



Introducing Play Framework

Java Web Application
Development

—

Second Edition

—

Prem Kumar Karunakaran

Apress®

Introducing Play Framework

**Java Web Application
Development**

Second Edition

Prem Kumar Karunakaran

Apress®

Introducing Play Framework: Java Web Application Development

Prem Kumar Karunakaran
Thiruvananthapuram, Kerala, India

ISBN-13 (pbk): 978-1-4842-5644-2
<https://doi.org/10.1007/978-1-4842-5645-9>

ISBN-13 (electronic): 978-1-4842-5645-9

Copyright © 2020 by Prem Kumar Karunakaran

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

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

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

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

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Editorial Operations Manager: Mark Powers

Cover designed by eStudioCalamar

Cover image by Mel Elias on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. 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 editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484256442. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To my wife, Teena, and my daughters, Nayana and Nandana. My gratitude to my parents for their support during all these years. Thanks to all my friends for their support and encouragement.

Table of Contents

- About the Authorxi**
- About the Technical Reviewerxiii**
- Preface xv**

- Chapter 1: Getting Started with Play 2 1**
 - Getting Ready..... 1
 - Installation 1
 - Prerequisites 2
 - Installing sbt..... 2
 - Installing conscript..... 2
 - Installing Giter8 3
 - Setting Up Play 4
 - Using Play Example Projects 4
 - Using sbt..... 6
 - Creating Your First Project 8
 - app..... 9
 - conf..... 10
 - build.sbt..... 11
 - project 11
 - public..... 11
 - lib..... 11
 - test 12

TABLE OF CONTENTS

Configuring Play to Work with Your Preferred IDE.....	12
Setting Up in Eclipse.....	12
Setting Up in IntelliJ	15
Hello World Application	17
Configuration	19
Controller and View	22
Testing Play Applications	27
Testing Views.....	28
Testing Controllers.....	29
Chapter 2: Build System	33
Scala Build Tool/Simple Build Tool	33
Core Principles.....	33
Benefits of sbt.....	34
Project Structure.....	35
Using sbt	36
Setting Definition.....	39
Resolvers.....	42
Complete build.sbt.....	42
Complete plugins.sbt.....	43
Quick Recap of SBT Commands.....	43
Chapter 3: Play Controllers and HTTP Routing	45
MVC Programming Model	45
Model.....	46
View	48
Controller.....	48

HTTP Routing	48
Static Definition	50
Dynamic Parts in a URL	50
Passing Fixed Values	52
Optional Parameters	53
Application Configuration Using application.conf	53
Controllers.....	54
Finishing the Bookshop Controller	56
saveComment Method.....	58
Testing the saveComment Action	59
Models	60
Scoped Objects	61
Session Scope	62
Flash Scope	64
Chapter 4: Play Views and Templating with Scala	65
Composite Views.....	66
Designing a General Template	68
Code Snippets Templating Basics	69
Comments	70
Template Parameters	70
Import Statement.....	71
Iterating a List	72
Iterating a Map	72
If Blocks.....	73
Escaping Dynamic Contents.....	74

TABLE OF CONTENTS

- Chapter 5: Concurrency and Asynchronous Programming.....75**
 - What Is Concurrency? 76
 - Executor 76
 - Example 1: Using Runnable 78
 - Example 2: Using Callable 80
 - Asynchronous Programming with Play 82
 - Writing an Asynchronous App 83
 - Configuring Asynchronous Scheduled Jobs..... 84
 - Akka Basics 84

- Chapter 6: Web Services, JSON, and XML.....87**
 - Consuming Web Services..... 88
 - Processing Large Responses 91
 - Handling JSON 94
 - Consuming JSON Request..... 94
 - Producing a JSON Response 97
 - Handling XML..... 98
 - Example 1: Simple XML Parsing 99
 - Example 2: XML Parsing Using JAXB..... 100

- Chapter 7: Accessing Databases105**
 - Configuring Database Support 105
 - Working with an ORM 109
 - ORM Concepts 109
 - Key Terms 111
 - Relationship Direction 111
 - Configuring JPA..... 113

Using Ebean in Play	118
Ebean Query	120
Common Select Query Constructs in Ebean	125
Using RawSql	128
Relationships in Ebean	130
Chapter 8: Complete Example.....	133
Chapter 9: Using Play Modules.....	157
Creating a Module	158
Third-Party Modules	161
Chapter 10: Application Settings and Error Handling.....	163
Filters	164
Action Composition	168
Error Handlers.....	170
Client Errors.....	170
Server Errors	170
How Global Settings Were Done Before Play 2.6.x.....	172
Chapter 11: Working with Cache	175
Configuring Caffeine	175
Adding Caffeine to a Project.....	176
Configuring EhCache.....	177
Using the Cache API.....	177
Chapter 12: Production Deployment.....	183
Configuring Apache httpd for Play	184
Load Balancing Using mod_proxy_balancer.....	184
Configuring Play with Nginx.....	185
Index.....	187

About the Author

Prem Kumar Karunakaran is an enterprise architect with about 20 years of industry experience. He holds a M.Tech in Software Systems from BITS Pilani and a bachelor's degree in electronics engineering from Cochin University of Science and Technology. He is also an Oracle Certified Java Enterprise Edition Master. He was involved in the architecture and design of many cutting-edge products used by clients around the globe. He has worked with organizations such as Infosys and IBS as an architect and has worked in many projects spanning airlines, logistics, travel, and retail domain. He is passionate about Java, Machine Learning, BigData processing and Cloud and loves to learn new technologies; he contributes his time to open source initiatives as well.

About the Technical Reviewer

Satheesh Madhavan has nearly 20 years of experience in the software industry and currently serves as a digital consultant in one of India's largest IT firms. He has experience working in Java and JEE technologies in providing enterprise solutions under various capacities. His qualifications include an M.S in software systems from BITS Pilani and a B.Tech in chemical engineering from the University of Kerala. His hobbies include reading fiction and non-fiction, Philately, technical blogs and podcasts, and astrophysics and pure sciences.

Preface

Software developers need to have a number of traits in order to practice their jobs well. A developer's primary job is to create software that solves business problems. Customers are operating in a dynamic business environment and they need to change their business solutions fast enough to retain and attract new customers. Hence, rapid application development is a critical aspect of today's software development methodology. You need better frameworks and tools that can help in developing quality software faster.

Play Framework is the new and impressive framework in web application development. By breaking the existing standards, it tries not to abstract away from HTTP, as with most web frameworks, but to tightly integrate with it.

Play Framework is a revolution in rapid application programming for Java web development. It has changed the rules of the game for Java web development. Java web development was a tedious, time consuming activity and there were many frameworks offering the benefits of MVC architecture. But there were hardly any frameworks capable of providing true rapid application development capabilities. Play has changed all that.

Play is a clean alternative to the existing Java Enterprise stacks for web development. Play focuses on developer productivity, scalability, adherence to modern web standards (REST, JSON, Web Sockets, and Comet, to name a few) and efficiency.

Using Play Framework, you just need to code your changes and hit the browser refresh button to test your changes. Play will automatically compile and deploy your changes. If there are any errors in your code, Play shows them nicely in the browser. You are no longer required to scan long

PREFACE

exception traces to find out the cause of the error. Above all, Play comes packaged with its own application server, which is lightweight, fast, and easy to use. In a single sentence, Play is the single framework you will need to start your Java web development. Java web development is a lot easier, more fun, and more powerful thanks to Play Framework.

This book is all you need to learn and use the new version of Play Framework, Play 2. You will be taken through all layers of Play Framework and you'll get in-depth knowledge via as many examples and applications as possible.

Further References

Play Framework official docs: www.playframework.com/

Google groups for Play: <https://groups.google.com/forum/#!forum/play-framework>

Questions related to Play: <http://stackoverflow.com/tags/playframework>

Ebeans: www.avaje.org/

Twirl: <https://github.com/playframework/twirl>

CHAPTER 1

Getting Started with Play 2

After reading the introductory section on Play 2, you should now have an idea of the capabilities of Play Framework and how it can accelerate Java web development. This chapter is about

- Installing Play Framework
- Setting up Play Framework on your machine
- Configuring Play Framework
- Creating the first sample project
- Setting up the IDE

Getting Ready

All you need is a browser and Internet connectivity. You can install Play on a wide variety of operating systems, including Microsoft Windows, Linux, and Mac. The operating system should have Java installed.

Installation

Play just needs the Play jars available at runtime to work, hence you can include the Play jars in any application using Maven or any such build tool.

But the recommended way to use Play is to install it using either sbt or Gradle because Play provides a better development experience when using sbt or Gradle.

Prerequisites

Play 2.x requires JDK 1.8 or later to be installed on the machine. Please note that JRE is not enough; you need JDK Version 8 or higher. You should ensure that the PATH variable points to the JDK bin directory and that javac and Java are accessible from everywhere.

You can check this by typing `javac -version` in the command prompt or shell and verify that the version is 1.8 or above.

You can get Java SE from the Oracle website at www.oracle.com/technetwork/java/javase/downloads/index.html.

Play also works with Open JDK version 1.8 and above. But you should make sure that any dependency you use in the project is compatible with the Open JDK.

Installing sbt

sbt (Scala build tool) is available for all OS versions at the scala-sbt website at www.scala-sbt.org/download.html. Please follow the instructions in the [installation guide](http://www.scala-sbt.org/1.x/docs/Setup.html) at www.scala-sbt.org/1.x/docs/Setup.html to install sbt.

Installing conscript

To install Play using sbt, you need to install conscript. Please follow the instructions at www.foundweekends.org/conscript/setup.html to install conscript. The instructions are available for Mac, Linux, and Windows.

The cross-platform installation using a jar is simple. Download the conscript jar from the foundweekends Maven release repo (https://dl.bintray.com/foundweekends/maven-releases/org/foundweekends/conscript/conscript_2.11/0.5.2/conscript_2.11-0.5.2-proguard.jar) and run the following from the command prompt:

```
java -jar /conscript_2.11-0.5.2-proguard.jar
```

This will start a splash screen and will install conscript. Ignore any error related to not finding 'cs.' Typically, conscript will get installed in the C:\Users\username\.conscript folder in Windows, where *username* is the home folder of the logged-in user in Windows. This location will be shown in the splash screen.

Now let's set up the environment variables for conscript:

```
export CONSCRIPT_HOME=" C:\Users\username\.conscript"
export CONSCRIPT_OPTS="-XX:MaxPermSize=512M -Dfile.encoding=UTF-8"
export PATH=%PATH%;%CONSCRIPT_HOME%\bin
```

CONSCRIPT_HOME is where conscript will download various files. For example, in Windows, it will be C:\Users\username\.

PATH is your OS's path variable. This will make the command cs available from everywhere.

Installing Giter8

After installing conscript, Giter8 can be installed using conscript itself. Giter8 is a command-line tool to generate files and directories from templates published on GitHub or any other git repository.

Go to a command prompt and type

```
cs foundweekends/giter8
```

This will download and install Giter8. More details of the installation is available from the foundweekends website for Giter8 (www.foundweekends.org/giter8/setup.html).

Setting Up Play

There are multiple ways to install Play. Let's look at the two most common ways:

- **Using Play example projects:** Use any of the example projects provided by Lightbend Tech Hub (<https://developer.lightbend.com/start/?group=play>).
- **Using sbt:** This approach depends on Giter8 templates for sbt. This method creates a fresh Play project without any example code. This is the preferred mode for a clean, lean project structure.

Using Play Example Projects

Play provides many example projects for making the installation easier. If you follow this path, installation of sbt is not required because the template provides sbt launchers for the Unix and Windows environments.

Let's try the first approach: installing the Play example project in Scala.

1. Go to <https://developer.lightbend.com/start/?group=play>.
2. Choose the project PLAY SCALA HELLO WORLD TUTORIAL.
3. Click the "Create a project for me" button (this will download the example project to your system).
4. Unzip the archive (play-samples-play-scala-hello-world-tutorial.zip).
5. Go to the folder named play-samples-play-scala-hello-world-tutorial and double-click the sbt.bat file if on Windows. If you are on a Mac or Unix-based system, run the sbt file.

It will take a while to download the sbt and all dependencies.

Once the sbt build is successful, you will see the message

```
sbt server started at local:sbt-server-
f50b441db70fa47676ba
```

Note that the last part of the message (f50b441db70fa47676ba) might be different for you but that is okay. You are now good to go and you will see the prompt as

```
[play-scala-seed] $
```

6. Type the run command to start the application:

```
[play-scala-seed] $ run
```

As soon as this step completes, the prompt will display (Server started, use Enter to stop and go back to the console...)

7. Go to <http://localhost:9000>. If all is good, you should see the Play documentation displayed in the browser.

Now let's see how to install a Play example project using Java.

1. Go to <https://developer.lightbend.com/start/?group=play>.
2. Choose the project PLAY JAVA HELLO WORLD TUTORIAL.
3. Click the "Create a project for me" button (this will download the example project to your system).

4. Unzip the archive (`play-samples-play-java-hello-world-tutorial.zip`).
 5. Go to the folder named `play-samples-play-java-hello-world-tutorial` and double-click the `sbt.bat` file if on Windows. If you are on a Mac or Unix-based system, run the `sbt` file.
 6. This will download all the dependencies and perform the build. Please note that it might take a while to download all the dependencies.
 7. Once the build is over, the command prompt will show
8. Type the run command to start the play application:

```
[play-java-hello-world-tutorial] $
```

```
[play-java-hello-world-tutorial] $ run
```

Using sbt

Now you'll learn how to install Play using `sbt` to get the basic Play application structure and dependent jars but not the example projects.

To use `sbt` to create a Play project, the following needs to be completed:

1. Install SBT.
2. Install `conscript`.
3. Install `Giter8`.

You can find instructions for installing the above software in the Installation section of this chapter.

Once the above software is installed, you can proceed with creating a Play project using sbt. Play supports Java and Scala as programming languages, so I will show examples for both using sbt.

Installing a Java Project Using sbt

Take a command prompt, change to the directory where you want the project to be created, and type

```
sbt new playframework/play-java-seed.g8
```

This will install a Java starter project structure.

The wizard will ask for the project name and organization. (By convention, this is a reverse domain name that you own, typically one specific to your project. It is used as a namespace for projects.) Once these names are supplied, the project will get created.

Installing a Scala Project Using sbt

Here is the command:

```
sbt new playframework/play-scala-seed.g8
```

Provide the project name and organization, and the project will get created.

You have now learned multiple ways to install a Play project for both Scala and Java:

- Using Play examples
- Using sbt

In this book, I use Java as the programming language for Play, so when you create your first project (a bookshop) in the next section, you will be using sbt and the Java seed.

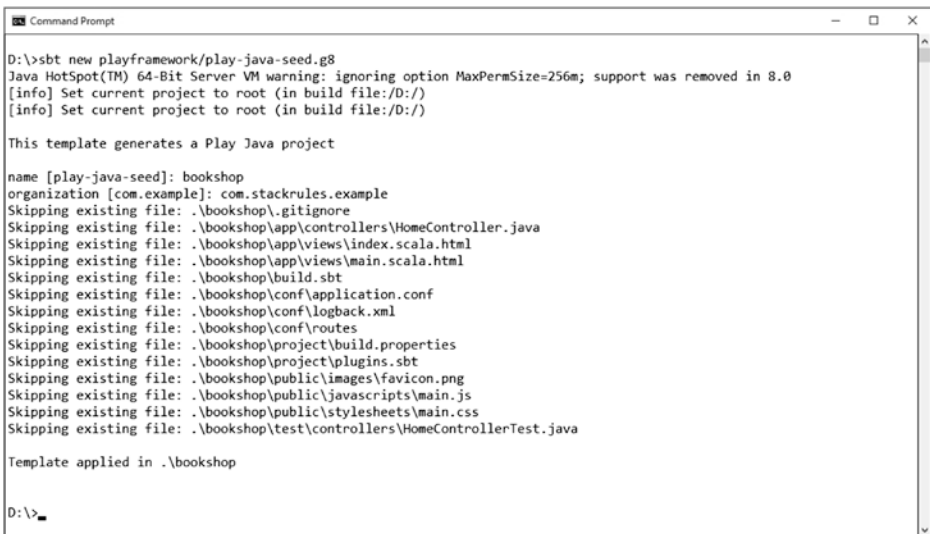
Creating Your First Project

Let's create a new project called `bookshop` to explore Play in detail. Name your project **bookshop** and, for the ease of explanation, let's make it an online retail application selling books, a domain familiar to everybody.

From a command prompt or shell, type

```
sbt new playframework/play-java-seed.g8
```

See Figure 1-1.



```

Command Prompt
D:\>sbt new playframework/play-java-seed.g8
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
[info] Set current project to root (in build file:/D:/)
[info] Set current project to root (in build file:/D:/)

This template generates a Play Java project

name [play-java-seed]: bookshop
organization [com.example]: com.stackrules.example
Skipping existing file: .\bookshop\.gitignore
Skipping existing file: .\bookshop\app\controllers\HomeController.java
Skipping existing file: .\bookshop\app\views\index.scala.html
Skipping existing file: .\bookshop\app\views\main.scala.html
Skipping existing file: .\bookshop\build.sbt
Skipping existing file: .\bookshop\conf\application.conf
Skipping existing file: .\bookshop\conf\logback.xml
Skipping existing file: .\bookshop\conf\routes
Skipping existing file: .\bookshop\project\build.properties
Skipping existing file: .\bookshop\project\plugins.sbt
Skipping existing file: .\bookshop\public\images\favicon.png
Skipping existing file: .\bookshop\public\javascripts\main.js
Skipping existing file: .\bookshop\public\stylesheets\main.css
Skipping existing file: .\bookshop\test\controllers\HomeControllerTest.java

Template applied in .\bookshop

D:\>_

```

Figure 1-1. Creating a project

Provide the following for the project name and organization:

```
name [play-java-seed]: bookshop
```

```
organization [com.example]: com.stackrules.example
```

For the organization name, you can give another domain name if you want to do so.

Go to the directory where you created project. For instance, in my machine, the project directory is `E:\workarea\bookshop`.

Let's look at the folder structure and its relevance in the overall project organization. See Figure 1-2.

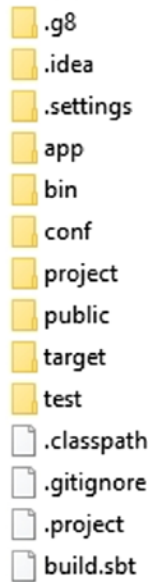


Figure 1-2. Project structure

app

The `app` folder contains all your server-side source files. This includes all your Java code, Scala code, dynamic Scala HTML templates, database access-related code, etc. By default, Play creates two folders, `controllers` and `views`, inside the `app` folder. The names are self-explanatory. The `controllers` folder holds your controller classes and `views` holds the dynamic screens (the HTML and Scala code snippets).

You are free to create subdirectories inside the `app` folder for better organization of your files. For example, you can create a `models` folder to hold all your ORM mapped POJOs, a folder named `helper` to hold your helper classes, etc.

Most of the project will have the following structure under the app folder:

app

└ assets: Compiled asset sources

└ stylesheets: For CSS source code (less CSS sources)

└ javascripts: Typically this is the folder where coffeescript sources are placed.

└ controllers: Application controllers

└ models: Application business layer

└ views: Templates

conf

The conf folder holds the configurations used by the Play application. It contains all the HTTP mappings, orm configurations, environmental variables, logging, etc.

Basically, the conf directory contains configuration and internationalization files, whereas the app folder has a subdirectory for its model definitions.

The most important files in this directory are

- `application.conf`: The main configuration file for the application, it contains standard configuration parameters.
- `routes`: Maps HTTP URL paths to methods in the controller. Handles all HTTP routing configurations.
- `logback`: Play uses logback for all logging configuration and this is the file you need to change to configure logback.

build.sbt

Play's build configuration is defined in two places: the `build.sbt` file in the project root and two files found inside the `/project` folder. The `build.sbt` files contain the build configuration.

project

The `project` folder contains project build configurations:

- `plugins.sbt`: Defines the sbt plugins used by this project
- `build.properties`: Contains the sbt versions to use to build your app and related sbt build information.

I will discuss more about sbt going forward. A basic understanding of sbt is good when you work with Play.

public

The `public` folder hosts all static files like Javascripts, images, and CSS style sheets that are directly served by the web server. The `public` folder has three subfolders, `images`, `javascripts`, and `stylesheets`, for storing these assets:

```
public
├─ stylesheets: CSS files (.css extension)
├─ javascripts: JavaScript files (.js files)
└─ images: Images
```

lib

The `lib` folder is not created by default. But you can create this folder and put any jar into it. All jars in this folder will be added to the application class path. It's ideal for including third-party dependencies that need to be managed out of the Play build system.

test

The `test` folder is a holder for storing all unit and functional test cases.

Configuring Play to Work with Your Preferred IDE

Since you have finished setting up the project structure and have generated a project template, let's now integrate the project with an IDE to make the development process easier and more rapid.

You don't need a sophisticated IDE to work with Play because Play compiles and refreshes the modifications you make to your source files automatically. This gives you the flexibility to use a simple editor like Notepad or a vi editor to work with Play. But this is not a practical case for real-world projects. You need an IDE that provides better navigation, autocompletion, debugging, refactoring, etc. By default, Play supports most of the popular IDEs like Eclipse, IntelliJ, NetBeans, and ENSIME.

For this book, I will use Eclipse as the IDE. You can use any IDE of your choice to try out the samples.

