



Quick answers to common problems

Play Framework Cookbook

Second Edition

Over 60 hands-on recipes to create dynamic and reactive
web-based applications with Play 2

Alexander Reelsen
Giancarlo Inductivo

[**PACKT**]
PUBLISHING open source*
community experience distilled

Play Framework Cookbook Second Edition

Table of Contents

[Play Framework Cookbook Second Edition](#)

[Credits](#)

[About the Authors](#)

[About the Reviewers](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers, and more](#)

[Why subscribe?](#)

[Free access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Sections](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[See also](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Downloading the color images of this book](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Basics of Play Framework](#)

[Introduction](#)

[Installing Play Framework](#)

[Getting ready](#)

[How to do it...](#)

[Creating a Play application using Typesafe Activator](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using the Play console](#)

[How to do it...](#)

[There's more...](#)

[Working with modules](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Working with controllers and routes](#)

[How to do it...](#)

[How it works...](#)

[Using Action parameters in controllers](#)

[How to do it...](#)

[How it works...](#)

[Using reverse routing and redirects](#)

[How to do it...](#)

[How it works...](#)

[Working with View templates](#)

[How to do it...](#)

[How it works...](#)

[Using helper tags](#)

[How to do it...](#)

[How it works...](#)

[Using View layouts and Includes](#)

[How to do it...](#)

[How it works...](#)

[Working with XML and text files](#)

[How to do it...](#)

[How it works...](#)

[Using Ebean \(Java\) with MySQL](#)

[How to do it...](#)

[Creating a record](#)

[Updating a record](#)

[Querying a record](#)

[Retrieving a record](#)

[Using Anorm \(Scala\) and database evolutions with MySQL](#)

[There's more...](#)

[Creating a new record](#)

[Updating a record](#)

[Deleting a record](#)

[Using a form template and web action](#)

[How to do it...](#)

[How it works...](#)

[Using form validation](#)

[How to do it...](#)

[How it works...](#)

[Securing form submission](#)

[How to do it...](#)

[How it works...](#)

[Testing with JUnit \(Java\) and specs2 \(Scala\)](#)

[How to do it...](#)

[How it works...](#)

[Testing models](#)

[How to do it...](#)

[How it works...](#)

[Testing controllers](#)

[How to do it...](#)

[How it works...](#)

[2. Using Controllers](#)

[Introduction](#)

[Using HTTP headers](#)

[How to do it...](#)

[How it works...](#)

[Using HTTP cookies](#)

[How to do it...](#)

[How it works...](#)

[Using the session](#)

[How to do it...](#)

[How it works...](#)

[Using custom actions](#)

[How to do it...](#)

[How it works...](#)

[Using filters](#)

[How to do it...](#)

[How it works...](#)

[Using path binders](#)

[How to do it...](#)

[How it works...](#)

[Serving JSON](#)

[How to do it...](#)

[How it works...](#)

[Receiving JSON](#)

[How to do it...](#)

[How it works...](#)

[Uploading files](#)

[How to do it...](#)

[How it works...](#)

[Using futures with Akka actors](#)

[How to do it...](#)

[How it works...](#)

[3. Leveraging Modules](#)

[Introduction](#)

[Dependency injection with Spring](#)

[How to do it...](#)

[How it works...](#)

[Dependency injection using Guice](#)

[How to do it...](#)

[How it works...](#)

[Utilizing MongoDB](#)

[How to do it...](#)

[How it works...](#)

[Utilizing MongoDB and GridFS](#)

[How to do it...](#)

[How it works...](#)

[Utilizing Redis](#)

[How to do it...](#)

[How it works...](#)

[Integrating Play application with Amazon S3](#)

[How to do it...](#)

[How it works...](#)

[Integrating with Play application Typesafe Slick](#)

[How to do it...](#)

[How it works...](#)

[Utilizing play-mailer](#)

[How to do it...](#)

[How it works...](#)

[Integrating Bootstrap and WebJars](#)

[How to do it...](#)

[How it works...](#)

[4. Creating and Using Web APIs](#)

[Introduction](#)

[Creating a POST API endpoint](#)

[How to do it...](#)

[How it works...](#)

[Creating a GET API endpoint](#)

[How to do it...](#)

[How it works...](#)

[Creating a PUT API endpoint](#)

[How to do it...](#)

[How it works...](#)

[Creating a DELETE API endpoint](#)

[How to do it...](#)

[How it works...](#)

[Securing API endpoints with HTTP basic authentication](#)

[How to do it...](#)

[How it works...](#)

[Consuming external web APIs](#)

[How to do it...](#)

[How it works...](#)

[Using the Twitter API and OAuth](#)

[How to do it...](#)

[How it works...](#)

[5. Creating Plugins and Modules](#)

[Introduction](#)

[Creating and using your own plugin](#)

[How to do it...](#)

[How it works...](#)

[Building a flexible registration module](#)

[How to do it...](#)

[How it works...](#)

Using the same model for different applications

[How to do it...](#)

[How it works...](#)

Managing module dependencies

[How to do it...](#)

[How it works...](#)

Adding private module repositories using Amazon S3

[How to do it...](#)

[How it works...](#)

6. Practical Module Examples

Introduction

Integrating a Play application with message queues

[How to do it...](#)

[How it works...](#)

Integrating a Play application with ElasticSearch

[How to do it...](#)

[How it works...](#)

Implementing token authentication using JWT

[How to do it...](#)

[How it works...](#)

7. Deploying Play 2 Web Apps

Introduction

Deploying a Play application on Heroku

[How to do it...](#)

[How it works...](#)

[Procfile](#)

[system.properties](#)

[There's more...](#)

Deploying a Play application on AWS Elastic Beanstalk

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Deploying a Play application on CoreOS and Docker](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Deploying a Play application with Dokku](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Deploying a Play application with Nginx](#)

[How to do it...](#)

[How it works...](#)

[8. Additional Play Information](#)

[Introduction](#)

[Testing with Travis CI](#)

[How to do it...](#)

[How it works...](#)

[Monitoring with New Relic](#)

[How to do it...](#)

[How it works...](#)

[Integrating a Play application with AngularJS](#)

[How to do it...](#)

[How it works...](#)

[Integrating a Play application with Parse.com](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Creating a Play development environment using Vagrant](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Coding Play 2 web apps with IntelliJ IDEA 14](#)

[How to do it...](#)

[How it works...](#)

[Index](#)

Play Framework Cookbook Second Edition

Play Framework Cookbook Second Edition

Copyright © 2015 Packt Publishing

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

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

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

First published: August 2011

Second edition: June 2015

Production reference: 1090615

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78439-313-7

www.packtpub.com

Credits

Authors

Alexander Reelsen

Giancarlo Inductivo

Reviewers

Didier Bathily

Yannick De Turck

Jérôme Leleu

Davor Sauer

Commissioning Editor

Usha Iyer

Acquisition Editor

Shaon Basu

Content Development Editor

Samantha Gonsalves

Technical Editor

Vivek Pala

Copy Editors

Pranjali Chury

Shambhavi Pai

Project Coordinator

Sanchita Mandal

Proofreader

Safis Editing

Indexer

Monica Ajmera Mehta

Graphics

Disha Haria

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Authors

Alexander Reelsen is a software engineer living in Munich, Germany, where he has been working on different software systems such as a touristic booking engine, campaign management, and messaging platform, and a B2B e-commerce portal. He began using the Play Framework in 2009, and was immediately astonished by the sheer simplicity of this framework, while still dealing with pure Java. Other interests include scaling shared-nothing web architectures and NoSQL databases.

Being a system engineer, when he started playing around with Linux at the age of 14, Alexander got to know about software engineering during his studies and decided that web applications are more interesting than system administration.

When not doing something hacky, he enjoys playing a good game of basketball or street ball.

Sometimes he even tweets at <http://twitter.com/spinscale> and can be reached anytime at [<alexander@reelsen.net>](mailto:alexander@reelsen.net)

If I do not thank my girlfriend for letting me spend more time with the laptop than with her while writing this book, I fear unknown consequences. So, thanks Christine!

Uncountable appreciation goes to my parents for letting me spend days and (possibly not knowing) nights with my PC, and to my brother, Stefan, for introducing me to this IT stuff —this combination worked out pretty well.

Thanks for the inspiration, fun, and fellowship to all my current and former colleagues, mainly of course to the developers. They always open up views and opinions to make developing enjoyable.

Many thanks go out to the Play Framework developers and especially Guillaume, but also to the other core developers. Additionally, to all of the people on the mailing list who provided good answers to the many questions, and all the people working on tickets and helping to debug issues I had while writing this book.

My last appreciation goes to everyone at Packt Publishing, who was involved with this book.

Giancarlo Inductivo is the founder of DYNAMIC OBJX, a web and mobile development firm based in Manila, Philippines, and is the current technology head. Over the last 10 years, Giancarlo has been based in San Francisco and Manila, working in software development for companies in various industries such as online recruitment, social networking, Internet media, and software consultancy.

With DYNAMIC OBJX, Giancarlo has been utilizing the Play Framework for client projects since version 1.2 and has successfully launched many client projects using version 2.x for systems such as inventory management, logistics, financials, social media content aggregation, mobile content management, and so on.

Giancarlo is also interested in other technologies such as Apache Spark, Docker, and

Arduino.

Apart from technology, Giancarlo is also obsessed with his Stratocasters, the Golden State Warriors, and his two kids, Makayla and Elijah.

You can reach Giancarlo on Quora:

<http://www.quora.com/Giancarlo-Inductivo>

The writing and completion of this book would not have been possible at all if not for my Lord and Savior, Jesus Christ. *From Him, through Him and for Him.*

This book is dedicated to the memory of my late father, Tony, who never gave up on me, and to Mannyboy, who has always believed in me.

Lastly, this book is for my wife, Jill, in whom I have found a good spouse, a good life, and even more, the favor of God.

About the Reviewers

Didier Bathily is a Malian software engineer living in France, who founded an IT development company (<http://www.njin.fr>) along with friends in 2011.

His studies and passion for new technologies have given him some versatility in software development. Indeed, for the customers of njin, he develops modern web applications in scala/playframework, mobile applications for iOS and Android, and games for iOS and Mac OS X applications.

He can be reached on Github at <https://github.com/dbathily>.

He can also be found on Twitter at <https://twitter.com/dbathily>.

Yannick De Turck (@YannickDeTurck) is a software developer based in Belgium. He has more than 5 years of experience working on a variety of projects using different Java technologies, Play Framework 2 with Scala, and iOS development with Objective C and Swift.

He holds a bachelor's degree in information management and multimedia, and he currently works as a Java consultant and coach at Ordina Belgium. As a coach, he follows up on junior developers helping them to kickstart their career.

Yannick is passionate about learning new technologies and frameworks and has a keen interest in innovative technologies, Java architectures, reactive programming, and DevOps.

Jérôme Leleu is a software architect living in Paris, France.

A consultant for 7 years, he has worked in many different companies, fields, and with many different people. He has participated in many IT projects as a developer, technical leader, and project manager, mostly in J2E technology.

Working for a French telecom company, he is a technical leader for several websites. Among them is the web SSO, which supports very high traffic; millions of authentications from millions of users every day.

He is involved in open source development as a CAS (web SSO) chairman. Interested in security/protocol issues, he has developed several libraries to implement client support for protocols such as CAS, OAuth, SAML, and OpenID. This can be found at <http://www.pac4j.org>.

Davor Sauer (@davor_sauer) started to code at an early age. Since then, he has always been interested in new techniques, technologies, and finding ways to incorporate them. By the day, Davor works as a software engineer but by night, his mistress is open source technology and other programming languages.

Davor has a master's degree in information science and an Oracle certification for Java professional. He loves to speak about Java, a lot. He has spoken at several conferences and meetups mostly about things related to Java.

He has lots of experience in software architecture and implementation, so whenever it is possible, he tries to bring his passion for Java and new approaches to his work. As he says, he just loves to convert complex situations into simple and elegant solutions.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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

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



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

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Web applications have come a long way in the last 5-10 years. From the glory days of Geocities and Friendster, the advent of massively popular websites, in the case of social media sites, such as Facebook, and Twitter, and the more utilitarian web applications that are Software as a service (SAAS) offerings such as SalesForce and Github, there is no denying that with all these advancements in consumer and enterprise web software, there arises a need for a solid web technology platform to build on top of, not only for end-user web clients but also with a variety of sophisticated and complex backend services, all of which can compose the modern web application.

This is where Play Framework 2.0 comes in to the scene. Play provides developers with a powerful and mature web development platform that is lightweight and stateless by nature, which was made with development speed and web application scalability in mind.

This book aims to give readers a deeper understanding of different parts of Play Framework through concise code recipes based on very common use cases and scenarios. You will be guided through the basic concepts and abstractions used in Play, and we will dive into more advanced and relevant topics such as creating RESTful APIs, using a third-party cloud storage service to store uploaded images, posting messages to external message queues, and deploying Play web applications using Docker.

By providing relevant recipes, this book hopes to equip developers with the necessary building blocks required for creating the next Facebook or SalesForce using Play Framework.

What this book covers

[Chapter 1](#), *Basics of Play Framework*, introduces Play Framework and its capabilities. This chapter also introduces the essential Play components, such as the Controller and View templates. Finally, we cover how to do unit testing for a Play Framework model and controller classes.

[Chapter 2](#), *Using Controllers*, discusses Play Controllers in depth. This chapter demonstrates how to use controllers with other web components such as request filters, sessions, and JSON. It also touches on how Play and Akka can be utilized together from a controller.

[Chapter 3](#), *Leveraging Modules*, looks into utilizing official Play 2 modules as well as other third-party modules. This should help developers speed up their development by reusing and integrating existing modules.

[Chapter 4](#), *Creating and Using Web APIs*, discusses how to create secure RESTful API endpoints with Play. This chapter also discusses how to consume other web-based APIs using the Play WS library.

[Chapter 5](#), *Creating Plugins and Modules*, discusses how to write Play Modules and Plugins, and also tells us how to publish Play modules to a private repository on Amazon S3.

[Chapter 6](#), *Practical Module Examples*, adds to the previous chapter regarding Play Modules and discusses more practical examples of integrating modules such as message queues and Search Services.

[Chapter 7](#), *Deploying Play 2 Web Apps*, discusses deployment scenarios for different environments for Play web applications using tools such as Docker and Dokku.

[Chapter 8](#), *Additional Play Information*, discusses topics relevant to developers such as integrating with IDEs and using other third-party Cloud services. This chapter also discusses how to build a Play developer environment from scratch using Vagrant.

What you need for this book

You will need the following software to work with recipes in this book:

- Java Development Kit 1.7
- Typesafe Activator
- Mac OS X Terminal
- Cygwin
- Homebrew
- curl
- MongoDB
- Redis
- boot2docker
- Vagrant
- IntelliJ IDEA

Who this book is for

This book is aimed at advanced developers who are looking to harness the power of Play 2.x. This book will also be useful for professionals looking to dive deeper into web development. Play 2.x is an excellent framework to accelerate your learning of advanced topics.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

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

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: ” Utilize this new filter by declaring it in the app/Global.scala file.”

A block of code is set as follows:

```
// Java
return Promise.wrap(ask(fileReaderActor, words, 3000)).map(
  new Function<Object, Result>() {
    public Result apply(Object response) {
      return ok(response.toString());
    }
  });
);
```

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

```
GET /dashboard controllers.Application.dashboard
GET /login controllers.Application.login
```

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

```
$ curl -v http://localhost:9000/admins
```

New terms and important words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: ” Access this new URL route again using the same web browser and you see the text **Found userPref: tw.**”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from

http://www.packtpub.com/sites/default/files/downloads/1234OT_ColorImages.pdf.

Errata

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

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

Piracy

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

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

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

Questions

If you have a problem with any aspect of this book, you can contact us at [<questions@packtpub.com>](mailto:questions@packtpub.com), and we will do our best to address the problem.

Chapter 1. Basics of Play Framework

In this chapter, we will cover the following recipes:

- Installing Play Framework
- Creating a Play application using Typesafe Activator
- Using the Play console
- Working with modules
- Working with controllers and routes
- Using Action parameters in controllers
- Using reverse routing and redirects
- Working with View templates
- Using helper tags
- Using View layouts and Includes
- Working with XML and text files
- Using Ebean (Java) with MySQL
- Using Anorm (Scala) and database evolutions with MySQL
- Using a form template and web actions
- Using a form validation
- Securing form submission
- Testing with JUnit (Java) and specs2 (Scala)
- Testing models
- Testing controllers

Introduction

Play is a developer-friendly and modern web application framework for both Java and Scala. This first chapter will take you through the steps in installing Play Framework for local development. This chapter will describe the Play application project directory structure, its various members and its function in a Play application.

