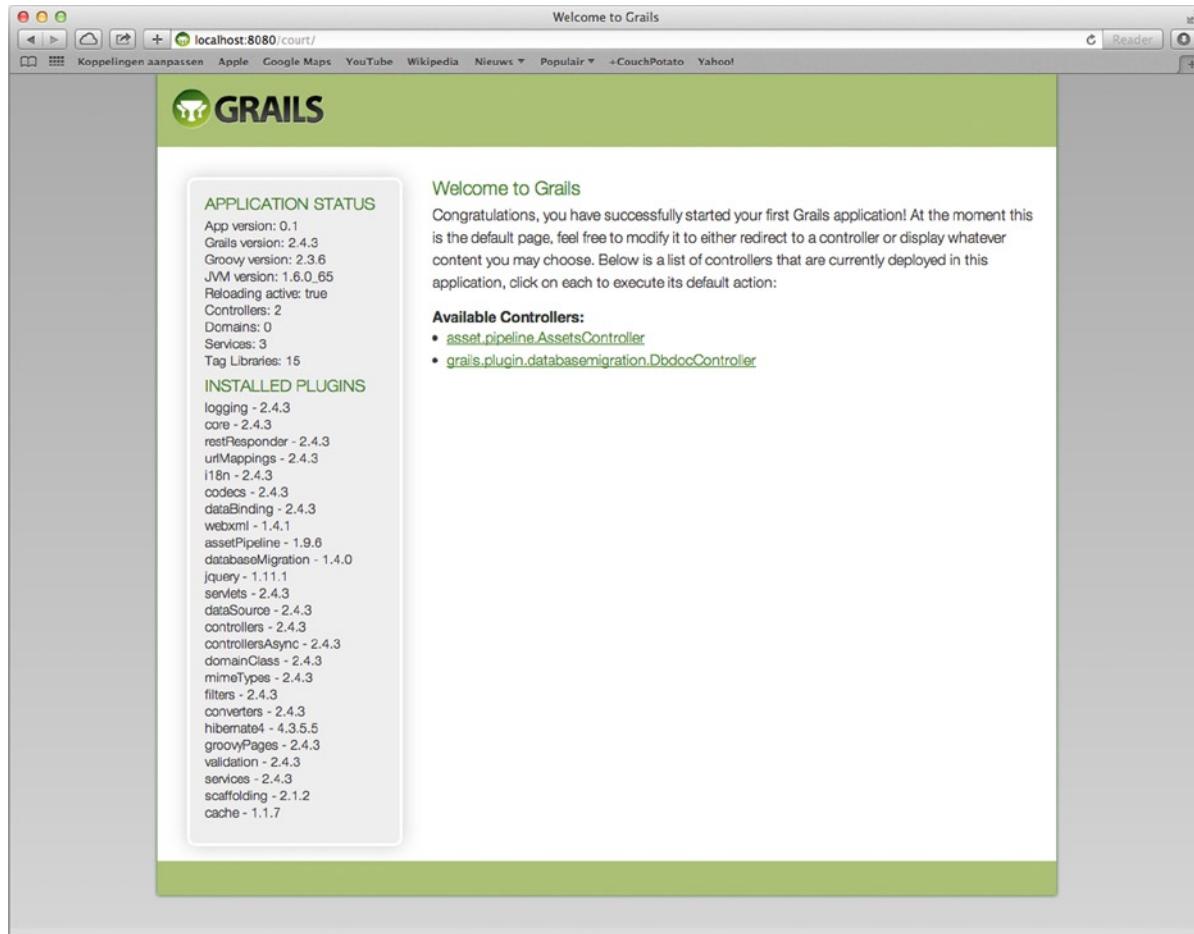


Figure 18-1. Default main screen for court Grails application

The application is still in its out of the box state. Next, we will illustrate how to create your first Grails construct in order to realize more time-saving procedures.

Creating Your First Grails Application Construct

Now that you have seen how easy it is to create a Grails application, let's incorporate an application construct in the form of a controller. This will further illustrate how Grails automates a series of steps in the development process of Java applications.

Placed under the root directory of a Grails application, invoke `grails create-controller welcome`. Executing this command will perform the following steps:

1. Create a controller named `WelcomeController.groovy` under the application directory `grails-app/controllers`.
2. Create a directory named `welcome` under the application directory `grails-app/views`.
3. Create a test class named `WelcomeControllerSpec.groovy` under the application directory `test/unit`.

As a first step, let's analyze the contents of the controller generated by Grails. The contents of the `WelcomeController.groovy` controller are the following:

```
class WelcomeController {
    def index = {}
}
```

If you're unfamiliar with Groovy, the syntax will seem awkward. But it's simply a class named `WelcomeController` with a method named `index`. The purpose is the same as the Spring MVC controllers you created back in Chapter 4, `WelcomeController`, represents a controller class whereas the method, `index`, represents a handler method. However, in this state the controller isn't doing anything. Modify it to reflect the following:

```
class WelcomeController {
    Date now = new Date()
    def index = {[today:now]}
}
```

The first addition is a `Date` object assigned to the `now` class field, to represent the system date. Since `def index = {}` represents a handler method, the addition of `[today:now]` is used as a return value. In this case, the return value represents a variable named `today` with the `now` class field, and that variable's value will be passed onto to the view associated with the handler method.

Having a controller and a handler method that returns the current date, you can create a corresponding view. If you place yourself under the directory `grails-app/views/welcome`, you will not find any views. However, Grails attempts to locate a view for the `WelcomeController` controller inside this directory, in accordance with the name of the handler method; this, once again, is one of the many conventions used by Grails.

Therefore, create a GSP page named `index.gsp` inside this directory with the following contents:

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>

<body>
    Welcome to Court Reservation System</h2>
    Today is <g:formatDate format="yyyy-MM-dd" date="${today}" />
</body>
</html>
```

This is a standard GSP page which makes use of the expressions and a tag library. When writing GSP pages the default tag library is available using the `g` tag. The `formatDate` tag renders a variable named `${today}`, which is precisely the name of the variable returned by the controller handler method named `index`.

Next, from the root directory of the Grails application, invoke the command `grails run-app`. This automatically builds the application, compiling the controller class and copying files where they are needed as well as starting the Apache Tomcat web container and deploying the application.

Following the same Grails convention process, the `WelcomeController` along with its handler methods and views will be accessible from the context path `http://localhost:8080/court/welcome/`. Since `index` is the default page used for context paths, if you open a browser and visit `http://localhost:8080/court/welcome/` or the explicit URL `http://localhost:8080/court/welcome/index`, you will see the previous JSP page that renders the current date as returned by the controller. Note the lack of view extensions in the URLs (i.e., `.jsp`). Grails hides the view technology by default, the reasons for this will become more evident in more advanced Grails scenarios.

As you repeat the simple steps needed to create an application controller and view, bear in mind you didn't have to create or modify any configuration files, manually copy files to different locations, or set up a web container to run the application. As an application moves forward, avoiding these scaffolding steps that are common in Java web applications can be a great way to reduce development time.

Exporting a Grails Application to a WAR

The previous steps were all performed in the confines of a Grails environment. That is to say, you relied on Grails to bootstrap a web container and run applications. However, when you want to run a Grails application in a production environment, you will undoubtedly need to generate a format in which to deploy the application to an external web container, which is a WAR file in the case of Java applications.

Placed under the root directory of a Grails application, invoke `grails war`. Executing this command generates a WAR file under the root directory in the form `<application-name>-<application-version>.war`. This WAR is a self-contained file with all the necessary elements needed to run a Grails application on any Java standard web container.

In the case of the court application, a file named `court-0.1.war` is generated in the root directory of the Grails application, and the application version is taken from the parameter `app.version` defined in the `application.properties` file.

In accordance with Apache Tomcat deployment conventions, a WAR named `court-0.1.war` would be accessible at a URL in the form `http://localhost:8080/court-0.1/`. WAR deployment to URL conventions may vary depending on the Java web container (e.g., Jetty or Oracle WebLogic).

18-3. Grails Plug-Ins

Problem

You want to use functionality from a Java framework or Java API inside Grails applications, while taking advantage of the same Grails techniques to save scaffolding.

The problem isn't simply using a Java framework or Java API in an application; this can be achieved by simply dropping the corresponding JARs into an application's `lib` directory. But rather having a Java framework or Java API *tightly* integrated with Grails, something that is provided in the form of Grails plug-ins.

By *tightly* integrated with Grails, we mean having the capacity to use short-cut instructions (e.g., `grails <plug-in-task>`) for performing a particular Java framework or Java API task or the ability to use functionality inside an application's classes or configuration files without resorting to scaffolding steps.

Solution

Grails actually comes with a few preinstalled plug-ins, even though this is not evident if you stick to using Grails out of the box functionality. However, there are many Grails plug-ins that can make working with a particular Java framework or Java API as productive a process as using Grails core functionality. Some of the more popular Grails plug-ins follow:

- *App Engine*: Integrates Google's App Engine SDK and tools with Grails.
- *Quartz*: Integrates the Quartz Enterprise Job Scheduler to schedule jobs and have them executed using a specified interval or cron expression.
- *Spring WS*: Integrates and supports the provisioning of web services, based on the Spring Web Services project.
- *Clojure*: Integrates Clojure and allows Clojure code to be executed in Grails artifacts.

To obtain a complete list of Grails plug-ins you can execute the command: `grails list-plugins`. This last command connects to the Grails plug-in repository and displays all the available Grails plug-ins. In addition, the command `grails plugin-info <plugin_name>` can be used to obtain detailed information about a particular plug-in. As an alternative, you can visit the Grails plug-in page located at <http://grails.org/plugin/home>.

Installing a Grails plug-in requires invoking the following command from a Grails application's root directory:
`grails install-plugin <plugin_name>`.

Removing a Grails plug-in requires invoking the following command from a Grails application's root directory:
`grails uninstall-plugin <plugin_name>`.

How It Works

A Grails plug-in follows a series of conventions that allow it to tightly integrate a particular Java framework or Java API with Grails. By default, Grails comes with the Apache Tomcat and Hibernate plug-ins preinstalled, and both of these are located under the `plugins` directory of the Grails distribution.

When you first create a Grails application, these plug-ins are copied to the Grails working directory `.grails/<grails_version>/plugins` located under a user's home directory. In addition, two files are generated under this same directory: `plugins-list-core.xml` and `plugins-list-default.xml`. The first of which contains plug-ins included in every Grails application, by default Apache Tomcat and Hibernate, and the second containing the entire list of available Grails plug-ins.

Every Grails application you create afterward includes the plug-ins defined in the `plugins-list-core.xml` file. This inclusion takes place by unzipping each plug-in under an application's working directory under `.grails/<grails_version>/projects/<project_name>/plugins/`. Once the plug-ins are unzipped, a Grails application is equipped to support whatever functionality is provided by a plug-in.

Besides these default plug-ins, additional plug-ins can be installed on a per-application basis. For example, to install the Clojure plug-in, you would execute the following command from an application's root directory:

```
grails install-plugin clojure
```

This last command downloads and initially copies the Clojure plug-in to the same Grails working directory, `.grails/<grails_version>/plugins`, located under a user's home directory. Once downloaded, the plug-in is copied as a zip file, as well as unzipped under an application's working directory under `.grails/<grails_version>/projects/<project_name>/plugins/`.

To remove a plug-in from a Grails application, you can invoke the following command:

```
grails uninstall-plugin clojure
```

This last command removes the unzipped plug-in from an application's working directory under `.grails/<grails_version>/projects/<project_name>/plugins/`, though the zipped plug-in file will remain. In addition, the downloaded plug-in remains under the Grails working directory `.grails/<grails_version>/plugins`, located under a user's home directory.

This installation and uninstallation of plug-ins is done on a per-application basis. In order for a plug-in to be automatically added to a Grails application upon creation, you need to modify the `plugins-list-core.xml` file manually. This copies whatever plug-ins you deem necessary on all subsequent application you create, along with the default Apache Tomcat and Hibernate plug-ins.

Caution Besides the steps outlined here, we do not recommend you modify the structure of a plug-in or a Grails application working directory. Plug-ins always alter the structure of a Grails application to provide it with the necessary functionality. If you encounter problems with a Grails plug-in, we recommend you consult the plug-in's documentation or ask the plug-in maintainers on the Grails development mailing list at <http://grails.org/Mailing%20lists>.

18-4. Developing, Producing, and Testing in Grails Environments

Problem

You want to use different parameters for the same application on the basis of the environment (e.g., development, production, and testing) it's being run in.

Solution

Grails anticipates that a Java application can undergo various phases that require different parameters. These phases, or "environments" as they are called by Grails, can be, for instance, development, production, and testing.

The most obvious scenario involves data sources, where you are likely to use a different permanent storage system for development, production, and testing environments. Since each of these storage systems will use different connection parameters, it's easier to configure parameters for multiple environments and let Grails connect to each one depending on an application's operations.

In addition to data sources, Grails provides the same feature for other parameters that can change in between application environments, such as server URLs for creating an application's absolute links.

Configuration parameters for a Grails application environment are specified in the files located under an application's `/grails-app/conf/` directory.

How It Works

Depending on the operation you're performing, Grails automatically selects the most suitable environment: development, production, or testing.

For example, when you invoke the command `grails run-app`, this implies that you are still developing an application locally, so a development environment is assumed. In fact, when you execute this command, among the output you can see a line that reads:

`Environment set to development`

This means that whatever parameters are set for a development environment are used to build, configure and run the application.

Another example is the `grails war` command. Since exporting a Grails application to a stand-alone WAR implies you will be running it on an external web container, Grails assumes a production environment. In the output generated for this command, you will find a line that reads

`Environment set to production`

This means that whatever parameters are set for a production environment are used to build, configure, and export the application.

Finally, if you run a command like `grails test-app`, Grails assumes a testing environment. This means that whatever parameters are set for a testing environment are used to build, configure, and run tests. In the output generated for this command, you will find a line that reads

Environment set to test

Inside the configuration files located in an application's directory `/grails-app/conf/`, you can find sections in the following form:

```
environments {
    production {
        grails.serverURL = "http://www.domain.com"
    }
    development {
        grails.serverURL = "http://localhost:8080/${appName}"
    }
    test {
        grails.serverURL = "http://localhost:8080/${appName}"
    }
}
```

This last listing belongs to the `Config.groovy` file. It's used to specify different server URLs for creating an application's absolute links on the basis of an application's environment. Another example would be the section contained in the `DataSource.groovy` file, which is used to define data sources and illustrated next:

```
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop','update' , 'validate', ''
            url = "jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            "jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000;DB_CLOSE_ON_EXIT=FALSE"
            properties { ... }
        }
    }
}
```

In this last listing, different connection parameters are specified for permanent storage systems on the basis of an application's environment. This allows an application to operate on different data sets, as you will surely not want development modifications to take place on the same data set used in a production environment. It should be noted that this doesn't mean these last examples are the only parameters that are allowed to be configured on the

basis of an application's environment. You can equally place any parameter inside the corresponding environments { <environment_phase> } section. These last examples simply represent the most likely parameters to change in between an application's environments.

It's also possible to perform programming logic (e.g., within a class or script) on the basis of a given application's environment. This is achieved through the grails.util.Environment class. The following listing illustrates this process.

```
import grails.util.Environment
...
...
switch(Environment.current) {
    case Environment.DEVELOPMENT:
        // Execute development logic
        break
    case Environment.PRODUCTION:
        // Execute production logic
        break
}
```

This last code snippet illustrates how a class first imports the grails.util.Environment class. Then, on the basis of the Environment.current value, which contains the environment an application is being run on, the code uses a switch conditional to execute logic depending on this value.

Such a scenario can be common in areas like sending out emails or performing geolocation. It would not make sense to send out e-mails or determine where a user is located on a development environment, given that a development team's location is irrelevant in addition to not requiring an application's e-mail notifications.

Finally, it's worth mentioning that you can override the default environment used for any Grails command.

For example, by default, the grails run-app command uses the parameters specified for a development environment. If for some reason you want to run this command with the parameters specified for a production environment, you can do so using the following instruction: grails prod run-app. If you wish to use the parameters specified for a test environment, you can also do so by using the following instruction: grails test run-app.

By the same token, for a command like grails test-app, which uses the parameters specified for a test environment, you can use the parameters belonging to a development environment by using the command grails dev test-app. The same case applies for all other commands, by simply inserting the prod, test, or dev keyword after the grails command.

18-5. Creating an Application's Domain Classes

Problem

You need to define an application's domain classes.

Solution

Domain classes are used to describe an application's primary elements and characteristics. If an application is designed to attend reservations, it's likely to have a domain class for holding reservations. Equally, if reservations are associated with a person, an application will have a domain class for holding persons.

In web applications, domain classes are generally the first things to be defined, because these classes represent data that is saved for posterity—in a permanent storage system—so it interacts with controllers, as well as representing data displayed in views.

In Grails, domain classes are placed under the `/grails-app/domain/` directory. The creation of domain classes, like most other things in Grails, can be carried out by executing a simple command in the following form:

```
grails create-domain-class <domain_class_name>
```

This last command generates a skeleton domain class file named `<domain_class_name>.groovy` inside the `/grails-app/domain/` directory.

How It Works

Grails creates skeleton domain classes, but you still need to modify each domain class to reflect the purpose of an application.

Let's create a reservation system, similar to the one you created back in Chapter 4 to experiment with Spring MVC. Create two domain classes, one named `Reservation` and another named `Player`. To do so, execute the following commands:

```
grails create-domain-class Player
grails create-domain-class Reservation
```

By executing these last commands, a class file named `Player.groovy` and another one named `Reservation.groovy`¹ are placed under an application's `/grails-app/domain/` directory. In addition, corresponding unit tests files are also generated for each domain class under an application's `test/unit` directory, though testing will be addressed in a Recipe 18-10.

Next, open the `Player.groovy` class to edit its contents. The following statements in bold represent the declarations you need to add to this domain class:

```
class Player {
    statichasMany = [ reservations : Reservation ]
    String name
    String phone
    static constraints = {
        name(blahn:false)
        phone(blahn:false)
    }
}
```

The first addition, `statichasMany = [reservations : Reservation]`, represents a relationship among domain classes. This statement indicates that the `Player` domain class has a `reservations` field that has many `Reservation` objects associated with it. The following statements indicate that the `Player` domain class also has two `String` fields, one called `name` and another called `phone`.

The remaining element, `static constraints = { }`, defines constraints on the domain class. In this case, the declaration `name(blahn:false)` indicates that a `Player` object's `name` field cannot be left blank. While the declaration `phone(blahn:false)` indicates that a `Player` object cannot be created unless the `phone` field is provided with a value.

Once you modify the `Player` domain class, open the `Reservation.groovy` class to edit its contents. The following statements in bold represent the declarations you need to add to this domain class:

```
class Reservation {
    static belongsTo = Player
    String courtName;
    Date date;
    Player player;
```