Setting Up in Eclipse

As an example, let's configure Play for Eclipse. To use Play with Eclipse, you need to first integrate sbteclipse to your project. To do this, open `plugins.sbt` (`project/plugins.sbt`) and add the following:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" %
"5.2.2")
```

You want to compile the project before you run the eclipse command to generate the eclipse import settings for the bookshop project. The manual way is to run `sbt compile` first and then do the generate eclipse

project part. But you can do it in a better and automated way; you can instruct it to run compilation first whenever you generate eclipse project settings. For this, open the `build.sbt` file and add the following:

```
EclipseKeys.preTasks := Seq(compile in Compile, compile in Test)
EclipseKeys.projectFlavor := EclipseProjectFlavor.Java
EclipseKeys.createSrc := EclipseCreateSrc.ValueSet(EclipseCreateSrc.
    ManagedClasses, EclipseCreateSrc.
    ManagedResources)
```

Please note the above is only for Java projects. If you have Scala sources, then you should use Scala IDE instead of the regular Eclipse IDE.

Save the `build.sbt` file and go to the sbt prompt by taking a command prompt, moving to the project root directory (E:\workarea\bookshop) and typing `sbt`.

This initializes the sbt prompt. This can take few minutes to complete because sbt will download the plugins and all related dependencies. Once the sbt prompt gets initialized, you should see the prompt as

```
[bookshop] $
```

Type `eclipse with-source=false` and press Enter. If you need all the source jars of the dependencies, you can issue `eclipse with-source=true` instead.

After the successful execution of the above command, start eclipse and import the project. See Figure 1-3.

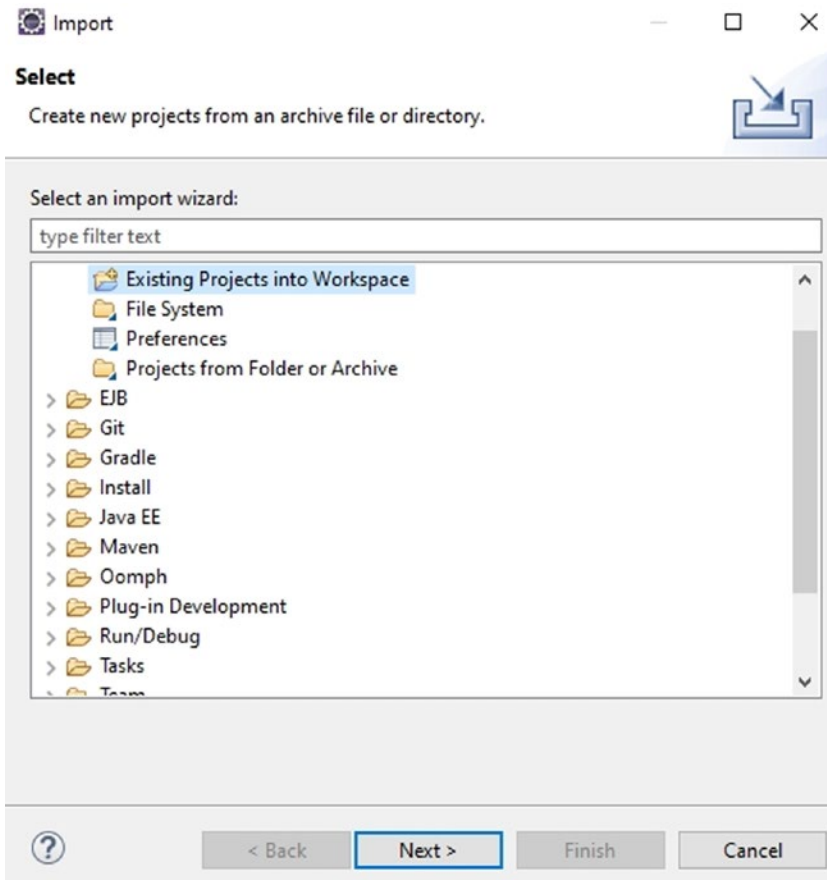


Figure 1-3. *Import Wizard*

1. Open Eclipse.
2. Click File ► Import ► General ► Existing Projects into Workspace.
3. Browse and choose the root project folder (bookshop).
4. Click Finish.

Setting Up in IntelliJ

Importing a Play project to IntelliJ is pretty straightforward. The only precondition is that the Scala plugin for IntelliJ should be installed, even when using Java as the language. This is because the Scala plugin for IntelliJ is required to resolve the sbt dependencies. So go ahead and install the Scala plugin for IntelliJ if you have not done so. Open IntelliJ and go to File ► Settings ► Plugins and search for Scala in the Marketplace tab. From the plugins listed, select the Scala plugin from JetBrains and click Install. See Figure 1-4.

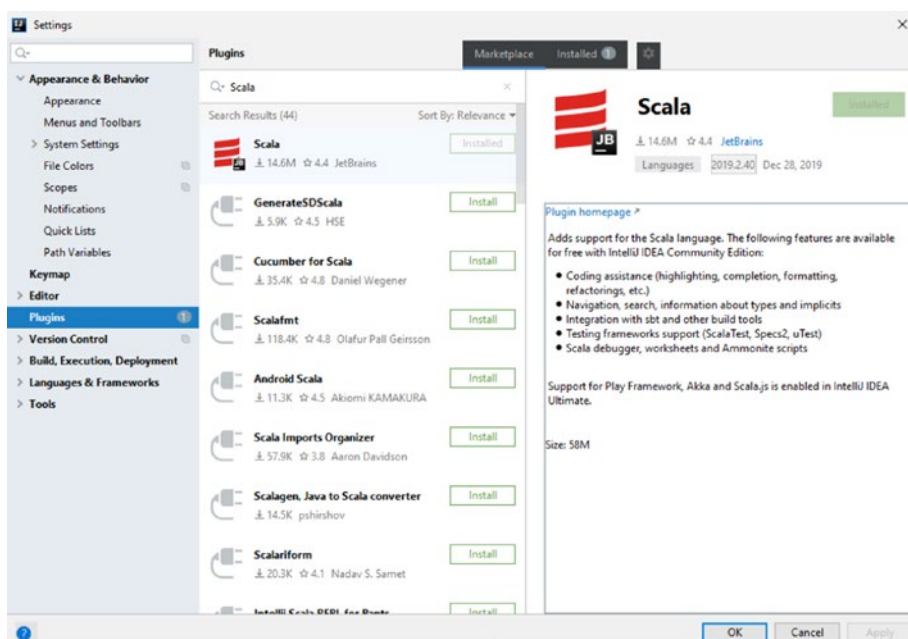


Figure 1-4. Importing the Scala plugin

After installing the Scala plugin, import the bookshop Play project to IntelliJ. See Figure 1-5.

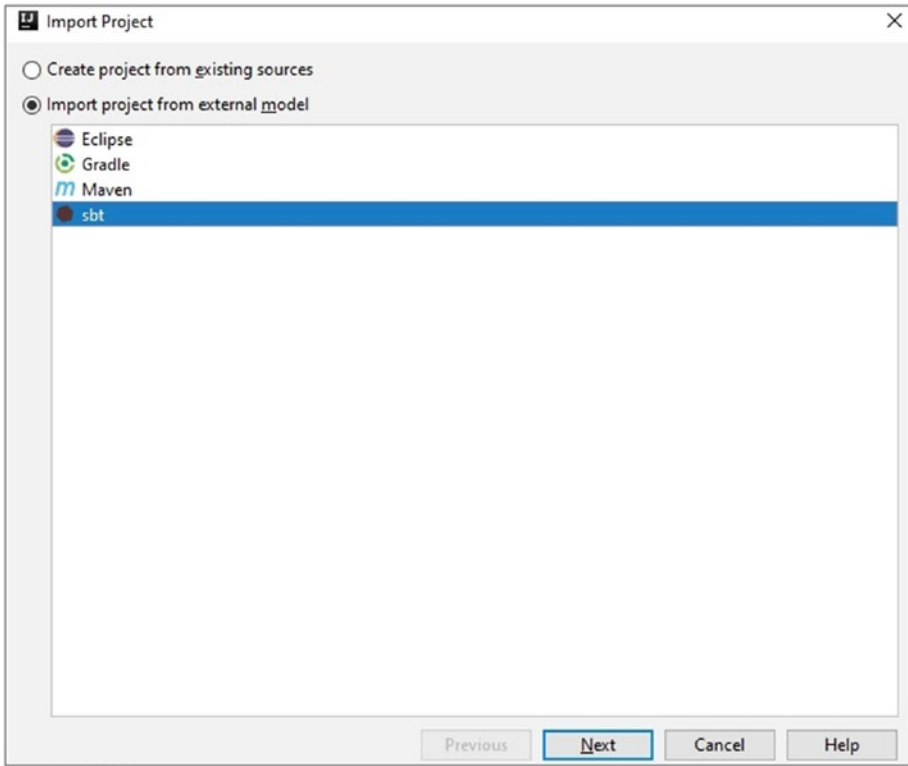


Figure 1-5. *Importing a project*

1. Open File ► New ► Project from Existing Sources.
2. Browse to the bookshop project root folder (E:\workarea\bookshop).
3. Select Import project from the external model.
4. Choose sbt.
5. Click Finish.

Wait for the build to sync and you will see that the project is imported to IntelliJ.

Hello World Application

You have your project ready and you have imported the IDE of your choice. Let's add some files to it and create the Hello World program. From there you will add more features to it and expand it.

Let's start the Play server and try the default application.

1. Open a command prompt.
2. Go to the project root folder (E:\workarea\bookshop).
3. Go to the Play console by typing `sbt` in the root folder.
4. This will open up the Play console prompt (`[bookshop] $`).
5. Type `run` to start the Play server. See Figure 1-6.

(If the server started, use `Enter` to stop, and go back to the console.)

You can even combine the commands together to start Play by using

`sbt run`

CHAPTER 1 GETTING STARTED WITH PLAY 2



```
C:\WINDOWS\system32\cmd.exe - sbt run
E:\workarea\bookshop> sbt run
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0
[info] Loading settings for project bookshop-build from plugins.sbt ...
[info] Loading project definition from D:\Apress\bookshop\project
[info] Loading settings for project root from build.sbt ...
[info] Set current project to bookshop (in build file:/D:/Apress/bookshop/)

--- (Running the application, auto-reloading is enabled) ---

[info] p.c.s.AkkaHttpServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000
(Server started, use Enter to stop and go back to the console...)

[info] Compiling 10 Scala sources and 18 Java sources to D:\Apress\bookshop\target\scala-2.13\classes ...
[info] Non-compiled module 'compiler-bridge_2.13' for Scala 2.13.0. Compiling...
[info]   Compilation completed in 8.809s.
[info] D:\Apress\bookshop\app\controllers\Application.java: Some input files use or override a deprecated API.
[info] D:\Apress\bookshop\app\controllers\Application.java: Recompile with -Xlint:deprecation for details.
[info] Done compiling.
[info] p.a.d.DefaultDBApi - Database [default] initialized at jdbc:h2:mem:play;DB_CLOSE_DELAY=-1
[info] application - Creating Pool for datasource 'default'
[info] p.a.d.HikariCPConnectionPool - datasource [default] bound to JNDI as DefaultDS
[info] p.a.h.EnabledFilters - Enabled Filters (see <https://www.playframework.com/documentation/latest/Filters>):

  play.filters.csrf.CSRFFilter
  play.filters.headers.SecurityHeadersFilter
  play.filters.hosts.AllowedHostsFilter

[info] play.api.Play - Application started (Dev) (no global state)
```

Figure 1-6. Application console

Open the browser and access `http://localhost:9000/` to access the home page. See Figure 1-7.

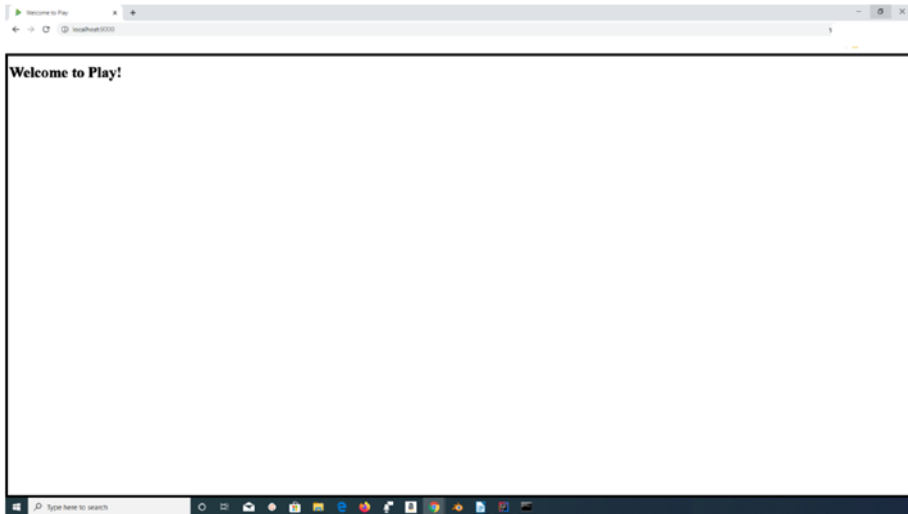


Figure 1-7. Welcome page

Configuration

Go to the `conf` folder and open the `routes` file. This is where all the URL mappings of the project need to be defined. You will examine URL mappings and routes configuration in detail in the next chapter. When you open up the `routes` file, you should see an entry like this:

```
GET      /                               controllers.HomeController.index
```

This means the root of the application points to the `index` method defined in the `HomeController` class.

If you type `localhost:portnumber` in the browser, the request will be routed to the `index` method defined in the `app/controllers/HomeController.java` file.

This is how Play routes the URL paths or URL patterns to the specific methods of the controller classes.

Open the `HomeController.java` file and you can see the method

```
public Result index() {
    return ok(views.html.index.render());
}
```

This method returns a Twirl template file and this template file generates the HTML output. Open the `index.scala.html` file found inside the `views` folder. Let's examine the contents of this file to understand what is happening in it.

```
@()
@main("Welcome to Play") {
    <h1>Welcome to Play!</h1>
}
```

Before we take a look at each element, it is important to understand what Twirl is and how it can be used.

WHAT IS TWIRL?

Twirl is the template engine developed for Play Framework. But it can also be used outside the Play environment. By default, Twirl is included as part of Play but if there is a need to use Twirl outside Play, then the sbt plugin for Scala can be installed. For example adding the below entry in the `plugins.sbt` file will make Twirl available to any sbt-based project:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-twirl" % "LATEST_VERSION")
```

Template files must be named `{name}.scala.{ext}` where `ext` can be `html`, `js`, `xml`, or `txt`. The templates can be used to generate various types of markup like HTML, XML, or TXT and are totally decoupled from the controller. Various kinds of markup can be plugged in as needed.

The Twirl template is just a normal text file that contains small blocks of Scala code. Templates help to create composite views and help in a component-based view generation.

The `@` character marks the beginning of the dynamic code in the template. Chapter 4 of this book provides detailed explanation of views and Twirl templates.

For the time being, let's understand what is defined in the `index.scala.html` file:

```
@main("Welcome to Play") {
  <h1>Welcome to Play!</h1>
}
```

`@main("Welcome to Play")` calls another template, `main.scala.html` and passes it the page title "Welcome to Play" and the HTML content in the second parameter, enclosed within the `{}`.

Hence you can infer that main template file should take two parameters: a string for the title and HTML content as the second parameter.

Open the `main.scala.html` to validate this. The file starts with `@(title: String)(content: Html)`; this is just like any normal method that accepts two parameters.

```
@*
```

```
* This template is called from the `index` template. This template
* handles the rendering of the page header and body tags. It takes
* two arguments, a `String` for the title of the page and an `Html`
* object to insert into the body of the page.
```

```
*@
```

```
@(title: String)(content: Html)
```

```
<!DOCTYPE html>
```

```
<html lang="en">
```

```
  <head>
```

```
    @* Here's where we render the page title `String`. *@
```

```
    <title>@title</title>
```

```
    <link rel="stylesheet" media="screen" href="@routes.
Assets.versioned("stylesheets/main.css")">
```

```
    <link rel="shortcut icon" type="image/png" href="@
routes.Assets.versioned("images/favicon.png")">
```

```
  </head>
```

```
  <body>
```

```
    @* And here's where we render the `Html` object containing
    * the page content. *@
```

```
    @content
```

```
    <script src="@routes.Assets.versioned("javascripts/
main.js")" type="text/javascript"></script>
```

```
  </body>
```

```
</html>
```

The title string is inserted to HTML `<title>` using `@title` and HTML content using `@content` markup.

Now you know the different elements and how they are wired together. Let's take this further by writing a new action and a view.

Let's create an entry for the Hello World method:

```
GET    /hello                               controllers.  
HomeController.hello()
```

Save the routes file. The next step is to code your controller to handle the request.

Controller and View

Inside the `app/controllers` folder, you should find the `HomeController.java` file. This is your default controller.

```
package controllers;  
  
import play.mvc.*;  
import views.html.*;  
  
import java.time.LocalDate;  
  
/**  
 * This controller contains an action to handle HTTP requests  
 * to the application's home page.  
 */  
public class HomeController extends Controller {  
  
    /**  
     * An action that renders an HTML page with a welcome message.  
     * The configuration in the <code>routes</code> file means that  
     * this method will be called when the application receives a
```

```

    * <code>GET</code> request with a path of <code>/</code>.
    */
    public Result index() {
        return ok(views.html.index.render());
    }
}
}

```

Let's examine the `HomeController` class in detail to understand what is happening in it. First of all, it extends from `play.mvc.Controller`. When you write a new controller, make sure you extend from `play.mvc.Controller`.

You've seen the `index` method before: it is pretty simple and it just has a single line. But it does a lot of smart things. The `ok()` method is closely related to HTTP status 200, or the success response. If you want to return a HTTP not found, you can use the `notFound` method. This is the beauty of Play; it closely resembles HTTP protocol and there is no need for any fancy code to convert your exceptions to corresponding HTTP status codes. You can code and talk the language of HTTP.

The `ok()` method returns the HTML output generated by the `render` method, defined in the view named `index`. You've seen this view in `index.scala.html`. Views in Play use Scala and follow the naming convention of "`viewname.scala.html`". This file gets compiled by the Play compiler into the corresponding Java class, which can be directly used in the controller. This ensures type safety and less bugs. Remember that in other frameworks like struts or Spring MVC, you typically return a string as view name and the framework resolves that to a file. In Play, there is no need for that. The views are straightaway available as Java class files and you can ensure compile-time type safety.

Since you have already added the routes entry for the `hello` method, let's proceed with adding the controller method and create the view.

View

Create a new file inside the `app/views` folder and name it `hello.scala.html` and add the following contents:

```
@(message: String)
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Hello World </title>
</head>
<body>
    <h1> @message </h1>
</body>
</html>
```

This template takes a single parameter and places that inside the HTML `<h1>` tags.

Controller

Now edit the `HomeController`:

```
package controllers;

import play.mvc.*;
import views.html.*;

import java.time.LocalDate;

/**
 * This controller contains an action to handle HTTP requests
 * to the application's home page.
 */
public class HomeController extends Controller {
```

```
/**
 * An action that renders an HTML page with a welcome message.
 * The configuration in the <code>routes</code> file means that
 * this method will be called when the application receives a
 * <code>GET</code> request with a path of <code>/</code>.
 */
public Result index() {
    return ok(views.html.index.render());
}
public Result hello() {
    return ok(views.html.hello.render("Today is
    " + LocalDate.now()));
}
}
```

Open your browser and go to

<http://localhost:9000/hello>

You will see the message “Today is” and the current date. See Figure 1-8.

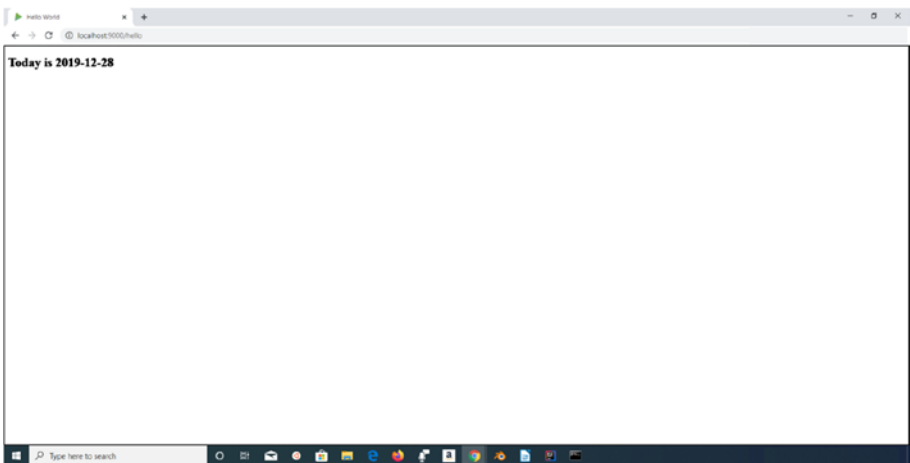


Figure 1-8. *The Hello page*

Just change any HTML code in the view and refresh the browser. Play will perform on-the-fly compilation and render the view.

Enhancing the View

Let's now enhance the `hello.scala.html` file to reuse the main template already available in the `view` folder. By using this template you will be able to get the common theming across pages because `main.scala.html` defines the common layout and style sheets.

Go to the `public/stylesheets` folder and open `main.css`. This is the master style sheet for the application. You will make the background black and the text white in color:

```
body {
  background: black;
}
h1,h2 {
  color: white;
}
```

Go to `http://localhost:9000/` to see the changes. But wait, if you go to `http://localhost:9000/hello`, the display shows the white background! Why?

The reason is your style sheet definitions and all site-wide settings are defined in the `main.scala.html` template but you never used it in the `hello.scala.html` file. Let's fix that.

Edit the `hello.scala.html` and replace its contents as follows:

```
@(message: String)
@main("Hello World") {
  <h2>@message</h2>
}
```

You want a common theme across the web application and all pages should inherit the site wide settings as is. You don't want to scatter the site-wide definitions across all pages, so put all common settings like header, footer, style sheet definitions, JavaScript inclusions, etc. in a single file; that is what `main.scala.html` is used for. It defines the common elements applicable to all pages.

The `main.scala.html` file is converted by Play into a method equivalent and can be invoked from other pages using its name. The name doesn't include the `.scala.html` part. For instance, `main.scala.html` is invoked using its name, `main`. The `hello.scala.html` file calls the `main.scala.html` as a method call and reuses the common definitions. Let's examine the contents of the `hello.scala.html` file in detail to understand what is going on.

The `hello.scala.html` file is very simple. It just invokes `main.scala.html` and passes it two arguments: the title of the page as a string and the HTML content to be included. The first argument to `main` is the string "Hello World" and the second argument is the content within `{}`. In short, you pass the title and the HTML content defined in `hello.scala.html` (the content within `{}`) to `main.scala.html`.

Now you have a common theme across all of your pages. Any other page you may define should follow the same approach. If you want to include a JavaScript library like JQuery or bootstrap to all pages, just define it in `main.scala.html` and it will be available to all pages, if it follows the approach mentioned above.

Testing Play Applications

Let's now understand how to unit test the important parts of a Play application: views and controller classes.

Play supports test cases using Junit and provides helper classes and utilities to make testing the application as easy as possible.

Tests should be created inside the tests folder within the project root. Let's write test cases for the views and controllers.

Testing Views

Let's write the test case for the `hello.scala.html` view. Create a new file inside the test folder and name it `HelloViewTest.java`:

HelloViewTest.java

```
import org.junit.Test;
import play.twirl.api.Content;

import static junit.framework.TestCase.assertEquals;

public class HelloViewTest {

    @Test
    public void renderTemplate() {
        Content html = views.html.hello.render("Welcome to Play!");
        assertEquals("text/html", html.contentType());
        assertEquals(html.body().toString().contains("Hello World"));
    }
}
```

The above test renders the `hello.scala.html` view and checks that the content type is indeed `html` and also inspects the content to confirm that it contains the string "Hello World".

You can run the test directly within your IDE. Figure 1-9 shows how you can run test from IntelliJ.

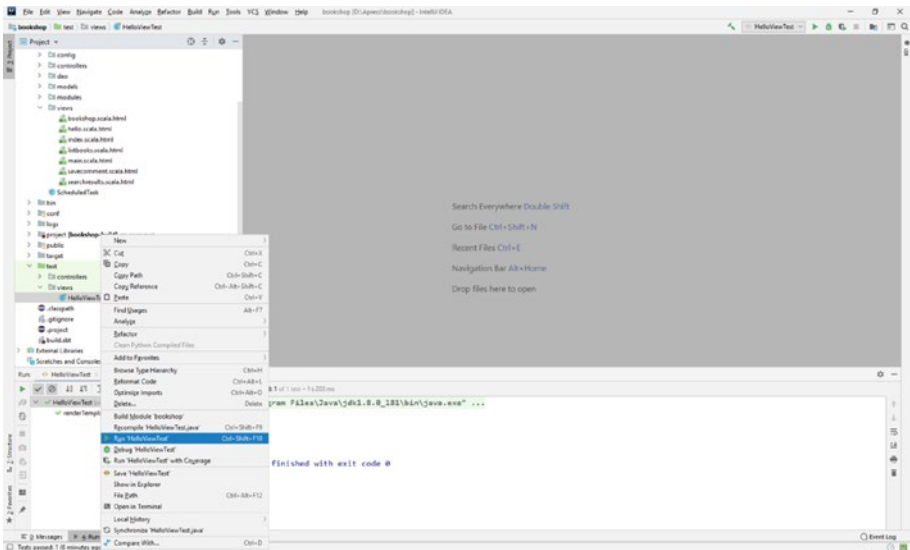


Figure 1-9. Running a test

You can also run a test from the command prompt using `sbt`. Open the command prompt, go to the project root directory, and run

```
sbt test
```

Note that `sbt test` will run all tests in the test folder. If you want to run only a particular test, use the `testOnly` command.

Testing Controllers

Create new file named `HomeControllerTest.java` inside the test folder and add the contents as below. This test validates the `index` method of the controller mapped to the `/URL` path.

HomeControllerTest.java

```
import org.junit.Test;
import play.Application;
```

CHAPTER 1 GETTING STARTED WITH PLAY 2

```
import play.inject.guice.GuiceApplicationBuilder;
import play.mvc.Http;
import play.mvc.Result;
import play.test.WithApplication;

import static org.junit.Assert.assertEquals;
import static play.mvc.Http.Status.OK;
import static play.test.Helpers.GET;
import static play.test.Helpers.route;

public class HomeControllerTest extends WithApplication {

    @Override
    protected Application provideApplication() {
        return new GuiceApplicationBuilder().build();
    }

    @Test
    public void testIndex() {
        Http.RequestBuilder request = new Http.RequestBuilder()
            .method(GET)
            .uri("/");

        Result result = route(app, request);
        assertEquals(OK, result.status());
    }
}
```

Run the test using the following sbt command:

```
sbt test
```

To run the `HomeControllerTest` alone, perform the following steps (Figure 1-10):

- Open the command prompt.
- Go to the project root.
- Type **sbt** to go to the sbt shell.
- **testOnly HomeControllerTest**

```

C:\WINDOWS\system32\cmd.exe - sbt
[bookshop] $ testOnly HomeControllerTest
[info] a.e.s.Slf4jLogger - Slf4jLogger started
[info] p.a.d.HikariCPConnectionPool - Creating Pool for datasource 'default'
[info] c.z.h.HikariDataSource - HikariPool-1 - Starting...
[info] c.z.h.HikariDataSource - HikariPool-1 - Start completed.
[info] p.a.d.HikariCPConnectionPool - datasource [default] bound to JNDI as DefaultDS
[info] o.h.j.i.u.LogHelper - HHH000204: Processing PersistenceUnitInfo [
  name: defaultPersistenceUnit
  ...]
[info] o.h.Version - HHH000412: Hibernate Core (5.3.7.Final)
[info] o.h.c.Environment - HHH000206: hibernate.properties not found
[info] o.h.a.c.Version - HCAN000001: Hibernate Commons Annotations (5.0.4.Final)
[info] o.h.d.Dialect - HHH000400: Using dialect: org.hibernate.dialect.H2Dialect
[info] o.h.v.i.u.Version - HV000001: Hibernate Validator 6.1.0.Final
[info] a.e.LoggingFilter - GET / took 94ms to complete and produced the status 200
[info] p.a.d.HikariCPConnectionPool - Shutting down connection pool.
[info] c.z.h.HikariDataSource - HikariPool-1 - Shutdown initiated...
[info] c.z.h.HikariDataSource - HikariPool-1 - Shutdown completed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 8 s, completed Jan 5, 2020 11:42:02 PM
[bookshop] $
  
```

Figure 1-10. *testOnly*

This concludes Chapter 1. Before we discuss the controllers and other Play Framework components, let's dive a little deeper into the build system (sbt) you are going to use.