This chapter will also introduce you to the Activator command, which replaces the old Play command. Activator is used for various stages during development, including compilation, downloading library dependencies, testing, and building. It is really quite similar to other build tools such as Ant or Maven.

This first chapter will also go about implementing **Model-View-Controller (MVC)** components available in Play Framework. This will be followed by source code to create controllers and routing actions using View templates and model components used to interface with an RDBMS (such as MySQL). This chapter will tackle basic HTTP forms, recognizing the importance of modern web applications being able to deal with user interactivity and data and how Play Framework provides various APIs to make life easier for developers.

By the end of the chapter, you should have a good grasp of how to implement basic web application functionalities such as form submissions and data access with MySQL, create URL routes to web actions, and create views composed of smaller, modular, and reusable view components.

Most of the recipes in this chapter assume that you have a level of familiarity with Java development, web application development, command-line interfaces, **Structured Query Language (SQL)**, development build tools, third-party library usage, dependency management, and unit testing.

Installing Play Framework

This recipe will guide you through installing Play Framework 2.3 for local development. This section will guide you on the prerequisite installations for Play Framework, such as the **Java Development Kit (JDK)**, and the necessary steps to ensure that Play Framework has access to the JDK's binaries.

Getting ready

Play Framework requires a JDK version of 6 or above. Head over to the Oracle website and download the appropriate JDK for your development machine at <http://www.oracle.com/technetwork/java/javase/downloads/index.html>.

Once you have downloaded a suitable JDK, ensure that the binary folder is added to the system path:

```
$ export JAVA_PATH=/YOUR/INSTALLATION/PATH  
$ export PATH=$PATH:$JAVA_HOME/bin
```

You can also refer to Oracle's online documentation for more information regarding setting environment variables at http://docs.oracle.com/cd/E19182-01/820-7851/inst_cli_jdk_javahome_t/index.html.

Here's how you can verify that the JDK is now accessible in the system path:

```
$ javac -version  
javac 1.7.0_71  
  
$ java -version  
java version "1.7.0_71"  
Java(TM) SE Runtime Environment (build 1.7.0_71-b14)  
Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
```

How to do it...

As of Play 2.3.x, Play is now distributed using a tool called Typesafe Activator (<http://typesafe.com/activator>), install it using following steps:

1. Download the *Typesafe Reactive Platform* distribution at <https://typesafe.com/platform/getstarted> and unzip it at your desired location that has write access.
2. After downloading and unzipping the distribution, add the Activator installation directory to your system path:

```
$ export ACTIVATOR_HOME=</YOUR/INSTALLATION/PATH>
$ export PATH=$PATH:$ACTIVATOR_HOME
```

3. Now, verify that Activator is now accessible in the system path:

```
$ activator --version
sbt launcher version 0.13.5
```

4. You should now be able to create a Play application using the activator command:

```
$ activator new <YOUR_APP_NAME>
```


Creating a Play application using Typesafe Activator

Once you have a JDK and Activator installed and properly configured, you should be ready to create Play 2.3.x applications. Beginning with Play 2.0, developers are now able to create Java- or Scala-based Play applications. Activator provides many Play project templates for both Java and Scala. For the first project, let us use the basic project templates. We will also be using the command-line interface of Activator across all recipes in this cookbook.

How to do it...

You need to perform the following for creating the templates for both Java and Scala:

- For Java, let's use the play-java template and call our first application `foo_java` by using the following command:

```
$ activator new foo_java play-java
```

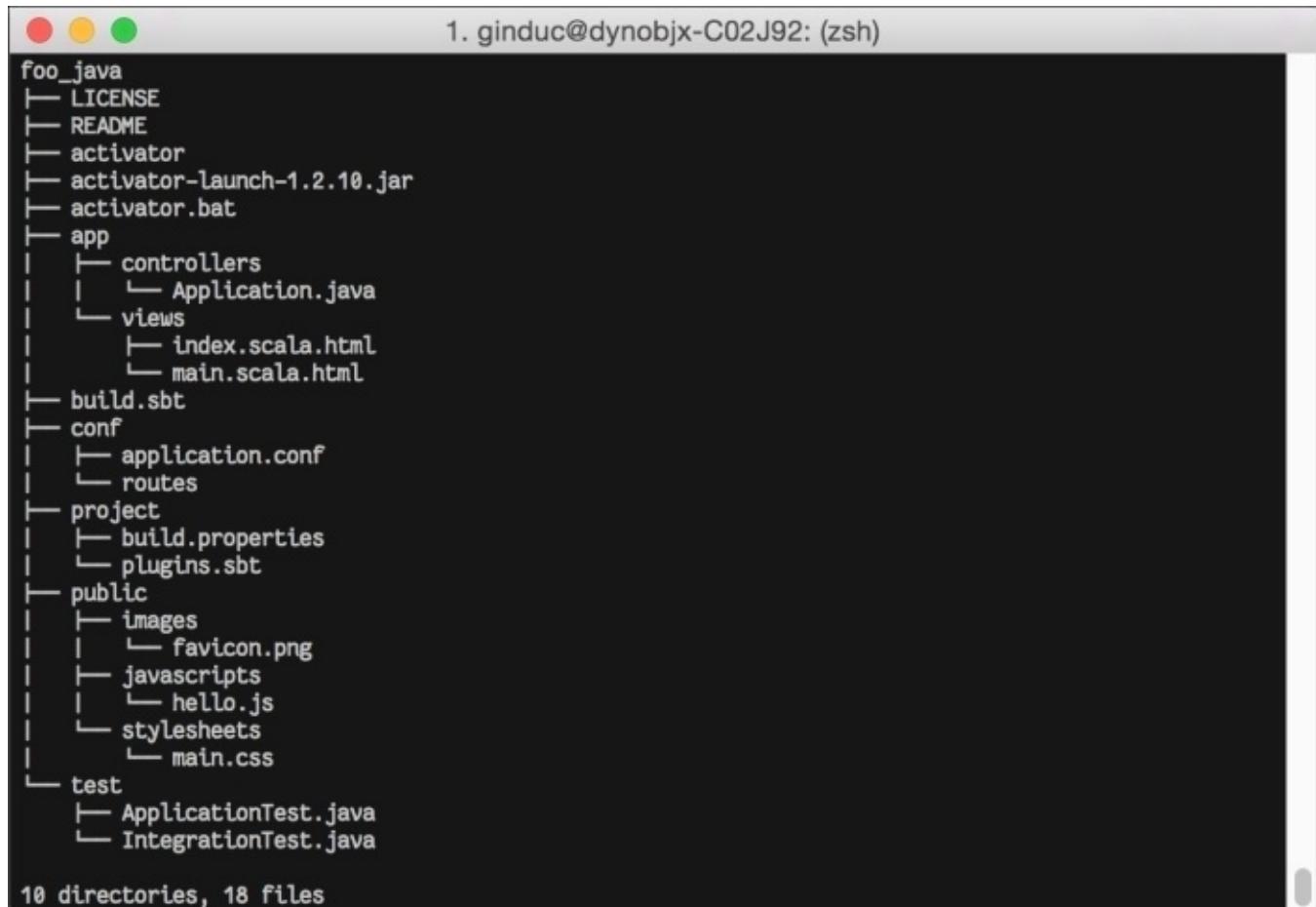
- For Scala, let's use the play-scala template and call our first application `foo_scala` by using the following command:

```
$ activator new foo_scala play-scala
```

How it works...

This Activator command creates the project's root directory (`foo_java` or `foo_scala`) and creates all the relevant subdirectories, config files, and class files:

The following screenshot shows the root directory for `foo_java`:



A screenshot of a terminal window titled "1. ginduc@dynobjx-C02J92: (zsh)". The window displays the directory structure of a generated Scala project named "foo_java". The structure includes sub-directories like LICENSE, README, activator, app, build.sbt, conf, project, public, and test, along with various configuration and source files. At the bottom of the terminal, the message "10 directories, 18 files" is visible.

```
foo_java
├── LICENSE
├── README
├── activator
├── activator-launch-1.2.10.jar
├── activator.bat
└── app
    ├── controllers
    │   └── Application.java
    ├── views
    │   ├── index.scala.html
    │   └── main.scala.html
    └── build.sbt
├── conf
│   ├── application.conf
│   └── routes
└── project
    ├── build.properties
    └── plugins.sbt
└── public
    ├── images
    │   └── favicon.png
    ├── javascripts
    │   └── hello.js
    └── stylesheets
        └── main.css
└── test
    ├── ApplicationTest.java
    └── IntegrationTest.java

10 directories, 18 files
```

The following screenshot shows the root directory for `foo_scala`:

```

1. ginduc@dynobjx-C02J92: (zsh)
foo_scala
├── LICENSE
├── README
├── activator
├── activator-launch-1.2.10.jar
├── activator.bat
└── app
    ├── controllers
    │   └── Application.scala
    ├── views
    │   ├── index.scala.html
    │   └── main.scala.html
└── build.sbt
└── conf
    ├── application.conf
    └── routes
└── project
    ├── build.properties
    └── plugins.sbt
└── public
    ├── images
    │   └── favicon.png
    ├── javascripts
    │   └── hello.js
    └── stylesheets
        └── main.css
└── test
    ├── ApplicationSpec.scala
    └── IntegrationSpec.scala

10 directories, 18 files

```

As you notice, both the Java and Scala project template generated an almost identical list of files, except for class files that are generated as `.java` files for the `play_java` template and as `.scala` files for the `play_scala` template.

For the project's directory structure, one of the more important aspects of Play Framework is its adherence to the concept of convention over configuration. This is best reflected by the standard project directory structure of every Play application it follows:

1st Level	2nd Level	3rd Level	Description
app/			Application source files
	assets/		Compiled JavaScript or style sheets
		stylesheets/	Compiled style sheet (such as LESS or SASS)
		javascripts/	Compiled JavaScript (such as CoffeeScript)
	controllers/		Application request-response controllers
	models/		Application domain objects
	views/		Application presentation views
conf/			Application configuration files

public/		Publicly available assets
	stylesheets/	Publicly available style sheet files
	javascripts/	Publicly available JavaScript files
project/		Build configuration files (such as <code>Build.scala</code> and <code>plugins.sbt</code>)
lib/		Unmanaged libraries and packages
logs/		Log files
test/		Test source files

Source code, configuration files, and web assets are organized in a predefined directory structure, making it easy for the developer to navigate through the project directory tree and find relevant files in logical placements.

There's more...

Go to <http://typesafe.com/activator/templates> for a comprehensive list of available project templates.

Using the Play console

The Play console is a command-line interface tool used to build and run Play applications. It is important for every developer to be familiar with the available commands, such as `clean`, `compile`, `dependencies`, and `run`, to fully utilize the power of the Play console.

How to do it...

You need to perform the following to use the Play console for both Java and Scala:

1. After Activator finishes setting up the Play project, you can enter the Play console of your Play application.

- Use the following command for Java:

```
$ cd foo_java  
$ activator
```

- Use the following command for Scala:

```
$ cd foo_scala  
$ activator
```

2. Once you have entered the Play console, you can run your application in the development mode:

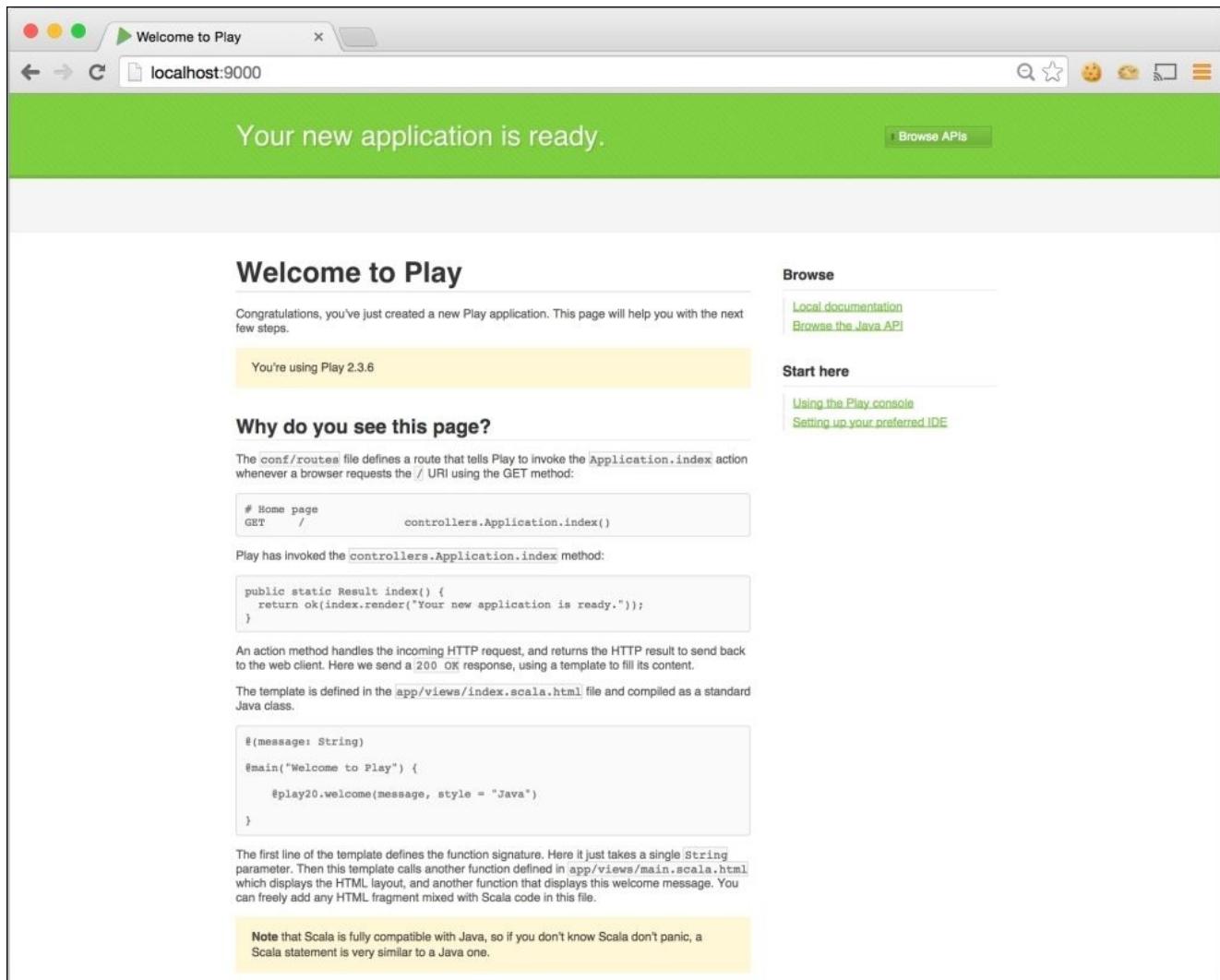
- Use the following command for Java:

```
[foo_java] $ run
```

- Use the following command for Scala:

```
[foo_scala] $ run
```

3. Now, open a web browser and head over to <http://localhost:9000>:



4. Use the following command line to start your Play application with Hot-Reloading enabled:

```
$ activator "~run"
```

5. Use the following command line to start your Play application on a different port:

```
$ activator "run 9001"
```

Note

Running your application in development mode configures your application to run in auto-reload, where Play will attempt to recompile any recent changes to the project files, removing the need to manually restart your application for every code edit. You are now ready to view your application using your web browser.

There's more...