```
String sportType;
static constraints = {
    sportType(inList:["Tennis", "Soccer"] )
    datevalidator: {
        if (it.getAt(Calendar.DAY_OF_WEEK) == "SUNDAY" &&←
            ( it.getAt(Calendar.HOUR_OF_DAY) < 8 || ←
              it.getAt(Calendar.HOUR_OF_DAY) > 22) ) {←
            return ['invalid.holidayHour']
        } else if ( it.getAt(Calendar.HOUR_OF_DAY) < 9 ||←
                    it.getAt(Calendar.HOUR_OF_DAY) > 21) {←
            return ['invalid.weekdayHour']
        }
    })
}
```

The first statement added to the Reservation domain class, static belongsTo = Player, indicates that a Reservation object always belongs to a Player object. The following statements indicate the Reservation domain class has a field named courtName of the type String, a field named date of the type Date, a field named player of the type Player and another field named sportType of the type String.

The constraints for the `Reservation` domain class are a little more elaborate than the `Player` domain class. The first constraint, `sportType(inList:["Tennis", "Soccer"])`, restricts the `sportType` field of a `Reservation` object to a string value of either Tennis or Soccer. The second constraint is a custom-made validator to ensure the `date` field of a `Reservation` object is within a certain hour range depending on the day of the week.

Now that you have an application's domain classes, you can create the corresponding views and controllers for an application.

Before proceeding, though, a word on Grails domain classes is in order. While the domain classes you created in this recipe provide you with a basic understanding of the syntax used to define Grails domain classes, they illustrate only a fraction of the features available in Grails domain classes.

As the relationship between domain classes grows more elaborate, more sophisticated constructs are likely to be required for defining Grails domain classes. This comes as a consequence of Grails relying on domain classes for various application functionalities.

For example, if a domain object is updated or deleted from an application's permanent storage system, the relationships between domain classes need to be well established. If relationships are not well established, there is a possibility for inconsistent data to arise in an application (e.g., if a person object is deleted, its corresponding reservations also need to be deleted to avoid an inconsistent state in an application's reservations).

Equally, a variety of constraints can be used to enforce a domain class's structure. Under certain circumstances, if a constraint is too elaborate, it's often incorporated within an application's controller prior to creating an object of a certain domain class. Though for this recipe, model constraints were used to illustrate the design of Grails domain classes.

18-6. Generating CRUD Controllers and Views for an Application's Domain Classes

Problem

You need to generate create, read, update, and delete (CRUD) controllers and views for an application's domain classes.

Solution

An application's domain classes by themselves are of little use. The data mapped to domain classes still needs to be created, presented to end users, and potentially saved for future use in a permanent storage system.

In web applications backed by permanent storage systems, these operations on domain classes are often referred to as CRUD operations. In the majority of web frameworks, generating CRUD controllers and views entails a substantial amount of work. This is on account of needing controllers capable of creating, reading, updating, and deleting domain objects to a permanent storage systems, as well as creating the corresponding views (e.g., JSP pages) for an end user to create, read, update, and delete these same objects.

However, since Grails operates on the basis of conventions, the mechanism for generating CRUD controllers and views for an application's domain classes is easy. You can execute the following command to generate the corresponding CRUD controller and views for an application's domain class:

```
grails generate-all <domain_class_name>
```

How It Works

Grails is capable of inspecting an application's domain classes and generating the corresponding controllers and views necessary to create, read, update, and delete instances belonging to an application's domain classes.

For example, take the case of the Player domain class you created earlier. In order to generate its CRUD controller and views, you only need to execute the following command from an application's root directory:

```
grails generate-all court.Player
```

A similar command would apply to the Reservation domain class. Simply execute the following command to generate its CRUD controller and views:

```
grails generate-all court.Reservation
```

So what is actually generated by executing these steps? If you saw the output for these commands you will have a pretty good idea, but I will recap the process here nonetheless.

1. Compile an application's classes.
2. Generate 12 properties files under the directory `grails-app/i18n` to support an application's internationalization(e.g., `messages_<language>.properties`).
3. Create a controller named `<domain_class>Controller.groovy` with CRUD operations designed for an RDBMS, placed under an application's `grails-app/controllers` directory.
4. Create four views corresponding to a controller class's CRUD operations named `create.gsp`, `edit.gsp`, `list.gsp`, and `show.gsp`. Note that the `.gsp` extension stands for "Groovy Server Pages," which is equivalent to JavaServer Pages except it uses Groovy to declare programmatic statements instead of Java. These views are placed under an application's `grails-app/views/<domain_class>` directory.

Once you finish these steps, you can start the Grails application using `grails run-app` and work as an end user with the application. Yes, you read correctly; after performing these simple commands, the application is now ready to for end users. This is the mantra of Grails operating on conventions, to simplify the creation of scaffolding code through one word commands.

After the application is started, you can perform CRUD operations on the Player domain class at the following URLs:

- Create: <http://localhost:8080/court/player/create>
- Read: <http://localhost:8080/court/player/list> (for all players) or http://localhost:8080/court/player/show/<player_id>
- Update: http://localhost:8080/court/player/edit/<player_id>
- Delete: http://localhost:8080/court/player/delete/<player_id>

The page navigation between each view is more intuitive than these URLs have it to be, but we will illustrate with a few screen shots shortly. An important thing to be aware about these URLs is their conventions. Notice the pattern `<domain>/<app_name>/<domain_class>/<crud_action>/<object_id>`, where `<object_id>` is optional depending on the operation.

In addition to being used to define URL patterns, these conventions are used throughout an application's artifacts. For example, if you inspect the `PlayerController.groovy` controller, you can observe there are handler methods named like the various `<crud_action>` values. Similarly, if you inspect an application's backing RDBMS, you can note that the domain class objects are saved using the same `<player_id>` used in a URL.

Now that you're aware of how CRUD operations are structured in Grails applications, create a Player object by visiting the address <http://localhost:8080/court/player/create>. Once you visit this page, you can see an HTML form with the same field values you defined for the Player domain class.

Introduce any two values for the name and phone fields and submit the form. You've just persisted a Player object to a RDBMS. By default, Grails come preconfigured to use HSQLDB, an in-memory RDBMS. A future recipe will illustrate how to change this to another RDBMS, for now HSQLDB will suffice.

Next, try submitting the same form but, this time, without any values. Grails will not persist the Player object, it will instead show two warning messages indicating that the name and phone fields cannot be blank. Figure 18-2 illustrates this screen shot.

The screenshot shows a 'Create Player' form on a Grails application. The top navigation bar includes links for 'Home' and 'Player List'. The main title is 'Create Player'. Below the title, there are two error messages in a red box: 'Property [name] of class [class court.Player] cannot be null' and 'Property [phone] of class [class court.Player] cannot be null'. The form has two input fields: 'Name *' and 'Phone *'. Below the inputs is a link 'Reservations Add Reservation'. At the bottom left is a 'Create' button.

Figure 18-2. Grails domain class validation taking place in a view (in this case, an HTML form)

This validation process is being enforced on account of the statements, `name(blank:false)` and `phone(blank:false)`, which you placed in the `Player` domain class. You didn't need to modify an application's controllers or views or even create properties files for these error messages; everything was taken care of by Grails convention-based approach.

Note When using a HTML5 capable browser you won't be allowed to submit the form. Both input elements are marked `required` which prevent the form submission in these browsers. These rules are also added based on statement mentioned above.

Experiment with the remaining views available for the `Player` domain class, creating, reading, updating, and deleting objects directly from a web browser to get a feel for how Grails handles these tasks.

Moving along the application, you can also perform CRUD operations on the `Reservation` domain class at the following URLs:

- Create: `http://localhost:8080/court/reservation/create`
- Read: `http://localhost:8080/court/reservation/list` (for all reservations) or `http://localhost:8080/court/reservation/show/<reservation_id>`
- Update: `http://localhost:8080/court/reservation/edit/<reservation_id>`
- Delete: `http://localhost:8080/court/reservation/delete/<reservation_id>`

These last URLs serve the same purpose as those for the `Player` domain class. The ability to create, read, update, and delete objects belonging to the `Reservation` domain class from a web interface.

Next, let's analyze the HTML form used for creating `Reservation` objects, available at the URL `http://localhost:8080/court/reservation/create`. Figure 18-3 illustrates this form.

The screenshot shows a Grails application interface. At the top, there is a green header bar with the Grails logo and the word "GRAILS". Below the header, there is a navigation bar with links for "Home" and "Reservation List". The main content area has a title "Create Reservation". Below the title are four form fields: "Sport Type * Tennis" (with a dropdown arrow), "Date * 4 August 2014" (with three dropdown arrows for day, month, and year), "Court Name *" (an empty text input field), and "Player * court.Player : 1" (with a dropdown arrow). At the bottom of the form is a "Create" button with a small icon.

Figure 18-3. Grails domain class HTML form, populated with domain objects from separate class

Figure 18-3 is interesting in various ways. Though the HTML form is still created on the basis of the fields of the Reservation domain class, just like the HTML form for the Player domain class, notice that it has various prepopulated HTML select menus.

The first select menu belongs to the sportType field. Since this particular field has a definition constraint to have a string value of either Soccer or Tennis, Grails automatically provides a user with these options instead of allowing open-ended strings and validating them afterward.

The second select menu belongs to the date field. In this case, Grails generates various HTML select menus representing a date and time to make the date-selection process easier, instead of allowing open-ended dates and validating them afterward.

The third select menu belongs to the player field. This select menu's options are different in the sense they are taken from the Player objects you've created for the application. The values are being extracted from querying the application's RDBMS; if you add another Player object, it will automatically become available in this select menu.

In addition, a validation process is performed on the date field. If the selected date does not conform to a certain range, the Reservation object cannot be persisted and a warning message appears on the form.

Try experimenting with the remaining views available for the Reservation domain class, creating, reading, updating, and deleting objects directly from a web browser.

Finally, just to keep things in perspective, realize what your application is already doing in a few steps: validating input; creating, reading, updating and deleting objects from a RDBMS; completing HTML forms from data in a RDBMS; and supporting internationalization. And you haven't even modified a configuration file, been required to use HTML, or needed to deal with SQL or object-relational mappers (ORMs).

18-7. Internationalization (I18n) Message Properties

Problem

You need to internationalize values used throughout a Grails application.

Solution

By default, all Grails applications are equipped to support internationalization. Inside an application's /grails-app/i18n/ folder, you can find series of *.properties files used to define messages in 12 languages.

The values declared in these *.properties files allow Grails applications to display messages based on a user's languages preferences or an application's default language. Within a Grails application, the values declared in *.properties files can be accessed from places that include views (JSP or GSP pages) or an application's context.

How It Works

Grails determines which locale (i.e., from an internationalization properties file) to use for a user based on two criteria:

- The explicit configuration inside an application's /grails-app/conf/spring/resource.groovy file
- A user's browser language preferences

Since the explicit configuration of an application's locale takes precedence over a user's browser language preferences, there is no default configuration present in an application's resource.groovy file.

This ensures that if a user's browser language preferences are set to Spanish (es) or German (de), a user is served messages from the Spanish or German properties files (e.g., messages_es.properties or messages_de.properties). On the other hand, if an application's resource.groovy file is configured to use Italian (it), it won't matter what a user's browser language preferences are; a user will always be served messages from the Italian properties file (e.g., messages_it.properties).

Therefore, you should define an explicit configuration inside an application's `/grails-app/conf/spring/resource.groovy` file, only if you want to coerce users into using a specific language locale. For example, maybe you don't want to update several internationalization properties files or maybe you simply value uniformity.

Since Grails internationalization is based on Spring's Locale Resolver, you need to place the following contents inside an application's `/grails-app/conf/spring/resource.groovy` file, in order to force a specific language on users:

```
import org.springframework.web.servlet.i18n.SessionLocaleResolver

beans = {
    localeResolver(SessionLocaleResolver) {
        defaultLocale= Locale.ENGLISH
        Locale.setDefault (Locale.ENGLISH)
    }
}
```

By using this last declaration, any visitor is served messages from the English properties files (e.g. `messages_en.properties`) irrespective of his or her browser's language preferences.

It's also worth mentioning that if you specify a locale for which there are no available properties files, Grails reverts back to using the default `messages.properties` file, which by default is written in English though you can easily modify its values to reflect another language if you prefer. This same scenario applies when a user's browser language preferences are the defining selection criteria (e.g., if a user browser's language preferences are set for Chinese and there is no Chinese properties file, Grails falls back to using the default `messages.properties` file).

Now that you know how Grails determines which properties file to choose from in order to serve localized content, let's take a look at the syntax of a Grails `*.properties` file.

```
default.paginate.next=Next
typeMismatch.java.net.URL=Property {0} must be a valid URL
default.blank.message=Property [{0}] of class [{1}] cannot be blank
default.invalid.email.message=Property [{0}] of class [{1}] with value
[{2}] is not a valid e-mail address
default.invalid.range.message=Property [{0}] of class [{1}] with value
[{2}] does not fall within the valid range from [{3}] to [{4}]
```

The first line is the simplest declaration possible in a `*.properties` file. If Grails encounters the property named `default.paginate.next` in an application, it will substitute it for the value `Next`, or whatever other value is specified for this same property based on a user's determining locale.

On certain occasions, it can be necessary to provide more explicit messages that are best determined from wherever a localized message is being called. This is the purpose of the keys `{0}`, `{1}`, `{2}`, `{3}`, and `{4}`; they are parameters used in conjunction with a localized property. In this manner, the localized message displayed to a user can convey more detailed information. Figure 18-4 illustrates localized and parameterized messages for the court application determined on a user browser's language preferences.

The figure displays four Grails application screens arranged in a 2x2 grid, illustrating localized and parameterized messages based on user preferences:

- Top Left (German):** 'Player anlegen' form. Validation errors: 'Die Eigenschaft [name] des Typs [class court.Player] darf nicht null sein' and 'Die Eigenschaft [phone] des Typs [class court.Player] darf nicht null sein'. Fields: Name * [text input], Phone * [text input]. Action button: Anlegen.
- Top Right (German):** 'Create Player' form. Validation errors: 'La propriété [name] de la classe [class court.Player] ne peut pas être nulle' and 'La propriété [phone] de la classe [class court.Player] ne peut pas être nulle'. Fields: Name * [text input], Phone * [text input]. Action button: Create.
- Bottom Left (Spanish):** 'Crear Player' form. Validation errors: 'La propiedad [name] de la clase [class court.Player] no puede ser nula' and 'La propiedad [phone] de la clase [class court.Player] no puede ser nula'. Fields: Name * [text input], Phone * [text input]. Action button: Crear.
- Bottom Right (Spanish):** 'Crea Player' form. Validation errors: 'La proprietà [name] della classe [class court.Player] non può essere nulla' and 'La proprietà [phone] della classe [class court.Player] non può essere nulla'. Fields: Name * [text input], Phone * [text input]. Action button: Crea.

Figure 18-4. Grails localized and parameterized messages, determined on a user browser's language preferences (Left-Right, Top-Down: Spanish, German, Italian, and French)

Armed with this knowledge, define the following four properties inside Grails message.properties files:

```
invalid.holidayHour=Invalid holiday hour
invalid.weekdayHour=Invalid weekday hour
welcome.title=Welcome to Grails
welcome.message=Welcome to Court Reservation System
```

Next, it's time to explore how property placeholders are defined in Grails applications.

Back in Recipe 18-5, "Defining an Application's Domain Classes," you might not have realized it, but you declared a localized property for the Reservation domain class. In the validation section (`static constraints = { }`), you created this statement in the form:

```
return ['invalid.weekdayHour']
```

If this statement is reached, Grails attempts to locate a property named `invalid.weekdayHour` inside a properties file and substitute its value on the basis of a user's determining locale.

It's also possible to introduce localized properties into an application's views. For example, you can modify the GSP page created in Recipe 18-2 and located under /court/grails-app/views/welcome/index.gsp to use the following

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="g" uri="http://grails.codehaus.org/tags" %>
<html>
<!DOCTYPE html>
<html>
<head>
    <title><g:message code="welcome.title"/></title>
</head>

<body>
<g:message code="welcome.message"/></h2>
Today is <g:formatDate format="yyyy-MM-dd" date="\${today}"/>
</body>
</html>
```

This GSP page uses the `<g:message>` tag then using the `code` attribute, the properties `welcome.title` and `welcome.message` are defined, both of which will be replaced with the corresponding localized values once the JSP is rendered.

18-8. Changing Permanent Storage Systems

Problem

You want to change a Grails application's permanent storage system to your favorite RDBMS.

Solution

Grails is designed to use a RDBMS as a permanent storage system. By default, Grails comes preconfigured to use HSQLDB. HSQLDB is a database that is automatically started by Grails upon deploying an application (i.e., executing `grails run-app`).

However, the simplicity of HSQLDB can also be its primary drawback. Every time an application is restarted in development and testing environments, HSQLDB loses all its data since it's configured to operate in memory. And even though Grails applications in a production environment are configured with HSQLDB to store data permanently on a file, HSQLDB feature set may be seen as a limited for certain application demands.

You can configure Grails to use another RDBMS by modifying an application's `DataSource.groovy` file, located under the `grails-app/conf` directory. Inside this file, you can configure up to three RDBMS, one for each environment—development, production, and testing—undertaken by an application. See Recipe 18-4, “Developing, producing, and testing Grails Environments,” for more on development, production, and testing environments in Grails applications.

How It Works

Grails relies on the standard Java JDBC notation to specify RDBMS connection parameters, as well as on the corresponding JDBC drivers provided by each RDBMS vendor, to create, read, update, and delete information.

One important aspect you need to be aware of if you change RDBMS, is that Grails uses an ORM called Groovy Object Relational Mapper (GORM) to interact with a RDBMS.

The purpose behind GROM is the same as all other ORM solutions—to allow you to concentrate on an application's business logic, without worrying about the particularities of an RDBMS implementation, which can range from discrepancies in data types to working with SQL directly. GROM allows you to design an application's domain classes and maps your design to the RDBMS of your choice.

Setting Up an RDBMS Driver

The first step you need to take in changing Grails default RDBMS is to install the JDBC driver for the RDBMS of your choice inside an application's lib directory. This allows the application access to the JDBC classes needed to persist objects to a particular RDBMS.

Configuring an RDBMS Instance

The second step consists of modifying the `DataSource.groovy` file located under an application's `grails-app/conf` directory. Inside this file, there are three sections for defining an RDBMS instance.

Each RDBMS instance corresponds to a different possible application environment: development, production, and testing. Depending on the actions you take, Grails chooses one of these instances to perform any permanent storage operations an application is designed to do. See Recipe 18-4, “Developing, producing, and testing Grails Environments,” for more on development, production, and testing environments in Grails applications.

However, the syntax used for declaring a RDBMS in each of these sections is the same. Table 18-1 contains the various properties that can be used in a `dataSource` definition for the purpose of configuring a RDBMS.

Table 18-1. *dataSource properties for configuring a RDBMS*

Property	Definition
<code>driverClassName</code>	Class name for the JDBC driver
<code>username</code>	Username to establish a connection to a RDBMS
<code>password</code>	Password to establish a connection to a RDBMS
<code>url</code>	URL connection parameters for a RDBMS
<code>pooled</code>	Indicates whether to use connection pooling for a RDMBS; defaults to <code>true</code>
<code>jndiName</code>	Indicates a JNDI connection string for a data source. (This is an alternative to configuring <code>driverClassName</code> , <code>username</code> , <code>password</code> , and <code>url</code> directly in Grails and instead relying on a data source being configured in a web container.)
<code>logSql</code>	Indicates whether to enable SQL logging
<code>dialect</code>	Indicates the RDBMS dialect to perform operations.
<code>properties</code>	Used to indicate extra parameters for RDBMS operation
<code>dbCreate</code>	Indicates auto-generation of RDBMS data definition language (DDL)
<code>dbCreate value</code>	Definition
<code>create-drop</code>	Drops and recreates the RDBMS DDL when Grails is run (Warning: Deletes all existing data in the RDBMS.)
<code>create</code>	Creates the RDBMS DDL if it doesn't exist, but doesn't modify if it does (Warning: Deletes all existing data in the RDBMS.)
<code>update</code>	Creates the RDBMS DDL if it doesn't exist or update if it does

If you've used a Java ORM, such as Hibernate or EclipseLink, the parameters in Table 18-1 should be fairly familiar. The following listing illustrates the `dataSource` definition for a MySQL RDBMS:

```
dataSource {
    dbCreate = "update"
    url = "jdbc:mysql://localhost/grailsDB"
    driverClassName = "com.mysql.jdbc.Driver"
    username = "grails"
    password = "groovy"
}
```

Of the properties in this last definition, the one you should be most careful with is `dbCreate`, since it can destroy data in a RDBMS. In this case, the `update` value is the most conservative of all three available values, as explained in Table 18-1.