CHAPTER 2

Build System

Play uses sbt as its build system, as you saw in Chapter 1. A basic understanding of sbt is a must when you work with Play-based projects. The objective of this chapter is to provide a quick introduction to sbt.

Scala Build Tool/Simple Build Tool

sbt is the build tool for Scala and Java projects. It is similar to the Maven build tool typically used in Java projects. sbt is heavily inspired by Maven. As a build tool, it provides capabilities to compile, run, test, and package projects. Play comes packaged with sbt, so there is no need to install sbt separately. Sbt was created by Mark Harrah. There is confusion about whether sbt stands for “Scala build tool” or “simple build tool.” In fact, when the sbt project started, it was announced as “simple build tool” but over the years it was widely called “Scala build tool.” So both usages are correct. My personal preference is “simple build tool” because it can be used as the build tool for both Scala and Java projects.

Core Principles

sbt always sticks to four core principles:

1. Everything should have a type, enforced as much as is practical.
2. Dependencies should be explicit.

3. Once learned, a concept should hold throughout all parts of sbt.
4. Parallel is the default.

Benefits of sbt

sbt has the following benefits:

1. It uses the Scala language to describe the build.
2. There's no need to write big and large `pom.xml` files. The build configuration is in code and not in XML.
3. It works for both Scala and Java.
4. It requires a minimal configuration.
5. It offers declarative dependency management (via Ivy).
6. It has sensible defaults.

sbt build uses something called tasks (yes, it is a task and nothing more) and a task can have dependencies. So a sbt build is a tree of task dependencies that is to be executed.

When you want to do something, you have to execute a task. By default, the tasks run in parallel in sbt. If you want to order the execution of tasks, it can be done by specifying dependencies between tasks. For example, `compile` and `test` are two tasks but `test` has a dependency with `compile`, so when you execute the `test` task, the `compile` task will get executed before it.

Chaining of tasks is a lot easier with sbt. You can pass the output of a task to another dependent task. Internally, sbt keeps an immutable map (key-value pairs) describing the build. Because of this you can see that most of the entries in the build file are key-value pairs. For instance, the project name is provided as a key-value pair and it maps to a string value, the name of your project.

The build configuration file in sbt is the `build.sbt` file and this is similar in intent to that of the `pom.xml` in Maven. The `build.sbt` contains the build definition of the project.

Let's look at some code and practical examples to get a better understanding.

Project Structure

A typical project structure has the `build.sbt` file in the root directory and the `build.properties` and `plugins.sbt` files inside the `/project` directory:

```
HelloProject
```

```
/project
```

```
build.properties: Specifies the sbt version to be
used in the project. If the particular sbt version is not
available locally, the sbt launcher will download it.
```

```
plugins.sbt: Definition for sbt plugins
```

```
build.sbt: File containing build and project
settings
```

build.properties

```
sbt.version=0.13.0
```

Before you examine the sbt file structure created by Play for your project, let's understand the structure of `build.sbt` in general and try few examples

Go to any directory, create a folder called `helloworldsbt`, create a file named `build.sbt`, and add the following contents:

```
name := "helloworldsbt"
organization := "com.domainname.example"
```

```
version := "1.0-SNAPSHOT"
lazy val hello = (project in file("."))
.settings(
  name := "HelloWorld Proj"
)
scalaVersion := "2.13.0"
```

The `build.sbt` file holds a sequence of key-value pairs called setting expressions and has a general structure of

```
key operator setting/task body
```

For example:

```
organization := "com.domainname"
```

Breaking it down a bit more:

- 1) The left-hand side is the key.
- 2) Operator (`:=`)
- 3) The right-hand side is the body.

The `organization`, `version`, `name`, etc. are predefined, readily available keys.

Using sbt

Go to the root folder of the `helloworldsbt` project and type `sbt` from a command prompt. You can see that `sbt` downloads the needed jars and you will be in the `sbt` prompt as

```
sbt:helloworldsbt>
```

You don't have any source files in the `helloworldsbt` project and it is fine. You can try many `sbt` commands and their usage without source code.

Here are the most common commands used in a typical project:

- `help`: Displays sbt help
- `compile`: Compiles the source code
- `test`: Executes the test cases
- `run`: Runs the main class
- `package`: Creates a jar file
- `exit`: Exits from the sbt prompt

When you execute `run`, you will get the following error:

```
No main class definition found
```

Ignore it for now as it is because you don't have any source code and `run` expects to find a main method to run.

As mentioned earlier, `compile`, `run`, `test`, etc. are readily available tasks in sbt that perform a particular action. Just like these tasks, sbt lets you create custom tasks and use them in the build. Let's try a custom task.

Modify `build.sbt` as follows:

```
name := "helloworldsbt"
organization := "com.domainname.example"

version := "1.0-SNAPSHOT"

lazy val hello = taskKey[Unit]("Custom Task")

lazy val root = (project in file(".")).settings(
  hello := { println("This is a custom task !!") }
)
scalaVersion := "2.13.0"
```

The following line defines the tasks and assigns it to the variable `hello`:

```
lazy val hello = taskKey[Unit]("Custom Task")
```

The next line adds this task and provides an implementation, which in this case is very simple, like printing a string to the console. That's it; you have defined a custom task.

```
lazy val root = (project in file(".")).settings(
  hello := { println("This is a custom task !!") }
)
```

To test this, go the sbt prompt and type **hello**.

```
sbt:helloworldsbt> hello
This is a custom task !!
[success] Total time: 1 s
```

From your project root folder you can even try **sbt hello**:

```
C:\Test\helloworldsbt>sbt hello
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option
MaxPermSize=256m; support was removed in 8.0
[info] Loading project definition from C:\Test\helloworldsbt\project
[info] Loading settings from build.sbt ...
[info] Set current project to helloworldsbt (in build file:/C:/
Test/helloworldsbt/)
This is a custom task !!
[success] Total time: 0 s, completed Nov 8, 2019 1:03:59 PM
```

Now let's look at the `build.sbt` definition created by Play for the bookshop project.

When you create a Play project, the build automatically creates the required sbt folder structures and files:

```
build.sbtname := "bookshop"
organization := "com.stackrules.example"
version := "1.0-SNAPSHOT"
lazy val root = (project in file(".")).enablePlugins(PlayJava)
```

```
scalaVersion := "2.13.0"
libraryDependencies += guice
EclipseKeys.preTasks := Seq(compile in Compile, compile in Test)
EclipseKeys.projectFlavor := EclipseProjectFlavor.
Java          // Java project. Don't expect Scala IDE
EclipseKeys.createSrc := EclipseCreateSrc.
ValueSet(EclipseCreateSrc.ManagedClasses, EclipseCreateSrc.
ManagedResources)
```

Let's analyze each line. The entries in the above files are settings. Each setting has to be separated by an empty line. That is very important.

Setting Definition

As you saw above, a setting contains a key, an operator, and an initialization. For instance, `name := "bookshop"` sets the project name to `bookshop`.

`name-key := assignment operator "bookshop" - the initialization value`

That is how you set a static value for a setting. The value of a setting is computed only once and kept around. A task is also similar to a setting having a key and a value. The important difference is that a task is recomputed every time when it is invoked. We will come back to tasks later.

In the `libraryDependencies` section in the `build.sbt` file, you can see this entry:

```
libraryDependencies += guice
```

This indicates the build is including `guice` Play module as a dependency. To proceed to the other sections in the chapters ahead, you need more modules to be available to Play. Let's modify the `libraryDependencies` section to add them now.

Replace the line

```
libraryDependencies += guice
```

with

```
libraryDependencies += Seq(  
  javaJdbc,  
  cacheApi,  
  guice  
)
```

Ebean ORM is available as an external plugin for Play so add the following to the `plugins.sbt` file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-play-ebean" % "5.0.2")
```

You are done with the changes to the build settings!

The typical library dependency format is

```
libraryDependencies += groupId % artifactID % revision
```

For example,

```
libraryDependencies += "org.apache.derby" % "derby" %  
"10.4.1.3"
```

You can find the `groupId` and `artifactId` of any jar by visiting Maven Central Repository at <https://mvnrepository.com/repos/central> and search using the name. For example, type **derby** and choose the version to see its dependency information. The definition for derby 10.4.1.3 is shown in Figure 2-1.

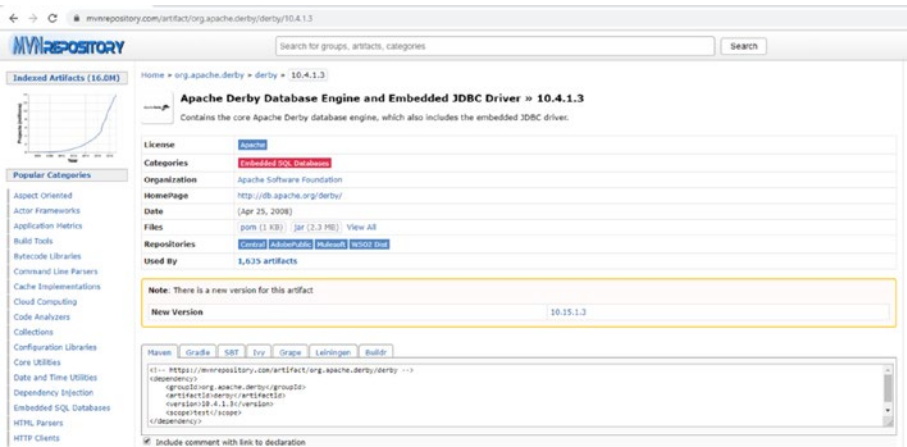


Figure 2-1. Maven Central Repository

The above line adds Derby version 10.4.1.3 as a dependency. If you need to add more dependencies, you have to add another line and so on. But there is also a shorthand technique to add dependencies all at once rather than specifying each one line by line:

libraryDependencies += Seq(

```

groupID % artifactID % revision,
groupID % otherID % otherRevision
)

```

sbt depends on plugins to extend the build settings defined in the build.sbt file. A plugin adds more functionality to the build and is reused across sbt projects. For example, someone could develop a plugin that performs code coverage during the build. To integrate sbt and Play, you need to use the sbt-plugin from Typesafe. The plugins.sbt file is where all plugin definitions go:

```

// Comment to get more information during initialization
logLevel := Level.Warn

```

```
// The Typesafe repository
resolvers += "Typesafe repository" at "http://repo.typesafe.
com/typesafe/releases/"
// Use the Play sbt plugin for Play projects
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.8.0")
```

Resolvers

By default, sbt uses the standard Maven 2 repository. But not all jars exist in the default repositories. If your jar is living in another repository, you can add that repository as a resolver.

Format for adding a resolver:

```
resolvers += name at location
```

There are three parts to this: the name and location separated by the at keyword.

Adding the Jboss repository:

```
resolvers += "JBoss repository" at "https://repository.jboss.
org/nexus/content/repositories/"
```

Complete build.sbt

```
name := """"bookshop""""
organization := "com.stackrules.example"
version := "1.0-SNAPSHOT"
lazy val root = (project in file(".")).enablePlugins(PlayJava)
scalaVersion := "2.13.0"
libraryDependencies ++= Seq(guice, javaJdbc, cache)
EclipseKeys.preTasks := Seq(compile in Compile, compile in Test)
```

```
EclipseKeys.projectFlavor := EclipseProjectFlavor.
Java                       // Java project. Don't expect Scala IDE
EclipseKeys.createSrc := EclipseCreateSrc.
ValueSet(EclipseCreateSrc.ManagedClasses, EclipseCreateSrc.
ManagedResources)
```

You can easily escape characters and symbols inside strings; you just need to wrap the text within triple quotes. This is why you see triple quotes for the name. If you need to include a space or a colon or an apostrophe in the name, this syntax helps.

Complete plugins.sbt

```
// The Play plugin
addSbtPlugin("com.typesafe.play" % "sbt-plugin" % "2.8.0")
// Defines scaffolding (found under .g8 folder)
// http://www.foundweekends.org/giter8/scaffolding.html
// sbt "g8Scaffold form"
addSbtPlugin("org.foundweekends.giter8" % "sbt-giter8-scaffold"
% "0.11.0")
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" %
"5.2.2")
addSbtPlugin("com.typesafe.sbt" % "sbt-play-ebean" % "5.0.2")
```

Quick Recap of SBT Commands

- `sbt`: Starts the sbt console
- `run`: Runs the main method of the application
- `compile`: Compiles the source code in the `src/main/scala` directory

CHAPTER 2 BUILD SYSTEM

- `test`: Executes all test cases
- `testOnly`: Provides the complete name of the test case to run only the specific test case
- `test:compile`: Compiles only the test sources (`src/test/scala`)
- `package`: Creates a jar containing the classes from the source folder and artifacts from the resource folders
- `doc`: Generates Scala docs
- `exit`: Quits the sbt prompt

CHAPTER 3

Play Controllers and HTTP Routing

This chapter focuses on the MVC part of the Play application: how MVC plays a key role in the Play application framework. Before we get into the details of the HTTP routing and controllers in Play, it is good to have a quick introduction to MVC. If you are familiar with MVC, you can skip this and head straight to the “HTTP Routing” section.

MVC Programming Model

MVC (model-view-controller) is a framework for building web applications using the famous MVC design pattern. MVC defines an application into three logical layers: the business layer (model), the display layer (view), and a routing and input control (controller). See Figure 3-1.

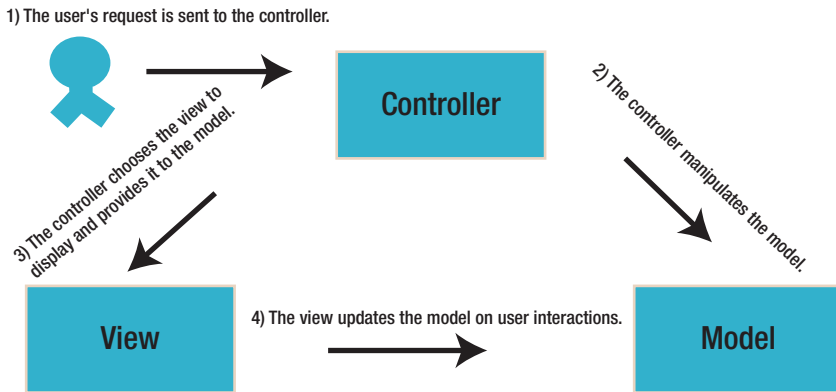


Figure 3-1. MVC

MVC architecture works like this. The user interacts with the view and changes are sent to the controller by the view. The controller receives the changes, invokes the model, and applies the validations and logic in the model. The controller chooses the view and sends the updated model to the view. The view is rendered and is send back to the browser client.

Both the view and the controller depend on the model. The model doesn't have any dependency with the view or controller. This is one of the key benefits of the separation. This separation allows the model to be constructed and tested independently of the view.

Model

The model is the application's data and the business logic on that data. The model is an independent component and doesn't depend on the controller or the view. This means the model can be reused without associated views or the controller.

The model handles the application's business logic and is responsible for retrieving the data from the database, performing updates, enforcing validations, and being the core logical and analytical part of the application.

Consider the model representing the customer of the application; it's responsible for handling all validations and logic related to the customer entity. The `deactivate` method in the following code is an example of business logic within a model. This method handles the deactivation logic of the customer, thereby encapsulating the domain logic within the model itself. The domain logic is encapsulated within the domain object, so the model can be reused across applications/subsystems.

```
package models;

import java.util.List;
import controllers.Order;

public class Customer {

    private long id;
    private String name;
    private boolean loyaltyMember;
    private boolean isActive;
    private List<Order> orders;

    public List<Order> getOrders() {
        //Logic
    }
    public boolean deactivate() {
        //Logic and validation to deactivate a customer
    }
}
```

The `customer` is a domain model and contains the data and the associated business logic and validations applicable to the customer.

Of course, if you follow domain-driven design, you can split the model into many business objects and they can handle domain-specific logic and validations. Spring beans, JPA beans, simple POJO, etc. are typically used for implementing domain models.

View

The view handles the display of the data and gets needed data attributes from the model. The sole purpose of the view is to display the data. Consider a screen showing the customer data of the application. This screen may show the customer details, orders made by the particular customer, and so on. If there is a new requirement to show the same customer data in a drill-down mode instead of a tabular way in another screen, only a new view needs to be created. There are no changes to be done in the model. Thus views can be created and modified independently of the model.

Controller

The controller handles interactions with the user. Controllers typically perform the following:

- Get input data from the view and send it to the model for persistence.
- When the model changes, the controller sends the updated model to the view and renders the view.

MVC helps make development in a team easy and manageable when working on large projects. It provides a clean separation of concerns.

HTTP Routing

The job of the HTTP router is to translate the incoming HTTP request to an action call. That is, it maps an HTTP request to a method defined in the controller class. This configuration is maintained in the routes files inside the conf folder. The routes file also gets compiled by Play and, because

of that, if you have any error in the routes file, it will be displayed in the browser. This is very helpful during development time.

Let's look at HTTP routing fundamentals. An HTTP request has two parts: the protocol part (GET, POST, PUT, etc.) and the request path that includes the query string.

To define a route, define the protocol and URL path in the `conf/routes` file. Before that, create a new controller class inside the `app/controllers` folder and name it `Application`. Make sure to extend the class from `play.mvc.Controller`.

```
package controllers;

import play.mvc.*;
import views.html.*;

public class Application extends Controller {

}
```

You will add methods to this controller after completing the routes section.

Edit the `conf/routes` file and add the following entries:

```
# Home page
GET    /bookshop                controllers.
Application.index

# Show details of a Book
GET / bookshop /book/:id/  controllers.
Application.getBook(id:String)

# Add a comment to the book
POST /bookshop/book/save/comment/ controllers.Application.
saveComment(request:Request)
```

```
#Search a book by Title
GET /bookshop/book/search/           controllers.
Application.searchByTitle(keyword:String)
```

This example features a basic application for managing books online. It utilizes HTTP methods and URIs appropriately.

The entries defined in the routes file show important features of routing. The first entry indicates that the home URL (say `http://yourdomainname/bookshop`) points to the `index` method defined in the `Application` class (a controller). The rest of the definitions show how to dynamically include parameters in path and as query string parameters. Let's examine this in more detail.

Static Definition

The definition

```
GET /           controllers.Application.index
```

indicates that the root path doesn't take any parameters and it is routed to the `index` method defined in the `Application` class. You are using the `GET` protocol for this example. If your webserver supports it, you can also use the `PUT` and `DELETE` HTTP protocols as well.

The overall structure of a routes entry is

```
Protocol  URLPATH  Controller Mapping
```

Dynamic Parts in a URL

Let's say you want to retrieve the details of a particular book but you don't know in advance which book a user might select, so the book id has to be dynamic:

```
GET /bookshop/book/:id/ controllers.  
Application.getBook(id:String)
```

The dynamic part is indicated by a colon (:) followed by the parameter name (in this example `:id`). Play will extract the dynamic part from the URI and supply it as a parameter to the `getBook` method. This means the URL paths `/bookshop/book/123/`, `/bookshop/book/12453/`, etc. will get mapped to the `getBook` method defined in the `Application` class.

Key point `:variablename` is used to define the dynamic parts of the URI. Example `:id`.

You can also configure dynamic parts spanning several forward slashes (/). For instance,

```
GET /bookshop/book/images/* controllers.Application.  
fetchImage(name:String)
```

matches the URLs

```
/bookshop/book/images/book1.jpg  
/bookshop/book/images/books/book2.jpg  
/bookshop/book/images/books/thumbnails/small/image1.jpg
```

The name parameters passed to the `fetchImage` method will be `book1.jpg`, `books/book2.jpg`, and `books/thumbnails/small/image1.jpg`, respectively.

You have used the fully qualified name of the controller class (`controllers.Application`). You can define and use another controller if you wish.

Play also supports regular expressions in the path definitions. To define a regular expression in the dynamic part, you can use the syntax

```
$id<regex > syntax
```

For example:

```
GET /bookshop/book/:id/page/$page<[0-9]+>/
  controllers.Application.fetchBookpage(bookid:String,
  page:Integer)
```

In this definition, there are two dynamic parts: one for id (book's unique id) and the other for the page number. The page number part is defined as a regular expression accepting only numbers. Play will validate and throw an error if invoked with a nonnumeral in the URI. For example, when invoked as `http://localhost:9000/bookshop/book/10/page/12/`, the page will get rendered properly because the page number is a valid number in the request. If the request is made as `http://localhost:9000/bookshop/book/10/page/a/`, Play will throw an action not found error because the page number in the request is not a valid number.

Key point `$variablename<regular expression>` is used to define dynamic parts to match regular expression.

Passing Fixed Values

For certain methods, you need to always pass a fixed value. You can configure such cases as

```
GET /bookshop//authors/          controllers.Application.
  authors(limit: Integer = 10)
```

Here you are passing a default of 10 to the show method. This means it shows only 10 books when a user invokes the show method.

Optional Parameters

A typical use case is when some parameters may be passed sometimes but are missing in certain situations. To handle such cases, you can use the optional definition:

```
# Shows all comments or comments made by a particular user
GET /bookshop//showcomment/ controllers.Application.
showComment(userid ?= null)
```

In this example, if `userid` is passed, it will retrieve the comments by the user and if not fetch all comments.

Key point `variable? = default-value` is used to define optional parameters.

I hope you now understand the static and dynamic URL configurations. Let's now look at how to declare configurations for the application, before you proceed to the controllers section and other chapters, because you will be making entries to the `application.conf` file in the coming chapters.

Application Configuration Using `application.conf`

Play provides a single file called `application.conf` found inside the `conf` directory to configure application-level configurations. This `conf/application.conf` file can be used to configure database connection strings, log levels, modules to enable additional functionality, etc. The `application.conf` file is an UTF-8 encoded file.

The entries in this file follow the scheme

```
# comment
key = value
```

By default, the Play application starts listening to HTTP requests on port 9000. By adding an `http.port` entry in the `application.conf` file, this default port can be changed:

```
#http.port =8085
```

The default logging level defined in the `application.conf` file is as follows. In production, do remember to change it to the appropriate values. The general practice is to set it to `ERROR`.