You can also use the Play console to manually compile class files using the `compile` command in the activator console (use the `activator` command):

- Use the following command for Java:

```
[foo_java] $ compile
```

- Use the following command for Scala:

```
[foo_scala] $ compile
```

You can also run Play commands directly instead of using the Play console:

- Use the following command for Java:

```
$ cd foo_java  
$ activator compile  
$ activator run
```

- Use the following command for Scala:

```
$ cd foo_scala  
$ activator compile  
$ activator run
```

Use the following command to generate an eclipse project file for your existing Play application using Activator:

```
$ activator eclipse  
[info] Loading project definition from /private/tmp/foo_scala/project  
[info] Set current project to foo_scala (in build  
file:/private/tmp/foo_scala/)  
[info] About to create Eclipse project files for your project(s).  
[info] Compiling 5 Scala sources and 1 Java source to  
/private/tmp/foo_scala/target/scala-2.11/classes...  
[info] Successfully created Eclipse project files for project(s):  
[info] foo_scala
```

Use the following command to generate an IntelliJ IDEA project file for your existing Play application using Activator:

```
$ activator idea  
[info] Loading project definition from /private/tmp/foo_java/project  
[info] Set current project to foo_java (in build  
file:/private/tmp/foo_java/)  
[info] Creating IDEA module for project 'foo_java' ...  
[info] Running compile:managedSources...  
[info] Running test:managedSources...  
[info] Excluding folder target  
[info] Created /private/tmp/foo_java/.idea/IdeaProject.iml  
[info] Created /private/tmp/foo_java/.idea  
[info] Excluding folder /private/tmp/foo_java/target/scala-2.11/cache  
[info] Excluding folder /private/tmp/foo_java/target/scala-2.11/classes  
[info] Excluding folder /private/tmp/foo_java/target/scala-  
2.11/classes_managed
```

```
[info] Excluding folder /private/tmp/foo_java/target/native_libraries  
[info] Excluding folder /private/tmp/foo_java/target/resolution-cache  
[info] Excluding folder /private/tmp/foo_java/targetstreams  
[info] Excluding folder /private/tmp/foo_java/target/web  
[info] Created /private/tmp/foo_java/.idea_modules/foo_java.iml  
[info] Created /private/tmp/foo_java/.idea_modules/foo_java-build.iml
```


Working with modules

You can utilize other Play Framework or third-party modules in your Play application. This is easily done by editing the build file (`build.sbt`) and declaring library dependencies in the style of sbt dependency declaration.

How to do it...

You need to perform the following steps to declare a module:

1. Open the `build.sbt` file and add the following lines, using the notation of the group ID % module name % version while declaring library dependencies:

```
libraryDependencies ++= Seq(  
    jdbc,  
    "mysql" % "mysql-connector-java" % "5.1.28"  
)
```

2. Once the changes to `build.sbt` have been saved, head over to the command line and have Activator download the newly declared dependencies:

```
$ activator clean dependencies
```

How it works...

In this recipe, we declare what our Play application will need and reference the **Java Database Connectivity (JDBC)** module provided by Play Framework and the MySQL Java Connector module provided by MySQL. Once we have our modules declared, we can run the activator dependencies command to make Activator download all declared dependencies from the public Maven repositories and store them in the local development machine.

There's more...

Please refer to the Play Framework website for a complete list of official Play modules (<https://www.playframework.com/documentation/2.3.x/Modules>). You can also refer to the Typesafe official release repository for other useful plugins and modules at your disposal (<http://repo.typesafe.com/typesafe/releases/>).

Working with controllers and routes

Play applications use controllers to handle HTTP requests and responses. Play controllers are composed of actions that have specific functionality. Play applications use a router to map HTTP requests to controller actions.

How to do it...

To create a new page, which prints out “Hello World” for a Play Java project, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
$ activator "~run"
```

2. Edit `foo_java/app/controllers/Application.java` by adding the following action:

```
public static Result hello() {  
    return ok("Hello World");  
}
```

3. Edit `foo_java/conf/routes` by adding the following line:

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

4. View your new hello page using a web browser:

```
http://localhost:9000/hello
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
$ activator "~run"
```

2. Edit `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def hello = Action {  
    Ok("Hello World")  
}
```

3. Edit `foo_scala/conf/routes` by adding the following line:

```
GET      /hello      controllers.Application.hello
```

4. View your new hello page using a web browser:

```
http://localhost:9000/hello
```

How it works...

In this recipe, we enumerated the steps necessary to create a new accessible page by creating a new web action in a controller and defined this new page's URL route by adding a new entry to the `conf/routes` file. We should now have a "Hello World" page, and all without having to reload the application server.

Using Action parameters in controllers

Web applications should be able to accept dynamic data as part of their canonical URL. An example of this is GET operations of RESTful API web services. Play makes it easy for developers to implement this.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Edit `foo_java/app/controllers/Application.java` by adding the following action:

```
public static Result echo(String msg) {  
    return ok("Echoing " + msg);  
}
```

3. Edit `foo_java/conf/routes` by adding the following line:

```
GET      /echo/:msg      controllers.Application.echo(msg)
```

4. View your new echo page using a web browser:

```
http://localhost:9000/echo/foo
```

5. You should be able to see the text **Echoing foo**.

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Edit `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def echo(msg: String) = Action {  
    Ok("Echoing " + msg)  
}
```

3. Edit `foo_scala/conf/routes` by adding the following line:

```
GET      /echo/:msg      controllers.Application.echo(msg)
```

4. View your new echo page using a web browser:

```
http://localhost:9000/echo/bar
```

5. You should be able to see the text **Echoing bar**.

How it works...

In this recipe, we made edits to just two files, the application controller, `Application.java` and `Application.scala`, and `routes`. We added a new web action, which takes in a `String` argument `msg` in `Application.scala` and returns the contents of the message to the HTTP response. We then add a new entry in the `routes` file that declares a new URL route and declares the `:msg` route parameter as part of the canonical URL.

Using reverse routing and redirects

One of the more essential tasks for a web application is to be able to redirect HTTP requests, and redirecting HTTP with Play Framework is quite straightforward. This recipe shows how developers can use reverse routing to refer to defined routes.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Edit `foo_java/app/controllers/Application.java` by adding the following action:

```
public static Result helloRedirect() {  
    return  
    redirect(controllers.routes.Application.echo("HelloWorldv2"));  
}
```

3. Edit `foo_java/conf/routes` by adding the following line:

```
GET      /v2/hello      controllers.Application.helloRedirect()
```

4. View your new echo page using a web browser:

```
http://localhost:9000/v2/hello
```

5. You should be able to see the text **Echoing HelloWorldv2**.
6. Notice that the URL in the web browser has also redirected to
`http://localhost:9000/echo/Helloworldv2`

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Edit `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def helloRedirect() = Action {  
    Redirect(routes.Application.echo("HelloWorldv2"))  
}
```

3. Edit `foo_scala/conf/routes` by adding the following line:

```
GET      /v2/hello      controllers.Application.helloRedirect
```

4. View your new echo page using a web browser:

```
http://localhost:9000/v2/hello
```

5. You should be able to see the text **Echoing HelloWorldv2**.
6. Notice that the URL in the web browser has also redirected to
`http://localhost:9000/echo/Helloworldv2`

How it works...

In this recipe, we utilized reverse routes while referring to existing routes inside other action methods. This is handy, as we will not need to hard code rendered URL routes from within other action methods. We also utilized our first HTTP redirect, a very common web application function, by which we were able to issue a 302 HTTP redirect, a standard HTTP status code handled by all standard web servers.

Working with View templates

You expect to be able to send some data back to the View itself in web applications; this is quite straightforward with Play Framework. A Play View template is simply a text file that contains directives, web markup tags, and template tags. The `View Template` files also follow standard naming conventions and they are placed in predefined directories within the Play project directory, which makes it easier to manage template files.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Create the view file `products.scala.html` in `foo_java/app/views/`. Add the contents of the view file:

```
@(products: Collection[String])  
  
<h3>@products.mkString(", ")</h3>
```

3. Edit `foo_java/app/controllers/Application.java` by adding the following action:

```
private static final java.util.Map<Integer, String> productMap =  
new java.util.HashMap<Integer, String>();  
  
static {  
    productMap.put(1, "Keyboard");  
    productMap.put(2, "Mouse");  
    productMap.put(3, "Monitor");  
}  
  
public static Result listProducts() {  
    return ok(products.render(productMap.values()));  
}
```

4. Edit `foo_java/conf/routes` by adding the following line:

```
GET      /products      controllers.Application.listProducts
```

5. View the products page using a web browser:

```
http://localhost:9000/products
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Create the view file `products.scala.html` in `foo_scala/app/views/`. Add the contents of the view file:

```
@(products: Seq[String])  
  
<h3>@products.mkString(", ")</h3>
```

3. Edit `foo_scala/app/controllers/Application.scala` by adding the following action:

```
private val productMap = Map(1 -> "Keyboard", 2 -> "Mouse", 3 ->  
"Monitor")  
def listProducts() = Action {  
    Ok(views.html.products(productMap.values.toSeq))  
}
```

4. Edit `foo_scala/conf/routes` by adding the following line:

```
GET      /products      controllers.Application.listProducts
```

5. View the products page using a web browser:

```
http://localhost:9000/products
```

How it works...

In this recipe, we were able to retrieve a collection of data from the server side and display the contents of the collection in our View template. For now, we use a static collection of String objects to display in the View template instead of retrieving some data set from a database, which we will tackle in the upcoming recipes.

We introduced declaring parameters in View templates by declaring them in the first line of code in our view template and passing data into our View templates from the controller.

Using helper tags

View tags allow developers to create reusable view functions and components and make the management of views a lot simpler and easier.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Create the tag file `productsIterator.scala.html` in `foo_java/app/views/tags`
3. Add the contents of the tag file:

```
@(products: Collection[String])  
  
<ul>  
  @for(product <- products) {  
    <li>@product</li>  
  }  
</ul>
```

4. Edit `foo_java/app/views/products.scala.html` by adding the following block:

```
@import tags._  
  
@productsIterator(products)
```

5. Reload the products page using a web browser to see the new product listing, using an unordered list HTML tag:

`http://localhost:9000/products`

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Create the tag file `productsIterator.scala.html` in `foo_scala/app/views/tags`
3. Add contents of the tag file:

```
@(products: Seq[String])  
  
<ul>  
  @for(product <- products) {  
    <li>@product</li>  
  }  
</ul>
```

4. Edit `foo_scala/app/views/products.scala.html` by adding the following block:

```
@import tags._  
  
@productsIterator(products)
```

5. Reload the products page using a web browser to see the new products listing, using an unordered list HTML tag:

`http://localhost:9000/products`

How it works...

In this recipe, we were able to create a new view tag in `app/views/tags`. We proceeded to use this tag in our View template.

First, we created a new tag that receives a collection of product titles, from which it is then displayed in the template as an unordered list. We then imported the tag in our products View template and invoked the helper function by calling it using its filename (`@productsIterator(products)`).

Using View layouts and Includes

For this recipe, we will create a main layout View template that will include a defined header and footer view. This will allow our View template to inherit a consistent look and feel by including this main View template and manage all UI changes in a single file. Our Products view will utilize the main layout view in this example.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Create the main layout view file `mainLayout.scala.html` in `foo_java/app/views/common`
3. Add the contents of the main layout view file:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>@title</title>
  </head>
  <body>
    <header>@header()</header>
    <section class="content">@content</section>
    <footer>@footer()</footer>
  </body>
</html>
```

4. Create the header view file `header.scala.html` in `foo_java/app/views/common` and add the following code:

```
<div>
  <h1>Acme Products Inc</h1>
</div>
```

5. Create the footer view file `footer.scala.html` in `foo_java/app/views/common` and add the following code:

```
<div>
  Copyright 2014
</div>
```

6. Edit the products view file `foo_java/app/views/products.scala.html` to use the main layout View template by replacing all the file contents with the following code:

```
@(products: Collection[String])

@import tags._
@import common._

@mainLayout(title = "Acme Products") {
  @productsIterator(products)
}
```

7. Reload the updated products page using a web browser:

```
http://localhost:9000/products
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.

2. Create the main layout view file `mainLayout.scala.html` in `foo_scala/app/views/common`
3. Add the contents of the main layout view file:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>@title</title>
  </head>
  <body>
    <header>@header()</header>
    <section class="content">@content</section>
    <footer>@footer()</footer>
  </body>
</html>
```

4. Create the header view file `header.scala.html` in `foo_scala/app/views/common` and add the following code:

```
<div>
  <h1>Acme Products Inc</h1>
</div>
```

5. Create the footer view file `footer.scala.html` in `foo_scala/app/views/common` and add the following code:

```
<div>
  Copyright 2014
</div>
```

6. Edit the products view file `foo_scala/app/views/products.scala.html` to use the main layout view template by replacing all the file contents with following code:

```
@(products: Seq[String])

@import tags._
@import common._

@mainLayout(title = "Acme Products") {
  @productsIterator(products)
}
```

7. Reload the updated products page using a web browser:

`http://localhost:9000/products`

How it works...

In this recipe, we created a main layout view template that can be reused throughout the Play application. A common layout view removes the need to duplicate the view logic in related views and makes it a lot easier to manage parent views and child views.

Working with XML and text files

Using View templates, we are also able to respond to HTTP requests in other content types such as text files and XML data formats. Play Framework has native handlers for XML and text file content type responses.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
$ activator "~run"
```

2. Create the text-based view template file `products.scala.txt` in `app/views/` and add the following content:

```
@(productMap: Map[Integer, String])
@for((id, name) <- productMap) {
    The Product '@name' has an ID of @id
}
```

3. Create the XML-based view template file `products.scala.xml` in `app/views/` and add the following content:

```
@(productMap: Map[Integer, String]) <products>
@for((id, name) <- productMap) {
    <product id="@id">@name</product>
}
</products>
```

4. Edit `foo_java/app/controllers/Application.java` by adding the following actions:

```
public static Result listProductsAsXML() {
    return ok(views.xml.products.render(productMap));
}

public static Result listProductsAsTXT() {
    return ok(views.txt.products.render(productMap));
}
```

5. Edit `foo_java/conf/routes` by adding the following lines:

```
GET      /products.txt      controllers.Application.listProductsAsTXT()
GET      /products.xml      controllers.Application.listProductsAsXML()
```

6. View the new routes and actions using a web browser:

- `http://localhost:9000/products.txt` and,
- `http://localhost:9000/products.xml`

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
$ activator "~run"
```

2. Create the text-based view template file `products.scala.txt` in `app/views/` and add the following content:

```
@(productMap: Map[Int, String])
```

```
@for((id, name) <- productMap) {  
    The Product '@name' has an ID of @id  
}
```

3. Create the XML-based view template file `products.scala.xml` in `app/views/` and add the following content:

```
@(productMap: Map[Int, String]) <products>  
@for((id, name) <- productMap) {  
    <product id="@id">@name</product>  
}  
</products>
```

4. Edit `foo_scala/app/controllers/Application.scala` by adding the following actions:

```
def listProductsAsTXT = Action {  
    Ok(views.txt.products(productMap))  
}  
  
def listProductsAsXML = Action {  
    Ok(views.xml.products(productMap))  
}
```

5. Edit `foo_scala/conf/routes` by adding the following lines:

```
GET      /products.txt      controllers.Application.listProductsAsTXT  
GET      /products.xml      controllers.Application.listProductsAsXML
```

6. View the new routes and actions using a web browser:

- `http://localhost:9000/products.txt` and
- `http://localhost:9000/products.xml`

How it works...

In this recipe, we utilized build-in support for other content types in Play Framework. We created new URL routes and web actions to be able to respond to requests for data in XML or text file formats. By following file naming standards and convention for views, we were able to create view templates in HTML, XML, and text file formats, which Play automatically handles, and then adds the appropriate content type headers in the HTTP response.

Using Ebean (Java) with MySQL

Play Framework 2.x includes an object-relational mapping tool called **Ebean** for Java-based Play applications. To be able to use Ebean, ensure that Ebean and a suitable MySQL driver are declared as project dependencies in `foo_java/build.sbt`.

For this recipe, we will be utilizing Ebean with database evolutions. Play Framework 2.x gives developers a way to manage database migrations. Database migrations are useful for tracking schema changes during the course of application development. Database evolutions are enabled by default but can be disabled in `conf/application.conf` with the following settings:

```
evolutionplugin=disabled
```

Evolution scripts are stored in the `conf/evolutions/default/` directory. For more information regarding database evolutions, please refer to Play's online documentation at

<https://www.playframework.com/documentation/2.3.x/Evolutions>.

How to do it...