If you're using a production RDBMS, then `dbCreate="update"` is surely to be your preferred strategy, since it doesn't destroy any data in the RDBMS. If, on the other hand, a Grails application is undergoing testing, you are likely to *want* data in a RDBMS being cleaned out on every test run, thus a value like `dbCreate="create"` or `dbCreate="create-drop"` would be more common. For a development RDBMS, which of these options you select as the better strategy depends on how advanced a Grails application is in terms of development.

Grails also allows you to use a RDBMS configured on a web container. In such cases, a web container, such as Apache Tomcat, is set up with the corresponding RDBMS connection parameters and access to the RDBMS is made available through JNDI. The following listing illustrates the `dataSource` definition to access RDBMS via JNDI:

```
dataSource {
    jndiName = "java:comp/env/grailsDataSource"
}
```

Finally, it's worth mentioning that you can configure a `dataSource` definition to take effect on an application's various environments, while further specifying properties for each specific environment. This configuration is illustrated in the following listing:

```
dataSource {
    driverClassName = "com.mysql.jdbc.Driver"
    username = "grails"
}
environments {
    production {
        dataSource {
            url = "jdbc:mysql://localhost/grailsDBPro"
            password = "production"
        }
    }
    development {
        dataSource {
            url = "jdbc:mysql://localhost/grailsDBDev"
            password = "development"
        }
    }
}
```

As this last listing illustrates, a `dataSource`'s `driverClassName` and `username` properties are defined globally, taking effect on all environments, while other `dataSource` properties are declared specifically for each individual environment.

18-9. Logging

Problem

You want to customize the logging output generated by a Grails application.

Solution

Grails relies on Java Log4J to perform its logging operations. In doing so, all Log4J configuration parameters are specified inside the `Config.groovy` file located under an application's `/grails-app/conf` directory.

Given Log4J's logging versatility, a Grails application logging can be configured in various ways. This includes creating custom appenders, logging levels, console output, logging by artifacts, and custom logging layouts.

How It Works

Grails comes preconfigured with a basic set of Log4J parameters. Defined inside the `Config.groovy` file located under an application's `/grails-app/conf` directory, these Log4J parameters are the following:

```
log4j = {
    error   'org.codehaus.groovy.grails.web.servlet',           // controllers
            'org.codehaus.groovy.grails.web.pages',             // GSP
            'org.codehaus.groovy.grails.web.sitemesh',          // layouts
            'org.codehaus.groovy.grails.web.mapping.filter',   // URL mapping
            'org.codehaus.groovy.grails.web.mapping',           // URL mapping
            'org.codehaus.groovy.grails.commons',              // core / classloading
            'org.codehaus.groovy.grails.plugins',               // plugins
            'org.codehaus.groovy.grails.orm.hibernate',        // hibernate integration
            'org.springframework',
            'org.hibernate'
    warn    'org.mortbay.log'
```

The notation for this last listing follows the convention `<logging_level> '<package_name>'`. This implies that any logging operation occurring at any of the cited packages will be logged so long as it occurs within the specified logging level or a more severe level.

In Log4J parlance, each package is known as a *logger*. Log4J also has the following logging levels: `fatal`, `error`, `warn`, `info`, `debug`, and `trace`. `fatal` is the most severe. Grails thus follows a conservative default logging policy by using the `error` level on most of its packages. Specifying a less severe level (e.g., `debug`) would result in greater volumes of logging information, which may not be practical for most cases.

By default, all logging message are sent to the `stacktrace.log` file located under an application's root directory. And if applicable, to the standard output (i.e., console) of a running application. When you execute a `grails` command, you will observe logging messages sent to standard output.

Configuring Custom Appenders and Loggers

Log4J relies on appenders and loggers to offer versatile logging functionality. An appender is a location where logging information is sent (e.g., a file or standard output), whereas a logger is a location where logging information is generated (e.g., a class or package).

Grails is configured with a root Log4J logger, from which all other loggers inherit their behavior. The default Log4J logger can be customized in a Grails application using the following statement within the `log4j { }` section of an application's `Config.groovy` file:

```
root {
    error()
    additivity = true
}
```

This last statement defines a logger so that messages of an `error` level, or a more severe one, are logged to standard output. This is the reason you can see logging message from other loggers (e.g., a class or package) being sent to standard output, they all inherit the root logger's behavior, in addition to specifying their own log level.

On the other hand, Log4J appenders provide a means to send logging messages to various locations. There are four types of appenders available by default:

- `jdbc`: An appender that logs to a JDBC connection.
- `console`: An appender that logs to standard output.
- `file`: An appender that logs to a file.
- `rollingFile`: An appender that logs to a rolling set of files.

In order to define Log4J appenders in a Grails application, you need to declare them within the `log4j { }` section of an application's `Config.groovy` file, as follows:

```
appenders {
    file name:'customlogfile', file:'/logs/grails.log'
    rollingFile name:'rollinglogfile', maxFileSize:1024,file:'/logs/rollinggrails.log'
}
```

As you can see, Log4J appenders are defined in the form `<appender_type> name:<appender_name> <additional_appender_options>`. In order to use appenders, you simply need to add them to a corresponding logger where they can receive input.

The following declaration illustrates how to put together the use of appenders, loggers and logging levels:

```
root {
    debug 'stdout', 'customlogfile'
    additivity = true
}
```

This last listing overrides the default root Log4J logger. It indicates to use a `debug` level for outputting logging messages to both to the `stdout` appender (i.e., standard output or console) as well as the `customlogfile` appender, the last of which represent a file defined in the appender section. Be aware that a `debug` level generates a lot of logging information.

If you simply want to use an appender for a particular package (i.e., logger), you can do so using the following syntax:

```
error customlogfile:'com.apress.springwebrecipes.grails'
```

This last syntax is similar to the default notation included in Grails `<logging_level> '<package_name>'`, except it prefixes the name of an appender to the package.

Configuring Layouts

In addition to custom loggers and appenders, Log4J can also be customized to use a particular logging layout. There are four types of layouts available by default:

- `xml`: An XML log file
- `html`: An HTML log file
- `simple`: A simple textual log
- `pattern`: A pattern layout

By default, Log4J uses a pattern layout. However, you can configure a different logging layout on a per-appender basis. The following listing illustrates how to assign layouts to appenders:

```
appenders {
    file name:'customlogfile', file:'/logs/grails.log'
    layout:pattern(conversionPattern: '%c{2} %m%n')
    console name:'stdout', layout:simple
}
```

The first appender, `customlogfile`, is assigned a pattern layout. Whereas the second appender, `stdout`, is assigned a `simple` layout. Note that `stdout` is the built-in appender for standard output (i.e., `console`).

18-10. Running Unit and Integration Tests

Problem

To make sure that your application's classes are working as specified, you need to perform unit and integration tests on them.

Solution

Grails has built-in support for running both unit and integration tests on an application. Earlier when you generated Grails artifacts, such as an application's domain classes, you might recall a series of test classes were automatically generated.

In a Grails application, tests are placed under an application's test directory. Inside this directory, you will find three more folders: `unit` used to place an application's unit test classes, `integration` used to place an application's integration test classes, and `reports` used to place the results of performing an application's tests.

Similar to other functionality offered by Grails, much of the drudgery involved in setting up and configuring application tests is handled by Grails. You simply need to concentrate on designing tests.

Once you've designed an application's tests, running tests in Grails is as simple as executing the `grails test-app` command from an application's root directory.

How It Works

Grails bootstraps an environment necessary to perform application tests. This environment includes the libraries (i.e., JARs), permanent storage system (i.e., RDBMS), as well as any other artifact necessary to carry out unit and integration tests.

Let's start by analyzing the output of executing the `grails test-app` command, illustrated next:

```
Running script /springrecipes/grails-1.2/scripts/TestApp.groovy
Environment set to test
  [mkdir] Created dir: /springrecipes/Ch11/court/test/reports/html
  [mkdir] Created dir: /springrecipes/Ch11/court/test/reports/plain

Starting unit tests ...
Running tests of type 'unit'
  [groovyc] Compiling 3 source files to /home/web/.grails/1.2/➥
    projects/court/test-classes/unit
-----
Running 3 unit tests...
Running test WelcomeControllerTests...PASSED
Running test ReservationTests...PASSED
Running test PlayerTests...PASSED
Tests Completed in 2302ms ...

-----
Tests passed: 3
Tests failed: 0

-----
Starting integration tests ...
  [copy] Copying 1 file to /home/web/.grails/1.2/➥
    projects/court/test-classes/integration
  [copy] Copying 1 file to /home/web/.grails/1.2/➥
    projects/court/test-classes
Running tests of type 'integration'
No tests found in test/integration to execute ...

[junitreport] Processing /springrecipes/sourcecode/Ch11/➥
  court/test/reports/TESTS-TestSuites.xml to /tmp/null993113113
[junitreport] Loading stylesheet jar:file:/springrecipes/grails-1.2/➥
  lib/ant-junit-1.7.1.jar!/org/apache/tools/ant/taskdefs/optional/➥
  junit/xsl/junit-frames.xsl
[junitreport] Transform time: 2154ms
[junitreport] Deleting: /tmp/null993113113
```

The script `TestApp.groovy`, included in the Grails distribution, starts the testing process. Immediately after, you can see the Grails environment is set to `test`, meaning configuration parameters are taken from this type of environment (e.g., test RDBMS parameters). Next, there are three sections.

The first section indicates the execution of unit tests, which are taken from the `test/unit` directory under an application's root directory. In this case, three successful unit tests are performed, which correspond to the three skeleton test classes generated upon the creation of an application's domain classes. Since these test classes contain an empty test, they automatically validate one unit test.

The second section indicates the execution of integration tests, which are taken from the `test/integration` directory under an application's root directory. In this case, there are no classes found in this last directory, so no integration tests are performed.

The third and last section indicates the creation of reports for both unit and integration tests. In this case, reports are placed in an XML format under the `test/reports` directory of an application's root directory, as well as reports in the more user-friendly HTML and plain text formats under the corresponding `test/reports/html` and `test/reports/plain` subdirectories of an application's root directory.

Now that you know how Grails executes tests, let's modify the preexisting unit test classes to incorporate unit tests based on a domain class's logic. Given that Grails testing is based on the foundations of the JUnit testing framework (<http://www.junit.org/>), if you're unfamiliar with this framework, we advise you to look over its documentation to grasp its syntax and approach. The following sections assume a basic understanding of JUnit.

Add the following methods (i.e., unit tests) to the `PlayerSpec.groovy` class located under an application's `/test/unit/` directory:

```
void "Test Non Empty Player"() {
    given: "A valid player is constructed"
        def player = new Player(name:'James',phone:'118-1111')
        mockForConstraintsTests(Player, [player])
    when: "validate is called"
        def result = player.validate();
    then: "it should be valid"
        assertTrue player.validate()
}

void testEmptyName() {
    given: "A player without a name is constructed"
        def player = new Player(name:'',phone:'118-1111')
        mockForConstraintsTests(Player, [player])
    when: "validate is called"
        def result = player.validate();
    then: "The name should be rejected"
        assertFalse result
        assertEquals 'Name cannot be null', 'nullable',player.errors['name']
}

void testEmptyPhone() {
    given: "A player without a phone is constructed"
        def player = new Player(name:'James',phone='')
        mockForConstraintsTests(Player, [player])
    when: "validate is called"
        def result = player.validate()
    then: "The phone number should be rejected."
        assertFalse result
        assertEquals 'Phone cannot be null', 'nullable', player.errors['phone']
}
```

The first unit test creates a `Player` object and instantiates it with both a name and phone fields. In accordance with the constraints declared in the `Player` domain class, this type of an instance should always be valid. Therefore, the statement `assertTrue player.validate()` confirms the validation of this object is always true.

The second and third unit tests also create a `Player` object. However, notice in one test the `Player` object is instantiated with a blank name field, and in another, the `Player` object is instantiated with a blank phone field. In accordance with the constraints declared in the `Player` domain class, both instances should always be invalid. Therefore, the statements `assertFalse player.validate()` confirm the validation of such objects are always false. The `assertEquals` statements provide detailed results as to why the `assertFalse` declarations are false.

Next, add the following methods (i.e., unit tests) to the `ReservationSpec.groovy` class located under an application's `/test/unit/` directory:

```
void testReservation() {
    given:
        def calendar = Calendar.instance
        calendar.with {
            clear()
            set MONTH, OCTOBER
            set DATE, 15
            set YEAR, 2009
            set HOUR, 15
            set MINUTE, 00
        }

        def validDateReservation = calendar.getTime()
        def reservation = new Reservation(
            sportType:'Tennis', courtName:'Main',
            date:validDateReservation,player:new Player(name:'James',phone:'118-1111'))
        mockForConstraintsTests(Reservation, [reservation])

    expect:
        assertTrue reservation.validate()
    }

void testOutOfRangeDateReservation() {
    given:
        def calendar = Calendar.instance
        calendar.with {
            clear()
            set MONTH, OCTOBER
            set DATE, 15
            set YEAR, 2009
            set HOUR, 23
            set MINUTE, 00
        }

        def invalidDateReservation = calendar.getTime()
        def reservation = new Reservation(
            sportType:'Tennis',courtName:'Main',
            date:invalidDateReservation,player:new Player(name:'James',phone:'118-1111'))
        mockForConstraintsTests(Reservation, [reservation])

    expect:
        assertFalse reservation.validate()
        assertEquals 'Reservation date is out of range', 'invalid.weekdayHour',
        reservation.errors['date']
    }
}
```

```

void testOutOfRangeSportTypeReservation() {
    given:
        def calendar = Calendar.instance
        calendar.with {
            clear()
            set MONTH, OCTOBER
            set DATE, 15
            set YEAR, 2009
            set HOUR, 15
            set MINUTE, 00
        }

        def validDateReservation = calendar.getTime()
        def reservation = new Reservation(
            sportType:'Baseball',courtName:'Main',
            date:validDateReservation,player:new Player(name:'James',phone:'118-1111'))
        mockForConstraintsTests(Reservation, [reservation])

    expect:
        assertFalse reservation.validate()
        assertEquals 'SportType is not valid', 'inList',reservation.errors['sportType']
}

```

This last listing contains three unit tests designed to validate the integrity of `Reservation` objects. The first test creates a `Reservation` object instance and confirms that its corresponding values pass through the `Reservation` domain class's constraints. The second test creates a `Reservation` object that violates the domain class's date constraint and confirms such an instance is invalid. The third test creates a `Reservation` object that violates the domain class's `sportType` constraint and confirms such an instance is invalid.

If you execute the `grails test-app` command, Grails automatically executes all the previous tests and outputs the test results to the application's `/test/reports/` directory.

Now that you've created unit tests for a Grails application, let's explore the creation of integration tests.

Unlike unit tests, integration tests validate more elaborate logic undertaken by an application. Interactions between various domain classes or operations performed against a RDBMS are the realm of integration testing. In this sense, Grails aids the integration testing process by automatically bootstrapping a RDBMS and other application properties to perform integration tests. The "Grails Differences for Running Unit and Integration Tests" sidebar contains more details on the different aspects provided by Grails for running both unit and integration tests.

GRAILS DIFFERENCES FOR RUNNING UNIT AND INTEGRATION TESTS

Unit tests are designed to validate the logic contained in a single domain class. Because of this fact, besides automating the execution of such tests, Grails provides no type of bootstrapping properties for performing these type of tests.

This is the reason the previous unit tests relied on the special method `mockForConstraintsTests`. This method creates a mock object from a domain class that is used to access a class's dynamic methods (e.g., `validate`) needed to perform unit tests. In this manner, Grails maintains a low overhead for performing unit tests by not bootstrapping anything, leaving even the creation of mock objects to the creator of a test.

Integration tests are designed to validate more elaborate logic that can span a series of application classes. Therefore, Grails bootstraps not only a RDBMS for the purpose of running tests against this type of permanent storage system but also bootstraps a domain class's dynamic methods to simplify the creation of such tests. This of course entails additional overhead for performing such tests, compared to unit tests.

It's also worth mentioning that if you look closely at the skeleton test classes generated by Grails for both unit and integration tests, there aren't any difference among them. The only difference is that tests placed inside the integration directory have access to the series of provisions mentioned earlier, whereas those inside the unit directory do not. You could go down the route of placing unit tests inside the integration directory, but this is a matter for you to decide by considering convenience versus overhead.

Next, create an integration class for the application by executing the following command: `grails create-integration-test CourtIntegrationTest`. This generates an integration test class inside the application's `/test/integration/` directory.

Incorporate the following method (i.e., the integration test) into this last class to validate the RDBMS operations performed by the application:

```
void testQueries() {
    given: "2 Existing Players"
        // Define and save players
        def players = [ new Player(name:'James',phone:'118-1111'),
                        new Player(name:'Martha',phone:'999-9999') ]
        players*.save()

        // Confirm two players are saved in the database
        Player.list().size() == 2
    when: "Player James is retrieved"
        // Get player from the database by name
        def testPlayer = Player.findByName('James')
    then: "The phone number should match"
        // Confirm phone
        testPlayer.phone == '118-1111'
    when: "Player James is Updated"
        // Update player name
        testPlayer.name = 'Marcus'
        testPlayer.save()

    then: "The name should be updated in the DB"
        // Get updated player from the database, but now by phone
        def updatedPlayer = Player.findByPhone('118-1111')

        // Confirm name
        updatedPlayer.name == 'Marcus'

    when: "The updated player is deleted"
        // Delete player
        updatedPlayer.delete()

    then: "The player should be removed from the DB."
        // Confirm one player is left in the database
        Player.list().size() == 1
}
```

```
// Confirm updatedPlayer is deleted
def nonexistantPlayer = Player.findByPhone('118-1111')
nonexistantPlayer == null
}
```

This last listing performs a series of operations against an application's RDBMS, starting from saving two Player objects and then querying, updating, and deleting those objects from the RDBMS. After each operation, a validation step (e.g., `assertEquals`, `assertNull`) is performed to ensure the logic—in this case contained in the PlayerController controller class—operates as expected (i.e., the controller `list()` method returns the correct number of objects in the RDBMS).

By default, Grails performs RDBMS test operations against HSQLDB. However, you can use any RDBMS you like. See Recipe 18-8, "Changing Permanent Storage Systems," for details on changing Grails RDBMS.

Finally, it's worth mentioning that if you wish to execute a single type of test (i.e., unit or integration), you can rely on the command flags `-unit` or `-integration`. Executing the `grails test-app -unit` command performs only an application's unit tests, whereas executing the `grails test-app -integration` command performs only an application's integration tests. This can be helpful if you have a large amount of both tests, since it can cut down on the overall time needed to perform tests.

18-11. Using Custom Layouts and Templates

Problem

You need to customize layouts and templates to display an application's content.

Solution

By default, Grails applies a global layout to display an application's content. This allows views to have a minimal set of display elements (e.g., HTML, CSS, and JavaScript) and inherit their layout behavior from a separate location.

This inheritance process allows application designers and graphic designers to perform their work separately, with application designers concentrating on creating views with the necessary data and graphic designers concentrating on the layout (i.e., aesthetics) of such data.

You can create custom layouts to include elaborate HTML displays, as well as custom CSS or JavaScript libraries.