```
# Logger used by the framework:
logger.play=INFO
# Logger provided to your application:
logger.application=DEBUG
```

Controllers

Controllers in Play model the controller element of the MVC design. You have already seen how every Play request is mapped to an action using the HTTP routing configuration defined in the `conf/routes` file. In all these configurations, you can see the use of a controller named `Application`. This is done just for convenience because Play automatically creates a class named `Application` in the controller folder. You can very well create another class and use it.

A controller groups related actions together. You can create any number of controllers as you wish depending on your application's functionality. In the case of a bookshop application, you can define a `BookController` to handle all public-facing interactions with the user and define a `BackOfficeController` to define all actions that are termed

back-office functionalities like maintaining the inventory, adding a book to the system, adding images to the book, etc. Grouping the functionalities into different controllers will help in better maintainability and modularization of your application.

In Play, the controller class extends from `play.mvc.Controller`. You can define an action method that returns a `Result`:

```
public Result index() {
    return ok("Hello World.");
}
```

The `Result` is nothing but an HTTP response back to the client.

An action method can also take parameters, and these parameters are resolved as per the routing logic defined in the `conf/routes` file:

```
public Result saveRating(String bookid,int rating) {
    //logic to save rating
    return ok("success");
}
```

A typical HTTP response consists of a status code, headers, and a body. Play provides plenty of helper methods to produce all kinds of HTTP responses. For example, the `ok()` method shown in the code snippet above sends an HTTP 200 response back to the client. Similarly, Play provides methods like `notFound()`, `badRequest()`, `internalServerError()`, etc. to return all variations of HTTP responses. These helper methods are defined in the `play.mvc.Results` class. Since the controller you write is extending from the `Controller` class that extends the `Results` class, all these methods are available to your controller by default.

Here is an example to return an internal server error to the client:

```
Result response = internalServerError("Server Error");
```

Another important thing to understand is how to redirect a request to another URL or to another action. This is also fairly simple with Play. It provides a `redirect` method to do so:

```
public Result hello() {
    return redirect("/book/login/");
}
```

This redirects the user to the login page. This method sends the HTTP status code 303 to the browser.

Finishing the Bookshop Controller

Let's now add all the required methods to the `Application` controller for the bookshop application and test it. The routes look like this:

```
#Home Page
GET    /bookshop                controllers.Application.index

# Show details of a Book
GET / bookshop /book/:id/           controllers.
Application.getBook(id:String)

# Add a comment to the book
POST /bookshop/book/save/comment/   controllers.Application.
saveComment(request:Request)

#Search a book by Title
GET /bookshop/book/search/         controllers.
Application.searchByTitle(keyword:String)
```

Now the application controller (`Application.java`):

```
package controllers;

import javax.inject.Inject;
```

```

import play.mvc.*;
import play.data.DynamicForm;
import play.data.FormFactory;
public class Application extends Controller{
/**
 * Process the home page
 * @return
 */
public Result index() {
    return ok(views.html.bookshop.render());
}
/**
 * Get the details of a book by id
 * @param id
 * @return
 */
public Result getBook(String id) {
    return ok(views.html.bookshop.render());
}
@Inject
FormFactory formFactory;
/**
 * Accept a form post and save the comment
 * Shows the usage of dynamic forms to retrieve data from html
    form posts
 * @return
 */
public Result saveComment(Http.Request request) {
    DynamicForm requestData = formFactory.form().
    bindFromRequest(request);
    String comment = requestData.get("comment");

```

```

        return ok(views.html.savecomment.render());
    }
    public Result searchByTitle(String title) {
        //Query db and get the book details or get from cache
        return ok(views.html.searchresults.render());
    }
}

```

saveComment Method

The `saveComment` method handles the form post action used to save reviews. In Play, there are many ways to handle and bind form post data to model classes. The above method demonstrates how to dynamically get data from forms using the `DynamicForm` class.

The most common approach is to define a class (model) for the form data and wrap it with `play.data.Form` in a controller. Play will auto-bind the form fields to the model attributes. For instance,

```

package models;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="comment")
public class Comment {

    @Id
    private Long id;
    private String comment;
}

```

```

public String getComment() {
    return comment;
}

public void setComment(String comment) {
    this.comment = comment;
}
}

```

Save this file as `Comment.java` inside the `models` package.

Take a look at the `Application.java` class shown above. In that class, you have injected the `FormFactory` and supplied it with the `comment` model. The data binding of the form data to the model class will thus be handled automatically by Play Framework.

Relevant portions of the code from the `Application.java` file that perform the automatic binding of form data to the model class are as follows:

```

@Inject
FormFactory formFactory;

DynamicForm requestData = formFactory.form().
bindFromRequest(request);
String comment = requestData.get("comment");

```

Testing the `saveComment` Action

Any HTTP test client can be used to test this action. In this book, Postman is used to test the services. Postman is a platform that helps in testing HTTP endpoints adhering to REST, SOAP, and more. For testing via the command line, you can use `curl`. You can download Postman from www.getpostman.com/. See Figure 3-2.

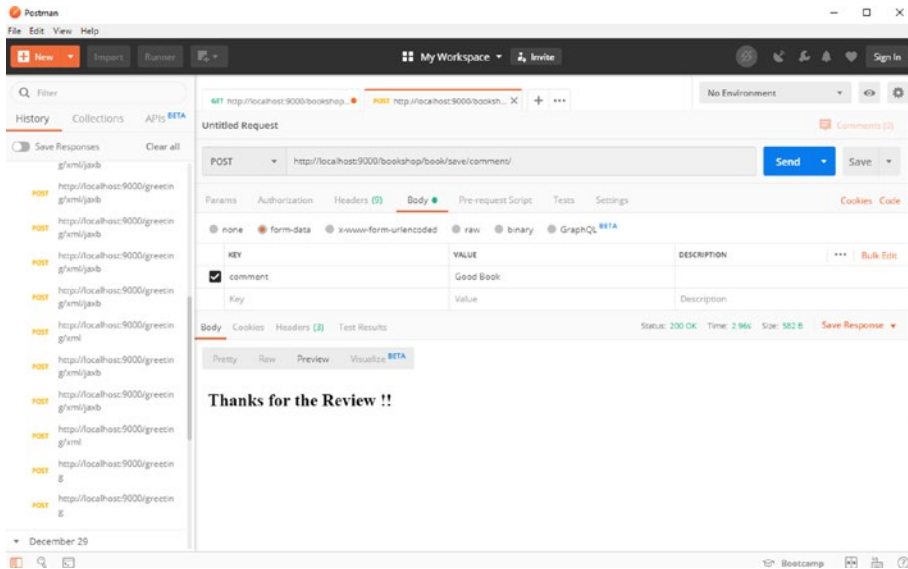


Figure 3-2. Testing saveComment

To test from a command prompt, use the `curl` command:

```
curl --location --request POST 'http://localhost:9000/bookshop/book/save/comment/' --form 'comment=Good Book'
```

Models

Since Play Framework follows the MVC principle, you should make sure that the controller layer is as thin as possible. Don't put any business logic in your controller. All your business logic should be done in your model or other helper classes. The convention in Play is to define your models inside the `app/models` folder.

By default, Play will create the views and controllers folders inside the app. The generally followed practice is to implement the model as JPA entities. In this chapter, we will just annotate the models with JPA annotations; we won't do any JPA configuration or persistence implementation. In later chapters in this book, we will explore JPA in detail.

```

package models;

import javax.persistence.*;

@Entity
@Table(name="book")
public class Book extends Model {

    @Id
    private String id;
    private String name;
    private String author;

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}

```

Scoped Objects

In any web application, data may need to be maintained across different pages of the application so as to maintain a conversational session with the user of the application. In certain cases, you may need to scope the data to only the next request and not the entire session. Play provides support for both cases. The controller class is responsible for storing and accessing scoped objects.

Play Framework provides two kinds of scoped objects:

- Session scope
- Flash scope

Data stored in the session scope spans multiple HTTP requests and is available during the whole user session. Data stored in the flash scope is only available to the next request.

Play implements the above two scopes by means of cookies and therefore it enforces a size limit of 4 KB. You can only store string values in these scopes.

Session Scope

In Play, there is no timeout for session. A session expires when the user closes the browser. But if your application needs to configure a timeout, it can be done by adding the session configuration in the `application.conf` file as follows:

application.conf

```
play.http {
  session {
    # Sets the max-age field of the cookie to 5 minutes.
    # maxAge = 300
  }
}
```

Play provides the following methods to work with session scope:

addingToSession

```
public Result dard(Http.Request request) {
    return ok("Welcome!").addingToSession(request,
        "dashboard login", "useremailaddress");
}
```

removingFromSession

```
public Result disconnect(Http.Request request) {
    return ok("disconnection success").
        removingFromSession(request, "disconnect");
}
```

To access session data, use the following code:

```
public Result index(Http.Request request) {
    return request
        .session()
        .get("dashboardlogin")
        .map(id -> ok("Welcome: " + user))
        .orElseGet(() -> unauthorized("Try login again"));
}
```

In this example, you use the Session API of Play to retrieve the value; if it is empty, you simply return an unauthorized response.

To discard the entire session, Play provides the `withNewSession` method:

```
public Result quit() {
    return ok("Logged out").withNewSession();
}
```

Flash Scope

The flash scope is similar to session but with the following differences.

Firstly, **data is kept for only one request**. For the first request, you set the flash scope, and when user moves to the next page or section by making another request, this data is available and can be retrieved in that request's process by a method in the controller. Secondly, it uses unsigned cookie. Both session and flash scope uses browser cookies to achieve the behavior, and with flash scope the cookie is an unsigned one. So don't use flash scope to store any sensitive data. Use it only for storing success messages or error messages. Use the following code to add data to the flash scope:

```
public Result about() {
    return redirect("/company").flashing("aboutinfo",
        "about info requested");
}
```

When a request arrives, data with the key "aboutinfo" is added to the flash scope and is redirected to the company page. In the company page, the controller method mapped to the /company URL can access this data as

```
public Result companyinfo(Http.Request request) {
    return ok(request.flash().get("aboutinfo").orElse
        ("Company Name"));
}
```

CHAPTER 4

Play Views and Templating with Scala

By default, views in Play are HTML and they can be made dynamic using the Scala expression language. In fact, you can return a JSON or XML response as well if necessary. Let's focus first on HTML Scala views and how the Play template engine helps you to create modular, expressive, and fluid views. Even though the template engine uses Scala as the expression language, this is not a problem for Java developers. The Scala language used in templates is very simple. As you saw in Chapter 1, the template engine used by Play is Twirl. Recall that the template files must be named `{name}.scala.{ext}`. The `ext` can be `html`, `js`, `xml`, etc. Play prefers convention or configuration. When you compile, the compiler will generate Scala source files and they will be compiled along with the rest of the source code.

Like all other classes in Play, the template is also compiled on the fly and you can see errors directly in the browser. You should keep in mind that a template/view is not a place to write complex logic. Write all your presentation-related logic in the controller or the helper classes. All your business logic should be in your business entity classes.

Most of the time you will only access data from your model objects. Stated another way, the Scala template language only gives limited options to write complex logic; its purpose is to provide a simpler means to access data in the model. This is by design to limit developers from writing complex code embedded in the views.

Key point Remember that the view is just to access data stored in the model. All logic, even something like maintaining a counter, is the job of the controller or presentation helper classes.

Composite Views

Remember that a Play view is named following the convention `viewname.scala.html`. For instance, you can create a view for the footer and name it as `footer.scala.html`. Let's examine the following page to understand this better:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
    @header("Home Page")
</head>
<body>
<div class="container">
<!-- Home Page content goes here -->
</div>
<div>
    @footer()
</div>
</body>
</html>
```

In this example, the home page uses two views to create a composite view. Take a note of the following: the header and footer are included from another view; the simplicity of incorporating the header and footer views; no complex configurations; just the use of @ symbol followed by the name of the view. You will find this simplicity all over Play Framework; it's one of the important goals of Play Framework.

The header view declares that it takes a String (title) as a parameter:

```
@(title:String)
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
<title>@title</title>
```

This parameter is supplied by the caller to render the page title. The parameter can be given a default value if required:

```
@(title : String ="Book Home")
```

@ SPECIAL CHARACTER

Scala templates use @ as the special character. Every time this character is encountered, it indicates the beginning of a dynamic statement. The end of the dynamic statement is automatically inferred by the Scala engine. Sometimes you may need to write multi-line statements and for that you can enclose the dynamic code in curly brackets or braces:

```
@(dynamic code )
@{ dynamic code }
```

Designing a General Template

Let's design the general template for all of your pages. For the sake of explanation, let's keep it very simple. All of your pages will have following content:

Header

Main Content

Footer

Let's define this template as `main.scala.html`:

```
@(title:String)(content:Html)

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
    @header(title)
</head>
<body>
    @content
    @footer()
</body>
</html>
```

Notice that along with the title, the template takes another parameter: content (of type `Html`). This is to incorporate HTML code into the template. The above template is like a method that takes two arguments: a title parameter and the HTML content.

Let's create a view that uses this template. 'The welcome view passes the two parameters from `main.scala.html` to the template to render the view. Please note that the HTML (the second parameter to the template) is passed inside the `{}`:


```
@maintemplate("welcome page") {
  <div>
    <h3>This is the welcome page</h3>
  </div>
}
```

The view (`welcome.scala.html`) uses the main template to introduce the page structure involving a header, the body, and a footer. If your views are complex, split them into multiple templates and compose the views with the help of these templates.

The rule is to keep recurring portions of the code in a separate template file. The Scala templating structure used in Play is more powerful and simpler than many existing templating engines, for example Tiles.

Code Snippets Templating Basics

Play views use a template engine called Twirl. By using a template engine, a view can render any markup like HTML, XML, and CSV with ease, even though for most of the web applications we only use HTML.

For example, in all the examples you only used HTML markup, so you named your views as `{name}.scala.html`. The general syntax for naming views is `{name}.scala.{ext}`, where `ext` can be `html`, `js`, `xml`, or `txt`. Thus you can keep different markup versions of the same page and, depending on the context, the controller may render the appropriate view. This lets you show HTML to a specific client and the XML markup to another client.

Play's template engine, Twirl, is designed in such a way that it makes front-end developers comfortable while working with dynamic parts of code. The HTML expert only bothers about the HTML markup in the template whereas another member may just deal with iterating the dynamic elements and so on. The simple Twirl syntax makes the code much easier to read and work with. Let's explore the most common Play templating elements.

Comments

```
@*****
* A comment block
*****@
```

Template Parameters

A template can take any number of parameters. It can take common Java types or custom user-defined types. For instance, consider `@(booklist <- List[models.Book])`. Here the template takes a list of book objects:

```
@(booklist <- List[models.Book])

@main("Book List") {
    @for(book <- booklist){
        <div>@book.getTitle()</div>
        <div>@book.getAuthor()</div>
    }
}
```

In this example, the template accepts a `List of Books` as the parameter and iterates the book and passes the HTML content to the `main.scala.html` file for rendering. Please note that you explored the `main.scala.html` file in previous chapters. Here is how it looks:

```
@*
* This template is called from the `index` template. This template
* handles the rendering of the page header and body tags. It takes
* two arguments, a `String` for the title of the page and an `Html`
* object to insert into the body of the page.
*@
@(title: String)(content: Html)
```

```

<!DOCTYPE html>
<html lang="en">
  <head>
    @* Here's where we render the page title `String`. *@
    <title>@title</title>
    <link rel="stylesheet" media="screen" href="@routes.
      Assets.versioned("stylesheets/main.css")">
    <link rel="shortcut icon" type="image/png" href="@
      routes.Assets.versioned("images/favicon.png")">
  </head>
  <body>
    @* And here's where we render the `Html` object
    containing
    * the page content. *@
    @content

    <script src="@routes.Assets.versioned("javascripts/
      main.js")" type="text/javascript"></script>
  </body>
</html>

```

You can provide default values to the template parameters using the syntax

```
@(title: String = "Book Home")
```

Import Statement

Consider the code `@(booklist <- List[models.Book])`. You might have noticed that the type is explicitly defined with its fully qualified package name. This becomes tedious and difficult to maintain when the number of parameters increases. Scala templates let you import the packages into the view so that you can refer to the classes by just their name.

Syntax:

```
@import packagename._
```

Example:

```
@import models._
```

You should use the `import` statement as the beginning of the template. If you declare it anywhere else, the compiler will throw an exception. The best place is to define the `import` just after the parameter declaration.

Iterating a List

Syntax:

```
@for(localvariable <- variable refering the List) {
  @localvariable.method()
}
```

Let's take the example of displaying the top N books on the home page. You can name the model as `Book`; it contains the methods `getTitle` and `getPicture`:

```
@(booklist <- List[models.Book])
@for(book <- booklist){
  <div>@book.getPicture()</div>
  <div>@book.getTitle()</div>
}
```

Iterating a Map

Syntax:

```
@for((key, value) <- mapreference) {
  Key is @key
  Value is @value
}
```

Let's take the example of iterating a Map having a String as a key and a List as its value. Your map holds the author name as the key and a list of books by the author as its value:

```
@(title: String,authorbookMap:Map[String,List[Book]])
@import models._

<!DOCTYPE html>
<html>
<head>
  @header("Main Home Page")
</head>
<body>
  @for((key,value) <- authorbookMap){
    Author - @key
    @for(book <- value){
      <div>
        @book.getTitle()
      </div>
    }
  }
</body>
</html>
```

If Blocks

Syntax:

```
@if(condition){
}else{
}
```

The if block is nothing special. It is the standard Scala if block.

Escaping Dynamic Contents

By default, the dynamic content generated by the Scala code snippets is escaped. This means it will behave as a plain text when the view is rendered. But if you need to output dynamic raw content, you can use the `@Html` tag.

Syntax:

```
@Html(code / variable)
```

CHAPTER 5

Concurrency and Asynchronous Programming

Before you dive deeper into Play, it is very important to have a good knowledge of concurrency and asynchronous programming in Java. This is essential because the examples in the chapters ahead will use lot of asynchronous programming practices.

Let's first understand the `java.util.concurrent` package, the core module in Java for dealing with concurrent programming. Knowing this will help in relating to how Play handles asynchronous web services using WS and Promise classes. You may skip this chapter if you are already familiar with concurrency, its challenges, and how Java handles it using the new classes available in the `java.util.concurrent` package since JDK 1.5.

Concurrent programming is now a lot easier in Java with the introduction of new classes in the `java.util.concurrent` package. The most important classes to focus on are

- 1) Executor
- 2) Callable<V>
- 3) Future<V>
- 4) CompletionStage<T>

Before looking into each of the above classes in detail, let's take a quick look at what concurrency is.

What Is Concurrency?

Concurrency is a technique that enables the execution of several tasks in parallel. These tasks can be parts of the same program or different programs. If a big task can be split into small chunks and if each of those chunks can be executed in parallel, it can result in better response times and throughput. When different parts of a program are run in parallel, usually lightweight processes called threads are used. The moment you use threads, you need to have proper mechanisms to allow access to shared resources, proper management of thread life cycles, scheduling, etc. The classes in the `java.util.concurrent` package help with many of these challenges.

Executor

Before the introduction of the Executor framework in JDK 1.5, thread management was the responsibility of the developer and there was no framework for that in JDK. Developers handled the creation of the threads and their management along with the actual business logic. From a design and coding perspective, this is not an ideal situation, when you take into account the benefits gained by following a “separation of concerns” design idea.

It makes sense to separate out the thread management and creation from the rest of the program, and that is what the Executor framework does. An Executor abstracts thread management activities like creation, scheduling, etc. Instead of directly creating a thread, a `Runnable` class is submitted to an Executor and it handles its execution. `ExecutorService`, a more refined interface than `Executor`, is used in practical scenarios because it provides support for `Callable` and `Future`. You will look at `Callable` and `Future` in the coming sections. Note: `ExecutorService` extends from `Executor`.

The `java.util.concurrent` package provides three Executor interfaces:

- `Executor`: A simple interface that supports launching new tasks
- `ExecutorService`: A subinterface of `Executor` that adds features to help manage the life cycle, both of the individual tasks and the executor itself
- `ScheduledExecutorService`: A subinterface of `ExecutorService` that supports future and/or periodic execution of tasks

`ThreadPoolExecutor`, `ScheduledThreadPoolExecutor`, and `ForkJoinPool` are some of the important `Executor` implementations available in JDK

At a high level, writing concurrent programs using the `Executor` framework involves the following:

- 1) Define a class/task that implements either a `Runnable` or `Callable` interface.
- 2) Configure and implement `ExecutorService` and submit the task.
- 3) Use the `Future` class to retrieve the result if the task is a `Callable` task.

Let's look at the difference between a `Runnable` and `Callable`. A `Runnable` interface does not return a result whereas a `Callable` allows for return values after completion. When a `Callable` is submitted to the `Executor` framework, it returns an object of type `java.util.concurrent.Future`. The `Future` can be used to retrieve results.

Example 1: Using Runnable

```

package com.domain.concurrency;

public class SimpleTask implements Runnable {

    @Override
    public void run() {
        System.out.println("SimpleTask, Runnable: Executing
        Logic "+System.currentTimeMillis());
    }
}

```

The SimpleTask class is the portion of logic that you want to execute in parallel by many clients. This logic is implemented in the run method and it prints the currentTime in milliseconds. In a traditional Java practice, in order to execute this in parallel by many clients, you need to create threads for each client and run those threads, along with thread coordination, stopping, etc. You also need to ensure that you don't create an unnecessary number of threads, and the threads should be reused. All of this adds to the complexity of concurrent programming and this where the ExecutorService helps because it takes care of all those aspects. Let's now create a client and ask it to run the SimpleTask in many threads:

```

package com.domain.concurrency;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Client {

    public static void main(String[] args) {
        // Step1 : Create a Runnable
        Runnable simpleTask = new SimpleTask();
        // Step 2: Configure Executor
    }
}

```

```

// Uses FixedThreadPool executor
ExecutorService executor = Executors.newFixedThreadPool(2);
for (int i = 0; i < 10; i++) {
    executor.submit(simpleTask);
}
executor.shutdown();
}
}

```

The code is very simple but it does a lot. You create an instance of `SimpleTask`, create the `ExecutorService`, and initialize it with a thread pool having two threads. Then you submit the task to the `ExecutorService`. `ExecutorService` takes care of the execution of the ten requests with the two threads it has in an efficient way.

Run `Client.java` as

```
java com.domain.concurrenc.Client
```

Output

```

SimpleTask, Runnable: Executing Logic 1578145676485
SimpleTask, Runnable: Executing Logic 1578145676485
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486
SimpleTask, Runnable: Executing Logic 1578145676486

```

By inspecting the output, you can see the time repeats in a pattern of two, meaning the `ExecutorService` schedules execution with the two threads it has and, once it becomes free, the next two requests gets served, and so forth.

Example 2: Using Callable

A `Callable` is similar to a `Runnable`. Both are designed to be executed by threads. The difference is a `Runnable` doesn't return anything and cannot throw a checked exception. The `Callable` returns a result and may throw an exception. The `Callable` interface defines the method:

```
V call() throws Exception;
```

Let's try a `Callable` using an example:

```
package com.domain.concurrency;

import java.util.concurrent.Callable;

public class CallableTask implements Callable<String> {

    @Override
    public String call() throws Exception {
        String s="Callable Task Run at "+System.
            currentTimeMillis();
        return s;
    }
}
```

The `CallableTask` returns a `String` and can also throw an exception if something goes wrong. The `CallableClient` defined below executes this task using the `ExecutorService`. The most important thing to understand is since these are asynchronous executions that return a result, `ExecutorService` returns a `Future` object, which is a proxy for the actual response. The caller has to check whether the execution has completed by invoking the `isDone` method on the `Future` and extracting the response. This is what is being done in the following code:

```

package com.domain.concurrency;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CallableClient {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // Step1 : Create a Runnable
        Callable callableTask = new CallableTask();
        // Step 2: Configure Executor
        // Uses FixedThreadPool executor
        ExecutorService executor = Executors.newFixedThreadPool(2);
        Future<String> future = executor.submit(callableTask);
        boolean listen = true;

        while (listen) {
            if (future.isDone()) {
                String result;
                try {
                    result = future.get();
                    listen = false;
                    System.out.println(result);
                } catch (InterruptedException |
                    ExecutionException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

```

    }
    executor.shutdown();
}
}

```

Here you use a `while` loop to continuously check for the completion and, once ready, the data is consumed and exits out of the loop. Note that `isDone` is a non-blocking method so you don't block the thread when you use it. There is also a method called `get` in the `Future` API; it is a blocking method and it blocks the main thread until a response arrives. You can try another variation of `get` that accepts a timeout:

```

String result =future.get();
String result =future.get(10, TimeUnit.SECONDS);

```

Asynchronous Programming with Play

The ability to handle requests asynchronously is a very important factor in choosing a framework for any project that has to scale well. Synchronous processing of a large number of concurrent requests can eat up server resources and, in certain extreme cases, if the capacity planning is not well done, can bring the server to a halt.

Normal page requests from a web server are typically synchronous and many times they are cached copies of the previous rendered page. But this is not the scenario always; there are many applications that do a lot of expensive processing to return results to the client. In such cases, obviously the client has to wait for the result to arrive, but the server can—in fact, does—the processing asynchronously in another thread of execution and frees up the scarce server resources.

Play does this elegantly. A Play action that is non-blocking at the server should return a `CompletionStage<Result>` instead of the normal `Result` object. The `CompletionStage` is a promise that the result will be available after some time. The beauty of the `CompletionStage` interface is that it provides a vast selection of methods to attach callbacks to in order to process the result once it becomes available.

The web client will be blocked while waiting for the response but nothing will be blocked on the server, and server resources can be used to serve other clients.

Writing an Asynchronous App

Let's find a scenario to understand this better. Consider an application that recommends gifts based on a person's Facebook profile. This application uses the data publicly available in the person's profile and then searches online to gather gifts matching their interests. The searching online and composing the gift list are expensive operations that can be very heavy on the server. So this piece of code is a good candidate for asynchronous programming:

```
public CompletionStage<Result> recomendGifts(final String
uid,final String age,final String relation) {
    return CompletableFuture.supplyAsync(this::getGifts)
        .thenApply(( List<GiftVO> gift) -> ok("Got " +
gift));
}
private List<GiftVO> getGifts() {
    List<GiftVO> gifts =new ArrayList<GiftVO>();
    gifts.add(new GiftVO());
    return gifts;
}
```

In this code, the private method `getGifts` deals with finding the gifts and returning the list of gifts. Of course, I have mocked the response for this example. In a real-world scenario, the gifts might be based on a person's social profile or wish list or similar logic. That logic is not in our scope; we are focusing only on asynchronous programming semantics. The public method `recomendGifts` invokes the `getGifts` method asynchronously using the `Completable<Future>` interface that is part of the `java.util.concurrent` package. This is done by the code segment `CompletableFuture.supplyAsync(this::getGifts)`.

The next step is to check whether you have a response and when it is available do further processing on the response and then send it back to the client. That is what the `thenApply` method does.

Configuring Asynchronous Scheduled Jobs

Play uses Akka to work with asynchronous jobs. In order to understand jobs in Play, a basic knowledge of Akka is required.

Akka Basics

The official definition of Akka is that it's "a toolkit and runtime for building **highly concurrent, distributed, and fault tolerant** event-driven applications on the JVM."

Akka introduces the actor model abstraction and provides a better platform to build correct, concurrent, and scalable applications. This means that it solves the hardships of writing multi-threaded, highly concurrent code. Akka uses a message flow model to achieve this.

The actor model is not a new concept. The idea was introduced in 1973 by Carl Hewitt, Peter Bishop, and Richard Steiger.

Traditional concurrent programming in Java involves many threads working together. When they need to work on a shared resource, it is done by acquiring locks on the shared object. The program deals directly with

the low-level tasks of getting locks, releasing them after use, and so on. This kind of code is hard to maintain and is error prone; many times it can lead to deadlocks. Another pain point is in scaling horizontally across JVMs. Akka tries to abstract these low-level programming methodologies using actors, ActorRefer, and ActorSystem. Actors provide the abstraction for transparent distribution and the basis for truly scalable and fault-tolerant applications.

Does this mean there are no threads and locking with Akka? Well, they are there, but you don't directly deal with them. Internally everything runs on lightweight threads and low-level concurrency primitives. Akka uses the `java.util.concurrent` library to handle the coordination.

Akka is written in Scala with language bindings provided for both Scala and Java. Let's first understand what an actor is. An actor is just an object that can receive messages and take an action to handle the messages. It is strictly decoupled from the source that generates the message.

To use the Akka actors, the first thing you need is to start the ActorSystem. The ActorSystem is the fundamental entity for interacting with actors and is responsible for actor life cycle management and is the entry point of an Akka application. When a Play application is started, an ActorSystem becomes available and can be accessed using dependency injection in classes that need to interact with the ActorSystem.

We will only look at how to schedule asynchronous jobs in Play via Akka. We won't be detailing out the internals or the configuration of Akka actor system in Play. This information is readily available from the official Play Framework documentation.