You need to perform the following steps to utilize Ebean:

1. Add the Ebean dependency in build.sbt:

```
libraryDependencies ++= Seq(
    javaJdbc,    javaEbean,
    "mysql" % "mysql-connector-java" % "5.1.28"
)
```

2. Ensure that Ebean and MySQL are configured properly in conf/application.conf:

```
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://<YOUR_MYSQL_HOST>/<YOUR_DB>"
db.default.user=<YOUR_USER>
db.default.password=<YOUR_PASSWORD>

ebean.default="models.*"
```

3. For the next recipes, we need to create our product table in our MySQL database. Create our first database evolution file 1.sql in conf/evolutions/default and add the following SQL statements:

```
# --- !Ups
CREATE TABLE Products (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Products;
```

4. The next step is to create the Ebean model for our entity Product:

```
package models;

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

@Entity
@Table(name = "Products")
public class Product extends Model {

    @Id
    public Long id;

    @Column
    @Constraints.Required
    public String name;

    public static Finder<Long, Product> find = new Finder<Long,
```

```

Product>(
    Long.class, Product.class
);

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

The following displays various database-oriented operations using Ebean.

Tip

Downloading the example code

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

Creating a record

The following code snippet will create a new record:

```

Product product = new Product();
product.name = "Apple iPhone";
product.save();

```

Updating a record

The following code snippet will update a record:

```

Product forUpdate = Product.find.ref(1L);
forUpdate.name = "Apple iPhone 6";
forUpdate.update();Deleting a record:
Product.find.ref(1L).delete();

```

Querying a record

The following code snippet will query a record:

```

Product p = Product.find.byId(1L);

```

Retrieving a record

The following code snippet will retrieve a record:

```
List<Product> products = Product.find.all();
```


Using Anorm (Scala) and database evolutions with MySQL

Play Framework 2.x includes Anorm, a useful data access library for Scala-based Play applications. To be able to use Anorm, ensure that Anorm and a suitable MySQL driver are declared as project dependencies in `foo_scala/build.sbt`.

For this recipe, we will be utilizing Anorm with database evolutions. Play Framework 2.x gives developers a way to manage database migrations. Database migrations are useful for tracking schema changes during the course of application development. Database evolutions are enabled by default but can be disabled in `conf/application.conf` using the following settings:

```
evolutionplugin=disabled
```

Evolution scripts are stored in the `conf/evolutions/default/` directory. For more information regarding database evolutions, please refer to Play's online documentation at <https://www.playframework.com/documentation/2.3.x/Evolutions>.

You need to perform the following steps to utilize Anorm:

1. Add the Anorm dependency to `build.sbt`:

```
libraryDependencies ++= Seq(
  jdbc,
  anorm,
  "mysql" % "mysql-connector-java" % "5.1.28"
)
```

2. Ensure that Anorm and MySQL are configured properly in `conf/application.conf`:

```
db.default.driver= com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://localhost/YOUR_DB"
db.default.user=YOUR_USER
db.default.password=YOUR_PASSWORD
```

3. For the next recipes, we need to create our products table in our MySQL database. Create our first database evolution file `1.sql` in `conf/evolutions/default` and add the following SQL statements:

```
# --- !Ups
CREATE TABLE Products (
  id INT NOT NULL AUTO_INCREMENT,
  name VARCHAR(100) NOT NULL,
  PRIMARY KEY (id)
);

# --- !Downs
DROP TABLE Products;
```

There's more...

The following sections display various database-oriented operations using Anorm.

Creating a new record

The following code snippet will create a new record:

```
DB.withConnection { implicit c =>
  SQL("INSERT INTO Products(id, name) VALUES ({id}, {name});")
    .on('id -> product.id, 'name -> product.name).executeInsert()
}
```

Updating a record

The following code snippet will update a record:

```
DB.withConnection { implicit c =>
  SQL("UPDATE Products SET name = {name} WHERE id = {id};")
    .on('name -> product.name, 'id -> product.id).executeUpdate()
}
```

Deleting a record

The following code snippet will delete a record:

```
DB.withConnection { implicit c =>
  SQL("DELETE FROM Products WHERE id={id};")
    .on('id -> id).executeUpdate()
}Querying a record
```

The following code snippet will query a record:

```
DB.withConnection { implicit c =>
  SQL("SELECT * FROM Products WHERE id={id};")
    .on('id -> id).executeQuery().singleOpt(defaultParser)
}
```

The following code snippet will retrieve a record:

```
DB.withConnection { implicit c =>
  SQL("SELECT * FROM Products;").executeQuery().list(defaultParser)
}
```

Finally, we can combine all of these functions in a companion object called Product:

```
package models

import play.api.db.DB
import play.api.Play.current
import anorm._
import anorm.SqlParser.{str, int}

case class Product(id: Long, name: String)
```

```

object Product {
    val defaultParser = int("id") ~ str("name") map {
        case id ~ name => Product(id, name)
    }

    def save(product: Product) = {
        DB.withConnection { implicit c =>
            SQL("INSERT INTO Products(id, name) VALUES ({id}, {name});")
                .on('id -> product.id, 'name ->
product.name).executeInsert()
        }
    }

    def update(product: Product) = {
        DB.withConnection { implicit c =>
            SQL("UPDATE Products SET name = {name} WHERE id = {id};")
                .on('name -> product.name, 'id ->
product.id).executeUpdate()
        }
    }

    def delete(id: Long) = {
        DB.withConnection { implicit c =>
            SQL("DELETE FROM Products WHERE id={id};")
                .on('id -> id).executeUpdate()
        }
    }

    def get(id: Long) = {
        DB.withConnection { implicit c =>
            SQL("SELECT * FROM Products WHERE id={id};")
                .on('id -> id).executeQuery().singleOpt(defaultParser)
        }
    }

    def all = {
        DB.withConnection { implicit c =>
            SQL("SELECT * FROM Products;").executeQuery().list(defaultParser)
        }
    }
}

```


Using a form template and web action

As with the majority of web applications, there will always be a need to accept an HTTP form, be it a registration form or a login form. Play Framework provides helper classes to manage and process HTTP form submissions. In this recipe, we will go over the steps to create a simple form and map the web action assigned to handle this form submission. We will also utilize the flash scope, which allows us to use the flash object to send messages from the controller to the view template on a per-request basis.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Create the form view template file `app/views/product/form.scala.html` and add the following contents:

```
@(productForm: Form[models.Product])  
  
@import common._  
  
@mainLayout(title = "New Product") {  
  
    @if(flash.get("success") != null) {  
        <p>@flash.get("success")</p>  
    }  
  
    @if(productForm.hasGlobalErrors) {  
        <ul>  
            @for(error <- productForm.globalErrors) {  
                <li>@error.message</li>  
            }  
        </ul>  
    }  
  
    @helper.form(action = routes.Products.postForm()) {  
        @helper.inputText(productForm("id"))  
        @helper.inputText(productForm("name"))  
  
        <input type="submit">  
    }  
}
```

3. Create the products controller `foo_java/app/controllers/Products.java` and add the following import, action, and Play form blocks:

```
package controllers;  
  
import play.*;  
import play.mvc.*;  
import play.data.*;  
import views.html.*;  
import models.*;  
  
public class Products extends Controller {  
  
    public static Result create() {  
        Form<Product> form = Form.form(Product.class);  
        return ok(views.html.product.form.render(form));  
    }  
  
    public static Result postForm() {  
        Form<Product> productForm =  
Form.form(Product.class).bindFromRequest();
```

```

        if (productForm.hasErrors()) {
            return
        badRequest(views.html.product.form.render(productForm));
    } else {
        Product product = productForm.get();
        product.save();

        flash("success", "Product saved!");
        return redirect(controllers.routes.Products.create());
    }
}
}
}

```

4. Edit `foo_java/conf/routes` by adding the following line:

```

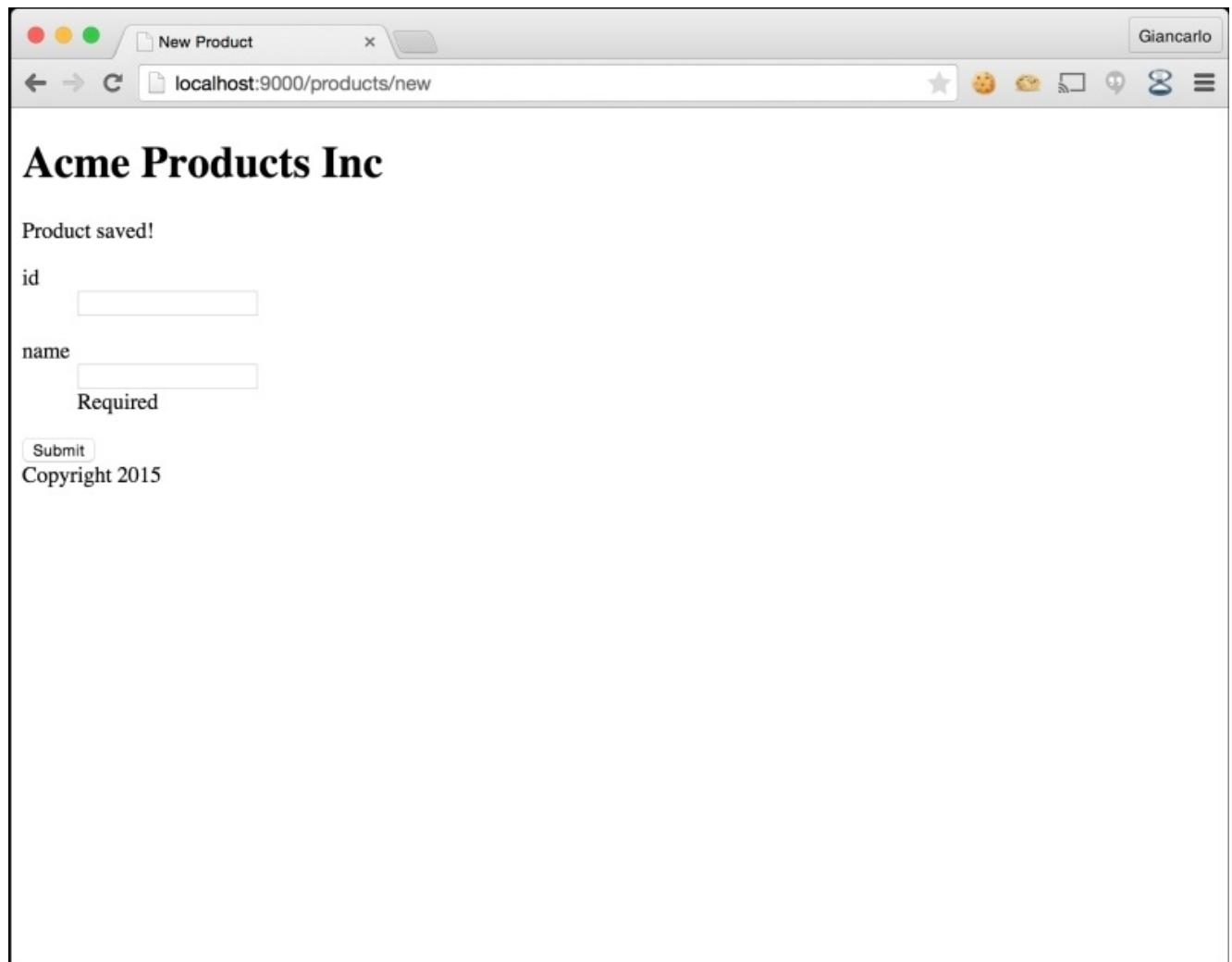
GET      /products/new      controllers.Products.create()
POST     /products          controllers.Products.postForm()

```

5. View your new product form using a web browser:

`http://localhost:9000/product/new`

6. Fill in a name for your new product and hit **submit**. You should now receive the success message:



For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Create the form view template file `app/views/product/form.scala.html` and add the following contents:

```
@(productForm: Form[models.Product])(implicit flash: Flash)

@import common._

@mainLayout(title = "New Product") {
  @flash.get("success").map { message =>
    <p>@message</p>
  }

  @if(productForm.hasGlobalErrors) {
    <ul>
      @for(error <- productForm.globalErrors) {
        <li>@error.message</li>
      }
    </ul>
  }

  @helper.form(action = routes.Products.postForm()) {
    @helper.inputText(productForm("id"))
    @helper.inputText(productForm("name"))

    <input type="submit">
  }
}
```

3. Create the products controller `foo_scala/app/controllers/Products.scala` and add the following import, action, and Play form blocks:

```
import play.api._
import play.api.mvc._
import models._
import play.api.data._
import play.api.data.Forms._

val form = Form(
  mapping(
    "id" -> longNumber,
    "name" -> text
  )(Product.apply)(Product.unapply)
)

def create = Action { implicit request =>
  Ok(views.html.product.form(form))
}

def postForm = Action { implicit request =>
  form.bindFromRequest.fold(
    formWithErrors => {
      BadRequest(views.html.product.form(formWithErrors))
    },
  )
}
```

```
        product => {Product.save(product)
                      Redirect(routes.Products.create).flashing("success" ->
                  "Product saved!")
                }
            }
        }
```

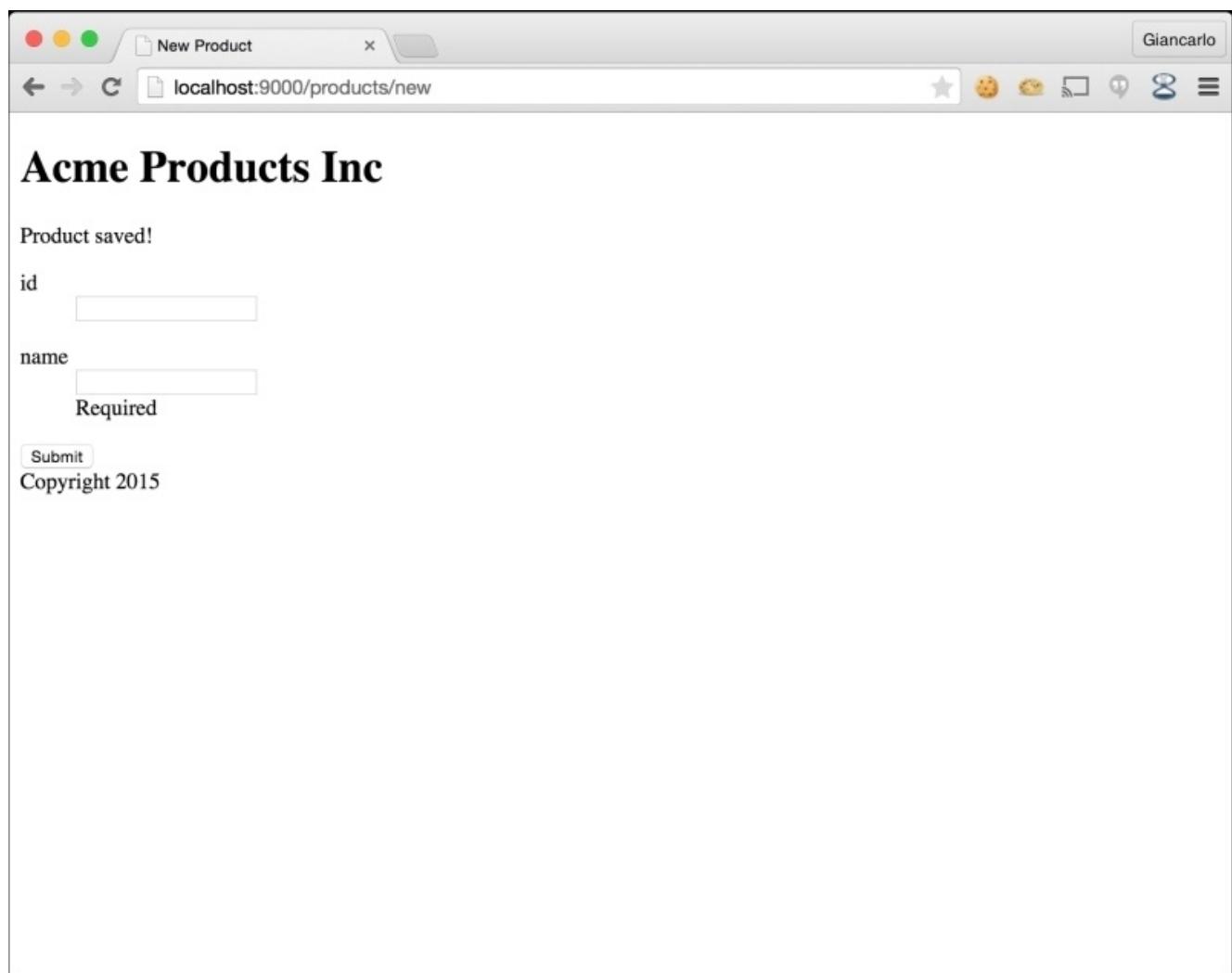
4. Edit `foo_scala/conf/routes` by adding the following lines:

```
GET      /products/new    controllers.Products.create
POST     /products        controllers.Products.postForm
```

5. View your new Product form using a web browser:

`http://localhost:9000/product/new`

6. Fill in a name for your new product and click on **submit**. You should now receive the following success message:



How it works...

In this recipe, we were able to create our first HTTP form using Play Framework. This recipe included steps in creating an HTML form view template and our Products controller. We declared two web actions and two URL routes and created the Play form object, which we used to bind request parameters to our model, Fruit. We were able to load the web form by accessing `http://localhost:9000/Products/new` on a web browser. After filling out our form details, we submitted the form itself and received a notification from the Products controller.

Using form validation

Play Framework provides an easy way to validate form submissions. For Play Java, we will add the validation to the model, which will check for a submitted field's length and return an error message if the validate condition is not satisfied. For Play Scala, we will add the form validation to the form object itself and define the validation parameters for each form field there.

How to do it...

For Java, we need to take the following steps:

1. Edit the Product model, `foo_java/app/models/Product.java` and add the `validate()` method:

```
public String validate() {  
    if (name.length() < 3 || name.length() > 100) {  
        return "Name must be at least 3 characters or a maximum of 100  
characters";  
    }  
    return null;  
}
```

2. Reload the Product form using a web browser:

`http://localhost:9000/products/new`

3. The product form should now accept only product names with a minimum of three characters and a maximum of 100, as shown in the following screenshot:

The screenshot shows a web browser window titled "New Product". The address bar displays "localhost:9000/products". The main content area is titled "Acme Products Inc". It contains a form with two fields: "id" (containing "100001") and "name" (containing "12"). Below the "name" field is the error message "Required". At the bottom of the form are "Submit" and "Cancel" buttons, and the text "Copyright 2015". Above the browser window, the Mac OS X menu bar is visible with the user "Giancarlo".

For Scala, we need to take the following steps:

1. Edit the products controller `foo_scala/app/controllers/Products.scala` and modify how the form is declared:

```
val form = Form(
  mapping(
    "id" -> longNumber,
    "name" -> nonEmptyText(minLength = 3, maxLength = 100)
  )(Product.apply)(Product.unapply)
)
```

2. Reload the Products form using a web browser:

<http://localhost:9000/products/new>

3. The product form should now accept only fruit names with a minimum of three characters and a maximum of 100, as shown in the following screenshot:

The screenshot shows a web browser window titled 'New Product'. The address bar displays 'localhost:9000/products/new'. The main content area is titled 'Acme Products Inc' and contains the following form fields:

- A message: 'Fruit saved!'
- A field labeled 'id' with a numeric input field.
- A field labeled 'name' with a text input field containing placeholder text: 'Minimum length: 3', 'Maximum length: 100', and 'Required'.
- A 'Submit' button.
- A copyright notice: 'Copyright 2015'.

How it works...

In this recipe, we added data validations for product name and the acceptable length submitted by users. For Java, we added a `validate()` method in the product model.

Our Java model can be validated by using JSR-303 JavaBean validation annotations and by defining a `validate()` method that Play invokes if it is present in the model class.

For Scala, we added data validation directives to the `Form` object in the controller. We used Play form helpers to define the minimum and maximum character count for the name property of the product.

Securing form submission

Play Framework has a CSRF filter module that developers can use to validate CSRF tokens during HTTP form submissions. This allows developers to be sure that the form was submitted with a valid session token and not tampered with in any way.

How to do it...

For Java, we need to take the following steps:

1. Add the Play filters module as a project dependency to build.sbt:

```
libraryDependencies += filters
```

2. Create a Global.java object file in the app/ directory:

```
import play.GlobalSettings;
import play.api.mvc.EssentialFilter;
import play.filters.csrf.CSRFFilter;

public class Global extends GlobalSettings {
    @Override
    public <T extends EssentialFilter> Class<T>[] filters() {
        return new Class[]{CSRFFilter.class};
    }
}
```

3. Declare the Global.java object in conf/application.conf:

```
application.global=Global
```

4. Update the template declaration by adding an implicit request object for the product form file app/views/product/form.scala.html:

```
@(productForm: Form[models.Product])
```

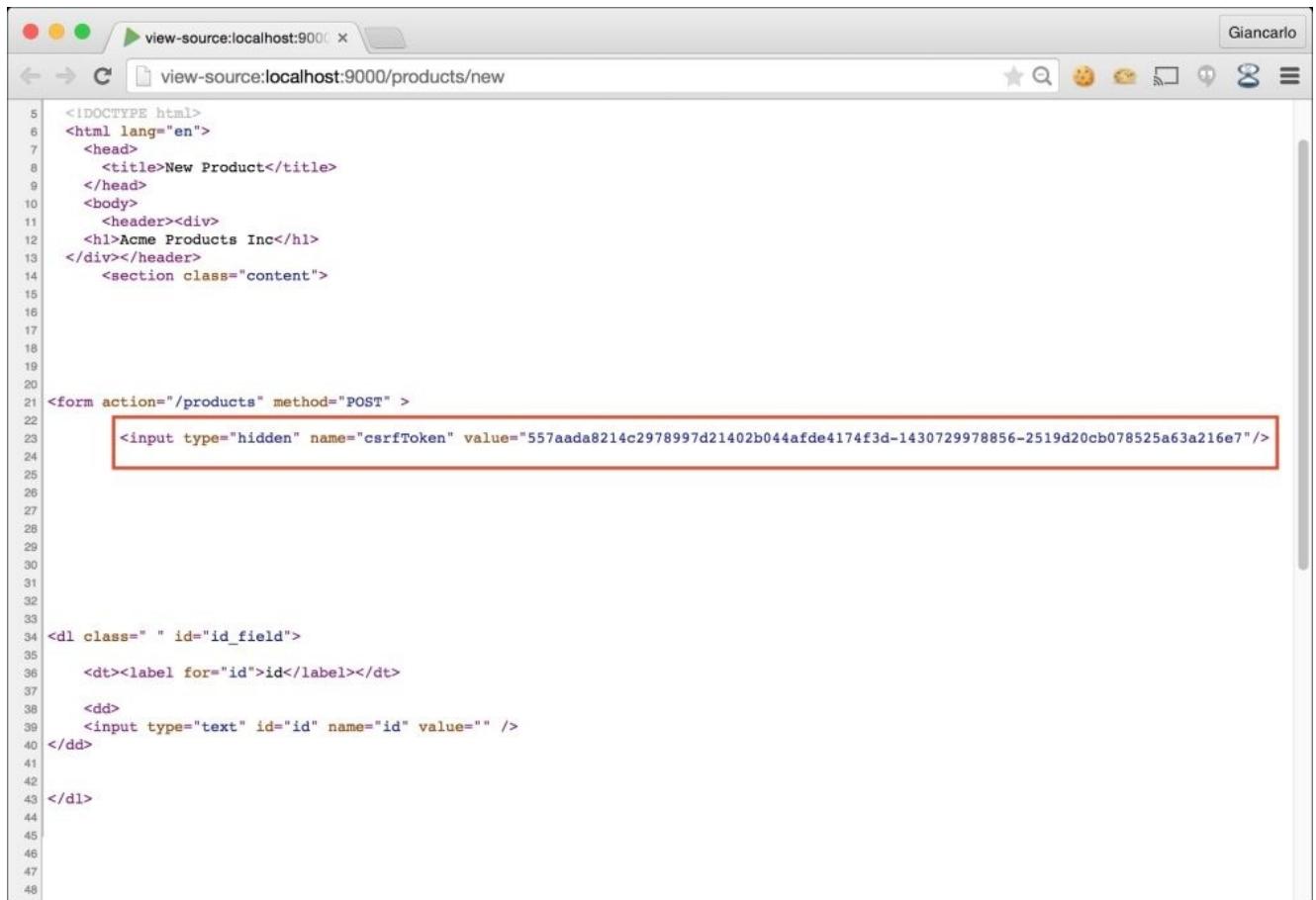
5. Add the CSRF token helper tag to the product form file app/views/product/form.scala.html:

```
@helper.form(action = routes.Products.postForm()) {
    @helper.CSRF.formField @* -Add the CSRF Token Helper Tag- *@
}
```

6. Reload the product form using a web browser:

```
http://localhost:9000/products/new
```

7. The product form should now contain a Play-generated CSRF token and should use this to validate form submissions, as shown in the following screenshot:



```

5  <!DOCTYPE html>
6  <html lang="en">
7  <head>
8      <title>New Product</title>
9  </head>
10 <body>
11     <header><div>
12         <h1>Acme Products Inc</h1>
13     </div></header>
14     <section class="content">
15
16
17
18
19
20 <form action="/products" method="POST" >
21
22     <input type="hidden" name="csrfToken" value="557aada8214c2978997d21402b044afde4174f3d-1430729978856-2519d20cb078525a63a216e7"/>
23
24
25
26
27
28
29
30
31
32
33 <dl class=" " id="id_field">
34     <dt><label for="id">id</label></dt>
35
36     <dd>
37         <input type="text" id="id" name="id" value="" />
38     </dd>
39 </dl>
40
41
42
43
44
45
46
47
48

```

For Scala, we need to take the following steps:

1. Add the Play filters module as a project dependency to `build.sbt`:

```
libraryDependencies += filters
```

2. Create a `Global.scala` object file in the `app/`:

```
import play.api._

object Global extends GlobalSettings { }
```

3. Declare the `Global.scala` object in `conf/application.conf`:

```
application.global=Global
```

4. Add the Play global CSRF filter by modifying the object declaration in `app/Global.scala`:

```
import play.api.mvc._
import play.filters.csrf._

object Global extends WithFilters(CSRFFilter()) with GlobalSettings
```

5. Update the template declaration by adding an implicit request object for the Product form file `app/views/product/form.scala.html`:

```
@(productForm: Form[models.Product])(implicit flash: Flash,  
request: play.api.mvc.Request[Any])
```

6. Add the CSRF token helper tag to the product form file

app/views/product/form.scala.html:

```
@helper.form(action = routes.Products.postForm()) {  
    @helper.CSRF.formField @* -Add the CSRF Token Helper Tag-*@  
}
```

7. Reload the product form using a web browser:

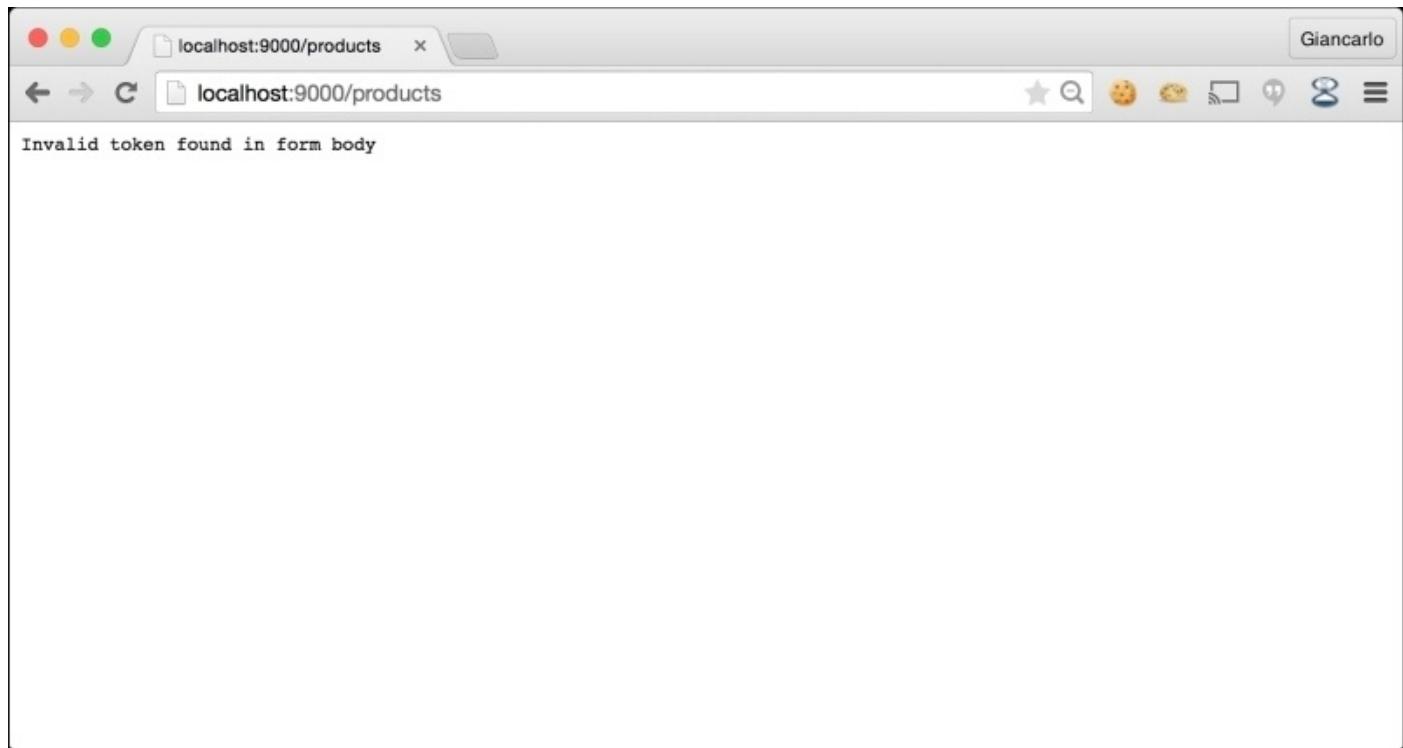
<http://localhost:9000/products/new>

8. The product form should now contain a Play-generated CSRF token and should use this to validate form submissions.

How it works...

In this recipe, we added the Play Framework filters module, which includes CSRF helpers. We added global CSRF support by declaring the `CSRFFilter` in the Play application global settings class, `app/Global.java` and `app/Global.scala`. The last step was to insert a CSRF token helper tag in our tag that the filter uses to validate form submissions.

Modifying or tampering with a valid CSRF token will now result in an error and will be rejected by Play, as shown in the following screenshot:



Testing with JUnit (Java) and specs2 (Scala)

It is quite important for a web framework to integrate testing as seamlessly as possible with the web framework itself. This minimizes the friction developers encounter when coding functional specs and writing tests to validate their work. For Play Java projects, we will be utilizing the popular test framework JUnit. We will be using it to do a simple unit test and to test our model and controller action. For Play Scala projects, we will be using specs2 to do a simple unit test and to test our model, a controller action, and a route mapping.

How to do it...

For Java, we need to take the following steps:

1. Create a new spec class, `ProductTest.java`, in `test/` and add the following content:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class ProductTest {

    @Test
    public void testString() {
        String str = "product";
        assertEquals(7, str.length());
    }
}
```

2. Run the first spec using Activator by running the command `test-only ProductTest`:

```
$ activator
[info] Loading project definition from
/private/tmp/foo_java/project
[info] Set current project to foo_java (in build
file:/private/tmp/foo_java/)
[foo_java] $ test-only ProductTest
[info] Compiling 3 Java sources to
/private/tmp/foo_java/target/scala-2.11/test-classes...
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 3 s, completed 09 29, 14 8:44:31 PM
```

For Scala, we need to take the following steps:

1. Create a new Spec class, `ProductSpec.scala`, in `test/` and add the following content:

```
import org.specs2.mutable._

class ProductSpec extends Specification {

    "The 'product' string" should {
        "contain seven characters" in {
            "product" must have size(7)
        }
    }
}
```

2. Run the first spec using Activator by running the command `test-only ProductSpec`:

```
$ activator
[info] Loading project definition from
/private/tmp/foo_scala/project
[info] Set current project to foo_scala (in build
file:/private/tmp/foo_scala/)
[foo_scala] $ test-only ProductSpec
```

```
[info] ProductSpec
[info]
[info] The 'product' string should
[info] + contain seven characters
[info]
[info] Total for specification ProductSpec
[info] Finished in 24 ms
[info] 1 example, 0 failure, 0 error
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
[success] Total time: 2 s, completed 09 29, 14 12:22:57 PM
```

How it works...

In this recipe, we created a brand new spec file that will contain our test specifications. We placed this file inside the test/ directory and ran the test using activator with the test-only command. The test command is used to run the test and it displays the results of the test.

Testing models

The following recipe focuses on writing a test for our model objects. We will create a new record and add assertions to validate the object's creation. We will then use the Activator command to run our test.

How to do it...

For Java, we need to take the following steps:

1. Edit the `ProductTest.java` file and add the following content:

```
// add new imports
import static play.test.Helpers.*;
import models.*;
import play.test.*;

// add new test
@Test
public void testSavingAProduct() {
    running(fakeApplication(), new Runnable() {
        public void run() {

            Product product = new Product();
            product.name = "Apple";
            product.save();
            assertNotNull(product.getId());
        }
    });
}
```

2. Execute the new spec by running the command `test-only ProductTest`:

```
[foo_java] $ test-only ProductTest
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 2 s, completed 09 29, 14 9:33:43 PM
```

For Scala, we need to take the following steps:

1. Edit the `ProductSpec.scala` file and add the following content:

```
import models._
import play.api.test.WithApplication

"models.Product" should {
    "create a product with save()" in new WithApplication {
        val product = Product(1, "Apple")
        val productId = Product.save(product)

        productId must not be None
    }
}
```

2. Execute the new spec by running the command `test-only ProductSpec`:

```
[foo_scala] $ test-only ProductSpec
[info] Compiling 1 Scala source to
/private/tmp/foo_scala/target/scala-2.11/test-classes...
[info] ProductSpec
[info]
[info] The 'product' string should
[info] + contain seven characters
```

```
[info]
[info] models.Product should
[info] + create a product with save()
[info]
[info] Total for specification ProductSpec
[info] Finished in 1 second, 90 ms
[info] 2 examples, 0 failure, 0 error
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 4 s, completed 09 29, 14 4:28:51 PM
```

How it works...