Grails also supports the concept of templates, which serve the same purpose as layouts, except applied at a more granular level. In addition, it's also possible to use templates for rendering a controller's output, instead of a view as in most controllers.

How It Works

Inside the `/grails-app/view/` directory of an application, you can find a subdirectory called `layouts`, containing the layouts available to an application. By default, there is a file named `main.gsp` whose contents are the following:

```
<!DOCTYPE html>
<!--[if lt IE 7 ]> <html lang="en" class="no-js ie6"> <![endif]-->
<!--[if IE 7 ]> <html lang="en" class="no-js ie7"> <![endif]-->
<!--[if IE 8 ]> <html lang="en" class="no-js ie8"> <![endif]-->
<!--[if IE 9 ]> <html lang="en" class="no-js ie9"> <![endif]-->
<!--[if (gt IE 9)|(IE)]><!-- <html lang="en" class="no-js"><!--<![endif]-->
```

```

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title><g:layoutTitle default="Grails"/></title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="shortcut icon" href="${assetPath(src: 'favicon.ico')}" type="image/x-icon">
  <link rel="apple-touch-icon" href="${assetPath(src: 'apple-touch-icon.png')}">
  <link rel="apple-touch-icon" sizes="114x114"
    href="${assetPath(src: 'apple-touch-icon-retina.png')}">
  <asset:stylesheet src="application.css"/>
  <asset:javascript src="application.js"/>
  <g:layoutHead/>
</head>
<body>
  <div id="grailsLogo" role="banner">
    <a href="http://grails.org"><asset:image src="grails_logo.png" alt="Grails"/></a>
  </div>
  <g:layoutBody/>
  <div class="footer" role="contentinfo"></div>
  <div id="spinner" class="spinner" style="display:none;">
    <g:message code="spinner.alt" default="Loading&hellip;"/>
  </div>
</body>
</html>

```

Though apparently a simple HTML file, this last listing contains several elements that are used as placeholders in order for application views (i.e., JSP and GSP pages) to inherit the same layout.

The first of such elements are Groovy tags appended to the `<g:>` namespace. The `<g:layoutTitle>` tag is used to define the contents of a layout's title section. If a view inherits the behavior from this layout and lacks such a value, Grails automatically assigns the `Grails` value, as indicated by the `default` attribute. On the other hand, if an inheriting view has such a value, it's displayed in its place.

The `<g:layoutHead>` tag is used to define the contents of a layout's head section. Any values declared in the head of a view head inheriting this layout are placed in this location upon rendering.

The `<asset:javascript library="application">` tag allows any view inheriting this layout automatic access to JavaScript libraries. Upon rendering, this element is transformed into the following: `<script type="text/javascript" src="/court/assets/application.js"></script>`. Bear in mind JavaScript libraries have to be placed inside the Grails `/<app-name>/web-app/assets/javascripts` subdirectory; `<app-name>` in this case corresponds to `court`.

Moving along, you will also find several declarations in the form `${assetPath*}` with a `src` attribute. Such statements are translated by Grails to reflect a resource contained in an application. So for example, the statement `${assetPath(src: 'favicon.ico') }` is transformed into `/court/assets/images/favicon.ico`. Notice the addition of the application's name (i.e., context path) to the transformed values. This allows the layout to be reused in several applications while referencing the same image, the last of which should be placed under the Grails `/court/web-app/assets/images` subdirectory.

Now that you know how a Grails layout is structured, let's take a look at how a view inherits its behavior. If you open any of the views generated by the application controllers created earlier—`player`, `reservation` or `welcome` (also located under the `views` directory)—you will find the following statement used to inherit behavior from a Grails layout:

```
<meta name="layout" content="main"/>
```

The `<meta>` tag is a standard HTML tag that has no effect on a page's display but is used by Grails to detect the layout from which a view should inherit its behavior. By using this last statement, a view is automatically rendered with the layout named `main`, which is precisely the template described earlier.

Looking further into a view's structure, you will notice that all generated views are structured as standalone HTML pages; they contain `<html>`, `<body>` and other such HTML tags, similar to the layout template. This doesn't mean, however, that a page will contain duplicate HTML tags upon rendering. Grails automatically sorts out the substitution process by placing a view's `<title>` content inside the `<g:layoutTitle>` tag, a view's `<body>` content inside the `<g:layoutBody />` tag, and so on.

What happens if you remove `<meta>` tag from a Grails view? On the face of it, the answer to this question is obvious: no layout is applied upon rendering a view, which also implies no visual elements are rendered (e.g., images, menus, and CSS borders). However, since Grails operates on the basis of conventions, Grails always attempts to apply a layout on the basis of a controller's name.

For example, even if the views corresponding to the `reservation` controller have no `<meta name="layout">` tag declaration's associated with them, if a layout named `reservation.gsp` is present inside an application's layout directory, it will be applied to all views corresponding to the controller.

Though layouts provide an excellent foundation on which to modularize an application's views, they are only applicable to a view's entire page. Providing a more granular approach, templates allow certain chunks of a view's page be made reusable.

Take the case of an HTML section used to display a player's reservations. You'd like to display this information on all views corresponding to this controller as a reminder. Placing this HTML section explicitly on all views not only results in more initial work but can also result in more ongoing work in case such an HTML section changes.

To facilitate this inclusion process, a template can be used. The following listing illustrates the contents of a template named `_reservationList.gsp`:

```
<table>
<g:each in="${reservationInstanceList}" status="i" var="reservationInstance">
    <tr class="${(i % 2) == 0 ? 'odd' : 'even'}">
        <td><g:link action="show" id="${reservationInstance.id}">
            ${fieldValue(bean:reservationInstance, field:'id')}</g:link></td>
        <td>${fieldValue(bean:reservationInstance, field:'sportType')}</td>
        <td>${fieldValue(bean:reservationInstance, field:'date')}</td>
        <td>${fieldValue(bean:reservationInstance, field:'courtName')}</td>
        <td>${fieldValue(bean:reservationInstance, field:'player')}</td>
    </tr>
</g:each>
</table>
```

This last template generates an HTML table relying on the Groovy tag `<g:each>` with a list of reservations. The underscore (`_`) prefix used to name the file is a notation by Grails to differentiate between templates and stand-alone views; templates are always prefixed with an underscore.

In order to use this template inside a view, you need to use the `<g:render>` tag illustrated here:

```
<g:render template="reservationList" model="[reservationList:reservationInstanceList]" />
```

In this case, the `<g:render>` tag takes two attributes: the `template` attribute to indicate the name of a template and the `model` attribute to pass reference data needed by a template.

Another variation of the `<g:render>` tag includes a template's relative and absolute locations. By declaring `template="reservationList"`, Grails attempts to locate a template in the same directory as the view in which it's declared. To facilitate reuse, templates can be loaded from a common directory for which absolute directories are used. For example, a view with a statement in the form `template="/common/reservationList"` would attempt to locate a template named `_reservationList.gsp` under an application's `grails-app/views/common` directory.

Finally, it's worth mentioning that a template can also be used by a controller to render its output. For example, most controllers return control to a view using the following syntax:

```
render view:'reservations', model:[reservationList:reservationList]
```

However, it's also possible to return control to a template using the following syntax:

```
render template:'reservationList', model:[reservationList:reservationList]
```

By using this last render statement, Grails attempts to locate a template by the name `_reservationList.gsp`.

18-12. Using GORM Queries

Problem

You want to perform queries against an application's RDBMS.

Solution

Grails performs RDBMS operations using GORM. GORM is based on the popular Java ORM Hibernate, allowing Grails applications to perform queries using Hibernate Query Language (HQL).

However, in addition to supporting the use of HQL, GORM also has a series of built-in functionalities that make querying a RDBMS very simple.

How It Works

In Grails, queries against a RDBMS are generally performed from within controllers. If you inspect any of the court application controllers, one of the simplest queries is the following:

```
Player.get(id)
```

This query is used to obtain a `Player` object with a particular ID. Under certain circumstances though, an application can be required to perform queries on another set of criteria.

For example, `Player` objects in the court application have the `name` and `phone` fields, as defined in the `Player` domain class. GORM supports the querying of domain objects on the basis of its field names. It does so by offering methods in the form `findBy<field_name>`, as illustrated here:

```
Player.findByName('Henry')
Player.findByPhone('118-1111')
```

These two statements are used to query a RDBMS and obtain a `Player` object on the basis of a name and phone. These methods are called dynamic finders, since they are made available by GORM on the basis of a domain class's fields.

In a similar fashion, the `Reservation` domain class having its own field names will have dynamic finders like `findByPlayer()`, `findByCourtName()`, and `findByDate()`. As you can see, this process simplifies the creation of queries against an RDBMS in Java applications.

In addition, dynamic finder methods can also use comparators to further refine a query's results. The following snippet illustrates how to use a comparator to extract `Reservation` objects in a particular date range:

```
def now = new Date()
def tomorrow = now + 1
def reservations = Reservation.findByDateBetween( now, tomorrow )
```

Besides the `Between` comparator, another comparator that can be of use in the court application is the `Like` comparator. The following snippet illustrates the use of the `Like` comparator to extract `Player` objects with names starting with the letter A.

```
def letterAPlayers = Player.findByNameLike('A%')
```

Table 18-2 describes the various comparators available for dynamic finder methods.

Table 18-2. GORM dynamic finder comparators

GORM comparator	Query
<code>InList</code>	If value is present in a given list of values
<code>LessThan</code>	For lesser object(s) than the given value
<code>LessThanEquals</code>	For lesser or equal object(s) than the given value
<code>GreaterThan</code>	For greater object(s) than the given value
<code>GreaterThanOrEqual</code>	For greater or equal object(s) than the given value
<code>Like</code>	For object(s) like the given value
<code>Ilike</code>	For object(s) like the given value in a case insensitive manner
<code>NotEqual</code>	For object(s) not equal to the given value
<code>Between</code>	For object(s) between to the two given values
<code>IsNotNull</code>	For not null object(s); uses no arguments
<code>IsNull</code>	For null object(s); uses no arguments

GORM also supports the use of Boolean logic (and/or) in the construction of dynamic finder methods. The following snippet demonstrates how to perform a query for `Reservation` objects that satisfy both a certain court name and a date in the future.

```
def reservations = Reservation.findAllByCourtNameLikeAndDateGreaterThan("%main%", new Date() + 7)
```

In a similar fashion, the `Or` statement (instead of `And`) could have been used in this last dynamic finder method to extract `Reservation` objects that satisfy at least one of the criteria.

Finally, dynamic finder methods also support the use of pagination and sorting to further refine queries.

This is achieved by appending a map to the dynamic finder method. The following snippet illustrates how to limit the number of results in a query, as well as define its sorting and order properties:

```
def reservations = Reservation.findAllByCourtName("%main%", [ max: 3, sort: "date", order: "desc" ] )
```

As outlined at the start of this recipe, GORM also supports the use HQL to execute queries against a RDBMS. Though more verbose and error prone than the preceding listing, the following one illustrates several equivalent queries using HQL:

```
def letterAPlayers = Player.findAll("from Player as p where p.name like 'A%'")
def reservations = Reservation.findAll("from Reservation as r
    where r.courtName like '%main%' order by r.date desc", [ max: 3 ] )
```

18-13. Creating Custom Tags

Problem

You want to execute logic inside a Grails view that is not available through a prebuilt GSP or JSTL tag and yet not resort to the inclusion of code in a view.

Solution

A Grails view can contain display elements (e.g., HTML tags), business logic elements (e.g., GSP or JSTL tags) or straightforward Groovy or Java code to achieve its display objectives.

On certain occasions, a view can require a unique combination of display elements and business logic. For example, displaying the reservations of a particular player on a monthly basis requires the use of custom code. To simplify the inclusion of such a combination and facilitate its reuse in multiple views a custom tag can be used.

How It Works

To create custom tags, you can use the `grails create-tag-lib <tag-lib-name>` command. This command creates a skeleton class for a custom tag library under an application's `/grails-app/tag-lib/` directory.

Knowing this, let's create a custom tag library for the court application designed to display special reservation offers. The first custom tag will detect the current date and based on this information display special reservation offers. The end result being the capacity to use a tag like `<g:promoDailyAd/>` inside an application's view, instead of placing in-line code in a view or performing this logic in a controller.

Execute the `grails create-tag-lib DailyNotice` command to create the custom tag library class. Next, open the generated `DailyNoticeTagLib.groovy` class located under an application's `/grails-app/tag-lib/` directory, and add the following method (i.e., custom tag):

```
def promoDailyAd = { attrs, body ->
    def dayoftheweek = Calendar.getInstance().get(Calendar.DAY_OF_WEEK)
    out << body() << (dayoftheweek == 7 ?
        "We have special reservation offers for Sunday!" : "No special offers")
}
```

The name of this method defines the name of the custom tag. The first declarations of the method (`attrs, body`) represent the input values of a custom tag—its attributes and body. Next, the day of the week is determined using a `Calendar` object.

After that, you can find a conditional statement based on the day of the week. If the day of the week is 7 (Saturday), the conditional statement resolves to the string "We have special reservation offers for Saturday!"; otherwise, it resolves to "No special offers".

The string is outputted through << and is first assigned through to the `body()` method, which represents the custom tag's body, then throughout, which represents the custom tag's output. In this manner, you declare the custom tag in an application's view using the following syntax:

```
<g:promoDailyAd/></h3>
```

When the view containing this custom tag is rendered, Grails executes the logic in the backing class method and supplants it with the results. This allows a view to display results based on more elaborate logic by means of a simple declaration.

Caution This type of tag is automatically available in GSP pages but not JSP pages. In order for this custom tag to function properly in JSP, it's necessary to add it to the corresponding Tag Library Definition (TLD) `grails.tld`. TLDs are located in an application's `/web-app/WEB-INF/tld/` directory.

Custom tags can also rely on input parameters passed in as tag attributes to perform a backing class's logic. The following listing illustrates another custom tag that expects an attribute named `offerdate` to determine its results:

```
def upcomingPromos = { attrs, body ->
    def dayoftheweek = attrs['offerdate']
    out << body() << (dayoftheweek == 7 ?
        "We have special reservation offers for Saturday!" : "No special offers")
}
```

Though similar to the earlier custom tag, this last listing uses the statement `attrs['offerdate']` to determine the day of the week. In this case, `attrs` represents the attributes passed as input parameters to the class method (i.e., those declared in the view). Therefore, to use this last custom tag, a declaration like the following is used:

```
<g:upcomingPromos offerdate='saturday' /></h3>
```

This type of custom tag allows more flexibility, since its logic is executed on the basis of data provided in a view. Inclusively, it's also possible to use a variable representing data passed by a controller into a view, as illustrated here:

```
<g:upcomingPromos offerdate='${promoDay}' /></h3>
```

Finally, a word about the namespace used in Grails custom tags—by default, Grails assigns custom tags to the `<g:>` namespace. To use a custom namespace, it's necessary to declare the `namespace` field at the top the custom tag library class.

```
class DailyNoticeTagLib {
    static namespace = 'court'
    def promoDailyAd = { attrs, body ->
        ...
    }
    def upcomingPromos = { attrs, body ->
        ...
    }
}
```

By using this last statement, a class's custom tags are assigned their own custom namespace named court. With the custom tag declarations made in a view requiring to be changed to the following:

```
<court:promoDailyAd/></h3>
<court:upcomingPromos offerdate='${promoDay}'/></h3>
```

Tip The Grails project has several contributed custom tags you can use in your applications.

Consult: <http://www.grails.org/Contribute+a+Tag>

18-14. Adding Security

Problem

You want to secure your application using Spring Security.

Solution

Use the Grails Spring Security plugin to have security applied to your application.

How It Works

To secure an application you need to add the Grails plugin `spring-security-core` to the application. To do this open the `grails-app\conf\BuildConfig.groovy` file and add the plugin to it.

```
grails.project.dependency.resolution = {
    ...
    repositories {
        ...
        mavenRepo 'http://repo.spring.io/milestone'
    }
    dependencies {
        ...
        plugins {
            // plugins for the compile step
            ...
            compile ":spring-security-core:2.0-RC4"
        }
    }
}
```

The plugin is added to the `plugin` section of the `grails.project.dependency.resolution` section. (Note as this isn't a final version but a release candidate you need to add the Spring milestone repository to the `repositories` section).

Now that the plugin is added running `grails compile` will download and install the plugin.

After installing the plugin, security can be set up using the `s2-quickstart` command. This command takes a package name and names of the classes to use for representing the user and its authorities.

```
grails s2-quickstart court SecUser SecRole
```

Running this will create a SecUser and SecRole domain object. It will also modify the grails-app/conf/Config.groovy file this will have a security section added.

```
// Added by the Spring Security Core plugin:
grails.plugin.springsecurity.userLookup.userDomainClassName = 'court.SecUser'
grails.plugin.springsecurity.userLookup.authorityJoinClassName = 'court.SecUserSecRole'
grails.plugin.springsecurity.authority.className = 'court.SecRole'
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    '/':
    '/index':
    '/index.gsp':
    '/assets/**':
    '**/js/**':
    '**/css/**':
    '**/images/**':
    '**/favicon.ico': ['permitAll']
]
```

When running the application with `grails run-app` the application will start and have security applied. If you try to access the application you will be prompted with a login screen asking for a username and password (see Figure 18-5).

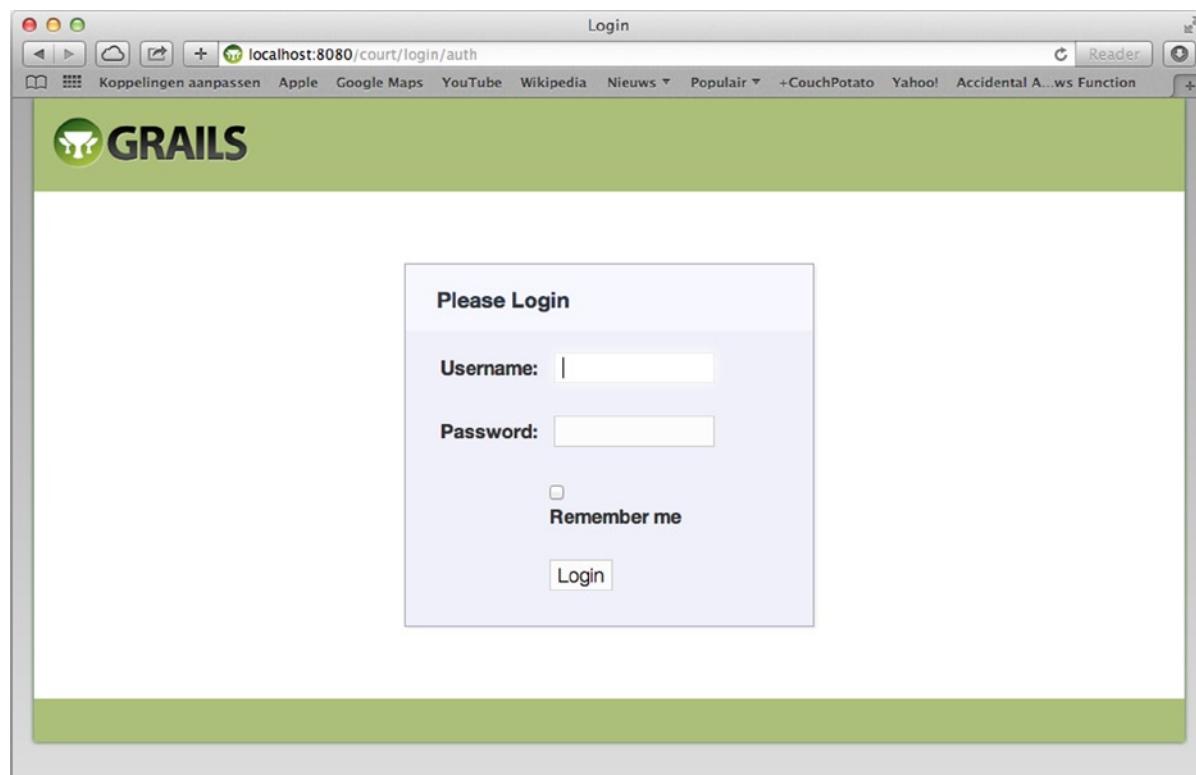


Figure 18-5. Login screen after adding security

Currently there are no users and roles in the system so logging on to the system isn't possible at the moment.