```
import akka.actor.ActorSystem;
import scala.concurrent.ExecutionContext;
import scala.concurrent.duration.Duration;

import javax.inject.Inject;
import java.util.concurrent.TimeUnit;
```

```

public class ScheduledTask {

    private final ActorSystem actorSystem;
    private final ExecutionContext executionContext;

    @Inject
    public ScheduledTask(ActorSystem actorSystem,
        ExecutionContext executionContext) {
        this.actorSystem = actorSystem;
        this.executionContext = executionContext;

        this.initialize();
    }

    private void initialize() {
        this.actorSystem
            .scheduler()
            .scheduleAtFixedRate(
                Duration.create(30 TimeUnit.SECONDS),
                // initialDelay
                Duration.create(1, TimeUnit.MINUTES),
                // interval
                () -> actorSystem.log().info("Time in
                    millis now "+System.currentTimeMillis()),
                this.executionContext);
    }
}

```

This example shows how to schedule a task that is to be run 30 seconds from now and then every minute after. The code uses dependency injection to get a reference to the default ActorSystem and ExecutionContext, and then uses the scheduler method of the ActorSystem to execute the code in the defined intervals.

CHAPTER 6

Web Services, JSON, and XML

Modern applications not only expose web service, they also consume other third-party web services. For example, your application may consume a Facebook or Twitter feed. Most modern web services are exposed using REST APIs that handle JSON payloads. A typical problem with calling web services is that they can block the caller and can result in blocking the server until a response is obtained or a timeout occurs. This is not a particularly good design because it can affect the throughput of the server and can result in missed SLAs. Play 2 has solved this problem by providing asynchronous, non-blocking APIs to invoke long-running tasks like an external web service. Play provides the `play.libs.ws` library to handle asynchronous web service calls. A call made by `play.libs.ws` returns `CompletionStage<WSResponse>`. Later you can extract the response by using callback functions available in the `CompletionStage` interface. Essentially they are asynchronous callback methods. The caller makes a web service call and, once the response is available, the rest of the code runs. You will explore this in more detail later in this chapter.

Note Going forward, I will use the term *promise* quite often. Hence it is important to understand what a promise is and what its intent is.

“A promise, also known as **CompletableFuture**, is design pattern widely used in concurrent, asynchronous programming situations. A promise represents a proxy for some response that is not known when the promise is created. Promises are a way to write asynchronous code so it appears as though executing in a synchronous manner.” The use of a promise is a way to avoid the callback hell found in processing asynchronous functional calls. Whenever I refer to a promise, it indicates the promise design pattern unless explicitly mentioned otherwise.

Consuming Web Services

Let’s learn by example. You are going to build a new method in the controller that calls a web service and returns the response back to the client. This new service just echoes the response from the back-end service.

Route configuration:

```
GET /bookshop/book/echo      controllers.Application.echoService
```

The following is `Application.java`:

```
package controllers;

import javax.inject.Inject;
import play.mvc.*;
import play.data.DynamicForm;
import play.data.FormFactory;
import play.libs.ws.*;
import play.mvc.Result;
import java.util.concurrent.CompletionStage;

public class Application extends Controller {

    @Inject WSClient ws;

    public CompletionStage<Result> echoService() {
```

```

return
  ws.url("http://www.mocky.io/v2/53c7ec8426e0e3fd14326b0d")
  .get()
  .thenApply(response -> ok("Feed Response: " + response.
    getBody()));
}
}

```

The `echoService` uses the `WS` library to invoke an external web service and returns the response. For the demonstration, I have created a simple JSON service using the `mocky.io` website.

Let's examine each of the steps in the `echoService` method.

First, the `WSClient` is injected to the controller using the `@Inject` annotation.

`ws.url().get` returns `CompletionStage<WSResponse>`. This object contains the complete response and provides methods to fetch the response in widely used formats like JSON, XML, etc. Since all invocations and content delivery use asynchronous, non-blocking semantics, the caller needs to use a futures block to retrieve the response when it is ready. `thenApply` is used to do this; whenever the response is available, the code inside `thenApply` will execute.

- Read response body as a string:
`response.getBody()`
- Read as JSON:
`response.asJson()`
- Read as XML:
`thenApply(r -> r.getBody(xml()));`

The `xml` method is available in `play.libs.ws.WSBodyReadables`.

The following are the `WSResponse` API's commonly used methods:

- Get the body as an array of bytes:
`byte[] asByteArray()`
- Get the body as a JSON node:
`com.fasterxml.jackson.databind.JsonNode asJson()`
- Return the body as XML:
`org.w3c.dom.Document asXml()`
- Return the content as raw bytes:
`akka.util.ByteString getBodyAsBytes()`
- Get the HTTP content type of the response:
`java.lang.String getContentType()`
- Return the HTTP status codes (200, 404, 201, 500, ...) of the response:
`int getStatus()`
- Return the textual representation of an HTTP status code (Ok, Not Found, Internal Server error, ...):
`java.lang.String getStatusText()`

The `echoServices` method uses the `get` method that returns a `CompletionStage<WSResponse>` object. When you invoke `get`, what you get is just a `Proxy` object. No actual invocation has occurred. The `Proxy` (`CompletionStage`) merely indicates that the framework has created a job that will be processed asynchronously in a non-blocking manner. When the call is made and a response is obtained, there needs to be some way by which the application code can get an intimation and do further

processing. This is accomplished by providing a callback method. This is what the rest of the code in `echoService` is for. The `thenApply` function is a callback function. Play will invoke this function with an instance of `WSResponse` when the back-end service returns a response.

Processing Large Responses

When receiving large HTTP responses, it is not a good idea to use `get` to load the response into memory. If the response is big and in the order of gigabytes, it can result in memory errors and can potentially crash the application. In such cases, the better choice is to use Akka streaming to consume the response incrementally. For this use case, `WSResponse` provides a method named `getBodyAsSource`:

```
Source<ByteString, ?> responseBody = res.getBodyAsSource();
```

This response body can be processed by chaining it with a sink. Let's see how this is done.

For this example, I have created a mock response using the `mocky.io` online service. This service lets you create mock responses for testing. Go to www.mocky.io/v2/5e08df833000005b0081a159. This URL returns a simple HTML response. Let's use Play and Akka streaming to process this response in an asynchronous, non-blocking, and efficient way. The intended output is to count the length of content in an incremental way and, once all the processing is over, send the response to the client. To use Akka streaming, you need to get a reference to the `Akka ActorSystem` and the Akka streaming materializer classes in the controller class. For the time being, just understand that the materializer is a helper class for Akka streams to work. This is the class that is responsible for materializing the Akka stream processing pipeline. If you are totally new to Akka and asynchronous programming, you can quickly read Chapter 8 of this book covering the basics of Akka and come back to this section. Before you get to the code, here are two terms you need to understand:

- **Source:** The element that produces the data. For example, a source may be created to stream data from a Twitter feed or a source for reading files in a directory or a web service.
- **Sink:** The end receiver of the data. For instance, you may read from a source, do some transformation, and write back to a file. In this case, the save to file is the sink. In Akka streams, you connect a source and sink, and perform transformations and data filtering in between. With this much information in hand, let's get to the code.

You will add a new method in `Application.java` (Controller class), namely `processLargeResponse`. This method reads the data from a rest endpoint using Akka streaming semantics and calculates the size of the data. You won't be reading the entire response and counting; instead, the data is read in chunks and the count is updated.

```
package controllers;

import actors.ActorModel;
import actors.PingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.stream.Materializer;
import akka.stream.javadsl.Sink;
import akka.stream.javadsl.Source;
import akka.util.ByteString;
import play.libs.ws.WSClient;
import play.libs.ws.WSResponse;
import play.mvc.Controller;
import play.mvc.Http;
import play.mvc.Result;
```



```

import scala.compat.java8.FutureConverters;

import javax.inject.Inject;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import static akka.pattern.Patterns.ask;

public class Application extends Controller {

    final ActorRef pingActor;

    @Inject
    public Application(ActorSystem system) {
        pingActor = system.actorOf(PingActor.getProps());
    }
    @Inject
    Materializer materializer;

    public CompletionStage<Result> processLargeResponse() {
        CompletionStage<WSResponse> futureResponse =
            ws.url("http://www.mocky.io/v2/5e08df83300005b0081a159")
                .setMethod("GET").stream();

        CompletionStage<Long> bytesReturned =
            futureResponse.thenCompose(
                res -> {
                    Source<ByteString, ?> responseBody = res.
                        getBodyAsSource();

                    // Count the number of bytes returned
                    Sink<ByteString, CompletionStage<Long>> bytesSum =

```

```

        Sink.fold(0L, (total, bytes) -> total +
            bytes.length());

        return responseBody.runWith(bytesSum, materializer);
    });
    return bytesReturned.thenApply(
        res -> ok((String)res.toString()));
}
}

```

This code calls the web service defined at `mocky.io` and processes the response as an Akka stream. This way it only loads chunks of the response into memory and processes the chunks. This mechanism ensures that you can process any large response without encountering memory and CPU bottlenecks.

To test this, add the following entry into the `routes.conf` file:

```
GET /bookshop/example/largerresponse controllers.Application.
processLargeResponse()
```

Handling JSON

Play has built-in support for JSON. It can automatically parse a JSON request or generate a JSON response. Let's explore the two scenarios:

- 1) Consuming a JSON request
- 2) Producing a JSON response

Consuming JSON Request

Play by default uses a body parser that can parse any valid content type. The default body parser is accessible from the request object and has method `asJson` to process the incoming request as JSON content.

JSON content will have the Content-Type header set as `text/json` or `application/json`.

```
JsonNode json = request().body().asJson();
```

This is, in fact, a Jackson node. Once you get this, you can use the methods available in the Jackson API to extract the data in the JSON request.

You can ask Play to process only the JSON body as a request and reject anything that is not valid with an HTTP 400 response code. For this, just annotate the method with `@BodyParser.Of (BodyParser.Json.class)`. This is the better way to process an incoming JSON request body.

Let's create a method that acknowledges a message it receives. The `acknowledgeGreeting` method in the controller accepts a JSON body and returns an HTML output:

```
package controllers;

import com.fasterxml.jackson.databind.JsonNode;
import play.mvc.Controller;
import play.mvc.Result;
import play.mvc.BodyParser;

public class Application extends Controller {

@BodyParser.Of(BodyParser.Json.class)
public Result acknowledgeGreeting(){

    JsonNode json = request().body().asJson();
    String greeting = json.findPath("greeting").textValue();
    if(greeting == null) {
        return badRequest("Missing parameter [greeting]");
    } else {
        return ok("Your greeting "+greeting+" is accepted" );
    }
}
}
```

Modify the routes file and add the mapping entry:

```
POST /greeting controllers.Application.acknowledgeGreeting()
```

To test the service, you need a client that can post a JSON body. There are many browser-based clients available and you can use the one of your choice. I used a Mozilla Firefox add-on named RestClient (a debugger for RESTful web services); you can also use Postman. If you want to do it via the command line, use curl. See Figure 6-1.

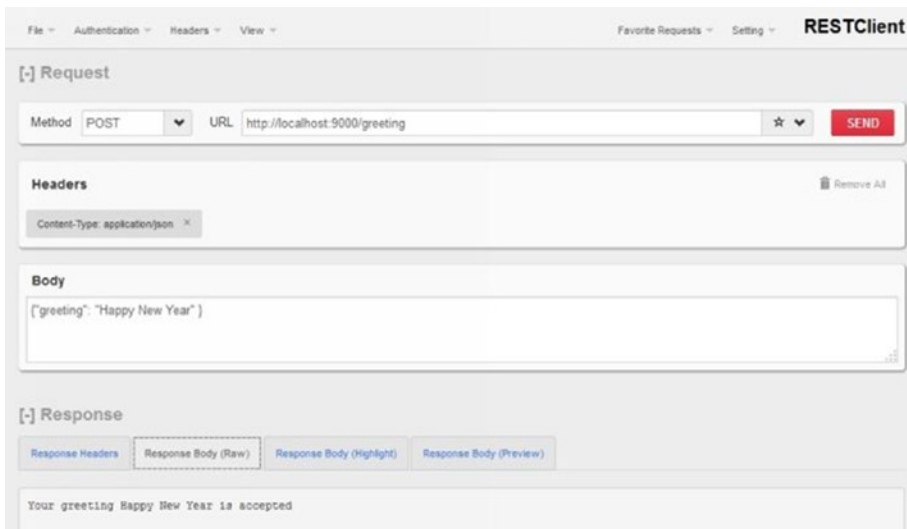


Figure 6-1. Testing with RestClient

Testing the service using curl:

```
$ curl -H "Content-Type: application/json" -X POST -d
{"greeting": "hello"} http://localhost:9000/greeting http://
localhost:9000/greeting
```

Response:

Your greeting hello is accepted

Producing a JSON Response

It is very simple to generate a JSON response with Play. Just create a JSON object and put key-value pairs into it:

```
ObjectNode result = Json.newObject();
result.put("status", "OK");
result.put("message", "Greetings" );
return ok(result);
```

The `acknowledgeGreeting` method you used above accepts a JSON request and it sends a text/plain response back. Let's create another version of the `acknowledgeGreeting` that returns an `application/json` response instead (see Figure 6-2):

@BodyParser.Of(BodyParser.Json.class)

```
public Result acknowledgeGreetingJSON(){
    JsonNode json = request().body().asJson();
    String greeting = json.findPath("greeting").textValue();
    ObjectNode result = Json.newObject();
    if(greeting == null) {
        result.put("status", "BAD_REQ");
        result.put("msg", "Missing parameter [greeting]");
    } else {
        result.put("status", "SUCCESS");
        result.put("msg", "Your greeting "+greeting+" is
        accepted");
    }
    return ok(result);
}
```

Routes entry:

```
POST /greeting/better controllers.Application.
acknowledgeGreetingJSON()
```

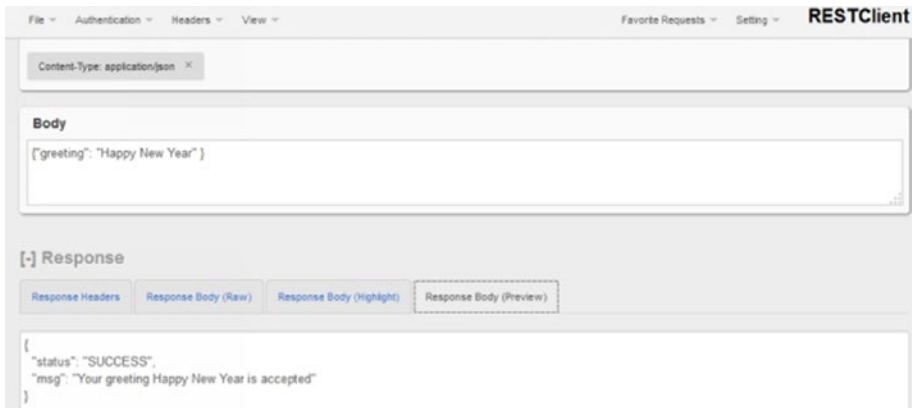


Figure 6-2. Testing the JSON response

If you examine the HTTP headers in the response, you can see that the Content-Type header is set as `application/json`.

Handling XML

Handling XML is very similar to the above case of handling JSON. It is enough to use the default body parser to convert an incoming XML request body to an `Object` for easier processing. The important factor to consider is that the incoming Content-Type header should be either `application/xml` or `text/xml`. By default, raw XML is converted to a valid W3C Document object:

```
Document dom = request().body().asXml();
```

Here the default body parser is used to convert the XML string to a parsable `Object`.

To create an XML response, you should use a valid JAXB implementation or, for very simple XML, you can even just return a `String` in the XML structure. Manually building XML using `String` manipulation is not extensible or maintainable, so use a proper JAXB implementation. In Play Framework, you don't need to add extra libraries because Play already supports JAXB via the JDK. Let's look at both ways. Example 1 shows simple XML parsing and example 2 uses JAXB.

Example 1: Simple XML Parsing

Use the following XML:

```
<message>
  <greeting>Value</greeting>
</message>
```

You will parse this XML and send back a text/plain response.

The method in the controller (`Application.java`):

```
import play.libs.XPath;
import play.mvc.BodyParser;
import org.w3c.dom.Document;

@BodyParser.Of(BodyParser.Xml.class)
public Result acknowledgeGreetingXML() {

    Document dom = request().body().asXml();
    if(dom == null) {
        return badRequest("Requires XML Input");
    } else {
        String greeting = XPath.selectText("//greeting", dom);
        if(greeting == null) {
            return badRequest("Missing parameter
                [greeting]");
        } else {
            return ok("Your greeting "+greeting+" is
                accepted");
        }
    }
}
```

Now the routes:

```
POST /greeting/xml controllers.Application.  
acknowledgeGreetingXML()
```

You can test this using Postman or RestClient or curl.

Testing using Curl:

```
curl -H "Content-Type: application/xml" -X POST -d "<message>  
<greeting>Value</greeting>  
</message>" http://localhost:9000/greeting/xml
```

Response: Your greeting Value is accepted

Instead of using XPath, if required you may directly use JAXB and map the incoming XML to a Java Object. This is what you will explore in example 2.

Example 2: XML Parsing Using JAXB

Let's add a new method in `Application.java` called `acknowledgeGreetingXMLJaxbVersion`. This method uses JAXB to bind the incoming XML to a Java object or the model. For this to work, first you need to create your model representing the XML content. Create new file named `Message.java` inside the `models` package. Add the following content to it and save the file:

Message.java

```
package models;  
  
import javax.xml.bind.annotation.XmlAccessType;  
import javax.xml.bind.annotation.XmlAccessorType;  
import javax.xml.bind.annotation.XmlElement;  
import javax.xml.bind.annotation.XmlRootElement;
```



```

@XmlRootElement(name = "message")
@XmlAccessorType(XmlAccessType.PROPERTY)
public class Message {

    private String greeting;

    public String getGreeting() {
        return greeting;
    }
    @XmlElement
    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}

```

This file represents the XML content. I have annotated it to instruct JAXB that the `<message>` element is the root element and marked the setter method using the `@XmlElement` annotation so that JAXB uses the setter instead of the property directly.

Now let's add the method to `Application.java`:

```

import actors.ActorModel;
import actors.PingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.stream.Materializer;
import akka.stream.javadsl.Sink;
import akka.stream.javadsl.Source;
import akka.util.ByteString;
import models.GiftVO;
import models.Message;
import modules.Factorial;
import org.w3c.dom.Document;

```

```
import play.data.DynamicForm;
import play.data.FormFactory;
import play.libs.XPath;
import play.libs.ws.WSClient;
import play.libs.ws.WSResponse;
import play.mvc.Controller;
import play.mvc.Http;
import play.mvc.Result;
import scala.compat.java8.FutureConverters;

import javax.inject.Inject;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import com.fasterxml.jackson.databind.JsonNode;
import play.mvc.BodyParser;
import static akka.pattern.Patterns.ask;
```

```

public class Application extends Controller {
    @BodyParser.Of(BodyParser.Xml.class)
    public Result acknowledgeGreetingXMLJaxbVersion() throws Exception {
        Document doc = request().body().asXml();
        if(doc == null) {
            return badRequest("Requires XML Input");
        }else {
            TransformerFactory tf = TransformerFactory.newInstance();
            Transformer transformer = tf.newTransformer();
            transformer.setOutputProperty(OutputKeys.OMIT_XML_
DECLARATION, "yes");
            StringWriter writer = new StringWriter();
            transformer.transform(new DOMSource(doc), new
            StreamResult(writer));
            String output = writer.getBuffer().toString();
            JAXBContext context = JAXBContext.newInstance
            (Message.class);
            Unmarshaller unMarshaller = context.createUnmarshaller();
            //JAXB Auto conversion- XML to Model
            Message msg = (Message)unMarshaller.unmarshal(new
            StringReader(output));
            if(msg == null) {
                return badRequest("Requires XML Input");
            } else {
                String greeting = msg.getGreeting();
                if(greeting == null) {
                    return badRequest("Missing parameter [greeting]");
                } else {
                    return ok("Your greeting "+greeting+" is
                    accepted");
                }
            }
        }
    }
}

```

```
        }  
    }  
}  
}
```

In the `acknowledgeGreetingXMLJaxbVersion` function, you take the XML document object, convert it to the string representation, and use the `Unmarshaller` class of JAXB to autoconvert the XML string to the Model object, `Message`. This is the efficient and simple way to convert XML to Model.

CHAPTER 7

Accessing Databases

Play 2 has excellent support for SQL databases. All the database-related configurations are maintained in the `conf/application.conf` file. By default, Play 2 provides support for JDBC connection pools (connection pools offer efficient and faster access to the database resources).

Configuring Database Support

Edit the `build.sbt` file and add

```
libraryDependencies += javaJdbc
```

This will enable the JDBC API for Play projects.

Now open the `application.conf` file and add the following content:

```
# Database configuration
# ~~~~~
# You can declare as many datasources as you want.
# By convention, the default datasource is named `default`
#
# db.default.driver=org.h2.Driver
# db.default.url="jdbc:h2:mem:play"
# db.default.user=sa
# db.default.password=""
```

You can start using the default data source (default) by configuring the database properties and uncommenting the configuration settings. The following are the settings to configure a MySQL database as the default datasource:

```
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://IP Address:port/books"
db.default.user=username
db.default.password="PASSWORD"
db.default.logStatements=true
```

Make sure that you have the appropriate MySQL connector jar (mysql-connector-java-5.1.24-bin.jar) in the application classpath. An easy way to do this is to create a folder named lib in the project and copy the connector jar there. Play will add any jar file put inside the lib folder to the classpath.

You can either work with the default data source or create any number of your own data sources. To create another data source, copy the properties defined for the default data source and change the name default to one of your choice. For instance,

```
db.book.driver=com.mysql.jdbc.Driver
db.book.url="jdbc:mysql://localhost:3306/book"
db.book.user=username
db.book.password="password"
db.book.logStatements=true
```

This datasource is referred to by the name book, so by providing this name a connection can be obtained from the pool.

Directly invoking JDBC calls is costly because they are blocking operations and cause thread waiting. Hence it is a best practice to run the JDBC calls in a separate execution context by providing a CustomExecutionContext. Here's an example that executes JDBC calls in a Play application:

```

package dao;

import akka.actor.ActorSystem;
import play.db.Database;
import play.libs.concurrent.CustomExecutionContext;
import javax.inject.Inject;
import javax.inject.Singleton;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

public class DatabaseExecutionContext extends
CustomExecutionContext {

    @javax.inject.Inject
    public DatabaseExecutionContext(ActorSystem actorSystem) {
        // uses a custom thread pool defined in
        application.conf
        super(actorSystem, "database.dispatcher");
    }

    @Singleton
    public static class JdbcExample {

        private Database db;
        private DatabaseExecutionContext executionContext;

        @Inject
        public JdbcExample(Database db,
            DatabaseExecutionContext context) {
            this.db = db;
            this.executionContext = context;
        }

        public CompletionStage<Integer> updateSomething() {
            return CompletableFuture.supplyAsync(
                () -> {

```