In this recipe, we added a new spec, where we created a new product and invoked the `save()` method. We then added assertion statements to validate that the value returned by the `save()` method is not equal to none. The `test` command is used to run the test and displays the results of the test.

Testing controllers

The following recipe focuses on writing a test for our controller objects. We will use a `FakeApplication` to create a mock HTTP request to the Product XML listing page and add assertions to validate that the response we receive is indeed an XML representing all of the products in our database. We will then use the `Activator` command to run our test.

How to do it...

For Java, we need to take the following steps:

1. Edit the ProductTest.java file and add the following content:

```
// add new imports
import play.mvc.*;
import static org.fest.assertions.Assertions.*;

//add new test
@Test
public void testProductListAsXml() {
    Result result =
callAction/controllers.routes.ref.Application.listProductsAsXML());
    assertThat(status(result)).isEqualTo(OK);
    assertThat(contentType(result)).isEqualTo("application/xml");
    assertThat(contentAsString(result)).contains("products");
}
```

2. Execute the new spec by running the command test-only ProductTest:

```
[foo_java] $ test-only ProductTest
[info] Compiling 1 Java source to
/private/tmp/foo_java/target/scala-2.11/test-classes...
[info] Passed: Total 3, Failed 0, Errors 0, Passed 3
[success] Total time: 3 s, completed 09 29, 14 9:37:03 PM
```

For Scala, we need to take the following steps:

1. Edit the ProductSpec.scala file and add the following spec code:

```
import controllers._
import play.api.test.FakeRequest
import play.api.test.Helpers._

"controllers.Application" should {
    "respond with XML for /products.xml requests" in new
WithApplication {
    val result = controllers.Application.listProductsAsXML()
(FakeRequest())

    status(result) must equalTo(OK)
    contentType(result) must beSome("application/xml")
    contentAsString(result) must contain("products")
}
}
```

2. Execute the new spec by running the test-only ProductSpec command:

```
$ test-only ProductSpec
[info] Compiling 1 Scala source to
/private/tmp/foo_scala/target/scala-2.11/test-classes...
[info] ProductSpec
[info]
[info] The 'product' string should
```

```
[info] + contain seven characters
[info]
[info] models.Product should
[info] + create a product with save()
[info]
[info] controllers.Application should
[info] + respond with XML for /products.xml requests
[info]
[info] Total for specification ProductSpec
[info] Finished in 1 second, 333 ms
[info] 3 examples, 0 failure, 0 error
[info] Passed: Total 3, Failed 0, Errors 0, Passed 3
success] Total time: 4 s, completed 09 29, 14 5:23:41 PM
```

How it works...

In this recipe, we created a new spec to test a URL route we created earlier. Then, we validated the `/products.xml` URL route by making sure that the response content type is `application/xml` and that it contains our root element `products`. The `test` command is used to run the test and it displays the results of the test.

Chapter 2. Using Controllers

In this chapter, we will cover the following recipes:

- Using HTTP headers
- Using HTTP cookies
- Using the session
- Using custom actions
- Using filters
- Using path binders
- Serving JSON
- Receiving JSON
- Uploading files
- Using futures and Akka actors

Introduction

In this chapter, we will dive in a little deeper into Play controllers and discuss some advanced topics regarding controllers in web applications. We will also learn how Play can handle and address more modern web app requirements besides the common use cases such as data manipulation and data retrieval. As we rely on controllers to route web requests and responses, we want to ensure that our controllers are as lightweight and as decoupled as possible, to ensure page responsiveness and predictable page load times. Providing a clean separation from the model and other data-related processing and services also provides developers a clearer understanding of what each layer's responsibilities are.

Using HTTP headers

For this recipe, we will explore how Play applications can manipulate HTTP headers. We will use the `curl` tool to validate if our changes to the HTTP response headers were applied correctly. For Windows users, it is recommended to install Cygwin in order to have a unix-like environment for Windows machines (<https://www.cygwin.com/>).

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_java/app/controllers/Application.java` by adding the following action:

```
public static Result modifyHeaders() {  
    response().setHeader("ETag", "foo_java");  
    return ok("Header Modification Example");  
}
```

3. Add a new routes entry for the newly-added action in `foo_scala/conf/routes`:

```
GET /header_example controllers.Application.modifyHeaders
```

4. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/header_example  
* Hostname was NOT found in DNS cache  
*   Trying ::1...  
* Connected to localhost (::1) port 9000 (#0)  
> GET /header_example HTTP/1.1  
> User-Agent: curl/7.37.1  
> Host: localhost:9000  
> Accept: */*  
>  
< HTTP/1.1 200 OK  
< Content-Type: text/plain; charset=utf-8  
< ETag: foo_java  
< Content-Length: 27  
<  
* Connection #0 to host localhost left intact  
Header Modification Example%
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def modifyHeaders = Action {  
    Ok("Header Modification Example")  
    .withHeaders(  
        play.api.http.HeaderNames.ETAG -> "foo_scala"  
    )  
}
```

3. Add a new routes entry for the newly-added action in `foo_scala/conf/routes`:

```
GET /header_example controllers.Application.modifyHeaders
```

4. Request our new routes and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/header_example
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /header_example HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< ETag: foo_scala
< Content-Length: 27
<
* Connection #0 to host localhost left intact
Header Modification Example%
```

How it works...

In this recipe, we created a new URL route and action. Within the action, we added a new HTTP header and assigned an arbitrary value to it. We then accessed this new action using the command-line tool, `curl` so that we can view the response HTTP headers in raw text. The output should contain our custom header key and its assigned arbitrary value.

Using HTTP cookies

For this recipe, we will explore how Play applications can manipulate HTTP cookies. We will use the `curl` tool to validate our changes to the HTTP response headers containing the new cookie we added to the response.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify `foo_java/app/controllers/Application.scala` by adding the following action:

```
public static Result modifyCookies() {
    response().setCookie("source", "tw", (60*60));
    return ok("Cookie Modification Example");
}
```

3. Add a new route entry for the newly-added action in `foo_java/conf/routes`:

```
GET /cookie_example controllers.Application.modifyCookies
```

4. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/cookie_example
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /cookie_example HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: source=tw; Expires=Sun, 23 Oct 2014 10:22:43 GMT; Path=/
< Content-Length: 27
<
* Connection #0 to host localhost left intact
Cookie Modification Example%
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def modifyCookies = Action {
    val cookie = Cookie("source", "tw", Some(60*60))
    Ok("Cookie Modification Example")
        .withCookies(cookie)
}
```

3. Add a new routes entry for the newly-added action in `foo_scala/conf/routes`:

```
GET /cookie_example controllers.Application.modifyCookies
```

4. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/cookie_example
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /cookie_example HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: source=tw; Expires=Sun, 23 Oct 2014 09:27:24 GMT; Path=/;
HTTPOnly
< Content-Length: 27
<
* Connection #0 to host localhost left intact
Cookie Modification Example%
```

How it works...

In this recipe, we created a new URL route and action. Within the action, we added a new cookie named source and assigned it an arbitrary value “tw” and an optional expiration time (in this recipe, an hour):

```
val cookie = Cookie("source", "tw", Some(60*60))
```

We then accessed this new action using the command-line tool curl so that we can view the response HTTP headers in raw text. The output should contain the Set-Cookie header with the cookie name and value we assigned in the action.

Using the session

For this recipe, we will explore how Play applications handle a session state. This sounds counterintuitive, as Play claims to be a stateless and lightweight web framework. However, as sessions and session states have become major components for web applications, Play implements sessions as cookies and therefore, are actually stored on the client side or the user browser.

How to do it...

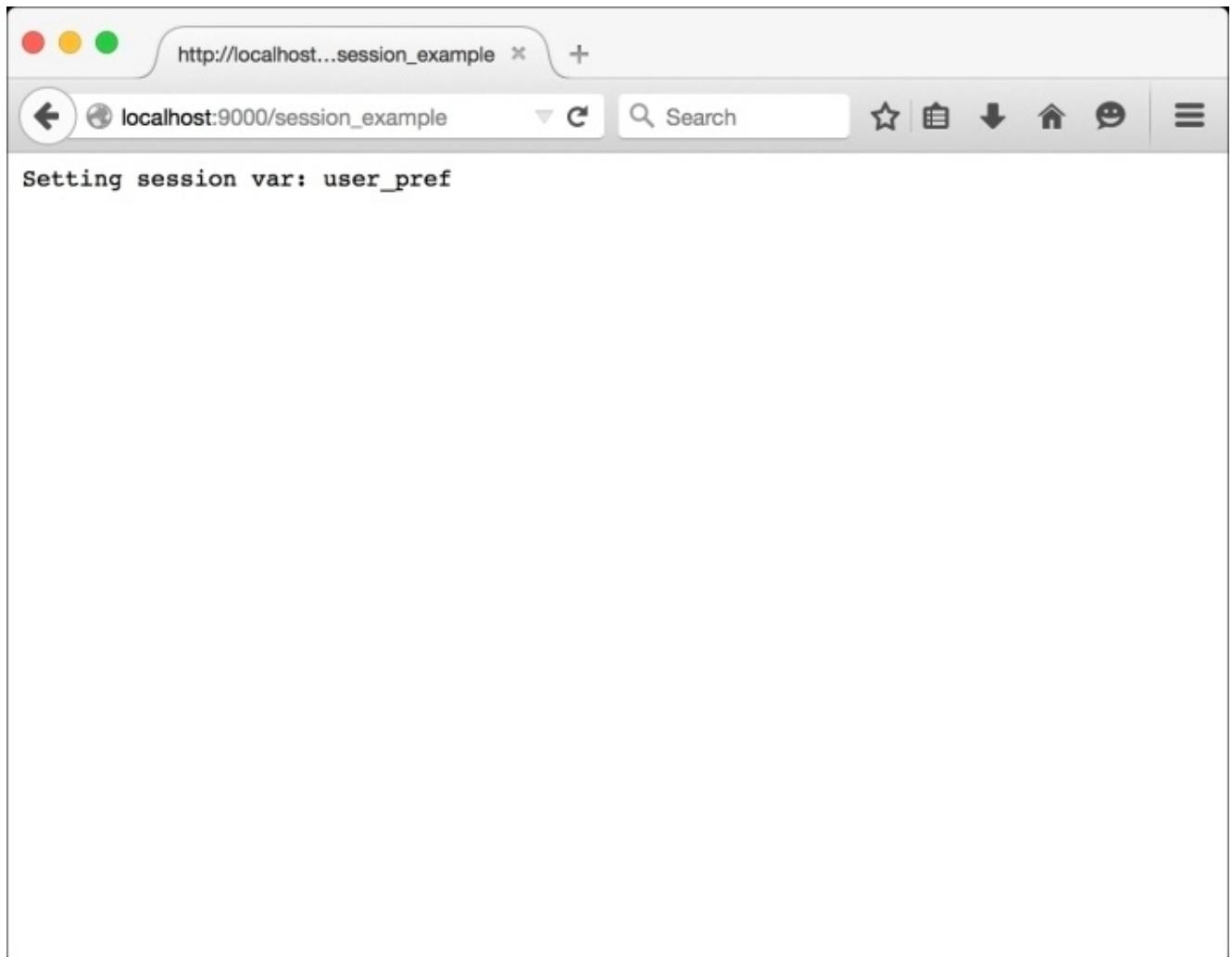
For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify `foo_java/app/controllers/Application.scala` by adding the following action:

```
public static Result modifySession() {  
    final String sessionVar = "user_pref";  
    final String userPref = session(sessionVar);  
    if (userPref == null) {  
        session(sessionVar, "tw");  
        return ok("Setting session var: " + sessionVar);  
    } else {  
        return ok("Found user_pref: " + userPref);  
    }  
}
```

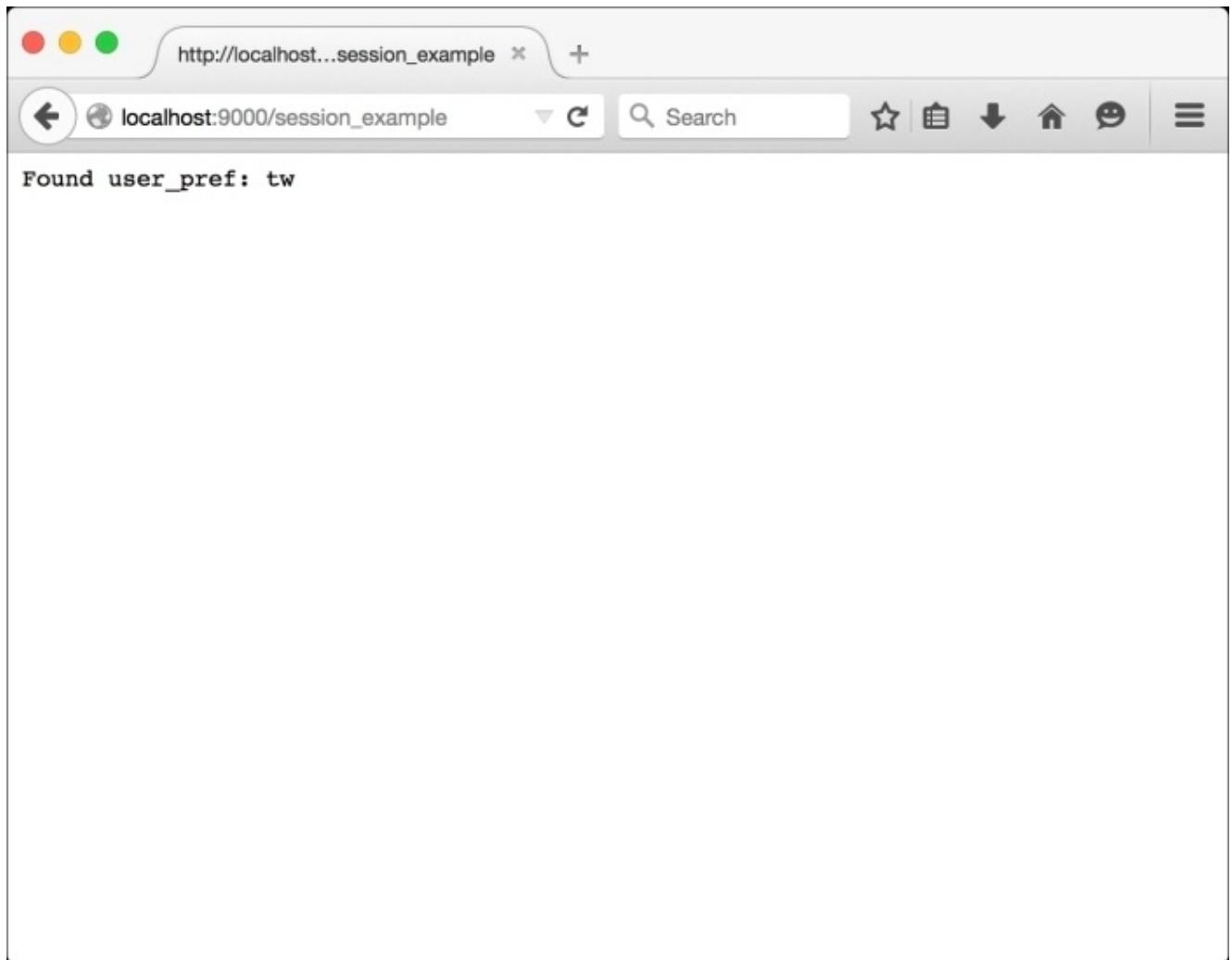
3. Add a new routes entry for the newly-added action in `foo_java/conf/routes`:

```
GET /session_example controllers.Application.modifySession
```



4. Access this new URL route (`http://localhost:9000/session_example`) using a

web browser. You should see the text **Setting session var: user_pref:**



5. Access this new URL route again using the same web browser and you see the text **Found userPref: tw**.
6. Our new session variable is assigned using curl:

```
$ curl -v http://localhost:9000/session_example
* Hostname was NOT found in DNS cache
*   Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /session_example HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: PLAY_SESSION="cadbccca718bbfcc11af40a2cfe8e4c76716cca1f-
user_pref=tw"; Path=/; HTTPOnly
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Setting session var: user_pref%
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def modifySession = Action { request =>
    val sessionVar = "user_pref"
    request.session.get(sessionVar) match {
        case Some(userPref) => {
            Ok("Found userPref: %s".format(userPref))
        }
        case None => {
            Ok("Setting session var: %s".format(sessionVar))
                .withSession(
                    sessionVar -> "tw"
                )
        }
    }
}
```

3. Add a new routes entry for the newly-added Action in `foo_scala/conf/routes`:

```
GET /session_example controllers.Application.modifySession
```

4. Access this new URL route (`http://localhost:9000/session_example`) using a web browser and you should see the text **Setting session var: user_pref**
5. Access this new URL route again using the same web browser and you see the text **Found userPref: tw**.
6. You can also see how our new session variable is assigned using curl:

```
$ curl -v http://localhost:9000/session_example
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /session_example HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Set-Cookie: PLAY_SESSION="64c6d2e0894a60dd28101e37b742f71ae332ed13-
user_pref=tw"; Path=/; HTTPOnly
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Setting session var: user_pref%
```

How it works...