Bootstrapping Security

To use the system first there need to be users that have passwords and roles in a live application. These would come from a database, ldap directory or maybe some files on the file system. You will add some users in the bootstrap script of the application. Open the `Bootstrap.groovy` file in the `grails-app/conf` directory and add 2 users and 2 roles to the system.

```
class BootStrap {

    def init = { servletContext ->

        def adminRole = new court.SecRole(authority: 'ROLE_ADMIN').save(flush: true)
        def userRole = new court.SecRole(authority: 'ROLE_USER').save(flush: true)

        def testUser = new court.SecUser(username: 'user', password: 'password')
        testUser.save(flush: true)

        def testAdmin = new court.SecUser(username: 'admin', password: 'secret')
        testAdmin.save(flush: true)

        court.SecUserSecRole.create testUser, userRole, true
        court.SecUserSecRole.create testAdmin, adminRole, true

    }
    ...
}
```

First 2 roles, `ROLE_ADMIN` and `ROLE_USER`, are added to the system. Next 2 users are added both with a username and password. Finally the link between the user and the role is made.

Now that everything is in place restart the application and logon to the system (see Figure 18-6).

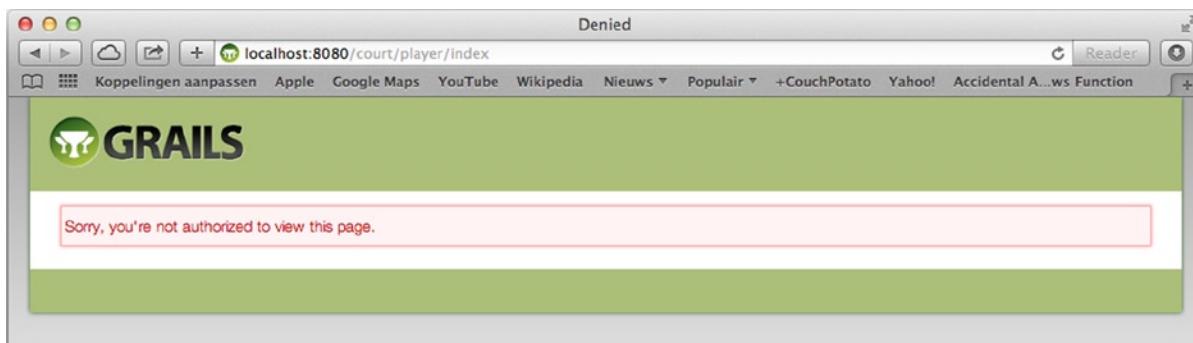


Figure 18-6. The system after logging in with a user

Although you are able to login you will be greeted with a page telling you that you aren't allowed to access the page you requested. Although there are now users in the system, our system doesn't know that those are allowed to access the page requests. For this we need to add configuration to make clear which URLs are allowed to be accessed.

Securing URLs

After creating the security configuration only some default URLs are added to it. It doesn't contain the specific URLs for your application. For this open the `Config.groovy` file in the `Grails-app/conf` directory.

```
grails.plugin.springsecurity.controllerAnnotations.staticRules = [
    '/': ['permitAll'],
    '/index': ['permitAll'],
    '/index.gsp': ['permitAll'],
    '/assets/**': ['permitAll'],
    '**/js/**': ['permitAll'],
    '**/css/**': ['permitAll'],
    '**/images/**': ['permitAll'],
    '**/favicon.ico': ['permitAll'],
    '/player/**': ['isAuthenticated()'],
    '/reservation/**': ['isAuthenticated()']
]
```

Notice the 2 new additions one for the players section of the site and one for the reservation section of the site. The expression, `/player/**`, is a so called ant-style pattern it matches everything that starts with `/player`. The `permitAll` means everyone even no users can access those parts of the website (mostly useful for static and public content). With `isAuthenticated()` only authenticated users are allowed to access the site. For more allowed expression see the recipes on Spring Security (Chapter 8).

After rebuilding and starting the application you should be able to access the player and reservations screens again.

Using Annotations for Security

Next to securing URLs it is also possible to secure methods, for this there you can use the `@Secured` annotation. Let's secure the `create` method so that only admins can create new players.

```
import grails.plugin.springsecurity.annotation.Secured

class PlayerController {
    ...

    @Secured(['ROLE_ADMIN'])
    def create() {
        respond new Player(params)
    }
}
```

Notice the addition of the `@Secured` annotation to the `create` method. The annotation takes an array of roles that are allowed access. Here `ROLE_ADMIN` is specified, as access is limited to administrators only. The `@Secured` annotation can also be placed on the class level instead of the method level and this will add security to all methods in the class.

Logging in as a normal user (using `user/password`) and trying to create a new player will result in an access denied page (the same as shown in Figure 18-6). When doing the same with an administrator (using `admin/secret`) you will be allowed to enter a new player.

Summary

In this chapter, you learned how to develop Java web applications using the Grails framework. You started by learning the structure of a Grails application and quickly followed that by working with a sample application that demonstrated the automation of several steps undertaken in a web application.

Throughout the various recipes, you also learned how Grails uses conventions in its automation process to create an application's views, controllers, models, and configuration files.

In addition, you also learned about the existence of Grails plug-ins to automate tasks for related Java APIs or frameworks in the context of Grails. You then explored how Grails separates configuration parameters and executes tasks on the basis of an application's working environment, which can be development, testing, or production.

You then learned how, from an application's domain classes, Grails generates the corresponding controller and views used to perform CRUD operations against a RDBMS. Next, you explored Grails internationalization, logging, and testing facilities.

Next, you explored Grails layouts and templates used to modularize an application's display and followed that with a look at the Grails Object Relational Mapping (GORM) facilities, as well as the creation of custom tags.

Finally you explored how to apply Spring Security to a Grails project and how to configure and use it to secure URLs and methods.

Index

A

- AbstractFactoryBean, 93, 171
- Access control decision (ACL), 381, 383–384
 - Access Decision Managers, 365
 - bean configuration, 366
 - board-security.xml, 367
 - decision manager, 365
 - for domain objects, 379–380
 - expression, 368–370
 - problem, 365
 - service, 376, 378–379
- Advice annotations
 - @After advice, 182
 - @AfterReturning advice, 183
 - @AfterThrowing advice, 183–184
 - @Around advice, 185
 - @Before advice, 180–182
- Annotations, 752
- Anonymous bean, 50
- Anonymous login, 350
- Apache James Server, 613
- Application context, 51
- Application programming interface (API), 484–486
- AspectJ compiler (AJC), 201
- AspectJ framework
 - configuration, 205–206
 - LTW, 201–205
- AspectJ pointcut language
 - ArithmeticCalculator interface, 192
 - combining with operators, 194
 - declaring, 195
 - @LoggingRequired annotation, 193
 - marker annotation, 192–193
 - Spring AOP, 191
 - type signature patterns, 193
- aspectOf(), 205–206
- Aspect orientated programming (AOP)
 - advice annotations, 178, 180–185
 - concrete class, 104
 - counter interface, 199–200
 - @DeclareParents annotation, 197–198
 - declaring advices, 103
 - declaring aspects, 101–102
 - declaring pointcuts, 103
 - definition, 178
 - description, 99
 - domain objects, 207–208
 - inheritance, 197
 - interfaces, 196–197
 - and POJOs, 178, 180
 - POJOs classes, 100–101
- Atom feed
 - AtomFeedView class, 297–298
 - buildFeedEntries method, 298
 - buildFeedMetadata method, 298
 - getAtomFeed(), 296
 - handler method, 296
 - Project Rome, 293, 295
 - RssFeedView class, 299
 - spring MVC controller, 298
 - structure, 294
- Atomicity, consistency, isolation and durability (ACID), 476
- AuthenticatedVoter, 365
- Authenticating users
 - against database, 353–356
 - encrypting passwords, 356–357
 - in-memory definitions, 352
 - Java Config, 353
 - LDAP repository, 358–361
 - LDAP Repository using Java Config, 361–362
 - problem, 351
 - solution, 351
- authoritiesByUsernameQuery methods, 356
- Auto-wire POJOs
 - advantages, 64
 - @Autowired annotation, 146–149

- Auto-wire POJOs (*cont.*)
 byName attribute, 65–66
 @Inject annotation, 152–153
 methods and constructors, 148
 primary attribute, 65
 @Resource annotation, 152
 supported by spring, 64
 @Autowired annotation, 261, 264
 auto-wire POJO fields, 146–147
 auto-wire POJO methods and constructors, 148
- B**
- B2B. *See* Business-to-business (B2B)
 Bean-managed transaction (BMT), 475
 Bean post processor
 addBeanPostProcessor(), 91
 characteristics, 90
 initialization callback method, 90
 postProcessBeforeInitialization() and
 postProcessAfterInitialization(), 91
 product bean instances, 92
 Bean scope, POJOs
 concept, 78–79
 description, 77–78
 prototype, 80
 singleton, 77, 79
 BeanShell, 123, 126
 Bean validation
 @NotNull, 266
 Reservation domain, 266
 Spring framework, 265
 submitForm method, 267
 Big Data
 DataAccessException, 549
 Hadoop, 568–573
 MongoDB (*see* MongoDB)
 Neo4j (*see* Big Data, Neo4j)
 Redis, 561–567
 SQL databases, 549
 BPM. *See* Business process management (BPM)
 Burlap service, 634–635
 Business process management (BPM), 536
 Business-to-business (B2B), 512
- C**
- Caching user, 362–364
 Checkout() method, 495
 CICS. *See* Customer Information Control System (CICS)
 Comma-separated value (CSV), 512, 518, 520, 522, 527, 536
 Communicate application events, POJOs
 define events, 110
 description, 109
- event-based communication model, 109
 listen to events, 111
 publish events, 110
 sender and receiver, 109
 Concurrency, 209–213, 215
 Constructor injection, 49
 Constructor, POJOs
 ambiguity, 56–59
 battery and disc, 143
 classes, 54
 definition, 53, 142
 java configuration class, 144–145
 product class, 143
 properties, 53
 XML configuration, 54–55
 Container-managed transaction (CMT), 475
 Content negotiation
 application/pdf., 249
 ContentNegotiatingViewResolver, 247
 reservationSummary, 248
 URL, 247
 contextConfigLocation, 340
 Controllers
 HTTP GET request, 252
 HTTP POST request, 252
 reservationForm.jsp, 252
 reservationSuccess.jsp, 253
 ResourceBundleMessageSource, 254
 Create POJOs
 AbstractFactoryBean, 93
 constructor, 53–59
 factory bean, 92
 instance factory method, 94–95
 object-creation process, 92
 Spring's factory bean, 96
 static factory method, 93–94
 Create, read, update, and delete (CRUD) controllers
 application's domain classes, 770
 grails generate-all court.Player, 770
 grails generate-all court.Reservation, 770
 HTML form, 771
 JSP pages, 769
 Player domain class, HTML, 772
 RDBMS, 771
 Reservation domain class, URLs, 772
 scaffolding code creation, 770
 SQL or ORMs, 773
 URLs, 771
 CSV. *See* Comma-separated value (CSV)
 Customer Information Control System (CICS), 511
 Customer relationship management (CRM), 700
 Custom ItemReader, 525–526
 Custom ItemWriter, 526–527
 Custom layouts and templates, Grails
 display elements (HTML/CSS/JavaScript), 787
 inherit behavior, 788–789

main.gsp contents, 787–788
 _reservationList.gsp, 789–790

Custom tags, Grails
 class method, 793
 custom namespace, 793–794
 DailyNoticeTagLib.groovy class, 792
 grails create-tag-lib <tag-lib-name> command, 792
 GSP or JSTL tag, 792
 HTML tags, 792

D

Data access
 AnnotationConfigApplicationContext, 469
 APIs, 419
 ComponentScan annotation, 469
 CurrentSessionContext, 468
 getCurrentSession(), 467
 Hibernate EntityManager, 419
 HibernateTemplate, 466
 HQL, 419
 JDBC (*see* Java Database Connectivity (JDBC))
 JPA, 419, 469–470, 472–474
 ORM (*see* Object/relational mapping (ORM))
 Repository annotation, 468
 SQL statements, 419
 Data Access Object (DAO) class, 139–140
 Data Access Object (DAO) pattern
 implementation, 422
 java.sql.SQLException, 422
 javax.sql.DataSource, 422
 Main class, 425
 RuntimeException, 423
 Database Connection Pooling Services (DBCP), 424
 Database in integration tests, 750, 752
 Data definition language (DDL), 513
 Data Tools Platform (DTP), 420
 Declare POJOs
 object properties, 106
 static fields, 104–105
 DefaultWebSecurityExpressionHandler, 370
 DelegatingFilterProxy instance, 339
 DispatcherServlet, 222
 Domain classes, Grails
 creation, 767
 declaration name/phone, 768–769
 description, 767
 reservation system, 768
 views and controllers, 769
 Domain objects, AOP
 AspectJ framework, 207
 ComplexFormatter, 207
 @Configurable annotation, 208–209
 Domain object security
 ACL, 376
 problem, 376

dotMobi factory method, 399
 Dynamic finders, 790

E

Eclipse IDE
 description, 12
 Gradle 'build' option, 25
 Gradle import, 23
 Gradle Integration, 17–18
 Gradle subproject, 24
 installation, 14
 marketplace option, 13
 Maven import, 19
 Maven integration, 17
 Maven project, 20
 pom.xml, 16
 project creation, 15
 springintro, 24
 spring project, 16
 springintro_mvn, 20
 springintro_mvn-1.0-SNAPSHOT.jar, 21
 E-mail message template, 617–618
 Enterprise application integration (EAI)
 ESB, 692–694
 JDBC, 693
 protocol, 692
 Remote Procedure Invocation, 692
 TCP/IP, 692
 Enterprise service bus (ESB)
 Axway Integrator, 693
 public static void main() method, 694
 SpringSource Portfolio, 693
 Excel and PDF views creation
 AbstractExcelView, 269
 Content-Disposition, 273
 date base, 272
 date parameter, 268
 HTML, 267
 @RequestMapping, 270–271
 resolving views, 272
 try/catch block, 269
 ExtendedWebSecurityExpressionHandler, 371
 Extensible Markup Language (XML), 229–232
 description, 275
 Marshalling View
 content negotiation, 280
 handler method, 276
 Java Architecture for XML Binding (JAXB), 277
 membertemplate, 277
 payload, 279
 Spring—Jaxb2Marshaller, 277–279
 URL extension, 280
 @PathVariable, 281, 283
 @ResponseBody, 280–281
 ResponseEntity, 283

F

Filters, WAF

- CityServiceImpl, 408
- CityServiceRequestAuditor, 409
- DelegatingFilterProxy, 410
- HttpRequestHandler interface, 407
- life cycle management, 411
- oriented programming, 406
- web.xml file, 406

Form-based login

- attribute, 347
- authentication-failure-url attribute, 348
- http://localhost:8080/board/, 347
- <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>, 346
- SPRING_SECURITY_LAST_EXCEPTION, 348
- URL /spring_security_login, 346

Form's service processing, 254–255

G

Gateways, Spring Integration

- accessor/mutators, 720
- client-facing interface, 720
- inboundHotelReservationSearchChannel, 722
- jms outbound-gateway, 719
- messaging middleware, 717
- POJO interface, 717
- service-activator, 722
- SimpleMessagingGateway, 718

getSportType() method, 260

Grails

- application
 - creation, 758–759
 - default main screen, 760–761
 - file and directory
 - structure, 759–760
 - first grails application
 - construct, 762
 - to WAR, 763
- custom layouts and templates, 787–789
- custom tags, creation, 792–793
- development, production, and testing
 - Config.groovy file, 766
 - configuration parameters, 765
 - grails test run-app, 767
 - grails war command, 765
 - permanent storage system, 765–766
- domain classes (*see* Domain classes, Grails)
- first grails application construct
 - handler method, 762
 - index.gsp, 762
 - JSP page, 762
 - WelcomeController, 761–762

getting and installing

- environment variables, 758
- Java applications, 757
- Windows or Linux workstation, 758

GORM queries (*see* Grails Object Relational Mapping)

Mapping (GORM) queries

logging, 779–780

permanent storage systems, 776–778

plug-ins

- Apache Tomcat and Hibernate
 - plug-ins, 764
- Clojure plug-in, 764
- installation and uninstallation, 764
- JARs, 763
- Java framework or Java API, 763

Spring Security, 794–797

unit and integration tests, 781–786

Grails Object Relational Mapping (GORM) queries

- Boolean logic, 791
- dynamic finders, 790–791
- facilities, 798
- HQL, 791
- Player.get(id), 790
- RDBMS operations, 790

Groovy, 122, 126

Groovy Object Relational Mapper (GROM), 776

H

Hadoop, Big Data

- ClassPathXmlApplicationContext, 572
- cluster resource management, 568
- core-site.xml, 570
- HADOOP_OPTS variable, 571
- IntSumReducer, 569
- map/reduce operations, 568
- pseudo cluster, 570
- WordCount program, 569, 573
- XML support, 571
- YARN, 573

Handler Interceptors

- afterCompletion() method, 236–237
- @Controllers, 238
- postHandle(), 236
- preHandle() method, 236–237
- Spring's web application, 236
- WebMvcConfigurerAdapter, 238

Handling security

- authentication information, 374
- JSP files, 374
- problem, 374
- view contents, 375–376

Hessian service

- HessianProxyFactoryBean instance, 634
- Spring MVC applications, 634

- web application, 632
- web container, 634
- Hibernate metadata (hbm)
 - configure() method, 450
 - Course.hbm.xml, 448
 - DAO, 449
 - hbm2ddl.auto property, 449
 - hibernate.cfg.xml, 448
 - JPA, 451–456
 - RuntimeException, 450
- Hibernate Query Language (HQL), 419, 790, 792
- Hypertext Transfer Protocol (HTTP)
 - Burlap service, 634–635
 - Hessian Service. (*see* Hessian Service)
 - Invoker service, 635

- I**
- I18N text messages
 - description, 84
 - getMessage(), 85
 - MessageSource, 86–87
 - resource bundles, 84
- Inherit POJO configurations
 - base sequence generator, 117
 - child beans, 116
 - child bean's collection, 119–120
 - parent <bean> element, 116
 - ReverseGenerator, 118
 - sequence generator instance, 116–117
- Inject Spring Beans
 - BeanShell, 126
 - Groovy, 126
 - JRuby, 125
- In-memory definitions, 352
- Instance factory method, POJOs, 94–95, 172–173
- Integration tests
 - AccountService, 740
 - AccountServiceImpl, 736
 - JUnit 4, 740, 742
 - TestContext framework, 739
 - TestNG, 740, 742–743
- IntelliJ IDE
 - application run, 35, 41
 - description, 26
 - 'Exclude build directory' 33
 - file/directory, import, 29
 - Gradle home, 37
 - Groovy application configuration, 38
 - import Maven project, 30
 - import project, 30
 - JetGradle project window, 37
 - Maven configuration, 31
 - Maven projects window, 32
 - pom.xml file, 28
- project dependencies, 28
- project structure, 39
- springintro_mvn., 36
- springintro-1.0-SNAPSHOT.jar, 40
- springintro_mvn., 30
- springintro_mvn-1.0-SNAPSHOT.jar, 34
- Spring project, 27
- InternalResourceViewResolver, 246
- Internationalization (I18N) message properties, Grails
 - browser's language preferences, 773, 775–776
 - default messages.properties file, 774
 - explicit configuration, 773–774
 - Grails *.properties file, 774
 - JSP or GSP pages, 773
 - welcome.message, 776
- internationalization (I18N) via annotations, 161–164
- IoC container
 - application context, 137
 - bean factory, 137
 - bean instance, 138
 - sequence generator application, 138