```

        return db.withConnection(
            connection -> {
                //perform the
                operations with
                //the connection
                return 1;
            });
    },executionContext);
}
}
}

```

You can see that this code uses `ActorSystem` as a constructor argument. `ActorSystem` is part of the Akka framework, and Play relies on Akka to implement the execution context (part of `play.libs.concurrent.CustomExecutionContext` class used above). A chapter on Akka is provided in this book and I recommend reading it if you are new to Akka.

Typically you don't deal directly with connection-related activities. It is a best practice to delegate the database activities to an ORM (object relational mapping) layer. It is recommended to use ORM for database access because it shields the programs from the complexities of persisting the object and fetching the object to and from the underlying data store.

ORM brings a rich object-oriented flavor to the code by enabling the programmer to concentrate on the business rather than writing code for database access and related connection management. An ORM layer is expected to generate efficient SQL statements given a situation and is expected to shield the code from changes to the data store or its structure. An ORM layer can also provide some sort of cache such that the actual database interaction happens only when absolutely necessary. Ebean, the ORM that comes bundled with Play, is an excellent choice considering the requirements stated above. You'll explore the Ebean ORM in the following sections.

Working with an ORM

Ebean is a lightweight, open source ORM and has a developer-friendly API. Play supports integration with JPA and can be easily configured to use Hibernate, iBatis or any other popular ORM. Let's see how to configure Hibernate as the JPA provider for Play 2 and then explore Ebean in detail.

In order to understand Ebean and any other ORM, it is important that you are well aware of the basic ORM concepts. A quick introduction to ORM is given in the next section. If you are already familiar with ORM concepts, you may skip this section.

ORM Concepts

Before the emergence of ORM, the only option available was JDBC. The JDBC API allows all kinds of database operations from Java. It supports updates, selects, database- and table-level metadata queries, and much more. Gradually the Java community realized the difficulties with this approach. The difficulties are

- Mismatch in representing the object models and relational database models. RDMS represent data in a tabular format whereas object-oriented languages like Java represent it as an interconnected graph of objects.
- State management: Synchronizing the state of the in-memory object model with a relational database
- Difficulties in modeling using pure object-oriented domain models because there was no elegant way to sync the model with the relational database
- Duplicate and repetitive SQL code
- Inefficient data traversal
- Ties the code to a particular database and SQL syntax

- The low-level SQL code gets embedded in the business logic
- Difficult to cache the data

An ORM framework tries to address these mismatches and acts as the middleman in bridging these mismatches. The most important features required from an ORM are the first three points; the rest are added advantages.

- No mismatch between the database model and the object model. ORM fixes this mismatch.
- ORM helps in state management by synchronizing the state of the in-memory object model with a relational database.
- ORM helps in modeling by using pure object-oriented domain models and syncing the model with the database.

As per my experience in building many systems over the years, if you are designing your project such that it uses an object-oriented domain model and the data store is relational, there is no other better choice than an ORM.

If you are not particular about object-oriented design and the domain model, there is no need for you to go for ORM. You may choose any other technology that helps in persisting data to a database from Java.

If you are always thinking about tables and rows and not thinking about domain objects, don't waste your time with ORM. It's better to use some other technology that can help in persisting data to RDMS from Java.

But if you are not using ORM, you are missing a lot and you will end up writing a lot of plumbing code. Hence even for smaller projects, go ahead with ORM and model your business objects appropriately. There is no reason for not using ORM because almost all ORM implementations have matured and can offer wider choices in data access and updates.

Key Terms

The possible relationships between entities in an object-oriented domain model are

- One to one
- One to many
- Many to one
- Many to many

One to Many

The relationship between an `Order` and the `LineItems`. An order may have many line items. So this is an example of one to many. If the `LineItem` doesn't have a property referring to the `Order`, it is a unidirectional association.

Many to One

The relationship from the perspective of `LineItem` to `Order` is an example of many to one.

Many to Many

This can be explained easily using the association between students and courses. A student can enroll in multiple courses, and a course can be attended by many students. Hence this is an example of many to many.

Relationship Direction

These relations can be unidirectional or bidirectional. Unidirectional means you can access the child only from the parent. Bidirectional means the parent and child can be traversed from both ends. That is, each entity has a relationship field that refers to the other entity.

A bidirectional relationship has an owning side and an inverse side whereas a unidirectional relation has only the owning side.

The **owning side of a relationship determines the updates to the relationship in the database**. An easier way to understand this is to consider that the entity that contains the foreign key of the table is the owner and is responsible for maintaining the relationship in the case of a bidirectional relationship.

For instance, `Order` has foreign keys to the line items so it is the owner of the relationship.

A bidirectional relationship should follow these rules:

- The inverse side of a bidirectional relationship must refer to its owning side by using the `mappedBy` element of the `@OneToOne`, `@OneToMany`, or `@ManyToMany` annotation. The `mappedBy` element designates the property or field in the entity that is the owner of the relationship.
- The many side of many-to-one bidirectional relationships must not define the `mappedBy` element. The many side is always the owning side of the relationship.
- For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.
- For many-to-many bidirectional relationships, either side may be the owning side.

The entity that contains the foreign key is the owner, and the child entities are the inverse side. The child entity should contain the `mappedBy` annotation. When entities are linked by relationships, it is also important to specify the appropriate cascade settings. For example, a line item is part of an order; if the order is deleted, the line item also should be deleted. This is called a cascade-delete relationship. I hope you have now attained a quick knowledge of working with ORM.

Configuring JPA

The Java Persistence API (JPA) mandates that the data source should be accessible via JNDI. The first thing you need to do is to make the data source accessible via JNDI by changing the data source configuration in the `conf/application.conf` file. A data source can be made available via JNDI by specifying the JNDI name in the configuration.

You are going to use the h2 in-memory database for the examples. The goal is to save a user's review comment on a book and persist it using JPA.

Open the `application.conf` file and add the following entries:

```
# This is the main configuration file for the application.
# https://www.playframework.com/documentation/latest/ConfigFile
# Default database configuration
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"

db.default.jndiName=DefaultDS
jpa.default=defaultPersistenceUnit
```

The next step is to choose a JPA implementation and add that as a dependency to the project. You are using Hibernate as the JPA provider and the in-memory h2 database for the examples. Open the `build.sbt` file and add

```
val appDependencies = Seq(
  javaJdbc,
  javaJpa,
  "org.hibernate" % "hibernate-entitymanager" % "5.3.7.Final",
  "com.h2database" % "h2" % "1.4.200"
)
```

`build.sbt` is located inside the project directory.

The next step is to create the `persistence.xml` file and configure it to use Hibernate as JPA provider. `persistence.xml` should be put inside the `conf/META-INF` directory.

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
version="2.1">

<persistence-unit name="defaultPersistenceUnit" transaction-
type="RESOURCE_LOCAL">
<provider>org.hibernate.jpa.HibernatePersistenceProvider</
provider>
<non-jta-data-source>DefaultDS</non-jta-data-source>
<properties>
<property name="hibernate.dialect" value="org.hibernate.
dialect.H2Dialect"/>
</properties>
</persistence-unit>
</persistence>
```

Tip It is a best practice to isolate all JPA operations in a separate execution context other than the default Play execution context. All DB operations are blocking, so it is important to isolate them in a separate execution context.

Play provides the `play.db.jpa.JPAApi` to work with Entity Manager. If you are not familiar with Entity Manager, read the next paragraph.

Entity Manager is part of JPA and it is used to interact with the persistence context to maintain the conversational state. In short, it is the interface we will use to create, remove, update, or find entities when using JPA. It is closely related to the session concept found in Hibernate.

If you want to know more about the latest things in JPA, you can download the specification at https://download.oracle.com/otn-pub/jcp/persistence-2_1-fr-eval-spec/JavaPersistence.pdf.

Your model class should not directly perform JPA or JDBC operations. Let's go ahead and create the repository classes. Before that, you need to create an entity named `Review` that models the real-world review comment by a user regarding a book.

package model;

```
import javax.persistence.*;
public class Review {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public Long id;
    public String comment;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getComment() {
        return comment;
    }
    public void setComment(String comment) {
        this.comment = comment;
    }
}
```

```

public String getUserId() {
    return userId;
}
public void setUserId(String userId) {
    this.userId = userId;
}
public String getBookId() {
    return bookId;
}
public void setBookId(String bookId) {
    this.bookId = bookId;
}
public String userId;
public String bookId;
}

```

Next, create an interface for abstracting the review operations (`ReviewRepository.java`):

```

:package dao;
import com.google.inject.ImplementedBy;
import model.Review;

import java.util.concurrent.CompletionStage;
import java.util.stream.Stream;

@ImplementedBy(JPAReviewRepository.class)
public interface ReviewRepository {

    CompletionStage<String> saveReview(Review review);
}

```

The `ReviewRepository` interface hides the JPA implementation details from the model classes.

Go ahead and create the implementation of this interface (JPAREviewRepository.java):

```

package dao;

import models.Review;
import play.db.jpa.JPAApi;

import javax.inject.Inject;
import javax.inject.Singleton;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import static java.util.concurrent.CompletableFuture.supplyAsync;

@Singleton
public class JPAREviewRepository implements ReviewRepository{

    private JPAApi jpaApi;
    private DatabaseExecutionContext executionContext;

    @Inject
    public JPAREviewRepository(JPAApi api,
DatabaseExecutionContext executionContext) {
        this.jpaApi = api;
        this.executionContext = executionContext;
    }

    @Override
    public CompletionStage<String> saveReview(Review review) {
        return CompletableFuture.supplyAsync(
            () -> {
                // lambda is an instance of
                //Function<EntityManager, Long>
                return jpaApi.withTransaction(
                    entityManager -> {

```

```

        entityManager.persist(review);
        return "saved";
    });
},
    executionContext);
}
}

```

JPA calls should be made inside a transaction, so you use the `jpaApi.withTransaction` method to execute the persistence operation of saving the `Review` entity.

Using Ebean in Play

My experience in using Ebean with Play is quite satisfactory. It offers easy to use apis and is much simpler in terms of architecture.

An architecture using `EntityManager` brings with it the concept of attached and detached entities. The entity state is persisted only when it is attached with an `EntityManager`. This also means if the conversation needs to be maintained across multiple transactions, the `EntityManager` needs to be managed by another object like an EJB Session bean. The same Entity Manager instance is used across transactions to maintain a single conversation. Ebean greatly simplifies this by following an `EntityManager` architecture. This means Ebean doesn't provide an Entity Manager nor does it provide the *merge*, *flush*, or *persist* operations normally found in JPA.

The point to understand here is that Ebean uses a totally different architecture compared to JPA. Please note that even though Ebean doesn't have an Entity Manager, it does have a "persistence context" that is transaction scoped and managed automatically.

I have worked with both JPA and Ebean, and in my experience both have merits. There is no doubt that the query language used by Ebean is much simpler to understand and work with. This simplicity is the selling point of Ebean.

Another important optimization employed in Ebean is its support for the partial object. In JPA, you would typically annotate a relationship to instruct whether the relationship needs to be loaded using an eager or lazy fetching strategy. In an eager fetch, the child objects are also retrieved and loaded when the parent is fetched. In a lazy loading strategy, only the parent gets loaded and the child is fetched only when it is used. The ability to configure these fetch strategies is very important and is a big factor in optimizing the code. JPA provides an annotation to configure them at the entity level. But there is a problem.

Consider a scenario in which for use case 1 you want the entity to adopt eager the fetch strategy and for use case 2 you want the same entity to use the lazy loading strategy. This is a problem because these annotations are declared at the entity level. This is highly inflexible. Most ORMs provide functionality similar to that of partial objects via “fetch groups.”

Having “partial object” support in the query language means you can write a query that is optimal for different use cases. That is, the lazy or eager loading of the dependent objects are not fixed but can be changed depending on the use case. This is a nice feature employed in Ebean and a big boost to optimizing queries for performance and efficiency. This feature was in the earlier JPA specifications.

In certain use cases, you may find that writing custom SQL will be much better and easier from a performance standpoint. Ebean has excellent support for custom SQL queries and provides a simpler API to work with.

To enable Ebean, add the plugin dependency to the project/plugins.sbt file:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-play-ebean" % "5.0.2")
```

Then modify the build.sbt file to add the Ebean dependency:

```
lazy val root = (project in file("."))
  .enablePlugins(PlayJava, PlayEbean)
```

Ebean Query

Ebean uses the same mapping as per the JPA specification. So you annotate your beans with `@Entity`, `@Table`, `@Column`, `@OneToMany`, etc. as per the JPA specification. In fact, most of the JPA annotations work with Ebean.

Let's consider a simple case of modeling a book as an entity. A book is unique with its ISBN (for this example, you denote the ISBN as `asin`, an Amazon notation). `ASIN` means a unique identifier for a book and no two books can have the same `ASIN`.

An Ebean model should extend from `io.ebean.Model`:

```
package models;

import java.util.*;
import javax.persistence.*;
import io.ebean.*;
import play.data.format.*;
import play.data.validation.*;

@Entity
@Table(name="book")

public class Book extends Model {
  @Id
  @Column(name="asin")
```

```
private String asin;
@Column(name="title")
private String title;
@Column(name="author")
private String author;
@Column(name="description")
private String description;
@Column(name="language")
private String language;
public String getLanguage() {
    return language;
}
public void setLanguage(String language) {
    this.language = language;
}
public String getDescription() {
    return description;
}
public void setDescription(String description) {
    this.description = description;
}
public String getAsin() {
    return asin;
}
public void setAsin(String asin) {
    this.asin = asin;
}
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
}
```

```
public String getAuthor() {
    return author;
}
public void setAuthor(String author) {
    this.author = author;
}
@Override
public int hashCode() {
    final int prime = 31;
    int result = super.hashCode();
    result = prime * result + ((asin == null) ? 0 : asin.
hashCode());
    return result;
}
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (!super.equals(obj))
        return false;
    if (getClass() != obj.getClass())
        return false;
    Book other = (Book) obj;
    if (asin == null) {
        if (other.asin != null)
            return false;
    } else if (!asin.equals(other.asin))
        return false;
    return true;
}
}
```

Note Play will automatically generate getter and setter methods during runtime, so unless you want them available during compilation time you don't need to generate them.

There's a lot of information in the annotations in the above code. The most important parts are

- `@Entity` informs the ORM (Ebean) that this class is associated with data stored in the underlying data store and asks the ORM to handle it.
- `@Table` informs the ORM provider that this entity is to be persisted in the table mentioned via the annotation, and in your case the table is `book`.
- `@Id` is very important information and declares that this field is the unique identifier to reference this entity. This typically means this field is the primary key in the underlying database.
- `@Column` maps the attribute to a particular column of the underlying table.

How to create a new book and persist to database:

```
Book book= new Book();
//set the attributes
book.setId("A0091234");
book.setTitle("Play 2");
book.setAuthor("Prem");
book.setDescription("Play 2");
book.setLanguage("English");
book.save();
```

Here you use the `save` method of the `Ebean` class to persist the book object.

How to fetch a book by its ID:

```
// find a Book by the id
Book book = Ebean.find(Book.class).where().idEq("id").
findOne();
```

How to fetch all books starting with Java in the title:

```
// find a list of books with title starting with "java"
List<Book> books =Ebean.find(Book.class).where().
like("title","java%").orderBy("tile desc").findList();
```

Let's explore the `find` query with the `where` and `like` clauses in detail. The above example uses a fluid API style where the methods are chained together. This kind of method chaining is one of the traits of good functional programming style and DSL.

You can optionally use `select()` and `fetch()` to specify only the properties you want to fetch. This returns "partially" populated beans. This is an important feature to help with the performance of your queries. For instance, if you are only interested in two columns, say `id` and `name`, and if they are already indexed by the database, the database is not required touch the data blocks; it can just serve the values from the index itself. This is a big performance improvement if the table has a large number of records.

```
List<Book> books = Ebean.find(Book.class).where("asin,author").
findList();
```

In this example, you only retrieve the `asin` and `author` properties.

Common Select Query Constructs in Ebean

Let us now explore the most commonly used methods available in Ebean that helps to frame Select Queries.

where()

The where clause is used to specify the where filtering condition. You can pass the complete where clause as a string (`where(" filter condition ")`) or use it with Expression objects. Using `where()` with an expression list is better from a programming standards perspective. The example listed above on “fetching books starting with Java in the title” shows the where clause used with Expression objects.

eq()

Checks whether the given property is equal to the supplied value:

```
eq(String propertyName, Object value)
```

Fetch all books written by Robin Cook:

```
List<Book> booksList = Ebean.find(Book.class).where().  
eq("author", "Robin Cook").findList();
```

like()

Specifies a property-like value where the value contains the SQL wild card characters % (percentage) and _ (underscore):

```
like(String propertyName,String value)  
List<Book> booksList = Ebean.find(Book.class).where().  
like("author", "%Cook").findList();
```

orderBy()

Sets the order by clause, replacing the existing order by clause if there is one:

```
orderBy(String orderByClause)
```

This follows the SQL syntax of using commas between each property, with the optional `asc` and `desc` keywords representing ascending and descending order, respectively:

```
List<Book> booksList = Ebean.find(Book.class).where().
eq("author", "Robin Cook")
.orderBy("title desc").findList();
```

findUnique()

Used to return only one record. Executes the query returning either a single bean or null (if no matching bean is found).

findList()

Executes the query, returning the list of objects.

LIMIT {max rows} [OFFSET {first row}]

Used to limit the number of records returned. By using it with the `offset` parameter, pagination can be easily implemented.

The Query Interface

There are a lot of methods in the Ebean Query interface. Please take a look at `com.avaje.ebean.Query` interface to explore the available options. Here is a list of a few of them:

- `Query<T> fetch(String path)`
Specifies a path to load, including all its properties
- `Query<T> fetch(String path, FetchConfig joinConfig)`
Additionally specifies a `JoinConfig` to specify a “query join” and or define the lazy loading query

- `Query<T> fetch(String path, String fetchProperties)`
Specifies a path to fetch with its specific properties to include (a.k.a. partial object)
- `Query<T> fetch(String assocProperty, String fetchProperties, FetchConfig fetchConfig)`
Additionally specifies a `FetchConfig` to use a separate query or lazy loading to load this path
- **`List<Object> findIds()`**
Executes the query returning the list of Ids
- **`List<T> findList()`**
Executes the query returning the list of objects
- **`Map<?,T> findMap()`**
Executes the query returning a map of the objects
- **`int findRowCount()`**
Returns the count of entities this query should return
- **`Set<T> findSet()`**
Executes the query returning the set of objects
- **`intger FirstRow()`**
Returns the first row value
- **`String getGeneratedSql()`**
Returns the SQL that was generated for executing this query
- **`intger MaxRows()`**
Returns the max rows for this query
- `RawSql getRawSql()`
Returns the `RawSql` that was set to use for this query

- `Query<T> select(String fetchProperties)`
Explicitly sets a comma-delimited list of the properties to fetch on the 'main' entity bean (aka partial object)
- `Query<T> setReadOnly(boolean readOnly)`
Set to true when you want the returned beans to be read only.
- `Query<T> setTimeout(int secs)`
Sets a timeout on this query

Using RawSql

You have seen that you need to have an entity for the typical Ebean queries to work. The query is on the entity and that in turn gets translated to a database query. In many situations, you may want to fetch results from many entities and the returned data cannot be translated to an entity. This is useful for reporting type requirements where you want to use aggregate functions such as `sum()`, `count()`, `max()`, etc. For whatever reason, when you want to write database-specific queries, you can use `RawSql`. The returned data from the query can be mapped to an object easily with Ebean thanks to its `@Sql` annotation. You can create an entity and mark it with a `@Sql` annotation to tell Ebean that this is just a SQL representation and not an entity that is related to a table.

Let's see an example. Say you want to find the total number of books written by all authors. You use the `RawSqlBuilder` class to specify the query and map the columns in the select clause to the corresponding property fields defined in the `PopularBook` class. This class is the entity you defined to map the result:

```
String sql="select author,count(author) as noofbooks from books
group by author"
RawSqlBuilder rawSqlBldr = RawSqlBuilder.parse(sql);
RawSql rawSql = rawSqlBldr.columnMapping("author", "bookauthor")
.columnMapping("noofbooks", "totalbooksbyauthor").create();
```

```
Query<PopularBook> query = Ebean.find(PopularBook.class);
query.setRawSql(rawSql);
List<PopularBook> list = query.findList();
```

Here is the PopularBook entity:

```
package models;
import javax.persistence.Entity;
import com.avaje.ebean.annotation.Sql;
@Entity
@Sql
public class PopularBook extends TrendingBook {
private String author;
private int totalbooksbyauthor;
//define the getters and setters
}
```

At times you may desire to use only the relational features and don't want any ORM features for whatever reason. For such cases, Ebean provides the `SqlQuery` class. Here you can write the plain old SQL and get the results just like you do via a JDBC resultset. `SqlQuery` is where you specify the exact SQL SELECT statement and return list, sets, or maps of `SqlRow` objects. A `SqlRow` is a `Map` where the key is the column name. This is a fairly lightweight API that you can use instead of going to raw JDBC. Again, let's see an example to understand this:

```
String sql = "select count(1) as count from book where author = ?";
SqlQuery query = Ebean.createSqlQuery(sql);
query.setParameter(1, author);
SqlRow row = query.findUnique();
int count = row.getInteger("count");
```

This is much easier than writing a plain old JDBC query using the JDBC API.

If the situation demands **writing raw SQL for an insert or update**, you can use the `SqlUpdate` class:

```
SqlUpdate query = Ebean.createSqlUpdate(sql);
```

Ebean also provides the `CallableSql` class to invoke database procedures and functions.

Relationships in Ebean

A one-to-many example:

```
@Entity
@Table(name="book")
public class Book implements Serializable {
    // unidirectional ...
    // ... can explicitly specify the join column if needed
    @OneToMany
    @JoinColumn(name="auth_id")
    List<Author> authors;
```

Consider the above case where you can find the author from the `Book` entity and you cannot traverse back to the book from the `Author` entity. Here the foreign key is in the book table and you can use the `@JoinColumn` to specify the table column that needs to be used for the key. If this was modeled as a bidirectional relationship, the code would be as follows:

```
@Entity
@Table(name="author")
public class Author implements Serializable {
    @OneToMany(mappedBy="authorId")
    List<Book> writtenBooks;
```

Notice that in this bidirectional model, the `Book` entity has the foreign key and is the owner of the relationship. So you mark the inverse side: the `Author` with the `mappedBy` attribute.

If you inspect the RDBMS, you can easily understand that there is no many-to-many relationship in RDBMS tables. The many-to-many is enforced as two one-to-many relationships in database tables via an intersection table. If you anticipate that the intersection table might need to support more attributes, it is better to model it as a separate entity and, in your model, include two one-to-many relationships. If there is no chance for any extra attributes in the intersection table, you can just use the default `@ManyToMany` annotation. Here's an example.

Consider the case of a user and role. A user can have many roles, and a role may be attributed to many users. In RDBMS, this is realized using an intersection table. Let's name the intersection table `user_role`. Let's assume the simple case that the intersection table won't contain any other attributes and in this case the entities will have the `ManyToMany` relationships as shown in the code:

```
@Entity
@Table(name="user")
public class User implements Serializable {
    ...
    @ManyToMany(cascade=CascadeType.ALL)
    List<Role> roles;
}

@Entity
@Table(name="role")
public class Role {
    ...
    @ManyToMany(cascade=CascadeType.ALL)
    List<User> users;
}
```

There so many more things to explain about Ebean as it is vast and may require a whole new book to explain it fully. I recommend you to go through the official Ebean online documentation to understand more. The docs are available at <https://ebean.io/docs/>.

CHAPTER 8

Complete Example

This chapter consolidates what you have learned so far into a complete example. The primary focus of this chapter is to present the code; we have already discussed all the concepts and portions of the code in prior chapters.

conf/application.conf

```
# This is the main configuration file for the application.
# https://www.playframework.com/documentation/latest/ConfigFile
# Default database configuration

play.modules.enabled += "modules.FirstModule"
play.http.filters=config.FilterConfig
play.http.errorHandler = "config.CustomErrorHandler"

db {
  default.driver = org.h2.Driver
  default.url = "jdbc:h2:mem:play"

  # Provided for JPA access
  default.jndiName=DefaultDS
}

# Point JPA at our database configuration
jpa.default=defaultPersistenceUnit
```


CHAPTER 8 COMPLETE EXAMPLE

```
# Number of database connections
# See https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing
# db connections = ((physical_core_count * 2) + effective_spindle_count)
fixedConnectionPool = 9

# Set Hikari to fixed size
play.db {
  prototype {
    hikaricp.minimumIdle = ${fixedConnectionPool}
    hikaricp.maximumPoolSize = ${fixedConnectionPool}
  }
}

# Job queue sized to HikariCP connection pool
database.dispatcher {
  executor = "thread-pool-executor"
  throughput = 1
  thread-pool-executor {
    fixed-pool-size = ${fixedConnectionPool}
  }
}
```

Here you configure a connection pool and a fixed thread pool for the database connections. This is very important because you are running the database invocations in a separate context with its own thread pool. This will make sure that you don't block default execution context threads.

Then you configure the h2 in-memory database as your storage and registered it with JNDI name of DefaultDS.

conf/routes