In this recipe, we created a new URL route and action. Within the action, we added some logic to understand whether the session var “user_pref” already existed in the session or not. If the session variable was indeed set, we print out the value of the session variable in the response body. If the session variable was not found in the current session, it would add the session variable to the session and display text, notifying the requestor that it did not find the session variable. We validated this by using a web browser and requesting the same URL route twice; first, to set the session variable and second, to print the value of the session variable. We also used curl to see how the session variable was set to the current session as an HTTP cookie header.

Using custom actions

For this recipe, we will explore how Play Framework provides the building blocks for creating reusable, custom actions.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify `foo_java/app/controllers/Application.java` by adding the following action:

```
@With(AuthAction.class)
public static Result dashboard() {
    return ok("User dashboard");
}

public static Result login() {
    return ok("Please login");
}
```

3. Add our new action class to `foo_java/app/controllers/AuthAction.java` as well:

```
package controllers;

import play.*;
import play.mvc.*;
import play.libs.*;
import play.libs.F.*;

public class AuthAction extends play.mvc.Action.Simple {
    public F.Promise<Result> call(Http.Context ctx) throws Throwable {
        Http.Cookie authCookie = ctx.request().cookie("auth");

        if (authCookie != null) {
            Logger.info("Cookie: " + authCookie);
            return delegate.call(ctx);

        } else {
            Logger.info("Redirecting to login page");
            return Promise.pure(redirect(controllers.routes.
Application.login()));
        }
    }
}
```

4. Add new routes for the newly added action in `foo_java/conf/routes`:

```
GET /dashboard    controllers.Application.dashboard
GET /login       controllers.Application.login
```

5. Access the dashboard URL route using a web browser and notice that it redirects you to the login URL route. You will also notice a log entry in our console where the request is about to be redirected to the login page:

```
[info] application - Redirecting to login page
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def dashboard = AuthAction {  
    Ok("User dashboard")  
}  
  
def login = Action {  
    Ok("Please login")  
}
```

3. Add our new action to `foo_scala/app/controllers/Application.scala` as well:

```
object AuthAction extends ActionBuilder[Request] {  
    import play.api.mvc.Results._  
    import scala.concurrent.Future  
  
    def invokeBlock[A](request: Request[A], block: (Request[A]) =>  
Future[Result]) = {  
    request.cookies.get("auth") match {  
        case Some(authCookie) => {  
            Logger.info("Cookie: " + authCookie)  
            block(request)  
        }  
        case None => {  
            Logger.info("Redirecting to login page")  
            Future.successful(Redirect(routes.Application.login()))  
        }  
    }  
}
```

4. Add new routes for the newly added action in `foo_scala/conf/routes`:

```
GET  /dashboard    controllers.Application.dashboard  
GET  /login       controllers.Application.login
```

5. Access the dashboard URL route using a web browser and notice that it redirects you to the login URL route. You will also notice a log entry in our console where the request is about to be redirected to the login page:

```
[info] application - Redirecting to login page
```

How it works...

In this recipe, we created two new URL routes and actions; one to display a user dashboard, and the other, to act as our login page. Within the dashboard action, we utilized our new action `AuthAction`. The `AuthAction` object checks for the existence of the auth cookie and if it does find the said cookie in the request, it calls the `ActionBuilder` in the chain:

```
// Java  
return delegate.call(ctx);  
  
// Scala  
block(request)
```

If the auth cookie is not found in the request, `AuthAction` redirects the current request to the login URL route, wrapping it around a completed `Future[Result]` object with `Future.successful()`:

```
// Java  
return Promise.pure(redirect(controllers.routes.Application.login()));  
  
// Scala  
Future.successful(Redirect(routes.Application.login()))
```


Using filters

For this recipe, we will explore how Play Framework provide APIs for HTTP request and response filters. HTTP filters provide a way to transparently decorate a HTTP request or response and is useful for lower-level services (such as response compression), gathering metrics, and more in-depth logging.

Note

It is also worth noting that presently (as of Play 2.3.7), HTTP filters are best implemented using the Play Scala API with the *play.api.mvc.EssentialFilter* trait. So for this recipe, we will implement a Scala-based filter for our Java recipe.

How to do it...

For Java, we need to take the following steps:

1. Run the foo_java application with Hot-Reloading enabled.
2. Create a new filter object by creating the file
foo_java/app/ResponseTimeLogFilter.scala, and adding the following contents:

```
import play.api.mvc._

object ResponseTimeLogFilter {
    def apply(): ResponseTimeLogFilter = {
        new ResponseTimeLogFilter()
    }
}

class ResponseTimeLogFilter extends Filter {
    import play.api.Logger
    import scala.concurrent.Future
    import
play.api.libs.concurrent.Execution.Implicits.defaultContext

    def apply(f: (RequestHeader) => Future[Result])(rh:
RequestHeader): Future[Result] = {
        val startTime = System.currentTimeMillis
        val result = f(rh)
        result.map { result =>
            val currDate = new java.util.Date
            val responseTime = (currDate.getTime() - startTime) / 1000F

            Logger.info(s"${rh.remoteAddress} - [${currDate}] -
${rh.method} ${rh.uri}" +
s" ${result.header.status} ${responseTime}")

            result
        }
    }
}
```

3. Utilize this new filter by declaring it in the app/Global.java file:

```
import play.GlobalSettings;
import play.api.mvc.EssentialFilter;

public class Global extends GlobalSettings {
    public <T extends EssentialFilter> Class<T>[] filters() {
        return new Class[]{
            ResponseTimeLogFilter.class
        };
    }
}
```

4. Access any previous URL routes we've defined
(http://localhost:9000/session_example) using a web browser. You will be able to see a new log entry with our response stats printed:

```
[info] application - 0:0:0:0:0:0:1 - [Mon Oct 24 23:58:44 PHT 2014] -  
GET /session_example 200 0.673
```

For Scala, we need to take the following steps:

1. Run the foo_scala application with Hot-Reloading enabled.
2. Create a new filter object by creating the file

foo_scala/app/controllers/ResponseTimeLogFilter.scala and adding the following contents:

```
import play.api.Logger  
import play.api.mvc._  
import play.api.libs.concurrent.Execution.Implicits.defaultContext  
  
object ResponseTimeLogFilter extends EssentialFilter {  
    def apply(nextFilter: EssentialAction) = new EssentialAction {  
        def apply(requestHeader: RequestHeader) = {  
            val startTime = System.currentTimeMillis  
            nextFilter(requestHeader).map { result =>  
                val currDate = new java.util.Date  
                val responseTime = (currDate.getTime() - startTime) / 1000F  
  
                Logger.info(s"${requestHeader.remoteAddress} -  
[$currDate] - ${requestHeader.method} ${requestHeader.uri}" +  
s" ${result.header.status} ${responseTime}")  
  
                result  
            }  
        }  
    }  
}
```

3. Utilize this new filter by declaring it in the app/Global.scala file:

```
import play.api._  
import play.api.mvc._  
import controllers.ResponseTimeLogFilter  
  
object Global extends WithFilters(ResponseTimeLogFilter) {  
    override def onStart(app: Application) {  
        Logger.info("Application has started")  
    }  
    override def onStop(app: Application) {  
        Logger.info("Application shutdown...")  
    }  
}
```

4. Access any previous URL routes we've defined

(http://localhost:9000/session_example) using a web browser. You will be able to see a new log entry with our response stats printed:

```
[info] application - 0:0:0:0:0:0:1 - [Mon Oct 24 23:58:44 PHT 2014] -  
GET /session_example 200 0.673
```

How it works...

In this recipe, we created a new Scala-based filter. The filter simply calculates the total response time for request and prints it out in the log file. We then utilized the filter by referring to it in the global application configuration class `Global.java`/`Global.scala`. This will be applied to all requests of the Play application.

Using path binders

For this recipe, we will explore how Play applications allow us to use custom binders for path parameters. These come in handy when you want to simplify your declaration of routes and corresponding actions by dealing with model classes in the routes file and a method signature of actions instead of individual properties and fields declared as parameters.

How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Create a new products controller as `foo_java/app/controllers/Products.scala`.

Once created, add a product case class and companion object, the product form object, and two routes (the first to display our selected product in a form and the second as the PUT action for form update submission):

```
package controllers;

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

public class Products extends Controller {
    private static Form<Product> productForm = Form.form(Product.class);

    public static Result edit(Product product) {
        return ok(views.html.products.form.render(product.sku,
productForm.fill(product)));
    }

    public static Result update(String sku) {
        return ok("Received update request");
    }
}
```

3. Add our new product model in `foo_java/app/models/Product.java` as well:

```
package models;

import play.mvc.*;

public class Product implements PathBindable<Product> {
    public String sku;
    public String title;

    private static final java.util.Map<String, String> productMap = new
java.util.HashMap<String, String>();
    static {
        productMap.put("ABC", "8-Port Switch");
        productMap.put("DEF", "16-Port Switch");
        productMap.put("GHI", "24-Port Switch");
    }

    public static void add(Product product) {
        productMap.put(product.sku, product.title);
    }

    public static java.util.List<Product> getProducts() {
        java.util.List<Product> productList = new
```

```

java.util.ArrayList<Product>();
    for (java.util.Map.Entry<String, String> entry :
productMap.entrySet()) {
        Product p = new Product();
        p.sku = entry.getKey();
        p.title = entry.getValue();
        productList.add(p);
    }
    return productList;
}

public Product bind(String key, String value) {
    String product = productMap.get(value);
    if (product != null) {
        Product p = new Product();
        p.sku = value;
        p.title = product;

        return p;
    } else {
        throw new IllegalArgumentException("Product with sku " + value +
" not found");
    }
}

public String unbind(String key) {
    return sku;
}

public String javascriptUnbind() {
    return "function(k,v) {\n" +
        "    return v.sku;" +
        "}";
}
}

```

4. Add new routes for the newly added action in `foo_java/conf/routes`:

```

GET /products/:product controllers.Products.edit(product:
models.Product)
PUT /products/:sku controllers.Products.update(sku)

```

5. Create the product form view template in

`foo_java/app/views/products/form.scala.html` with the following content:

```

@(sku: String, productForm: Form[models.Product])

@helper.form(action = routes.Products.update(sku)) {
    @helper.inputText(productForm("sku"))
    @helper.inputText(productForm("title"))

    <input type="submit" />
}

```

6. Access our edited products URL route (`http://localhost:9000/products/ABC`). You should be able to view the edit form for our first product. Access our next edit

products URL route (`http://localhost:9000/products/DEF`) and you should see the relevant product details loading in the form.

7. Access the URL `http://localhost:9000/products/XYZ` and see how Play automatically generates the error message with the custom message we specified:

For request 'GET /products/XYZ' [Product with sku XYZ not found]

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Create a new products controller as `foo_scala/app/controllers/Products.scala`. Once created, add a product case class and companion object, the product form object, and two routes (the first to display our selected product in a form and the second as the PUT action for the form update submission):

```
package controllers

import play.api._
import play.api.data._
import play.api.data.Forms._
import play.api.mvc._

case class Product(sku: String, title: String)

object Product {
    implicit def pathBinder(implicit stringBinder: PathBindable[String]) = new PathBindable[Product] {
        override def bind(key: String, value: String): Either[String, Product] = {
            for {
                sku <- stringBinder.bind(key, value).right
                product <- productMap.get(sku).toRight("Product not found").right
            } yield product
        }
        override def unbind(key: String, product: Product): String = {
            stringBinder.unbind(key, product.sku)
        }
    }
}

def add(product: Product) = productMap += (product.sku -> product)

val productMap = scala.collection.mutable.Map(
    "ABC" -> Product("ABC", "8-Port Switch"),
    "DEF" -> Product("DEF", "16-Port Switch"),
    "GHI" -> Product("GHI", "24-Port Switch")
)
}

object Products extends Controller {
    val productForm: Form[Product] = Form(
        mapping(
            "sku" -> nonEmptyText,
```

```

        "title" -> nonEmptyText
    )(Product.apply)(Product.unapply)
)

def edit(product: Product) = Action {
    Ok(views.html.products.form(product.sku,
productForm.fill(product)))
}

def update(sku: String) = Action {
    Ok("Received update request")
}
}

```

- Add new routes for the newly-added action in foo_scala/conf/routes:

```

GET /products/:product controllers.Products.edit(product:
controllers.Product)
PUT /products/:sku     controllers.Products.update(sku)

```

- Create the product form view template in foo_scala/app/views/products/form.scala.html with the following content:
- ```

@(sku: String, productForm: Form[controllers.Product])

@helper.form(action = routes.Products.update(sku)) {
 @helper.inputText(productForm("sku"))
 @helper.inputText(productForm("title"))

 <input type="submit" />
}

```
- Access our edit product URL route (<http://localhost:9000/products/ABC>) and you should be able to view the edit form for our first product. Access our next edit product URL route (<http://localhost:9000/products/DEF>) and you should see the relevant product details loading in the form.
  - Access the URL <http://localhost:9000/products/XYZ> and see how Play automatically generates the error message with the custom message we specified:

For request 'GET /products/XYZ' [Product not found]

# How it works...

In this recipe, we utilized Play's PathBindable interface to use custom path binders. We created a new route, controller, and model to represent a product. We implemented the PathBindable bind and unbind methods for the product:

Form binding is pretty straightforward for Java:

```
// Java
 private static Form<Product> productForm = Form.form(Product.class);
 public static Result edit(Product product) {
 return ok(views.html.products.form.render(product.sku,
productForm.fill(product)));
 }
```

As for Scala, we override two methods in the PathBindable class. During form binding, we first retrieve the product identifier sku, and then pass this same sku to retrieve the corresponding product in our product map:

```
// Scala
 implicit def pathBinder(implicit stringBinder: PathBindable[String]) =
new PathBindable[Product] {
 override def bind(key: String, value: String): Either[String, Product] =
 {
 for {
 sku <- stringBinder.bind(key, value).right
 product <- productMap.get(sku).toRight("Product not found").right
 } yield product
 }
 override def unbind(key: String, product: Product): String = {
 stringBinder.unbind(key, product.sku)
 }
}
```

We defined a route that required the custom path binding:

```
GET /products/:product controllers.Products.edit(product:
controllers.Product)
```

You will notice that in defining the earlier route, we mapped its product parameter to a product instance. The PathBindable class does all the work here in converting the passed sku to a product instance.



# Serving JSON

For this recipe, we will explore how Play Framework allows us to easily convert our model objects to JSON. Being able to write web services that deliver data in the JSON data format has been a very common requirement for modern web applications. Play provides a JSON processing library that we will utilize in this recipe.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify the products controller in `foo_java/app/controllers/Products.java` by adding our product listing action:

```
public static Result index() {
 return ok(Json.toJson(Product.getProducts()));
}
```

3. We need to add the following import statement for Play's JSON libraries:

```
import play.libs.Json;
```

4. Add a new route for the product listing action in `foo_java/conf/routes`:

```
GET /products controllers.Products.index
```

5. Access our product listing URL route (`http://localhost:9000/products`) using curl:

```
$ curl -v http://localhost:9000/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 117
<
* Connection #0 to host localhost left intact
[{"sku": "ABC", "title": "8-Port Switch"}, {"sku": "DEF", "title": "16-Port
Switch"}, {"sku": "GHI", "title": "24-Port Switch"}]%
```

6. As we look at the output of our curl command, you will notice that our content type was automatically set accordingly (`application/json`) and that the response body contains an array of JSON products.