- J, K**
- Java, 341
 - @ComponentScan, 343
 - hasAnyRole or hasRole method, 344
 - MessageBoardApplicationInitializer, 342
 - @Order(1), 341
 - package com.apress.springrecipes.board.config\;, 342
 - public class MessageBoardWebConfiguration, 343
 - SpringServletContainerInitializer, 341
 - Java Architecture for XML Binding (JAXB), 277
 - Java-based configuration transaction management, 493
 - JavaBeans API
 - property editors, 112–116
 - JAVA build tools, 3–4
- Java collection attributes, POJOs
 - array, 69
 - concrete classes for collections, 75–77
 - data type, 73–74
 - description, 68
 - lists, 68–69
 - maps, 70, 72
 - properties, 68, 72–73
 - set, 70
- Java Collections framework, 68
- Java configuration class, POJOs
 - annotations with Spring's IoC container, 136
 - @Bean, 136–137
 - bean definition, 136
 - @Component annotation, 139–140
 - @Configuration, 136–137
 - IoC container, 137–141
 - @Profile annotation, 176–177

- Java configuration class, POJOs (*cont.*)
 - reference, 146
 - SequenceGenerator, 136
 - XML configuration, 137
 - XML file removal and annotation usage, 141
- Java Database Connectivity (JDBC)
 - API, 426
 - application database, 420
 - DAO, 421–423
 - DBCP, 424
 - Derby, 420
 - DriverManagerDataSource, 424
 - DTP, 420
 - InsertVehicleStatementCreator class, 428
 - javax.sql.DataSource, 424
 - JdbcTemplate class, 427
 - JndiDataSourceLookup, 425
 - ORM, 426
 - PreparedStatementCreator, 427
 - PreparedStatementSetter, 428
 - queries, 431–436
 - Spring, 441–446
 - SQL, 429–431
 - templates, 436–437, 439–441
 - update() template methods, 426, 429
 - Vehicle class, 420
- JavaMail
 - API, 614
 - ErrorNotifier interface, 613
 - James Server, POP3, 615
- Java Management Extensions Remote API, 592
- Java Message Service (JMS)
 - AMQP messages
 - Message Listeners, 688–690
 - RabbitTemplate, 686
 - spring's template support, 686–688
 - AMQP messages (*see* AMQP messages)
 - applications, 659
 - cache and pool, 685–686
 - conversion, 673–675
 - default destination, 669–672
 - definition, 659
 - JmsGatewaySupport, 672
 - MDBs, 694
 - MDP, 695–698
 - message-driven POJO (MDP) (*see* Message-driven POJO (MDP))
 - middleware, 694
 - template
 - back office bean, 668
 - createMessage() method, 665
 - front desk bean, 665–666
 - MessageCreator object, 664
 - null message, receive() method, 668–669
 - receive() method, 667
 - "topics" and "queues", 659
- transaction management, 675–677
- without Spring
 - ActiveMQ message broker, 663
 - BackOfficeImpl class, 662–663
 - FrontDeskImpl class, 661
 - point-to-point communication model, 662
 - POJO configuration files, 663
 - sendMail() method, 662
- with spring
 - Apache ActiveMQ, 661
 - front desk and back office subsystem, 660
 - JMS 1.1, 660
 - tasks, 659
 - template-based solution, 660
- Java Persistence API (JPA)
 - classpath root, 453
 - CourseDao interface, 454
 - CourseRepository, 473
 - createEntityManagerFactory(), 455, 460
 - DAO implementation, 455
 - EE environment, 454
 - EJB component, 469
 - EnableJpaRepositories annotation, 473
 - EntityManagerFactory, 472
 - hibernate.cfg.xml, 452
 - HibernateJpaVendorAdapter, 462
 - JndiLocatorDelegate object, 461
 - JpaCourseDao, 460
 - LocalEntityManagerFactoryBean, 460
 - mapping metadata, 451
 - META-INF directory, 453
 - org.hibernate.DuplicateMappingException, 453
 - packagesToScan, 462
 - PersistenceContext annotation, 469
 - persistence.xml, 452
 - programming interfaces, 452
- JavaScript Object Notation (JSON)
 - MappingJackson2JsonView, 284–285, 287
 - payload, 284–285
 - @ResponseBody, 287–288
- JavaServer Faces (JSF)
 - DelegatingVariableResolver, 416
 - DistanceApplicationInitializer, 413, 417
 - DistanceBean class, 414
 - distance.xhtml, 415
 - find() method, 415
 - IoC container, 412
 - SpringBeanFacesELResolver, 412, 416
- JavaServer Pages [JSP] pages, 757, 762, 770
- JConsole
 - JMX-enabled applications, 597
 - MBean, RMI, 601
 - MX-enabled applications, 597
 - notification event, RMI, 611
 - Spring bean operation, 598
- JDBC. *See* Java Database Connectivity (JDBC)

JMX MBeans
 AnnotationMBeanExporter, 604
 annotations, 604–605
 @EnableMBeanExport, 605
 file replication, 592
 InterfaceBasedMBeanInfoAssembler, 602
 Java Config class, 594
 @ManagedResource, 605
 management interface, 592, 599, 602
 MetadataMBeanInfoAssembler, 604
 Remote access, RMI, 600–601
 replicate() method, 594
 source and destination directories, 592
 Spring IoC container, 598
 @Value annotations, 595

JMX Notifications
 MBean events, JConsole, 607
 MBean exporter, 608
 NotificationPublisher interface, 606
 types, 607

JobLaunch
 ApplicationContext, 541
 com.apress.springrecipes.springbatch, 545
 CommandLineJobRunner, 544
 DefaultBatchConfigurer, 543
 ideal solution, 542
 JobExecution, 542
 JobParameters, 542
 OSGi, 541
 Quartz, 546
 runRegistrationsJob(java.util.Date date) method, 546
 scheduling framework, 544
 SimpleAsyncTaskExecutor, 542
 system scheduler, 541
 TaskExecutor, 543

JobParameters
 batch code, 546
 ExecutionContexts, 547
 hard-coded paths, 547
 JobInstance, 546
 Spring Expression Language, 548
 Spring Framework, 547

Join points, 186–187

JRuby, 121–122, 125

JUnit 4
 @BeforeClass or @AfterClass, 725
 setUp() and tearDown() methods, 725
 TestCase, 725

JUnit and TestNG
 Java application, 724
 solution, 724
 @Test annotation, 724
 TestCase, 724

L

Lazy initialization, 87, 89, 164, 167
 LDAP repository, 358, 360–361
 Legacy user, 355
 Load-time weaving (LTW)
 and JVM, 201
 AspectJ weaver, 204
 caching, 203
 Complex(int,int) constructor, 203
 interface, 202
 post-compile-time weaving, 201
 Spring load-time weaver, 204–205
 toString() method, 201

Locales
 accept-language, 240
 CookieLocaleResolver, 240
 cookieName and cookieMaxAge, 240
 defaultLocale property, 240
 HTTP request header, 240
 language parameter, 242
 paramName property, 241

Logger, 779

Logging, Grails
 appenders and loggers, 779–780
 Config.groovy file, 779
 Java Log4J parameters, 779
 layouts configuration, 781
 output, Grails application, 779

Logout service, 349

M

Managed beans (MBeans). *See also* JMX MBeans
 documentReplicator, 596
 JMX-enabled applications, 597
 management interface, 595–596
 ModelMBeanInfo object, 597

Mapping, Big Data
 AnnotationConfigApplicationContext, 586
 cleanUp method, 584
 Neo4jConfiguration class, 585
 Neo4jTemplate, 584
 NodeEntity annotation, 581
 Planet and Character, 581
 repository, 583

Mapping exceptions
 defaultErrorView, 250
 @ExceptionHandler, 251–252
 HandlerExceptionResolver, 249
 java.lang.Exception, 250
 reservationNotAvailable, 250
 WebConfiguration, 250

MBeanProxyFactoryBean, [611–612](#)
 Message-driven beans (MDBs), [694](#)
 Message-driven POJO (MDP)
 JMS messages conversion, [681, 683](#)
 JMS transactions, [683–684](#)
 message listeners, [678–681](#)
 synchronous reception, [677](#)
 Message Driven Pojo (MDP)
 ApplicationContext, [695](#)
 connectionFactory, [695–696](#)
 EJB, [698](#)
 inboundHelloJMSMessageChannel, [696](#)
 inboundHelloWorldJMSPingServiceActivator, [696](#)
 javax.jms.Message, [697](#)
 service-activator, [696](#)
 MessageSource, [162, 164](#)
 Model-view-controller (MVC)
 Bean validation (*see* Bean validation)
 content negotiation (*see* Content negotiation)
 controllers (*see* Controllers) Excel and PDF views
 creation (*see* Excel and PDF views creation)
 Excel files, [217](#)
 Handler Interceptors (*see* Handler Interceptors)
 Java Standard Tag Library (JSTL), [217](#)
 locales (*see* Locales)
 locale-sensitive text messages, [242–243](#)
 PDF files, [217](#)
 @RequestMapping (*see* @RequestMapping)
 Rich Internet Application (RIA), [217](#)
 template's name and location, [244](#)
 Web application development (*see* Web
 application development)
 MongoDB
 BasicDBObject, [553](#)
 connection, [550](#)
 CrudRepository, [560](#)
 DBCollection, [553](#)
 findByVehicleNo method, [561](#)
 getCollection method, [553](#)
 installation, [549](#)
 mapping information, [558–560](#)
 MongoDBVehicleRepository, [554, 560](#)
 MongoTemplate, [555, 557–558](#)
 Spring configuration, [554–555](#)
 MongoTemplate
 API, [555](#)
 DBObject, [557](#)
 MongoConverter, [557](#)
 MongoFactoryBean, [558](#)
 repository, [555](#)
 SimpleMongoDbFactory, [558](#)
 Multipurpose Internet Mail Extensions (MIME)
 JavaMail features, [612](#)
 Spring Resource object, [619](#)
 Spring's bean configuration file, [618](#)
 MVC. *See* Model-view-controller (MVC)

N

Neo4j, Big Data
 CrudRepository, [587](#)
 GraphRepository, [587](#)
 Location relationship, [578](#)
 Mac and Linux, [574](#)
 mapping, [581–586](#)
 MATCH (n) RETURN, [577](#)
 nodes, [575](#)
 Planet and Character class, [577](#)
 RelationshipType, [577](#)
 remote database, [589](#)
 repository interfaces, [588](#)
 StarwarsRepositor, [578](#)

O

Object/relational mapping (ORM)
 configLocation property, [457](#)
 Core Programming Elements, [447](#)
 CourseDao interface, [447](#)
 DAO, [464](#)
 DataAccessException, [465](#)
 entity manager factory, [456](#)
 entity/persistent class, [446](#)
 getHibernateTemplate(), [465](#)
 hbm (*see* Hibernate metadata (hbm))
 HibernateCourseDao, [456, 464](#)
 HibernateTemplate class, [463](#)
 JDBC access, [446](#)
 JPA, [447, 460–462](#)
 LocalSessionFactoryBean, [457–458](#)
 mappingLocations property, [459](#)
 packagesToScan property, [458](#)
 ResourcePatternResolver, [459](#)
 RuntimeException, [448](#)
 sessionFactory property, [464](#)
 setHibernateTemplate(), [466](#)
 transaction management, [463, 465](#)
 XML mapping files, [457](#)

P

Passwords encryption, [356–357](#)
 Permanent storage systems, Grails
 dataSource definition, [778](#)
 DataSource.groovy file, [777](#)
 GROM, [776](#)
 HSQLDB database, [776](#)
 JDBC driver, installation, [777](#)
 MySQL RDBMS, [778](#)
 properties, RDBMS, [777](#)
 RDBMS, [776](#)
 Plain old java objects (POJOs)
 AOP, [99–104](#)

auto-wire, 146–149, 151–153
 auto-wiring mode, 64–66
 communicate application events, 109–111
 constructor, 142–145
 create (see Create POJOs)
 declare (see Declare POJOs)
 default profiles, 177
 define script sources, 128–129
 environments and profiles, 97–99
 external resources, 80, 82–83
 gradle, 135
 I18N text messages, 84–87
 inherit POJO configurations, 116–120
 initialization and destruction, 88–89
 @Bean, 164–165
 @DependsOn, 167–168
 lazy initialization, 167
 @PostConstruct and @PreDestroy, 166
 inject spring POJOs into scripts, 124–126
 instance factory method, 172–173
 java configuration class (see Java configuration class, POJOs)
 lazy initialization, 89
 load profile, 177
 post processors
 characteristics, 168
 every bean instance, 169
 @Required annotation, 170–171
 selected bean instances, 169–170
 post processors to validate and modify, 90–92
 properties file data, 80–81
 references, 60–63, 67, 146
 refresh POJOs from scripts, 127
 resolve auto-wire ambiguity, 149–150
 scope, 77–80, 154–156
 SpEL, 129–132
 Spring's factory bean, 173–175
 spring's IoC container
 getBean() method, 52
 instance, 51
 resources, 107–109
 SequenceGenerator class, 48–49
 XML configuration, 49–51
 XML file, 48
 static factory method, 171
 with Java collection attributes, 68–70, 72–77
 with scripting languages, 120–123
 PlatformTransactionManagers, 494
 Pointcuts, 372–373
 POJOs references
 constructor arguments, 62–63
 declare inner/anonymous, 63
 multiple configuration files, 67
 prefix generation operation, 60
 setter methods, 61–62

Propagation transaction attribute
 behaviors, 494–495
 BOOK_STOCK Table, 496
 Cashier interface, 495
 Property editors
 bean configurations, 112
 custom, 112
 CustomDateEditor, 113
 CustomEditorConfigurer, 114–115
 DateFormat object, 113
 DateFormat.parse(), 112
 parse(), 112
 product class, 115
 ProductEditor class, 115
 product ranking class, 112

Q

Quartz, 211
 Quartz scheduler
 API, 621
 Cron expression, 622
 job interface, 620
 schedule types, 622
 Spring support, 623–624
 Queries, JDBC
 DAO interface, 435
 findAll() method, 434
 java.lang.Class type, 435
 JdbcTemplate, 431
 PreparedStatementSetter, 431
 processRow() method, 432
 queryForList() method, 434
 queryForObject(), 433
 RowCallbackHandler, 432
 RowMapper<T>, 432

R

Redirect prefix, 247
 Redis, Big Data
 ByteArrayOutputStream, 563
 DataAccessException, 565
 enableTransactionSupport property, 567
 JDBC driver, 562
 localhost, 562
 ObjectInputStream, 563
 ObjectMapper, 565
 RedisConnectionFactory, 565
 RedisSerializer, 566
 redis-server command, 561
 rpush and lpush, 563
 SerializationUtils, 564
 Strings/byte, 563
 unix binaries, 561

- Redis, Big Data (*cont.*)
 - Vehicle class, [563](#)
 - `writeValueAsString` method, [565](#)
- Reference POJOs
 - `@Import` annotation, [151](#)
 - java configuration class, [146](#)
- Remember-me support, [350](#)
- Remote MBean access
 - MBeanProxy, [611–612](#)
 - MBean Server Connection, [608–611](#)
- Remote method invocation (RMI)
 - Java-based remoting technology, [628](#)
 - Java Config classes, [630](#)
 - production application, [628](#)
 - service URL property, [631](#)
 - weather service, [629](#)
- Remote procedure call (RPC), [525, 534](#)
- Remoting technologies
 - HTTP (*see* Hypertext Transfer Protocol (HTTP))
 - multitier enterprise applications, [591](#)
 - RMI (*see* Remote method invocation (RMI))
 - Spring-WS (*see* Spring web services (Spring-WS))
- Representational state transfer (REST)
 - browser performs, [289](#)
 - `getForObject` method, [291](#)
 - HTTP protocol's request methods, [290](#)
 - Hypertext Transfer Protocol (HTTP), [275](#)
 - JSON (*see* JavaScript Object Notation (JSON))
 - machine-to-machine communication, [275](#)
 - mapped object, [292](#)
 - parameterized URL, [291–292](#)
 - RestTemplate class methods, [289, 291](#)
 - RSS/Atom feeds (*see* Atom feeds)
 - WADL, [290](#)
 - web services, [275](#)
- `@RequestMapping`
 - controller class, [234](#)
 - DispatcherServlet, [232](#)
 - memberLogic, [235](#)
 - `@PathVariable`, [235](#)
 - URL wildcards, [234–235](#)
- REQUIRED propagation behavior
 - `purchase()` method, [497](#)
 - `@Transactional` annotation, [497](#)
- REQUIRES_NEW propagation behavior
 - `DefaultTransactionDefinition`, [499](#)
 - `purchase()` method, [498](#)
- Reservation class, [258](#)
- Reservation domain, [256](#)
- Resolve auto-wire ambiguity
 - `@Primary` annotation, [149](#)
 - `@Qualifier` annotation, [149–150](#)
- Resource bundles, [84, 245–246](#)
- Retrying
 - AOP advisor, [536](#)
 - `BackOffPolicy`, [535](#)
- fault tolerant step, [533](#)
- remote service, [533](#)
- `RetryTemplate`, [534](#)
- rollback exceptions, [533](#)
- RPC layer, [534](#)
- transactional resources, [533](#)
- RMI. *See* Remote method invocation (RMI)
- RoleVoter, [365](#)
- Rollback transaction attribute, [506–507](#)
- RPC. *See* Remote procedure call (RPC)
- Runtime Metadata Model
 - `JobExecution`, [512](#)
 - `JobInstance`, [512](#)
 - `JobRepository`, [512](#)

■ S

- Scaffolding code or scaffolding steps, [757, 770](#)
- Schedule tasks, Spring's scheduling, [625–627](#)
- Scripting languages with POJOs
 - BeanShell, [123](#)
 - Groovy, [122](#)
 - implementing modules, [120](#)
 - InterestCalculator interface, [120](#)
 - JRuby, [121–122](#)
 - lang schema, [120](#)
- Securing methods
 - annotations, [373](#)
 - pointcuts, [372–373](#)
 - problem, [371](#)
 - `@Secured` annotation, [371](#)
- Security interceptor, [371–372](#)
- Service Data Object (SDO), [212](#)
- `setPlayer` method, [257](#)
- Setter injection, [49](#)
- `setupForm` handler method, [259](#)
- `setupForm` method, [256](#)
- Simple Mail Transfer Protocol (SMTP), [613](#)
- Singleton, [77, 79](#)
- Site switching
 - `dotMobi` factory method, [399](#)
 - factory methods, [398](#)
 - `SitePreferencesHandlerInterceptor`, [399](#)
 - standard factory method, [400](#)
 - `urlPath` factory method, [400](#)
- SOAP web services
 - CXF, [638–640](#)
 - file generation, XSD, [642–643](#)
 - Java's JDK JAX-WS, [636–638](#)
 - sample XML message creation, [641–642](#)
 - Spring's JaxWsPortProxyFactoryBean, [640](#)
 - Spring-WS (*see* Spring Web Services (Spring-WS))
 - WSDL, [644–645](#)
 - XSD file optimization, [644](#)
- Social networking. Spring social