```
# Routes
```

```

# This file defines all application routes (Higher priority
routes first)
# ~~~~

# An example controller showing a sample home page
GET /bookshop controllers.Application.index
GET /bookshop/book/:id/ controllers.Application.getBook(id:String)
GET /bookshop/book/search/ controllers.Application.search
                                ByTitle(keyword:String)
GET /bookshop/book/echo controllers.Application.
                                echoService
GET /bookshop/book/:id/page/$page<[0-9]+>/ controllers.
Application.fetchBookpage(id:String, page:Integer)
GET /bookshop/example/largeresponse controllers.Application.
                                processLargeResponse()
GET /bookshop/authors/ controllers.Application.authors
                                (limit: Integer = 10)
GET /bookshop/showcomment/ controllers.Application.showComment
                                (userid != null)
GET /bookshop/contact controllers.Application.contact()
GET /bookshop/book/top controllers.Application.topThreeBooks()
POST /bookshop/book/save/comment/ controllers.Application.
                                saveComment(request:Request)
POST /greeting controllers.Application.acknowledgeGreeting
(request:Request)
POST /greeting/xml controllers.Application.acknowledgeGreeting
XML(request:Request)
POST /greeting/xml/jaxb controllers.Application.acknowledgeGr
eetingXMLJaxbVersion(request:Request)
# Map static resources from the /public folder to the /assets URL path
GET /assets/*file controllers.Assets.versioned
                                (path="/public", file: Asset)

```

Application.java

```
package controllers;

import actors.ActorModel;
import actors.PingActor;
import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.stream.Materializer;
import akka.stream.javadsl.Sink;
import akka.stream.javadsl.Source;
import akka.util.ByteString;
import config.EchoAction;
import dao.ReviewRepository;
import models.GiftVO;
import models.Message;
import models.Review;
import modules.Factorial;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.w3c.dom.Document;
import play.cache.AsyncCacheApi;
import play.cache.Cached;
import play.data.DynamicForm;
import play.data.FormFactory;
import play.libs.Akka;
import play.libs.XPath;
import play.libs.ws.WSClient;
import play.libs.ws.WSResponse;
import play.mvc.*;
import scala.compat.java8.FutureConverters;
```

```

import javax.inject.Inject;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringReader;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import com.fasterxml.jackson.databind.JsonNode;

import static akka.pattern.Patterns.ask;

/**
 * Main controller
 */
public class Application extends Controller {

    final ActorRef pingActor;
    private static final Logger log = LoggerFactory.getLogger(
        (Application.class));

    private AsyncCacheApi cache;
    private ReviewRepository reviewRepo;

    @Inject
    public Application(ActorSystem system, AsyncCacheApi cache,
        ReviewRepository reviewRepo) {

```

CHAPTER 8 COMPLETE EXAMPLE

```
        pingActor = system.actorOf(PingActor.getProps());
        this.cache = cache;
        this.reviewRepo = reviewRepo;
    }

@Inject
Materializer materializer;

/**
 * Simple ping method, demonstrating the use of Futures and
 * asynchronous processing
 *
 * @param msg
 * @return
 */
public CompletionStage<Result> ping(String msg) {
    return FutureConverters.toJava(
        ask(pingActor, new ActorModel.Ping(msg), 1000)
        ).thenApply(response -> ok((String) response));
}

/**
 * Method demonstrating processing large http responses in chunks
 *
 * @return
 */
public CompletionStage<Result> processLargeResponse() {
    CompletionStage<WSResponse> futureResponse =
        ws.url("http://www.mocky.io/
        v2/5e08df833000005b0081a159")
```

```

        .setMethod("GET").stream());
CompletionStage<Long> bytesReturned =
    futureResponse.thenCompose(
        res -> {
            Source<ByteString, ?> responseBody
            = res.getBodyAsSource();

            // Count the number of bytes returned
            Sink<ByteString,
            CompletionStage<Long>> bytesSum =
                Sink.fold(0L, (total, bytes)
                    -> total + bytes.length());

            return responseBody.runWith
                (bytesSum, materializer);
        });
return bytesReturned.thenApply(response -> ok((String)
response.toString()));
}

/**
 * Process the home page
 *
 * @return
 */
public Result index() {
    System.out.println(fact.fact(10));
    return ok(views.html.bookshop.render());
}

/**
 * Get the details of a book by id

```

CHAPTER 8 COMPLETE EXAMPLE

```
*
* @param id
* @return
*/
public Result getBook(String id) {
    return ok(views.html.bookshop.render());
}

@Inject
FormFactory formFactory;
@Inject
Factorial fact;

/**
 * Accept a form post and save the comment
 * Shows the usage of dynamic forms to retrieve data from
 * html form posts
 *
 * @return
 */
public Result saveComment(Http.Request request) {
    DynamicForm requestData = formFactory.form().
        bindFromRequest(request);
    String comment = requestData.get("comment");
    Review review = new Review();
    // We hardcoded the ids, but you can take it as
    // exercise to make book id and user id as request
    // parameters.
    // Also modify repository to save those two against comment
    review.setBookId("123456789");
    review.setUserId("U1");
}
```

```

        review.setComment(comment);
        review.save(reviewRepo);
        return ok(views.html.savecomment.render());
    }

    /**
     * Search a book by title.
     * Demonstrating the various route configuration semantics
     *
     * @param title
     * @return
     */
    public Result searchByTitle(String title) {

        //Query db and get the book details or get from cache
        return ok(views.html.searchresults.render());
    }

    /**
     * Demonstrates ActionComposition
     */
    @Inject
    WSClient ws;

    @With(EchoAction.class)
    public CompletionStage<Result> echoService() {

        return
            ws.url("http://www.mocky.io/
                v2/5e0edec33400003c0f2d7d27")
                .get()
                .thenApply(response -> ok("Feed
                    Response: " + response.getBody()));
    }
}

```



```

/**
 * Example for asynchronous programming
 *
 * @param uid
 * @param age
 * @param relation
 * @return
 */
public CompletionStage<Result> recomendGifts(final
String uid, final String age, final String relation) {

    return CompletableFuture.supplyAsync(this::getGifts)
        .thenApply((List<GiftVO> gift) -> ok("Got " +
            gift));
}

private List<GiftVO> getGifts() {

    List<GiftVO> gifts = new ArrayList<GiftVO>();
    gifts.add(new GiftVO());
    return gifts;
}

/**
 * Example for various route configuration semantics
 *
 * @param bookid
 * @param pageNumber
 * @return
 */
public Result fetchBookpage(String bookid, int pageNumber)
{
    //Query db and get the book details or get from cache

```

```

        return ok(views.html.searchresults.render());
    }

    /**
     * Example for handling Json
     *
     * @param request
     * @return
     */
    @BodyParser.Of(BodyParser.Json.class)
    public Result acknowledgeGreeting(Http.Request request) {

        JsonNode json = request.body().asJson();
        String greeting = json.findPath("greeting").textValue();
        if (greeting == null) {
            return badRequest("Missing parameter [greeting]");
        } else {
            return ok("Your greeting " + greeting + " is accepted");
        }
    }

    @BodyParser.Of(BodyParser.Xml.class)
    public Result acknowledgeGreetingXML(Http.Request request) {

        Document dom = request.body().asXml();
        if (dom == null) {
            return badRequest("Requires XML Input");
        } else {
            String greeting = XPath.selectText("//greeting", dom);
            if (greeting == null) {
                return badRequest("Missing parameter [greeting]");
            } else {

```

```

        return ok("Your greeting " + greeting + " is
        accepted");
    }
}

/**
 * Example for handling XML
 *
 * @param request
 * @return
 * @throws Exception
 */
@BodyParser.Of(BodyParser.Xml.class)
public Result acknowledgeGreetingXMLJaxbVersion(Http.Request
request) throws Exception {

    Document doc = request.body().asXml();
    if (doc == null) {
        return badRequest("Requires XML Input");
    } else {
        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();
        transformer.setOutputProperty(OutputKeys.OMIT_XML_
DECLARATION, "yes");
        StringWriter writer = new StringWriter();
        transformer.transform(new DOMSource(doc), new
StreamResult(writer));
        String output = writer.getBuffer().toString();
        System.out.println("XML " + output);
        JAXBContext context = JAXBContext.newInstance
(Message.class);

```

```

Unmarshaller unMarshaller = context.
    createUnmarshaller();
//JAXB Auto conversion- XML to Model
Message msg = (Message) unMarshaller.unmarshal(new
StringReader(output));
if (msg == null) {
    return badRequest("Requires XML Input");
} else {
    String greeting = msg.getGreeting();
    if (greeting == null) {
        return badRequest("Missing parameter
[greeting]");
    } else {
        return ok("Your greeting " + greeting + "
is accepted");
    }
}
}
}

public Result authors(Integer count) {
    return ok("Top selling authors");
}

/**
 * Example for Caching Http Response
 *
 * @return
 */
@Cached(key = "contactus")
public Result contact() {
    log.info("contact us method: processing");
}

```

CHAPTER 8 COMPLETE EXAMPLE

```
        return ok("Apress Media, LLC\n" +
                "\n" +
                "One New York Plaza, Suite 4600\n" +
                "\n" +
                "New York, NY 10004-1562");
    }

    public CompletionStage<Result> topThreeBooks() {
        return cache.getOrElseUpdate("topthree",
            this::getTopBooks)
            .thenApply((List<String> books) -> ok(books.
                toString()));
    }

    private CompletionStage<List<String>> getTopBooks() {
        List<String> topThreeBooks = new ArrayList<String>();
        topThreeBooks.add("Book 1");
        topThreeBooks.add("Book 2");
        topThreeBooks.add("Book 3");
        return CompletableFuture.completableFuture(topThreeBooks);
    }

    public Result showComment(String userId) {
        return ok("Recent Comments by user");
    }
}
```

models/Book.java

```
package models;

import io.ebean.Model;

import javax.persistence.*;
```

```
import io.ebean.*;

@Entity
@Table(name="book")
public class Book extends Model {

    @Id
    private String id;
    private String author;
    private String title;
    private String picture;

    public String getTitle() {
        return title;
    }

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

    public String getPicture() {
        return picture;
    }

    public void setPicture(String picture) {
        this.picture = picture;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

CHAPTER 8 COMPLETE EXAMPLE

```
    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }
}

class Test {
    void test() {
        Book book = new Book();
        book.save();
        Book b = Ebean.find(Book.class).where().idEq("").findOne();
        Ebean.find(Book.class).where().like("title","java%").
            orderBy("tile desc").findList();
    }
}
```

models/Comment.java

```
package models;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="comment")
public class Comment {

    @Id
    private Long id;
    private String comment;

    public String getComment() {
```

```

        return comment;
    }

    public void setComment(String comment) {
        this.comment = comment;
    }
}

```

models/Customer.java

```

package models;

import controllers.Order;
import java.util.List;

public class Customer {

    private long id;
    private String name;
    private boolean loyaltyMember;
    private boolean isActive;
    private List<Order> orders;

    public List<Order> getOrders() {
        return null;
    }

    public boolean deactivate() {
        //Logic and validation to deactivate a customer
        return false;
    }
}

```

models/GiftVO.java

```

package models;

```



```
public class GiftVO {  
    private String giftName;  
    private String giftImageUrl;  
    private float price;  
    private String category;  
  
    public String getGiftName() {  
        return giftName;  
    }  
  
    public void setGiftName(String giftName) {  
        this.giftName = giftName;  
    }  
  
    public String getGiftImageUrl() {  
        return giftImageUrl;  
    }  
  
    public void setGiftImageUrl(String giftImageUrl) {  
        this.giftImageUrl = giftImageUrl;  
    }  
  
    public float getPrice() {  
        return price;  
    }  
  
    public void setPrice(float price) {  
        this.price = price;  
    }  
  
    public String getCategory() {  
        return category;  
    }  
}
```

```

        public void setCategory(String category) {
            this.category = category;
        }
    }
}

```

models/Message.java

```

package models;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "message")
@XmlAccessorType(XmlAccessType.PROPERTY)
public class Message {

    private String greeting;

    public String getGreeting() {
        return greeting;
    }

    @XmlElement
    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }

}

```

models/Review.java

```

package models;

import dao.ReviewRepository;

import javax.persistence.*;

```

CHAPTER 8 COMPLETE EXAMPLE

```
public class Review {  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    public Long id;  
    public String comment;  
  
    public Long getId() {  
        return id;  
    }  
    public void setId(Long id) {  
        this.id = id;  
    }  
    public String getComment() {  
        return comment;  
    }  
    public void setComment(String comment) {  
        this.comment = comment;  
    }  
    public String getUserId() {  
        return userId;  
    }  
    public void setUserId(String userId) {  
        this.userId = userId;  
    }  
    public String getBookId() {  
        return bookId;  
    }  
}
```

```

public void setBookId(String bookId) {
    this.bookId = bookId;
}

public String userId;
public String bookId;

public void save(ReviewRepository reviewRepo) {
    System.out.println("review repo "+reviewRepo);
    reviewRepo.saveReview(this);
}
}

```

dao/DatabaseExecutionContext.java

```

package dao;

import akka.actor.ActorSystem;
import play.db.Database;
import play.libs.concurrent.CustomExecutionContext;

import javax.inject.Inject;
import javax.inject.Singleton;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

public class DatabaseExecutionContext extends
CustomExecutionContext {

    @Inject
    public DatabaseExecutionContext(ActorSystem actorSystem) {
        // uses a custom thread pool defined in application.conf
        super(actorSystem, "database.dispatcher");
    }
}

```

CHAPTER 8 COMPLETE EXAMPLE

```
@Singleton
public static class JdbcExample {

    private Database db;
    private DatabaseExecutionContext executionContext;

    @Inject
    public JdbcExample(Database db,
        DatabaseExecutionContext context) {
        this.db = db;
        this.executionContext = executionContext;
    }

    public CompletionStage<Integer> updateSomething() {
        return CompletableFuture.supplyAsync(
            () -> {
                return db.withConnection(
                    connection -> {
                        //perform the operations
                        with the connection
                        return 1;
                    });
            },
            executionContext);
    }
}
}
```

dao/JPARepository.java

```
package dao;

import models.Review;
import play.db.jpa.JPAApi;
```

```

import javax.inject.Inject;
import javax.inject.Singleton;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;

import static java.util.concurrent.CompletableFuture.supplyAsync;

@Singleton
public class JPAREviewRepository implements ReviewRepository{

    private JPAApi jpaApi;
    private DatabaseExecutionContext executionContext;

    @Inject
    public JPAREviewRepository(JPAApi api,
        DatabaseExecutionContext executionContext) {
        this.jpaApi = api;
        this.executionContext = executionContext;
    }

    @Override
    public CompletionStage<String> saveReview(Review review) {
        return CompletableFuture.supplyAsync(
            () -> {
                // lambda is an instance of
                // Function<EntityManager, Long>
                return jpaApi.withTransaction(
                    entityManager -> {
                        entityManager.persist(review);
                        return "saved";
                    });
            },
            executionContext);
    }
}

```

models/ReviewRepository.java

```
package dao;

import com.google.inject.ImplementedBy;
import models.Review;

import java.util.concurrent.CompletionStage;

@ImplementedBy(JPAReviewRepository.class)
public interface ReviewRepository {

    CompletionStage<String> saveReview(Review review);
}
```

The views are same as those listed in previous chapters; they are not listed again in this chapter to avoid repetition.

CHAPTER 9

Using Play Modules

A Play application can be assembled from several application modules. This helps in reusing the application components across several applications. The modules also help in splitting a large application into several smaller applications.

A module is just another Play application. Modules can introduce feature-specific abilities such as adding a different persistence mechanism, integrating other view techniques, or integrating a new caching framework.

The modules framework got a revamp since Play 2.5.x. There is not much difference between a module and a library in Play. The only real difference is that modules use the Play API directly. Play modules use dependency injection frameworks to work. If you want to write a custom Play module, it can be done using any dependency injection framework. Note that Play uses guice as its default dependency injection framework.

But the module shouldn't be tied to any particular dependency injection framework and should work in a dependency framework-agnostic way. For this, Play provides a lightweight Bindings API to abstract the module from underlying dependency injection framework.

The dependencies are declared in components using the `@Inject` annotation. For example, if a controller class is dependent on a `WSClient` module, it should declare it using the `@Inject WSClient` syntax. The wiring of the dependent class is done automatically by Play using the underlying dependency injection framework.

Creating a Module

Let's head straight to the creation of a simple module to understand the process. As mentioned, a module is like any other Play application. An important factor to remember is that a module doesn't have its own configuration; it uses the main application's configuration. This means module-specific configuration information should be set in the application's `conf/application.conf` file.

The module you are going to develop is simple. It calculates the factorial of a number less than 25. So follow these steps:

- 1) Create a package and name it `modules` in the `app` directory.
- 2) Define the contract of the module as an interface:

```
package modules;

public interface Factorial {

    int fact(int num);
}
```

- 3) Provide the implementation for the `Factorial` interface:

```
package modules;

import play.inject.ApplicationLifecycle;
import javax.inject.Inject;

public class FactorialImpl implements Factorial{

    @Inject
    public FactorialImpl(ApplicationLifecycle lifecycle) {
        //implement plugin lifecycle methods
    }
}
```

```

public int fact(int number) {
    return calculateFactorial(number);
}
private int calculateFactorial(int num) {
    if(num > 25) {
        throw new RuntimeException("Our of range");
    }
    if(num == 1)
        return num;
    return num * calculateFactorial(num-1);
}
}

```

- 4) Create a module and name it FirstModule:

```

package modules;

import play.api.Configuration;
import play.api.Environment;
import play.api.inject.Binding;
import play.api.inject.Module;

import javax.inject.Inject;
import play.inject.ApplicationLifecycle;
import scala.collection.Seq;

public class FirstModule extends Module {
    public Seq<Binding<?>> bindings(Environment
environment, Configuration configuration) {
        return seq(
            bind(Factorial.class).to(FactorialImpl.class)
        );
    }
}

```

Here you've declared a module and added your `Factorial` implementation to the module. Now, for Play to detect it as a module, a few changes are required in the `application.conf` file:

```
play.modules.enabled += "modules.FirstModule"
```

Save the file. That's it! You have created a custom Play module. The `Factorial` implementation defined in this module can be injected to any class and can be used. For example, if it is needed in a controller class, just declare it as

```
@Inject Factorial fact;
```

Play will do the rest and make it available during runtime. If you want to make this module available to other Play applications, it needs to be published to repositories. In such a case, only include the module classes and related dependent classes in the Play application. All unused components like views, public assets, etc. can be removed.

Start the command prompt and go to the directory where you created the `factorial` Play project. For example, if the project exists in `C:\Users\username\playexamples\bookshop\`, type `sbt` from the command prompt. This will load the Play console. You need to publish the module so that it is accessible by other applications. For the sake of simplicity, you will publish this module to a local repository in your computer:

```
bookshop] $ publishLocal
```

Play will compile the files and publish to the local Play repository. From the Play console, do take a note of the location where Play publishes the module. In my machine, it is published under `C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\docs\bookshop_2.13-javadoc.jar`. See Figure 9-1.

```

Command Prompt - sbt run - sbt run - sbt
[bookshop] $
[bookshop] $
[bookshop] $
[bookshop] $ publishLocal
[info] Wrote C:\Users\premk\playexamples\bookshop\target\scala-2.13\bookshop_2.13-1.0-SNAPSHOT.pom
[info] Main Scala API documentation to C:\Users\premk\playexamples\bookshop\target\scala-2.13\api...
model contains 38 documentable templates
[info] Main Scala API documentation successful.
[info] Packaging C:\Users\premk\playexamples\bookshop\target\scala-2.13\bookshop_2.13-1.0-SNAPSHOT-javadoc.jar ...
[info] Done packaging.
[info] :: delivering :: com.stackrules.example#bookshop_2.13;1.0-SNAPSHOT :: 1.0-SNAPSHOT :: integration :: Tue Nov 12 2
2:45:34 IST 2019
[info] delivering ivy file to C:\Users\premk\playexamples\bookshop\target\scala-2.13\ivy-1.0-SNAPSHOT.xml
[info] published bookshop_2.13 to C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\poms\boo
kshop_2.13.pom
[info] published bookshop_2.13 to C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\jars\boo
kshop_2.13.jar
[info] published bookshop_2.13 to C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\srcs\boo
kshop_2.13-sources.jar
[info] published bookshop_2.13 to C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\docs\boo
kshop_2.13-javadoc.jar
[info] published ivy to C:\Users\premk\.ivy2\local\com.stackrules.example\bookshop_2.13\1.0-SNAPSHOT\ivys\ivy.xml
[success] Total time: 10 s, completed Nov 12, 2019 10:45:35 PM
[bookshop] $

```

Figure 9-1. *Published module*

The module is now ready to be consumed by other applications.

Third-Party Modules

Often you need to use other software tools for various reasons, and in Play they are integrated into the application using modules. There are many third-party modules available for most enterprise requirements. Play modules make it possible to extend Play functionality with a plug-and-play architecture. Here are some popular ones:

- Redis: Integrates Redis to provide a cache implementation
- Deadbolt: Role-based authorization
- PDF: Adds support for PDF output based on HTML templates

There are many more such modules available and you can find more information about them by visiting www.playframework.org.

CHAPTER 10

Application Settings and Error Handling

Before Play 2.6, the practice was to use a global object to define application-level configurations and processing logic like filtering, error handling, etc. This approach was deprecated in Play 2.5.x and dropped since Play 2.6.x. The recommended approach since Play 2.6.x is to use dependency injection. Hence I won't go into the details of the approach followed prior to 2.6.x and instead will direct your attention to the dependency injection approach. I will give you an example at the end of this section so that you are aware how it was done before, which will be helpful if you're migrating an older Play framework project to the newer version.

The main class of a Play application is `play.Application`. This class is created by Play Framework during a Play application start up. You need not do anything special to make it happen; it's handled by the framework. You can hook various application-level behaviors by binding handlers to specific instances of classes using dependency injection. This is the recommended approach to configure application-level bindings in Play. Let's look at examples to understand this in more depth.

Filters

A filter is intended for applying cross-cutting concerns of the application to all classes, such as logging, inspecting security headers, compressing the response, analytics etc. These are some of the places where a filter is useful. Play also has something known as action composition and it is intended to be used in situations where the concern is specific to a route. For example, you may decide to secure certain routes and allow public access to all other routes. In such a case, for specific routes you may need to check authentication/authorization semantics. For such cases, use action composition. You will look more into action composition in later sections of this chapter. Let's focus on filters first.

Logging Filter:

```
package config;

import akka.event.LoggingFilter;
import akka.stream.Materializer;

import java.util.concurrent.CompletionStage;
import java.util.function.Function;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import play.mvc.Filter;
import play.mvc.Http;
import play.mvc.Result;

import javax.inject.Inject;

/**
 * A simple filter implementation that logs how long it took to
 * process a request
 */
```

```

public class ApplicationFilter extends Filter {
    @Inject
    public ApplicationFilter(Materializer mat) {
        super(mat);
    }
    private static final Logger log = LoggerFactory.getLogger(
        LoggingFilter.class);

    @Override
    public CompletionStage<Result> apply(Function<Http.
        RequestHeader,
                                CompletionStage<Result>> next,
                                Http.RequestHeader requestHeader) {

        long startTime = System.currentTimeMillis();
        return next
            .apply(requestHeader)
            .thenApply(
                result -> {
                    long endTime =
                        System.currentTimeMillis();
                    long requestTime = endTime -
                        startTime;
                    log.info(
                        "{} {} took {}ms to complete
                        and produced the status {}",
                        requestHeader.method(),
                        requestHeader.uri(),
                        requestTime,
                        result.status());
                }
            );
    }
}

```

```

        return result;
    });
}
}

```

The `ApplicationFilter` class extends from the `play.mvc.Filter` and overrides the `apply` method. The `apply` method takes two parameters:

- 1) A function that takes `HttpRequestHeader` as a parameter and returns `CompletionStage<Result>` as the output
- 2) `HttpRequestHeader`: The request header of the incoming request

Let's understand the first argument and look at how it is used in the code. The next parameter you use in the code represents the next action in the filter chain. Invoking it will result in allowing the request to flow forward and eventually reach the intended class, the controller in most cases. In your logging case, before you pass control, you log the request time and then allow the request to flow forward. Once the request completes, you use the `thenApply` function to log the response time and return the result back to the caller. Please note that `CompletionStage<Result>` is the generic Promise API provided by Java 8 for handling asynchronous responses. Since Play uses asynchronous programming, the responses from Play APIs use it. This is the reason why you need to do processing using the `thenApply` method. More details of the Promise class and asynchronous programming are provided in the chapter on asynchronous programming.