SportTypeConverter, 260
 Spring batch. *See also* Batch process
 audit file, 537
 batch process, 511
 BatchStatuses, 539
 BPM, 536
 chunk-oriented process, 520
 CICS, 511
 convenience class, 529
 CSV, 527
 Customer, 530
 Custom ItemReader, 525–526
 Custom ItemWriter, 526–527
 daily sales report, 536
 end element, 539
 ExitStatus, 538
 fixed-width data, 512
 helper methods, 539
 infrastructure, 513
 input, 520–521
 ItemProcessor, 528
 Java configuration, 541
 JobExecutionDecider, 540
 JobLaunch, 541–542, 544–546
 jobParameters, 546–548
 output, 521–524
 partitioning, 538
 property editors, 112–116
 reader output, 527
 retrying, 533–536
 Runtime metadata model, 512
 runtime profile, 536
 salient bits, 528
 solution, 511
 split() and flow() method, 538
 spring-batch-integration project, 538
 TaskExecutor, 538
 Tasklet implementation, 520
 telephone numbers, 528
 transactions, 530, 532
 two parameterized types, 529
 type information, 529
 U.S.zip codes, 528
 virtual machine, 537
 XML configuration, 540

Spring batch infrastructure
 components, 514
 DDL, 513
 java based configuration, 516
 Jobrepository, 513
 main class, 517
 Mapjobregistry, 515
 MapJobRepositoryFactoryBean, 513
 modular attribution, 517
 PropertySource annotation, 517
 SimpleJobLauncher, 515

SimpleJobRepository, 517
 spring property schema, 515
 Transactionmanager, 515

Spring development tools
 eclipse IDE (*see* Eclipse IDE)
 Gradle, 43–45
 Gradle Wrapper, 45
 IntelliJ IDE (*see* IntelliJ IDE)
 Maven, 42–43
 STS (*see* Spring tool suite (STS))
 toolboxes, 1

Spring expression language (SpEL)
 annotation configurations, 132
 description, 129
 POJO's life cycle, 130
 syntax, features, 130–131
 XML configurations, 131–132

Spring Integration
 BPM engine, 715
 CRM, 700
 customerBatchChannel, 717
 Customer object, 705
 customErrorChannel, 710
 EAI, 691–694
 EDAs, 702
 error-channel attribute, 707
 ESB, 691
 exception-type-router, 709
 file-system-based processing, 702
 gateways, 717–722
 HeaderAttributeCorrelationStrategy, 712
 inbound-channel-adapter, 703
 IntegrationMessageHeaderAccessor, 699
 java.util.Collection, 711
 JMS, 694, 696–698
 JobLaunchingMessageHandler, 715
 LoggingHandler, 707
 MessageBuilder<T> class, 706
 message enrichment, 700
 MessagingException, 709
 output-channel, 705
 routers, 713–714
 SEDA, 715
 SequenceSizeCompletionStrategy, 712
 service-activator, 698, 708
 splitter component, 710
 switch-laden class, 709
 transformer component, 704
 XPathMessageSplitter, 710

Spring IoC container
 filter expressions, 141
 POJO instances/beans, 141

Spring JDBC
 Apache Derby's, 443
 DataAccessException, 441–442
 databaseProductName property, 444

- Spring JDBC (*cont.*)
 - DataIntegrityViolationException, 445
 - DuplicateKeyException, 442
 - error code and SQL state, 444
 - getDatabaseProductName(), 444
 - java.sql.SQLException, 441
 - Main class, 442
 - NestedRuntimeException, 443
 - org.springframework.jdbc.support, 444
 - setExceptionTranslator() method, 446
 - sql-error-codes.xml file, 445
 - SQLException, 443
- Spring MBeanExporter approach, 600
- Spring mobile
 - Chrome, 389
 - ContextLoaderListener, 388
 - DeviceResolverRequestFilter, 387, 390
 - DeviceUtils, 395
 - DispatcherServlet, 388, 390
 - @EnableWebMvc, 388
 - Filter implementation, 385
 - getServletConfigClasses method, 388
 - home.jsp, 386
 - InternalResourceViewResolver, 397
 - iPhone 4, 389
 - LiteDeviceDelegatingViewResolver, 397
 - mobile and tablet directory, 394–395
 - MobileConfiguration, 391
 - Site preferences, 392–394
 - User-Agent, 386
- Spring MVC Application, 333–334
- Spring MVC controllers, 753
 - DepositController, 737–739
 - deposit test method, 756
 - DispatcherServlet, 737
 - JUnit, 754
 - MockMvc, 755
 - web controllers, 737
- Spring's annotation driven tasks
 - AOP (*see* Aspect orientated programming (AOP))
 - AspectJ framework, 205, 207
 - AspectJ pointcut language, 191–195
 - aspect precedence with @Order annotation, 187–189
 - with concurrency and TaskExecutor, 209–215
- external resources
 - banner.txt, 159–161
 - properties file, 157–158
- internationalization (I18N) via annotations, 161–164
- join points, 186–187
- LTW, 201–205
- @Pointcut annotation, 189–191
- POJOs (*see* Plain old java objects (POJOs))
- Spring Security
 - access control decisions (*see* Access control decisions)
- annotations and OpenID, 331
- Authenticating Users (*see* Authenticating users)
- authorization, 331
- configuration files, 335–336
- controllers and page views, 336–339, 341
- Grails
 - annotations for security, 797
 - bootstrapping security, 796
 - Grails plugin, 794
 - login screen, 795
 - securing URLs, 797
 - SecUser and SecRole domain object, 795
- Java, 341–344
- URL access (*see* URL access)
- web applications, 344
- Spring's factory bean, POJOs, 96, 173–175
- Spring's MailSender, 616–617
- Spring social
 - dependency, 304
 - Facebook
 - configure spring social, 312–313
 - connection, 310
 - register an application, 310–312
 - integrating spring social and spring security, 321, 323–325
 - modules, 304
 - PersistentUsersConnectionRepository, 319, 321
 - service provider connection status, 313–316, 318
 - sign in, 326–327, 329–330
 - StaticUserIdSource, 305
 - Twitter
 - API, 318–319
 - configure spring social, 308–309
 - connection, 305
 - register an application, 306–308
 - username, 325
- Spring testing
 - description, 723
 - integration tests (*see* Integration tests)
 - JUnit and TestNG (*see* JUnit and TestNG)
 - spring MVC controllers, 737
 - test context, 723
 - test context manager, 723
 - test execution listener, 723
 - testing frameworks, 723
 - unit tests and integration tests (*see* Unit tests and integration tests)
 - unit tests for dependent classes, stubs and mock objects (*see* Unit tests for dependent classes, stubs and mock objects)
- Spring tool suite (STS)
 - build.gradle file, 3
 - com.apress.springrecipes.hello.Main, 7
 - Dashboard, 3
 - gradle extension installation process, 9
 - gradle import, 10

- gradle subplot, 11
- installation confirmation, 8–9
- maven import, 4
- maven project, 5
- operating system (OS), 2
- ‘Package Explorer’, 5
- pom.xml file, 3
- springintro-1.0-SNAPSHOT.jar., 11
- springintro_mvn, 5
- springintro_mvn-1.0-SNAPSHOT.jar, 6
- startup screen, 2
- Spring transaction management
 - ACCOUNT Table, 480
 - API, 484
 - auto-commit, 481
 - BMT, 475
 - BOOK_STOCK Table, 478, 480
 - CMT, 475
 - commit() and rollback() methods, 481–482
 - declarative transaction management, 475
 - DriverManagerDataSource, 479–480
 - entity relational (ER), 476
 - HibernateTransactionManager, 483
 - JdbcBookShop class, 478
 - JDBC properties, 476
 - JtaTransactionManager, 483
 - PlatformTransactionManager, 482
 - programmatic transaction management, 475
 - purchase() method, 487, 489
 - purchase() operation, 479, 481
 - setAutoCommit() method, 482
 - TransactionTemplate, 486
- Spring Web Services (Spring-WS)
 - and XML Marshalling (*see* XML marshalling technology)
 - Configuration class, 647
 - service endpoints, 647–649
 - web application, 646
 - WSDL, 649–650
 - XML messages, 650
- SQL statement, JDBC
 - batchUpdate() template method, 430
 - JdbcTemplate class, 429
 - Main cMain class, 431
 - update() methods, 429
- Staged event-driven architecture (SEDA), 715
- Standard factory method, 400
- Static factory method, POJOs, 93–94, 171
- interface, 212
- Java EE and Java SE, 212
- JCA components, 211
- Quartz, 211
- Runnable, 212–213
- ScheduledExecutorFactoryBean, 215
- SimpleAsyncTaskExecutor, 215
- SyncTaskExecutor, 215
- TaskExecutorAdapter, 215
- Templates, JDBC
 - DAO, 437
 - insert() method, 437
 - JdbcDaoSupport class, 438–439
 - jdbcTemplate property, 436
 - MapSqlParameterSource, 440
 - NamedParameterJdbcTemplate, 439
 - setDataSource() method, 439
 - SQL, 439
 - SqlParameterSource array, 441
- Template’s name and location, 244
- TestContext Framework in JUnit 4, 748–749
- TestContext Framework in TestNG, 749
- Test fixtures
 - AbstractJUnit4SpringContextTests, 744
 - @Autowired, 744
 - TestContext framework, 744
- TestNG
 - AbstractTestNGSpringContextTests, 745
 - TestNG test, 726–727
- Timeout and Read-Only Transaction Attributes, 507–508
- Transaction attribute
 - description, 499
 - increase() operation, 501
 - isolation levels, 500
 - Main class, 502
 - nonrepeatable read, 505
 - println statements, 501
 - READ_UNCOMMITTED, 502
 - READ_UNCOMMITTED isolation, 503
 - REPEATABLE_READ isolation level, 504
 - SERIALIZABLE Isolation level, 506
 - transaction advices and APIs, 506
- Transaction management
 - BookShop interface, 491
 - description, 489
 - load-time weaving, 509–510
 - transaction advice, 490
 - transactional annotation, 492–493
- Transactions
 - capability, 530
 - Java based configuration, 531
 - message queue, 531
 - rollbacks, 532
 - spring core framework, 530
 - TransactionManager and BasicDataSource, 530

T

- TaskExecutors
 - abstraction, 209
 - application programming interface (API), 209
 - explore Java SE Executors and Spring’s, 210–211

Transactions Management, integration tests
 Derby server, [747–748](#)
 SimpleJdbcTemplate, [746](#)
 @Transactional, [746](#)
 Transactions Management, integration tests, [745](#)
 TransactionTemplate, [486](#)
 Transmission control protocol/internet protocol (TCP/IP), [692](#)

U

Unit and integration tests, Grails
 application's tests, [781](#)
 executes tests, [783](#)
 Grails differences, [785–786](#)
 permanent storage system, RDBMS, [781–782](#)
 PlayerSpec.groovy class, [783](#)
 RDBMS operations, [786–787](#)
 reports, creation, [783](#)
 ReservationSpec.groovy class, [784–785](#)
 script TestApp.groovy, [782](#)
 Unit tests and integration tests
 behavior verification, [728](#)
 mock object, [728](#)
 state verification, [728](#)
 Unit tests for dependent classes, stubs and mock objects
 AccountService, [732](#)
 AccountServiceImpl, [734](#)
 findAccount() and updateAccount(), [733](#)
 InsufficientBalanceException, [732](#)
 Mockito.verify, [736](#)
 null, [736](#)
 Unit tests for isolated classes
 Account, [729](#)
 AccountNotFoundException, [729](#)
 DAO implementation, [730–732](#)
 equals(), [729](#)
 RuntimeException, [729](#)
 URL access
 domain class Message, [332–333](#)
 Spring Security, [331–332](#)
 unauthorized web page, [331](#)

V

Validator interface, [262](#)

W

Web Application Description Language (WADL), [290](#)
 Web application development
 Autowired annotation, [226](#)
 concepts and configurations, [217](#)
 configuration file, [222](#)

contextConfigLocation, [222](#)
 ContextLoaderListener in web.xml., [223](#)
 @Controller, [218](#)
 @Controller annotation, [224](#)
 court-service.xml, [223](#)
 court-service.xml file, [226](#)
 development process, [229](#)
 DispatcherServlet, [222](#)
 domain subpackage, [219–220](#)
 front controller, [217](#)
 handler method, [219](#)
 HTTP GET method, [225](#)
 HTTP POST, [227](#)
 Java EE specification, [221](#)
 java.util.Date, [225](#)
 JSP views, [227–228](#)
 MVC annotation scanning, [224](#)
 request handling, [218](#)
 @RequestMapping, [225](#)
 @RequestMapping annotation, [218, 224–225](#)
 setupForm() method, [227](#)
 Spring MVC, [221](#)
 XML, [229–232](#)
 Web application framework (WAF)
 ContextLoaderListener, [401](#)
 filters, [406, 408–410, 412](#)
 getRequiredWebApplicationContext(), [404](#)
 HTTP GET method, [403](#)
 IoC container, [401](#)
 Java map, [402](#)
 javax.servlet.ServletContainerInitializer, [405](#)
 JSE, [401, 412–417](#)
 service interface, [401](#)
 ServletContainerInitializer interface, [405](#)
 Web applications
 anonymous login, [350](#)
 form-based login, [345–348](#)
 HTTP basic authentication, [345](#)
 logout service, [349](#)
 remember-me support, [350](#)
 solution, [344](#)
 Web archive (WAR) files, [760](#)

X, Y, Z

XML. See Extensible Markup Language (XML)
 XML configuration
 create POJOs via constructor, [54–55](#)
 POJO class, [49–51](#)
 SpEL, [131–132](#)
 XML marshalling technology
 service endpoints, [654–657](#)
 web services, [657–658](#)

Spring Recipes

A Problem-Solution Approach
Third Edition



Marten Deinum
Josh Long
Gary Mak
Daniel Rubio

Apress®

Spring Recipes: A Problem-Solution Approach

Copyright © 2014 by Marten Deinum, Josh Long, Gary Mak and Daniel Rubio

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-5908-4

ISBN-13 (electronic): 978-1-4302-5909-1

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

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

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

Managing Director: Welmoed Spahr

Lead Editor: Steve Anglin

Development Editor: Matthew Moodie

Technical Reviewer: Felipe Gutierrez

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Anamika Panchoo

Copy Editor: Lori Cavanaugh

Composer: SPi Global

Indexer: SPi Global

Artist: SPi Global

Cover Designer: Anna Ishchenko

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

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

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

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781430259084. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

*For my wife Djoke, my daughter Geeske and my, at the moment, unborn child.
I love you.*

—Marten Deinum

Contents

About the Authors.....	xxxiii
About the Technical Reviewer	xxxv
Acknowledgments	xxxvii
Introduction	xxxix
■ Chapter 1: Spring Development Tools.....	1
1-1. Build a Spring application with the Spring Tool Suite.....	1
Problem	1
Solution	1
How It Works.....	2
1-2. Build a Spring application with the Eclipse IDE	12
Problem	12
Solution.....	12
How It Works.....	12
1-3. Build a Spring application with the IntelliJ IDE.....	26
Problem	26
Solution.....	26
How It Works.....	26
1-4. Build a Spring application with the Maven command line interface	42
Problem	42
Solution	42
How It Works	42

1-5. Build a Spring application with the Gradle command line interface.....	43
Problem	43
Solution	43
How It Works	43
1-6. Build a Spring application with the Gradle Wrapper	45
Problem	45
Solution.....	45
How It Works.....	45
Summary.....	46
■ Chapter 2: Spring Core Tasks	47
2-1. Manage and Configure POJOs with the Spring IoC Container	48
Problem	48
Solution.....	48
How It Works.....	48
2-2. Create POJOs by Invoking a Constructor	53
Problem	53
Solution.....	53
How It Works.....	53
2-3. Use POJO References, Auto-Wiring, and Imports to Interact with Other POJOs	59
Problem	59
Solution.....	59
How It Works.....	60
2-4. Configure POJOs with Java Collection Attributes	68
Problem	68
Solution.....	68
How It Works.....	68
2-5. Set a POJOs Scope	77
Problem	77
Solution.....	77
How It Works.....	78

2-6. Use Data from External Resources (Text Files, XML Files, Properties Files, or Image Files)	80
Problem	80
Solution.....	80
How It Works.....	80
2-7. Resolve I18N Text Messages for Different Locales in Properties Files	84
Problem	84
Solution.....	84
How It Works.....	84
2-8. Customize POJO Initialization and Destruction.....	87
Problem	87
Solution.....	87
How It Works	87
2-9. Create Post Processors to Validate and Modify POJOs	90
Problem	90
Solution.....	90
How It Works.....	91
2-10. Create POJOs with a Factory (Static Method, Instance Method, Spring's FactoryBean)	92
Problem	92
Solution.....	93
How It Works.....	93
2-11. Use Spring Environments and Profiles to Load Different Sets of POJOs	97
Problem	97
Solution.....	97
How It Works.....	97
2-12. Aspect Orientated Programming.....	99
Problem	99
Solution.....	99
How It Works.....	99

2-13. Declare POJOs from Static Fields or Object Properties	104
Problem	104
Solution.....	104
How It Works.....	104
2-14. Making POJOs Aware of Spring's IoC Container Resources.....	107
Problem	107
Solution.....	107
How It Works.....	108
2-15. Communicate Application Events Between POJOs.....	109
Problem	109
Solution.....	109
How It Works.....	110
2-16. Use Property Editors in Spring.....	112
Problem	112
Solution.....	112
How It Works.....	112
2-17. Inherit POJO Configurations.....	116
Problem	116
Solution.....	116
How It Works.....	116
2-18. Implement POJOs with Scripting Languages.....	120
Problem	120
Solution.....	120
How It Works.....	120
2-19. Inject Spring POJOs into Scripts	124
Problem	124
Solution.....	124
How It Works.....	124

2-20. Refresh POJOs from Scripts	127
Problem	127
Solution.....	127
How It Works.....	127
2-21. Define Script Sources as Inline Code and not in External Files	128
Problem	128
Solution.....	128
How It Works.....	128
2-22. Use the Spring Expression Language to Configure POJOs	129
Problem	129
Solution.....	129
How It Works.....	130
Summary.....	133
Chapter 3: Spring Annotation Driven Core Tasks.....	135
3-1. Using Java Config to configure POJOs	136
Problem	136
Solution.....	136
How It Works.....	136
3-2. Create POJOs by Invoking a Constructor	142
Problem	142
Solution.....	142
How It Works.....	143
3-3. Use POJO References and Auto-Wiring to Interact with other POJOs.....	145
Problem	145
Solution.....	145
How It Works.....	146
3-4. Auto-wire POJOs the @Resource and @Inject annotation	151
Problem	152
Solution.....	152
How It Works	152

3-5. Set a POJOs Scope with the @Scope annotation	154
Problem	154
Solution.....	154
How It Works.....	155
3-6. Use data from External Resources (Text files, XML files, properties files, or image files)	157
Problem	157
Solution.....	157
How It Works.....	157
3-7. Resolve I18N Text Messages for different locales in properties files	161
Problem	161
Solution.....	162
How It Works.....	162
3-8. Customize POJO Initialization and Destruction with annotations	164
Problem	164
Solution.....	164
How It Works.....	164
3-9. Create Post Processors to validate and modify POJOs	168
Problem	168
Solution.....	168
How It Works.....	168
3-10. Create POJOs with a factory (Static method, Instance method, Spring's FactoryBean) ..	171
Problem	171
Solution.....	171
How It Works.....	171
3-11. Use Spring Environments and Profiles to load different sets of POJOs	175
Problem	175
Solution.....	175
How It Works.....	175

3-12. Aspect Orientated programming with Annotations	178
Problem	178
Solution.....	178
How It Works.....	178
3-13. Accessing the Join Point Information	186
Problem	186
Solution.....	186
How It Works.....	186
3-14. Specifying Aspect Precedence with the @Order annotation	187
Problem	187
Solution.....	187
How It Works.....	187
3-15. Reuse Aspect Pointcut Definitions	189
Problem	189
Solution.....	189
How It Works.....	189
3-16. Writing AspectJ Pointcut Expressions.....	191
Problem	191
Solution.....	191
How It Works.....	192
3-17. AOP introductions for POJOs.....	196
Problem	196
Solution.....	196
How It Works.....	196
3-18. Introduce states to your POJOs with AOP	198
Problem	198
Solution.....	198
How It Works.....	199

3-19. Load-Time Weaving AspectJ Aspects in Spring	201
Problem	201
Solution.....	201
How It Works.....	201
3-20. Configuring AspectJ Aspects in Spring.....	205
Problem	205
Solution.....	205
How It Works.....	205
3-21. Inject POJOs into Domain Objects with AOP	207
Problem	207
Solution.....	207
How It Works.....	207
3-22. Concurrency with Spring and TaskExecutors.....	209
Problem	209
Solution.....	209
How It Works.....	209
Summary.....	215
■ Chapter 4: Spring @MVC	217
4-1. Developing a Simple Web Application with Spring MVC	217
Problem	217
Solution.....	217
How It Works.....	219
4-2. Mapping requests with @RequestMapping.....	232
Problem	232
Solution.....	233
How It Works.....	233
4-3. Intercepting Requests with Handler Interceptors	236
Problem	236
Solution.....	236
How It Works.....	236

4-4. Resolving User Locales	239
Problem	239
Solution.....	239
How It Works.....	240
4-5. Externalizing Locale-Sensitive Text Messages	242
Problem	242
Solution.....	242
How It Works.....	242
4-6. Resolving Views by Names	243
Problem	243
Solution.....	243
How It Works.....	244
4-7. Views and Content Negotiation	247
Problem	247
Solution.....	247
How It Works.....	247
4-8. Mapping Exceptions to Views	249
Problem	249
Solution.....	249
How It Works.....	249
Mappings exceptions using @ExceptionHandler.....	251
4-9. Handling Forms with Controllers	252
Problem	252
Solution.....	252
How It Works.....	252
4-10. Bean validation with Annotations (JSR-303).....	265
Problem	265
Solution.....	265
How It Works.....	266

4-11. Creating Excel and PDF Views	267
Problem	267
Solution.....	267
How It Works.....	268
Summary.....	273
■Chapter 5: Spring REST	275
5-1. Publishing XML with REST Services275
Problem	275
Solution.....	275
How It Works.....	276
5-2. Publishing JSON with REST services.....	.284
Problem	284
Solution.....	284
How It Works.....	284
5-3. Accessing a REST Service with Spring	289
Problem	289
Solution.....	289
How It Works.....	289
5-4. Publishing RSS and Atom feeds	293
Problem	293
Solution.....	293
How It Works.....	293
Summary.....	302
■Chapter 6: Spring Social.....	303
6-1. Setting Up Spring Social	303
Problem	303
Solution.....	303
How It Works.....	303

6-2. Connecting to Twitter	305
Problem	305
Solution.....	305
How It Works.....	305
6-3. Connecting to Facebook	310
Problem	310
Solution.....	310
How It Works.....	310
6-4. Showing Service Provider Connection Status.....	313
Problem	313
Solution.....	313
How It Works.....	313
6-5. Using the Twitter API.....	318
Problem	318
Solution.....	318
How It Works.....	318
6-6. Using a Persistent UsersConnectionRepository	319
Problem	319
Solution.....	319
How It Works.....	320
6-7. Integrating Spring Social and Spring Security	321
Problem	321
Solution.....	321
How It Works.....	322
Summary	330
Chapter 7: Spring Security	331
7-1. Securing URL Access	331
Problem	331
Solution.....	331
How It Works.....	332

7-2. Logging In to Web Applications	344
Problem	344
Solution.....	344
How It Works.....	345
7-3. Authenticating Users.....	351
Problem	351
Solution.....	351
How It Works.....	352
7-4. Making Access Control Decisions	365
Problem	365
Solution.....	365
How It Works.....	366
7-5. Securing Method Invocations	371
Problem	371
Solution.....	371
How It Works.....	371
7-6. Handling Security in Views	374
Problem	374
Solution.....	374
How It Works.....	374
7-7. Handling Domain Object Security	376
Problem	376
Solution.....	376
How It Works.....	376
Summary.....	384
■ Chapter 8: Spring Mobile.....	385
 Recipe 8-1. Device detection without Spring Mobile	385
Problem	385
Solution.....	385
How It Works.....	385

Recipe 8-2. Device detection with Spring Mobile	390
Problem	390
Solution.....	390
How It Works.....	390
8-3. Using Site preferences.....	392
Problem	392
Solution.....	392
How It Works.....	392
8-4. Using the Device Information to Render Views.....	394
Problem	394
Solution.....	394
How It Works.....	394
8-5. Site Switching.....	398
Problem	398
Solution.....	398
How It Works.....	398
Summary	400
■ Chapter 9: Spring with Other Web Frameworks	401
9-1. Accessing Spring in Generic Web Applications.....	401
Problem	401
Solution.....	401
How It Works.....	401
9-2. Using Spring in Your Servlets and Filters.....	406
Problem	406
Solution.....	407
How It Works.....	407
9-3. Integrating Spring with JSF	412
Problem	412
Solution.....	412
How It Works.....	412
Summary	417

Chapter 10: Data Access	419
Problems with Direct JDBC	420
Setting Up the Application Database	420
Understanding the Data Access Object Design Pattern	421
Implementing the DAO with JDBC	422
Configuring a Data Source in Spring	424
Running the DAO	425
Taking It a Step Further	426
10-1. Using a JDBC Template to Update a Database.....	426
Problem	426
Solution.....	426
How It Works.....	427
10-2. Using a JDBC Template to Query a Database.....	431
Problem	431
Solution.....	431
How It Works.....	432
10-3. Simplifying JDBC Template Creation.....	436
Problem	436
Solution.....	436
How It Works.....	437
10-4. Using Named Parameters in a JDBC Template	439
Problem	439
Solution.....	439
How It Works.....	439
10-5. Handling Exceptions in the Spring JDBC Framework	441
Problem	441
Solution.....	441
How It Works.....	442
10-6. Problems with Using ORM Frameworks Directly	446
Problem	446
Solution.....	446

How It Works.....	446
Persisting Objects Using the Hibernate API with Hibernate XML Mappings	448
Persisting Objects Using the Hibernate API with JPA Annotations	451
Persisting Objects Using JPA with Hibernate as the Engine.....	452
10-7. Configuring ORM Resource Factories in Spring.....	456
Problem	456
Solution.....	456
How It Works.....	456
10-8. Persisting Objects with Spring's ORM Templates	463
Problem	463
Solution.....	463
How It Works.....	464
10-9. Persisting Objects with Hibernate's Contextual Sessions.....	466
Problem	466
Solution.....	467
How It Works.....	467
10-10. Persisting Objects with JPA's Context Injection.....	469
Problem	469
Solution.....	469
How It Works.....	470
10-11. Simplify JPA with Spring Data JPA	472
Problem	472
Solution.....	472
How It Works.....	473
Summary.....	474
■ Chapter 11: Spring Transaction Management	475
11-1. Problems with Transaction Management.....	476
Managing Transactions with JDBC Commit and Rollback.....	481

11-2. Choosing a Transaction Manager Implementation.....	482
Problem	482
Solution.....	482
How It Works.....	483
11-3. Managing Transactions Programmatically with the Transaction Manager API	484
Problem	484
Solution.....	484
How It Works.....	484
11-4. Managing Transactions Programmatically with a Transaction Template	486
Problem	486
Solution.....	486
How It Works.....	486
11-5. Managing Transactions Declaratively with Transaction Advices.....	489
Problem	489
Solution.....	489
How It Works.....	490
11-6. Managing Transactions Declaratively with the @Transactional Annotation.....	492
Problem	492
Solution.....	492
How It Works.....	492
11-7. Setting the Propagation Transaction Attribute	494
Problem	494
Solution.....	494
How It Works.....	495
11-8. Setting the Isolation Transaction Attribute	499
Problem	499
Solution.....	499
How It Works.....	500

11-9. Setting the Rollback Transaction Attribute.....	506
Problem	506
Solution.....	506
How It Works.....	507
11-10. Setting the Timeout and Read-Only Transaction Attributes	507
Problem	507
Solution.....	508
How It Works.....	508
11-11. Managing Transactions with Load-Time Weaving.....	509
Problem	509
Solution.....	509
How It Works.....	509
Summary.....	510
Chapter 12: Spring Batch	511
Runtime Metadata Model	512
12-1. Setting Up Spring Batch's Infrastructure	513
Problem	513
Solution.....	513
How It Works.....	513
12-2. Reading and Writing.....	518
Problem	518
Solution	518
How It Works.....	518
12-3. Writing a Custom ItemWriter and ItemReader	525
Problem	525
Solution.....	525
How It Works.....	525
12-4. Processing Input Before Writing	527
Problem	527
Solution.....	527
How It Works.....	527

12-5. Better Living through Transactions	530
Problem	530
Solution.....	530
How It Works.....	530
12-6. Retrying	533
Problem	533
Solution.....	533
How It Works.....	533
12-7. Controlling Step Execution.....	536
Problem	536
Solution.....	536
How It Works.....	536
12-8. Launching a Job	541
Problem	541
Solution.....	541
How It Works.....	541
12-9. Parameterizing a Job	546
Problem	546
Solution.....	546
How It Works.....	546
Summary.....	548
■ Chapter 13: NoSQL and BigData	549
13-1. Using MongoDB	549
Problem	549
Solution.....	549
How It Works.....	549

13-2. Using Redis	561
Downloading and Starting Redis	561
13-3. Using Hadoop.....	568
Problem	568
Solution.....	568
How It Works.....	568
13-4. Using Neo4j.....	574
Problem	574
Solution.....	574
How It Works.....	574
Summary	590
Chapter 14: Spring Java Enterprise Services and Remoting Technologies	591
14-1. Register Spring POJOs as JMX MBeans	591
Problem	591
Solution.....	592
How It Works.....	592
14-2. Publish and Listen to JMX Notifications	606
Problem	606
Solution.....	606
How It Works.....	606
14-3. Access Remote JMX MBeans in Spring	608
Problem	608
Solution.....	608
How It Works.....	608
14-4. Send E-mail with Spring's E-mail Support	612
Problem	612
Solution.....	612
How It Works.....	613

14-5. Schedule tasks with Spring's Quartz Support	620
Problem	620
Solution.....	620
How It Works.....	620
14-6. Schedule tasks with Spring's Scheduling.....	625
Problem	625
Solution.....	625
How It Works.....	625
14-7. Expose and Invoke Services through RMI.....	627
Problem	627
Solution.....	628
How It Works.....	628
14-8. Expose and Invoke Services through HTTP.....	632
Problem	632
Solution.....	632
How It Works.....	632
14-9. Expose and invoke SOAP Web Services with JAX-WS	635
Problem	635
Solution.....	636
How It Works.....	636
14-10. Introduction to contract first SOAP Web Services.....	641
Problem	641
Solution.....	641
How It Works.....	641
14-11. Expose and invoke SOAP Web Services with Spring-WS.....	646
Problem	646
Solution.....	646
How It Works.....	646

14-12. Develop SOAP Web Services with Spring-WS and XML Marshalling	653
Problem	653
Solution.....	653
How It Works.....	654
Summary.....	658
■ Chapter 15: Spring Messaging	659
15-1. Send and Receive JMS Messages with Spring.....	659
Problem	659
Solution.....	660
How It Works.....	660
15-2. Convert JMS Messages	673
Problem	673
Solution.....	673
How It Works.....	673
15-3. Manage JMS Transactions.....	675
Problem	675
Approach	675
Solution.....	676
15-4. Create Message-Driven POJOs in Spring.....	677
Problem	677
Solution.....	678
How It Works.....	678
15-5. Cache and pool JMS connections.....	685
Problem	685
Solution.....	685
How It Works.....	685
15-6. Send and Receive AMQP Messages with Spring	686
Problem	686
Solution.....	686
How It Works.....	686
Summary.....	690

■ Chapter 16: Spring Integration	691
16-1. Integrating One System with Another Using EAI.....	692
Problem	692
Solution.....	692
How It Works.....	692
16-2. Integrating Two Systems Using JMS.....	694
Problem	694
Solution.....	694
How It Works.....	695
16-3. Interrogating Spring Integration Messages for Context Information	698
Problem	698
Solution.....	698
How It Works.....	698
16-4. Integrating Two Systems Using a File System	701
Problem	701
Solution.....	702
How It Works.....	702
16-5. Transforming a Message from One Type to Another	704
Problem	704
Solution.....	704
How It Works.....	704
16-6. Error Handling Using Spring Integration	707
Problem	707
Solution.....	707
How It Works.....	707
16-7. Forking Integration Control: Splitters and Aggregators.....	710
Problem	710
Solution	710
How It Works.....	710

16-8. Conditional Routing with Routers	713
Problem	713
Solution	714
How It Works	714
16-9. Staging Events Using Spring Batch	714
Problem	714
Solution.....	715
How It Works.....	715
16-10. Using Gateways	717
Problem	717
Solution.....	717
How It Works.....	717
Summary.....	722
■ Chapter 17: Spring Testing	723
17-1. Creating Tests with JUnit and TestNG	724
Problem	724
Solution.....	724
How It Works.....	724
17-2. Creating Unit Tests and Integration Tests.....	728
Problem	728
Solution.....	728
How It Works.....	729
17-3. Unit Testing Spring MVC Controllers	737
Problem	737
Solution.....	737
How It Works.....	737
17-4. Managing Application Contexts in Integration Tests	739
Problem	739
Solution.....	739
How It Works.....	740

17-5. Injecting Test Fixtures into Integration Tests.....	743
Problem	743
Solution.....	743
How It Works.....	744
17-6. Managing Transactions in Integration Tests.....	745
Problem	745
Solution.....	745
How It Works.....	746
17-7. Accessing a Database in Integration Tests	750
Problem	750
Solution.....	750
How It Works.....	750
17-8. Using Spring's Common Testing Annotations.....	752
Problem	752
Solution.....	752
How It Works.....	752
17-9. Integration Testing Spring MVC Controllers	753
Problem	753
Solution.....	753
How It Works.....	753
Summary.....	756
■ Chapter 18: Grails.....	757
18-1. Getting and Installing Grails.....	757
Problem	757
Solution.....	757
How It Works.....	757
18-2. Creating a Grails Application.....	758
Problem	758
Solution.....	758
How It Works.....	759

18-3. Grails Plug-Ins.....	763
Problem	763
Solution.....	763
How It Works.....	764
18-4. Developing, Producing, and Testing in Grails Environments	765
Problem	765
Solution.....	765
How It Works.....	765
18-5. Creating an Application's Domain Classes.....	767
Problem	767
Solution.....	767
How It Works.....	768
18-6. Generating CRUD Controllers and Views for an Application's Domain Classes.....	769
Problem	769
Solution.....	770
How It Works.....	770
18-7. Internationalization (I18n) Message Properties	773
Problem	773
Solution.....	773
How It Works.....	773
18-8. Changing Permanent Storage Systems	776
Problem	776
Solution.....	776
How It Works.....	776
18-9. Logging	779
Problem	779
Solution.....	779
How It Works.....	779

18-10. Running Unit and Integration Tests	781
Problem	781
Solution.....	781
How It Works.....	781
18-11. Using Custom Layouts and Templates	787
Problem	787
Solution.....	787
How It Works.....	787
18-12. Using GORM Queries.....	790
Problem	790
Solution.....	790
How It Works.....	790
18-13. Creating Custom Tags	792
Problem	792
Solution.....	792
How It Works.....	792
18-14. Adding Security.....	794
Problem	794
Solution.....	794
How It Works.....	794
Summary.....	798
Index.....	799

About the Authors



Marten Deinum is a Java/software consultant working for Conspect. He has developed and architected software, primarily in Java, for small and large companies. He is an enthusiastic open source user and longtime fan, user and advocate of the Spring Framework. He has held a number of positions including Software Engineer, Development Lead, Coach, and also as a Java and Spring Trainer. When not working or answering questions on StackOverflow, he can be found in the water training for open water swimming events or under the water diving or guiding other people around.



Josh Long is the Spring developer advocate for SpringSource, an editor for <http://www.InfoQ.com>, and author/co-author of many works (including *Spring Recipes: A Problem-Solution Approach, Second Edition*, published by Apress). Josh has spoken at numerous industry conferences, including Geecon, TheServerSide Java Symposium, SpringOne, OSCON, JavaZone, Devoxx, JAX, and Java2Days. When he is not hacking on Spring Integration and other open-source code (see <http://Git.SpringSource.org> and <http://GitHub.com/JoshLong>), he can be found at the local Java user group, a coffee shop, or the airport. Josh likes solutions that push the boundaries of the technologies that enable them. His interests include scalability, big data, business process management, grid processing, rich Internet applications, mobile computing, and so-called “smart systems”. He blogs at <http://blog.springsource.org> and <http://www.JoshLong.com>, and can be reached at josh@joshlong.com.



Gary Mak, founder and chief consultant of Meta-Archit Software Technology Limited, has been a technical architect and application developer on the enterprise Java platform for more than seven years. He is the author of the Apress books *Spring Recipes: A Problem-Solution Approach* and *Pro SpringSource dm Server*. In his career, Gary has developed a number of Java-based software projects, most of which are application frameworks, system infrastructures, and software tools. He enjoys designing and implementing the complex parts of software projects. Gary has a master's degree in computer science. His research interests include object-oriented technology, aspect-oriented technology, design patterns, software reuse, and domain-driven development. Gary specializes in building enterprise applications on technologies including Spring, Hibernate, JPA, JSF, Portlet, AJAX, and OSGi. He has been using the Spring Framework in his projects since Spring version 1.0. Gary has been an instructor of courses on enterprise Java, Spring, Hibernate, Web Services, and agile development. He has written a series of Spring and Hibernate tutorials as course materials, parts of which are open to the public, and they're gaining popularity in the Java community. In his spare time, he enjoys playing tennis and watching tennis competitions.



Daniel Rubio is an independent consultant with over 10 years of experience in enterprise and web-based software. More recently, Daniel is founder and technical lead at MashupSoft.com.

About the Technical Reviewer

Felipe Gutierrez is an active, expert Spring and enterprise Java applications developer. He has used Spring Framework for several years now.

Acknowledgments

The acknowledgements are probably the hardest thing to write. It is impossible to name everyone personally and I probably will forget someone. For that I want to apologize beforehand.

Although this is the second book I wrote I couldn't have done it without the team at Apress; without you this book would have never seen the light of day. A special thanks to Mark Powers and Anamika Panchoo for keeping me focused and on schedule.

I thank **Felipe Gutierrez**, without whose comments and suggestions this book would never have become what it is now.

Thanks to my family and friends for the times they missed me, and to my dive-buddies for all the dives and trips I missed over the last year.

Last but definitely not least I thank my wife, Djoke Deinum, and daughter, Geeske, for their endless support, love, and dedication, despite the long evenings, sacrificed weekends and holidays to finish the book. Without your support I probably would have abandoned the endeavor long ago.

—Marten Deinum