Now that you have created a filter, let's see how you can tell Play to use it. Create a class named `FilterConfig` as follows and save it inside the `config` package:

FilterConfig.java

```
package config;

import play.http.DefaultHttpFilters;
import javax.inject.Inject;

public class FilterConfig extends DefaultHttpFilters {
    @Inject
    public FilterConfig(ApplicationFilter logging) {
        super(logging);
    }
}
```

You have defined this class in a package, `config`. Because of this, Play needs to be told which class it should use to configure the filters. This is done by adding the following entry to the `application.conf` file:

```
play.http.filters=config.FilterConfig
```

If you saved the `FilterConfig` file in the root folder, then Play will automatically use it and there is no need add the entry in `application.conf`, as above. But it is better to put the classes into corresponding packages for ease of organization and modularity, and that's the reason for saving in the `config` package instead of the root folder. Observe that the custom filter was passed as an argument to the `FilterConfig` class using constructor-level dependency injection. If you have written more filters, you can pass them in the same way as constructor arguments to the `FilterConfig` class. This works because the `FilterConfig` class extends from `DefaultHttpFilters`

and it has a constructor that takes *N*, the number of *Filter* instances via *vargs*.

That is it. You have created a response logging filter. Let's test it. Open `http://localhost:9000/bookshop` and observe the console output and logs (`projectroot\logs\application.log`). In both places you will see an entry as follows:

```
[info] a.e.LoggingFilter - GET /bookshop took 7ms to complete  
and produced the status 200
```

If you don't see any output, check the logger settings and make sure it is set to the INFO log level. For this, open the `conf/logback.xml` file and ensure you have the following settings:

```
<logger name="play" level="INFO" />  
<logger name="application" level="DEBUG" />  
  
<root level="INFO">  
  <appender-ref ref="ASYNCFILE" />  
  <appender-ref ref="ASYNCSTDOUT" />  
</root>
```

Action Composition

Action composition is useful when you want to decorate a specific class or certain group of classes with additional logic. Let's learn by an example:

EchoAction.java

```
package config;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import play.mvc.Http;
```

```

import play.mvc.Result;

import java.util.concurrent.CompletionStage;

/**
 * An example of Action composition
 */
public class EchoAction extends play.mvc.Action.Simple {

    private static final Logger log = LoggerFactory.getLogger(EchoAction.class);

    public CompletionStage<Result> call(Http.Request req) {
        log.info("Request Method {} ", req.method());
        return delegate.call(req);
    }
}

```

This is a very simple Action that merely echoes the HTTP request method to the logs. This is not a filter, so it won't be applied to methods by default. The method that needs to use this action has to explicitly ask for it via annotations. Let's use the EchoAction in one of the controller methods. Open the Application.java file, go to echoService method, and add the annotation @With(EchoAction.class):

```

@Inject WSClient ws;
@With(EchoAction.class)
public CompletionStage<Result> echoService() {
    return
        ws.url("http://www.mocky.io/v2/5e0edec3340003c0f2d7d27")
            .get()
            .thenApply(response -> ok("Feed Response: " + response.getBody()));
}

```

Open `http://localhost:9000/bookshop/book/echo` and observe the console and logs. You will see the following log entry:

```
[info] c.EchoAction - Request Method GET
```

Error Handlers

As you know, Play is an HTTP application framework and such an application can experience two types of errors:

1. Client errors
2. Server errors

Client Errors

Client errors are due to mistakes made by a caller or the connecting client of the application, such as if a client didn't send the Content-Type header, didn't use the proper HTTP method, invalid URLs, etc. In all such cases, Play automatically detects the error and redirects to error pages.

Server Errors

Server errors are due to something wrong on the server, such as lack of resources, a crash, application class-generated errors like a null pointer, code throwing other forms of exceptions, etc. Play intelligently handles such server errors by catching them and generating an error page.

In certain cases, the application may decide not to use the default error handling mechanism as is and instead would like to extend it and provide custom error handling logic. Let's look at how you can do that.

CustomErrorHandler.java

```

package config;

import play.http.HttpErrorHandler;
import play.mvc.*;
import play.mvc.Http.*;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import javax.inject.Singleton;

/**
 * A simple error handler
 */

@Singleton
public class CustomErrorHandler implements HttpErrorHandler {
    public CompletionStage<Result> onClientError(
        RequestHeader request, int statusCode, String
        message) {
        return CompletableFuture.completedFuture(
            Results.status(statusCode, "Invalid Request " +
            message));
    }

    public CompletionStage<Result> onServerError(RequestHeader
    request, Throwable exception) {
        return CompletableFuture.completedFuture(
            Results.internalServerError("Cannot process the
            request due to " + exception.getMessage()));
    }
}

```

You have now configured a custom error handler to use in cases of client and server errors. Let's test it.

Open `http://localhost:9000/bookshop/book/echos`. This URL is an invalid URL and is a client-side error. This request will be intercepted by the `onClientError` method defined in your `CustomErrorHandler` and you will see the response in browser as

Invalid Request

How Global Settings Were Done Before Play 2.6.x

Global objects allow for the handling of global settings of the application. A global object is the place where you write your intercepting logic. Tasks like filtering certain request parameters, logging requests, etc. can be easily done here. You can also configure global configurations for not found errors, error pages, etc.

I will not go much deeper into `GlobalSettings` and `Global` because Play deprecated them since the 2.5.0 version and dependency injection is now the recommended way. The example below is just to show how it was done in earlier Play versions. Please note that the below code will work only in Play versions 2.5.x and below; it won't work with Play 2.8.x, which is the version this book is based on. The code is just for reference in case you want to migrate from an earlier Play version to the latest and wish to know how it was done earlier.

To create a Global object, just write a class extending from `play.GlobalSettings`. The Global object is typically saved in the root package of your application but you can save it in any package and then configure it in the `application.conf` file by using the `application.global` property:

```
import play.Application;
import play.GlobalSettings;
import play.libs.Akka;
import play.mvc.Http.RequestHeader;
```

```

import play.mvc.Result;
import play.mvc.Results;
import scala.concurrent.duration.Duration;
import akka.actor.ActorRef;
import akka.actor.Props;
import controllers.Preloader;

public class Global extends GlobalSettings {

public Result onError(RequestHeader request, Throwable t) {
    return Results.notFound(views.html.errorpagetemplate.render());
}

public Result onHandlerNotFound(RequestHeader request) {
    return Results.notFound(views.html.errorpagetemplate.render());
}

public Result onBadRequest(RequestHeader request, String error) {
    return Results.notFound(views.html.errorpagetemplate.render());
}

public void onStart(Application app) {
    //initialize all
    ActorRef preloader = Akka.system().actorOf(new
    Props(Preloader.class));
    Akka.system().scheduler().schedule(
        Duration.create(0, TimeUnit.MILLISECONDS), //Initial delay 0
                                                    milliseconds
        Duration.create(5, TimeUnit.MINUTES),      //Frequency 30
                                                    minutes

        preloader,
        "preload",
        Akka.system().dispatcher()
    );
}

```

```
public void onStop(Application app) {  
    Logger.info("Application shutdown");  
}  
}
```

To migrate to the latest Play version, you should write custom filters and include the logic in those filters or error handlers. You learned both of these things at the beginning of this chapter.

CHAPTER 11

Working with Cache

Caching is one of the most important aspects in ensuring the scalability and responsiveness of the application. A good cache can take load off your database and can serve the user pretty fast. A framework that doesn't provide a good abstraction to work with cache is no longer fit for enterprise-class development. Play, with its in-built support for cache, is an ideal framework for building highly scalable, responsive, enterprise-class applications.

Play provides a global cache object and it works with the underlying cache implementation. This abstraction enables you to change your caching provider without any modification to your application code. By default, Play comes with a Cache API implementation using Caffeine and also provides implementation for EhCache 2.x. For in-process processing, Play recommends using Caffeine.

Configuring Caffeine

Caffeine is a high performance, near-optimal caching library and is based on Java 8.

The following are Caffeine's important features at a glance:

- Automatic loading of entries into the cache, optionally asynchronously
- Size-based eviction when a maximum is exceeded based on frequency and recency

- Time-based expiration of entries, measured since last access or last write
- Asynchronously refresh when the first stale request for an entry occurs
- Keys automatically wrapped in weak references
- Values automatically wrapped in weak or soft references
- Notification of evicted (or otherwise removed) entries
- Writes propagated to an external resource
- Accumulation of cache access statistics

You can find more about Caffeine at <https://github.com/ben-manes/caffeine/>.

Adding Caffeine to a Project

Open the `build.sbt` file and add the dependency for Caffeine:

```
libraryDependencies += Seq(guice,javaJdbc,cacheApi,
    javaWs,javaJpa,caffeine,org.hibernate" % "hibernate-
    entitymanager" % "5.3.7.Final",com.h2database" % "h2" %
    "1.4.200" )
```

This is all that is required to use Caffeine as the cache implementation for Play.

Configuring EhCache

EhCache has been in existence for a long time and is one of the most widely used cache implementations in Java projects. It requires Java 8 or above. You can learn about the internal workings of EhCache at www.ehcache.org/.

Configuring EhCache is as simple as adding an entry to the `build.sbt` file, just like you did for Caffeine:

```
libraryDependencies += Seq(guice, javaJdbc, cacheApi,
    javaWs, javaJpa, ehcache, org.hibernate" % "hibernate-
    entitymanager" % "5.3.7.Final", com.h2database" % "h2" %
    "1.4.200" )
```

Using the Cache API

The Cache API is pretty simple and easy to use:

- Putting data:
 - `Cache.set("key", object)`
- Retrieving data:
 - `Cache.get("key")`
- Removing data:
 - `Cache.remove("key")`

You can also control how long data needs to be retained in the cache (cache expiry) by using the `set` method with three arguments:

```
Cache.set("key", object, int timeinminutes)
```

To use the Cache API in the controller, it needs to be injected. Modify the `Application.java` file as follows to get a reference to the Cache API:

```
private AsyncCacheApi cache;
@Inject
public Application(ActorSystem system, AsyncCacheApi cache) {
    pingActor = system.actorOf(PingActor.getProps());
    this.cache = cache;
}
```

Now you can use the Cache API to cache data in any method of the controller. For instance,

```
CompletionStage<Done> result = cache.set("framework",
"Playframework");
//Cache for 10 minutes
CompletionStage<Done> result = cache.set("framework",
"Playframework", 60 * 10);
```

Whenever you work with cache, remember that a cache can expire or get lost anytime. It is not persistent data, hence you should always check whether the data is in cache and, if not, populate it in cache and then retrieve it. This way your program is safe even if the cache doesn't work or expires sooner than expected. In Play, the underlying cache implementation can automatically remove objects from cache if there is a low memory scenario in the JVM. So code wisely and use cache effectively.

Play uses a callable method to load data into cache if it is not present. This is accomplished by providing a callable method to the Cache API that gets invoked to generate the cached data when data is absent in the cache. Let's modify the `Application.java` file to add such a method:

```
public CompletionStage<Result> topThreeBooks() {
    return cache.getOrElseUpdate("topthree",this::getTopBooks)
```

```

        .thenApply(( List<String> books) -> ok(books.
            toString()));
    }
    private CompletionStage<List<String>> getTopBooks() {
        List<String> topThreeBooks = new ArrayList<String>();
        topThreeBooks.add("Book 1");
        topThreeBooks.add("Book 2");
        topThreeBooks.add("Book 3");
        return CompletableFuture.completedFuture(topThreeBooks);
    }

```

The private method `getTopBooks` is the callable method and it is invoked by the Cache API if the data is not present in the cache. This method computes the top three books and returns the response. The public method `topThreeBooks` uses the `getOrCreateUpdate` method of the Cache API to get the data from the cache and convert it to the HTTP response.

Edit routes.conf

```
GET /bookshop/book/top controllers.Application.topThreeBooks()
```

Test the method using `http://localhost:9000/bookshop/book/top` and you will see the response as

```
[Book 1, Book 2, Book 3]
```

Play also provides the ability to cache entire HTTP responses. This is useful to cache content that doesn't change per request. Let's create a new route and cache its response. You'll use the Contact Us data that is static

(you show the same content to all users). Open the `Application.java` file and add the `contactus` method:

```
@Cached(key="contactus")
public Result contact() {
    log.info("contact us method: processing");
    return ok("Apress Media, LLC\n" +
        "\n" +
        "One New York Plaza, Suite 4600\n" +
        "\n" +
        "New York, NY 10004-1562");
}
```

Add the route configuration in the `routes.conf` file as

```
GET /bookshop/contact controllers.Application.contact()
```

To test, open the URL `http://localhost:9000/bookshop/contact` to observe the response. This response is cached and is coming from the in-memory cache. To test this, check the logs in the `application.log` file or console you will see the entry

```
[info] c.Application - contact us method: processing
```

Access the URL again and you won't see the log entry because from the second time onwards, the response is served from the cache, bypassing the processing logic which was already computed and cached.

This example is very simple and was used just to explain the point. Response caching is very useful if the data (the contact info, as above) is retrieved from a database or a web service or some other way that involves some amount of processing. In such cases, for the first time, the processing will happen, but from the next time onwards you can avoid all processing because the response will be served from the cache. This generates a faster response to the user and also doesn't waste the processing time of the servers.

There are also many scenarios where it is not practical to cache the entire HTTP response. For instance, you may have certain elements in a page that change according to the user, based on his login. This scenario can also be easily handled by caching only the part of the data. You should design your content for this. Consider a home page with 75% static content and 25% dynamic content. In this case, split the static and dynamic parts into separate objects and cache the static object. But in such a scenario you can only cache the object and not the generated static HTML. That is, you cannot take part of the HTML response from the cache and mix it with the dynamic HTML response. You have to cache at the object or data level rather than the HTTP response level in such cases. For pure static content that doesn't change with each request or user, use HTTP response caching, as explained in the contact method above. You also learned object caching, as explained in the **topThreeBooks** example above.

The default Caffeine implementation is good enough for most cases and is highly scalable. But if you need to use a distributed caching framework like memcached, Play provides hooks to plug in memcached with its Cache API. For memcached, Play provides an implementation by default but depends on third-party plugins. To use memcached with Play, use the plugin available at <https://github.com/mumoshu/play2-memcached>. I won't detail a memcached implementation in this book because the third-party plugin documentation is not yet updated with Play 2.7 and above.

CHAPTER 12

Production Deployment

An easier way to deploy Play is to use the command `play run`. For this to work, you need to copy the entire Play project to the production box and then, from within the project, issue `play run`. This is not a good method for production deployment because you don't want to move your source files and only the executable in the form of jar files needs to be pushed to production.

To achieve this, you can use the `play dist` command in your build box to generate the Play executable jar files. Once the `dist` command succeeds, it will create a `dist` directory and the jar file will be placed there. Just copy and unzip the jar to the production box and you will see a start file. Make the start file executable by assigning it the required execute permissions (Unix/Linux box) and run it.

You can edit the start file and pass it additional arguments like `-Xms` `-Xmx` etc. to configure the JVM memory parameters. If you need to change the default listen port of the Play application, you can pass `-Dhttp.port=port` to the start script.

You can run Play upfront without the need for any other web servers. But this is not the ideal production deployment scenario. You may have a web server like Apache HTTP server or Nginx before your Play server to handle various production configuration and access requirements. You may want to configure Apache to handle SSL requests and offload the SSL load from Play, or maybe you just don't want to expose your Play server to the Internet. The reasons are many. Let's see how you can use Apache `httpd` as a proxy for your Play application.

Configuring Apache httpd for Play

It is assumed that you already have an Apache installation and it is up and running. Make sure you have the following modules enabled: `mod_ssl`, `mod_headers`, `mod_proxy`, `mod_proxy_http`, and `mod_proxy_balancer`.

Here is a simple Apache configuration for redirecting from external port 80 to an internal IP and port 9000 of your Play application:

```
<VirtualHost *:80>
ProxyPreserveHost on
ProxyPass / http://192.168.1.4:9000/
ProxyPassReverse / http://192.168.1.4:9000/
</VirtualHost>
```

The above configuration will route the requests to port 80 to the Play application running on port 9000.

Load Balancing Using `mod_proxy_balancer`

Production deployments usually have multiple server instances for load balancing and high availability requirements. You can start multiple Play instances and use Apache to perform the load balancing. You need the `mod_proxy_balancer` module for this:

```
<Proxy balancer://prodcluster>
BalancerMember http://192.168.1.4:9000
BalancerMember http://192.168.1.5:9000
</Proxy>
<VirtualHost *:80>
ProxyPreserveHost on
ProxyPass / balancer://prodcluster/
ProxyPassReverse / balancer://prodcluster/
</VirtualHost>
```

Configuring Play with Nginx

Nginx (pronounced engine-x) is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. Unlike traditional servers, Nginx doesn't rely on threads to handle requests.

Instead it uses a much more scalable event-driven (asynchronous) architecture. This architecture uses small, but more importantly, predictable amounts of memory under load.

Nginx should be already installed and up and running. Execute the following on any Debian-derived distribution to install Nginx in your server:

```
apt-get install nginx
```

The file you want to edit is the `nginx.conf` file. Edit the `/etc/nginx/nginx.conf` file and configure it for usage with Play. Here is a sample configuration for your reference:

```
http {
    proxy_buffering    off;
    proxy_set_header  X-Real-IP $remote_addr;
    proxy_set_header  X-Scheme $scheme;
    proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header  Host $http_host;
    upstream my-backend {
        server 127.0.0.1:9000;
    }
    server {
        server_name www.yourserverdomainname.com;
    }
    server {
        keepalive_timeout    70;
```

```
server_name www.yourserverdomainname.com;
location / {
proxy_pass http://my-backend;
}
}
}
```

Note Replace `yourserverdomainname` with the domain name of your website.

After saving the file, you need to restart Nginx:

```
sudo /etc/init.d/nginx restart
```

Index

A

- Action composition, [164](#), [168–169](#)
- Application-level configurations, [53](#), [54](#)
- Asynchronous programming
 - Akka, [84–86](#)
 - application, [83](#), [84](#)
 - CompletionStage, [83](#)
 - configuration, [84](#)
 - getGifts method, [84](#)
 - Play Framework
 - documentation, [85](#)
 - synchronous processing, [82](#)

B

- Bidirectional
 - relationship, [112](#), [113](#), [130](#)
- Bookshop project
 - app folder, [9](#), [10](#)
 - build.sbt files, [11](#)
 - configuration (conf) file, [10](#)
 - folder structure, [9](#)
 - lib folder, [11](#)
 - project folder, [11](#)
 - project creation, [8](#)
 - public folder, [11](#)
 - unit and functional test, [12](#)

C

- Caching object, [175](#)
 - API
 - Application.java file, [178](#)
 - callable method, [178](#)
 - contactus method, [180](#)
 - getTopBooks, [179](#)
 - HTTP and HTML response, [181](#)
 - use of, [177](#)
 - Caffeine
 - build.sbt file, [176](#)
 - features, [175](#), [176](#)
 - EhCache, [177](#)
- Cascade-delete relationship, [112](#)
- Composite views, [66](#)
- Concurrency programming,
 - See* Asynchronous programming
 - callable interface, [80–82](#)
 - classes, [75](#)
 - java.util.concurrent
 - package, [77](#)
 - runnable, [78](#), [79](#)
 - differences (Runnable and Callable), [77](#)
 - threads, [76](#)
 - executor framework, [76](#), [77](#)

INDEX

Controllers

- bookshop application
 - Application.java, 56
 - routers, 56
 - saveComment
 - method, 58, 59
 - testing, 59, 60
- JPA configuration, 61
- model/helper classes, 60
- scoped objects
 - application.conf file, 62
 - flash scope, 64
 - messages/error
 - messages, 64
 - session scope, 62, 63
 - storing and accessing
 - class, 61

D

Database connections

- Application.java, 136–146
- conf/application.conf, 133, 134
- dao
 - DatabaseExecutionContext.
 - java, 153
 - JpaRepository.java, 154
- http responses, 138
- models
 - Book.java, 146–148
 - Comment.java, 148, 149
 - Customer.java, 149
 - GiftVO.java, 149
 - Message.java, 151

Review.java, 151

ReviewRepository.java, 156

ping method, 138

routes, 134, 135

thread pool, 134

Database resources

ActorSystem code, 108

application.conf file, 105

build.sbt file, 105

CustomExecutionContext

method, 106, 107

Ebean (*see* Ebean model)

Java Persistence API (JPA),
113–118

MySQL database, 106

ORM (*see* Object relational
mapping (ORM))

Dependency injection, 85, 86, 157,
163, 167, 173

E

Ebean model

annotations, 123

EntityManager, 118

eq(), 125

findList(), 126

findUnique(), 126

interface, 126–128

io.ebean.Model, 120

like(), 125

LIMIT {max rows} [OFFSET
{first row}], 126

optimization, 119

- orderBy(), 125
- persistence context, 118
- plugin dependency, 120
- query, 120–124
- RawSql, 128–130
- relationship, 130, 131
- save method, 124
- where() clause, 125

Eclipse, 12–14

EhCache, 175, 177

Error handlers

- application.global property, 172
- client errors, 170
- global objects, 172–174
- server, 170–172
- types of, 170

ExecutorService, 76–79

Extensible Markup Language (XML)

- handling JSON, 98
- JAXB, 100–104
- Message.java file, 100
- parsing, 99, 100
- text/plain response, 99

F, G

Filters

- action composition, 164, 168–170
- application.conf file, 167
- ApplicationFilter class, 166

- conf/logback.xml file, 168
- echoService method, 169
- implementation, 164, 165
- package configuration, 167
- parameters, 166

H

Hello World application

- configuration, 19–22
- console, 18
- controller
 - app/controllers folder, 22
 - editing process, 24–26
 - Hello page, 25
 - hello.scala.html file, 26, 27
 - HomeController.java file, 22, 23
- default application, 17
- main.scala.html, 21
- routes file, 19
- Twirl template, 19
- view folder, 24
- welcome page, 18

HTTP routing fundamentals

- configuration, 48–50
- conf/routes file, 49
- dynamic part (URL), 51–53
- parameters, 53
- passing fixed values, 52
- protocols, 49
- static definition, 50

INDEX

I

Integrated development
environment (IDE), 12–17,
28, 39, 43

IntelliJ

import project, 15
Scala plugin, 15
steps, 16

J, K, L

Java Persistence API (JPA)

application.conf file, 113
build.sbt, 113
entity manager, 114
implementation, 113
JDBC operations, 115
JPReviewRepository.java, 117,
118
review operations, 116

JavaScript Object Notation (JSON)

acknowledgeGreeting method, 97
request object, 94–96
response, 97, 98
scenarios, 94
testing, 96, 98

M, N

Model-view-controller (MVC)

architecture, 46
controller, 48
deactivate method, 47
design pattern, 45, 46

model, 46, 47

view, 48

Modules

application.conf file, 160
creation, 158
dependencies, 157
Factorial interface, 158
FirstModule method, 159
interface, 158
publishes, 160, 161
third-party, 161

O

Object relational mapping (ORM)

advantages, 110
approach, 109
bidirectional
relationship, 112, 113
concepts, 109, 110
entities, 111
many to many, 111
many to one, 111
one to many, 111
unidirectional relationship,
111, 112

P, Q, R

Play framework

bookshop (*see* Bookshop
project)
conscript installation, 2
Giter8, 3

- Hello World (*see* Hello World application)
- IDE (*see* Integrated development environment (IDE))
- installation, 1
- Java project, 5, 6
- JDK version 1.8, 2
- play setup, 4
- sbt (Scala build tool)
 - Java project structure, 7
 - project creation, 6, 7
 - Scala and Java
 - installation, 7
 - website details, 2
- Scala approach, 4, 5
- testing
 - hello.scala.html view, 28, 29
 - HomeControllerTest.java, 29–31
 - testOnly, 31
 - views and controller classes, 27
- web development (Java), 1
- Production deployment
 - Apache installation, 184
 - jar files, 183
 - load balancing, 184
 - mod_proxy_balancer
 - module, 184
 - Nginx, 185, 186

S

- saveComment method
 - curl command, 60
 - DynamicForm class, 58, 59
 - testing, 59, 60
- Scala build tool/simple build tool (sbt), 2
 - benefits, 34
 - build.sbt file, 36, 42, 43
 - built tools, 33
 - commands, 37, 43
 - core principles, 33
 - definition, 39–42
 - folder structures and files, 38
 - helloworldsbt project, 36–39
 - Maven central repository, 41
 - plugin.sbt, 43
 - project structure, 35, 36
 - resolvers, 42
 - root folder, 38

T

- Template engine, 65
 - comments, 69
 - content design, 68
 - dynamic contents, 74
 - getTitle and getPicture
 - method, 72
 - HTML code, 68

INDEX

Template engine (*cont.*)

- if block, [73](#)
- import statement, [71](#)
- list iteration, [72](#)
- map iteration, [73](#), [74](#)
- parameters, [70](#)
- Scala template language, [66](#)
- Twirl, [69](#)

Third-party modules, [161](#)

Twirl, [19](#), [20](#), [65](#), [69](#)

U

Unidirectional relationship, [111](#), [112](#)

V

Views, *See* Composite views

W, X, Y, Z

Web services, [87](#)

- Application.java, [88–92](#)
- back-end service, [88](#)
- CompletableFuture, [88](#)
- getBodyAsSource, [91](#)
- processLargeResponse, [92–95](#)
- reponses, [91](#)
- xml method, [89](#)