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify the products controller in `foo_scala/app/controllers/Products.scala` by adding our product listing action:

```
def index = Action {
 Ok(Json.toJson(Product.productMap.values))
}
```

3. We need to add the following import statement for Play's JSON libraries:

```
import play.api.libs.json._
import play.api.libs.json.Json._
```

4. We also need to add our implementation of writes for our product model inside our products controller:

```
implicit val productWrites = new Writes[Product] {
 def writes(product: Product) = Json.obj(
 "sku" -> product.sku,
 "title" -> product.title
)
}
```

5. Add a new route for the product listing action in foo\_scala/conf/routes:

```
GET /products controllers.Products.index
```

6. Access our product listing URL route (<http://localhost:9000/products>) using curl:

```
$ curl -v http://localhost:9000/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 117
<
* Connection #0 to host localhost left intact
[{"sku": "ABC", "title": "8-Port Switch"}, {"sku": "DEF", "title": "16-Port Switch"}, {"sku": "GHI", "title": "24-Port Switch"}]%
```

7. As we look at the output of our curl command, you will notice that our content type was automatically set accordingly (application/json) and that the response body contains an array of JSON products.

## How it works...

In this recipe, we modified our products controller and added a new route that returns an array of products in the JSON format. We created the action in the controller and the new route entry in the `conf/routes` file. We then declared an implicit `writes` object which tells Play how to render our product model in the JSON format:

```
implicit val productWrites = new Writes[Product] {
 def writes(product: Product) = Json.obj(
 "sku" -> product.sku,
 "title" -> product.title
)
}
```

In the preceding code snippet, we explicitly declare the JSON key labels for the rendered JSON.

The action then converts the retrieved product to JSON as the response to the action request:

```
Ok(toJson(Product.productMap.values))
```



# Receiving JSON

For this recipe, we will explore how Play Framework allow us to receive JSON objects easily and enable us to automatically convert them into model instances.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify the products controller in `foo_java/app/controllers/Products.java` by adding our product creation action:

```
@BodyParser.Of(BodyParser.Json.class)
public static Result postProduct() {
 JsonNode json = request().body().asJson();
 String sku = json.findPath("sku").textValue();
 String title = json.findPath("title").textValue();

 Product p = new Product();
 p.sku = sku;
 p.title = title;
 Product.add(p);
 return created(Json.toJson(p));
}
```

3. We need to add the following import statement for Play's JSON libraries:

```
import com.fasterxml.jackson.databind.JsonNode;
```

4. Add a new route for the product listing action in `foo_java/conf/routes`:

```
POST /products controllers.Products.postProduct
```

5. Access our product creation URL route (`http://localhost:9000/products`) using curl:

```
$ curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" --data '{"sku":"JKL", "title":"VPN/Router"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 35
>
* upload completely sent off: 35 out of 35 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 34
<
* Connection #0 to host localhost left intact
{"sku":"JKL","title":"VPN/Router"}%
```

6. As we look at the output of our `curl` command, you will notice that our response body now contains our newly added product.

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify the products controller in `foo_scala/app/controllers/Products.scala` by adding our product creation action:

```
def postProduct = Action(BodyParsers.parse.json) { request =>
 val post = request.body.validate[Product]
 post.fold(
 errors => {
 BadRequest(Json.obj("status" ->"error", "message" ->
JsError.toJson(errors)))
 },
 product => {
 Product.add(product)
 Ok(toJson(product))
 }
)
}
```

3. We need to add the following import statement for Play's JSON libraries:

```
import play.api.libs.functional.syntax._
```

4. We will also need to add our implementation of reads for our product model inside our products controller:

```
implicit val productReads: Reads[Product] = (
 (JsPath \ "sku").read[String] and
 (JsPath \ "title").read[String]
)(Product.apply _)
```

5. Add a new route for the product listing action in `foo_scala/conf/routes`:

```
POST /products controllers.Products.postProduct
```

6. Access our product creation URL route (`http://localhost:9000/products`) using curl:

```
$ curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" --data '{"sku": "JKL", "title": "VPN/Router"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 35
>
* upload completely sent off: 35 out of 35 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 34
<
```

```
* Connection #0 to host localhost left intact
{"sku":"JKL","title":"VPN/Router"}%
```

7. As we look at the output of our curl command, you will notice that our response body now contains our newly added product.

## How it works...

In this recipe, we looked into how we can use Play to consume JSON objects by using the JSON BodyParser and be able to convert them to the appropriate model objects. For Java, we traversed the JSON tree, retrieved each property value, and assigned it to our local variables:

```
JsonNode json = request().body().asJson();
String sku = json.findPath("sku").textValue();
String title = json.findPath("title").textValue();
```

For Scala, this was a little more straightforward, using the JSON BodyParser from Play:

```
val post = request.body.validate[Product]
post.fold(
 errors => {
 BadRequest(Json.obj("status" ->"error", "message" ->
JsError.toJson(errors)))
 },
 product => {
 Product.add(product)
 Ok(toJson(product))
 }
)
```



# Uploading files

For this recipe, we will learn how to upload files in Play applications. The ability to upload files is one of the more important aspects of web applications, and we will see here how Play makes file uploads straightforward and easy to handle.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Add a new view template in `foo_java/app/views/form.scala.html` for the file upload form with the following contents:

```
@helper.form(action = routes.Application.handleUpload, 'enctype ->
"multipart/form-data") {
 Profile Photo: <input type="file" name="profile">

 <div>
 <input type="submit">
 </div>

}
```

3. Modify `foo_java/app/controllers/Application.java` by adding the following actions:

```
import play.mvc.Http.MultipartFormData;
import play.mvc.Http.MultipartFormData.FilePart;
import java.nio.file.*;
import java.io.*; public static Result uploadForm() {
 return ok(form.render());
}

public static Result handleUpload() {
 MultipartFormData body =
request().body().asMultipartFormData();
 FilePart profileImage = body.getFile("profile");

 if (profileImage != null) {
 try {
 String fileName = profileImage.getFilename();
 String contentType = profileImage.getContentType();
 File file = profileImage.getFile();

 Path path = FileSystems.getDefault().getPath("/tmp/" +
fileName);
 Files.write(path, Files.readAllBytes(file.toPath()));
 return ok("Image uploaded");
 } catch(Exception e) {
 return internalServerError(e.getMessage());
 }
 } else {
 flash("error", "Please upload a valid file");
 return redirect(routes.Application.uploadForm());
 }
}
```

4. Add a new routes entry for the newly-added action in `foo_java/conf/routes`:

```
GET /upload_form controllers.Application.uploadForm
```

```
POST /upload controllers.Application.handleUpload
```

5. Access the upload form URL route (`http://localhost:9000/upload_form`) using a web browser. You will now be able to select a file to upload in your filesystem.
6. You can verify that the file was indeed uploaded by taking a look at the `/tmp` directory:

```
$ ls /tmp
B82BE492-0BEF-4B2D-9A68-2664FB9C2A97.png
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Add a new view template in `foo_scala/app/views/form.scala.html` for the file upload form with the following contents:

```
@helper.form(action = routes.Application.handleUpload, 'enctype ->
"multipart/form-data") {

 Profile photo: <input type="file" name="profile">

 <div>
 <input type="submit">
 </div>

}
```

3. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
def uploadForm = Action {
 Ok(views.html.form())
}

def handleUpload = Action(parse.multipartFormData) { request =>
 import java.io.File

 request.body.file("profile") match {
 case Some(profileImage) => {
 val filename = profileImage.filename
 val contentType = profileImage.contentType
 profileImage.ref.moveTo(new File(s"/tmp/$filename"))
 Ok("Image uploaded")
 }
 case None => {
 Redirect(routes.Application.uploadForm).flashing(
 "error" -> "Please upload a valid file")
 }
 }
}
```

4. Add a new routes entry for the newly added action in `foo_scala/conf/routes`:

```
GET /upload_form controllers.Application.uploadForm
POST /upload controllers.Application.handleUpload
```

5. Access the upload form URL route (`http://localhost:9000/upload_form`) using a web browser. You will now be able to select a file to upload in your filesystem.
6. You can verify that the file was indeed uploaded, by taking a look at the `/tmp` directory:

```
$ ls /tmp
B82BE492-0BEF-4B2D-9A68-2664FB9C2A97.png
```

## How it works...

In this recipe, we created two new actions and URL routes; the first to display our upload form template and the second to handle the actual file upload action. We added our upload view form template `form.scala.html` in the `app/views` directory. We then handled the actual file upload submission by using Play's helper methods to retrieve the uploaded file, and then proceeded to store the file in a predefined location.



# Using futures with Akka actors

For this recipe, we will explore how Play Framework allows us to create asynchronous controllers using futures, in conjunction with Akka actors. The ability to create asynchronous controllers provides a way for developers to trigger background jobs and execute long-running operations asynchronously without sacrificing endpoint responsiveness. Adding Akka to the mix brings a new dimension to fault-tolerant, resilient data services, which become valuable tools in a developer's toolchain in the age of maturing and sophisticating web application requirements.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled.
2. Modify the application controller in `foo_java/app/controllers/Application.java` by adding the following contents:

```
public static Promise<Result> asyncExample() {
 ActorRef fileReaderActor =
 Akka.system().actorOf(Props.create(FileReaderActor.class));
 FileReaderProtocol words = new
 FileReaderProtocol("/usr/share/dict/words");

 return Promise.wrap(ask(fileReaderActor, words, 3000)).map(
 new Function<Object, Result>() {
 public Result apply(Object response) {
 return ok(response.toString());
 }
 }
);
}
```

3. We need to add the following import statement to include the necessary libraries, specifically the Akka libraries that we will be utilizing for this recipe:

```
import java.util.*;
import play.libs.Akka;
import play.libs.F.Function;
import static play.mvc.Results.*;
import static akka.pattern.Patterns.ask;
import play.libs.F.Promise;
import akka.actor.*;
```

4. We also need to add our Akka actor in

`foo_java/app/actors/FileReaderActor.java` with the following contents:

```
package actors;

import java.util.*;
import java.util.concurrent.Callable;
import java.nio.charset.*;
import java.nio.file.*;
import java.io.*;
import scala.concurrent.ExecutionContext;
import scala.concurrent.Future;
import scala.concurrent.Await;
import scala.concurrent.duration.*;
import akka.dispatch.*;
import akka.util.Timeout;
import akka.actor.*;
import play.libs.Akka;

import static akka.dispatch.Futures.future;
```

```

public class FileReaderActor extends UntypedActor {
 public void onReceive(Object message) throws Exception {
 if (message instanceof FileReaderProtocol) {
 final String filename = ((FileReaderProtocol)
message).filename;

 Future<String> future = future(new Callable<String>() {
 public String call() {
 try {
 Path path = Paths.get(filename);
 List<String> list = Files.readAllLines(path,
StandardCharsets.UTF_8);
 String[] contents = list.toArray(new String[list.size()]);

 return Arrays.toString(contents);
 } catch(Exception e) {
 throw new IllegalStateException(e);
 }
 }
 }, Akka.system().dispatcher());

 akka.pattern.Patterns.pipe(
future, Akka.system().dispatcher()).to(getSender());
 }
 }
 }
}

```

5. We also need to create our actor protocol class for the `FileReaderActor` in `foo_java/app/actors/FileReaderProtocol.java` with the following content:

```

package actors;

public class FileReaderProtocol implements java.io.Serializable {
 public final String filename;
 public FileReaderProtocol(String filename) { this.filename =
filename; }
}

```

6. Add a new route for the `async example` action in `foo_java/conf/routes`:

```
GET /async_example controllers.Application.asyncExample
```

7. Add the default Akka configs in `foo_java/conf/application.conf`:

```
akka.default-dispatcher.fork-join-executor.pool-size-max = 64
akka.actor.debug.receive = on
```

8. Access the `async example` URL route (`http://localhost:9000/async_example`) using a web browser. You should see the contents of the local file `/usr/share/dict/words` displayed in the web browser.

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled.
2. Modify the application controller in

foo\_scala/app/controllers/Application.scala by adding the following contents:

```
 val fileReaderActor = Akka.system.actorOf(Props[FileReaderActor],
name = "fileReader")

 def asyncExample = Action.async {
 implicit val timeout = Timeout(3 seconds)

 (fileReaderActor ?
FileReaderProtocol("/usr/share/dict/words")).mapTo[String].map{ words
=>
 Ok("Words: \n" + words)
 }
}
```

3. We need to add the following import statements to include the necessary libraries, specifically the Akka libraries we will be utilizing for this recipe:

```
import play.api.libs.concurrent.Akka
import play.api.Play.current
import akka.pattern.ask
import akka.pattern.pipe
import akka.util.Timeout
import akka.actor.{Props, Actor, ActorLogging}
import scala.concurrent.duration._
import scala.concurrent._
import play.api.libs.concurrent.Execution.Implicits.defaultContext
```

4. We also need to add our Akka actor in the application controller for convenience:

```
case class FileReaderProtocol(filename: String)

class FileReaderActor extends Actor with ActorLogging {

 def receive = {
 case FileReaderProtocol(filename) => {
 val currentSender = sender
 val contents = Future {
 scala.io.Source.fromFile(filename).mkString
 }
 contents pipeTo currentSender
 }
 }
}
```

5. Add a new route for the async example action in foo\_scala/conf/routes:

```
GET /async_example controllers.Application.asyncExample
```

6. Add the default Akka configs in foo\_scala/conf/application.conf:

```
akka.default-dispatcher.fork-join-executor.pool-size-max = 64
akka.actor.debug.receive = on
```

7. Access the async example URL route ([http://localhost:9000/async\\_example](http://localhost:9000/async_example))

using a web browser. You should see the contents of the local file /usr/share/dict/words displayed in the web browser.

# How it works...

In this recipe, we modified our application controller and added a new route that returns the contents of a local file /usr/share/dict/words. We created the action in the controller and the new route entry in the conf/routes file. We then created the Akka actor class and protocol class that will do the actual work of reading the file and returning its contents.

For Java, we need to take the following steps:

```
// Java
Path path = Paths.get(filename);
List<String> list = Files.readAllLines(path, StandardCharsets.UTF_8);
```

For Scala, we need to take the following steps:

```
// Scala
val contents = Future {
 scala.io.Source.fromFile(filename).mkString
}
```

We then configured our new action to invoke the Actor and configured it in such a way that it returns the results asynchronously:

```
// Java
return Promise.wrap(ask(fileReaderActor, words, 3000)).map(
 new Function<Object, Result>() {
 public Result apply(Object response) {
 return ok(response.toString());
 }
 }
);

// Scala
(fileReaderActor ? FileReaderProtocol("/usr/share/dict/words"))
.mapTo[String].map{ words =>
 Ok("Words: \n" + words)
}
```

We also added default Akka configuration settings in conf/application.conf:

```
akka.default-dispatcher.fork-join-executor.pool-size-max =64
akka.actor.debug.receive = on
```

The preceding settings allow us to set the maximum size of our default dispatcher's thread pool. In this recipe, it is set to 64. For more information about Akka dispatchers, please refer to <http://doc.akka.io/docs/akka/snapshot/java/dispatchers.html>.



# Chapter 3. Leveraging Modules

In this chapter, we will cover the following recipes:

- Dependency injection with Spring
- Dependency injection using Guice
- Utilizing MongoDB
- Utilizing MongoDB and GridFS
- Utilizing Redis
- Integrating Play application with Amazon S3
- Integrating Play application with Typesafe Slick
- Utilizing play-mailer
- Integrating Bootstrap and WebJars

# Introduction

In this chapter, we will look at utilizing Play and other third-party modules to address commonly required functionalities of modern web applications. As web applications and web application frameworks mature and evolve, the need for a modular and extensible system as part of the core web application framework becomes increasingly important. This is achievable and straightforward with Play Framework 2.0.



# Dependency injection with Spring

For this recipe, we will explore how to integrate the popular Spring Framework with a Play application. We will use Spring for bean instantiation and injection using Play controllers and service classes.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare Spring as a project dependency in `build.sbt`:

```
"org.springframework" % "spring-context" % "3.2.2.RELEASE",
"org.springframework" % "spring-aop" % "3.2.2.RELEASE",
"org.springframework" % "spring-expression" % "3.2.2.RELEASE"
```

3. Create a new admin controller in

`foo_java/app/controllers/AdminController.java` with the following code:

```
package controllers;

import play.*;
import play.mvc.*;
import org.springframework.beans.factory.annotation.Autowired;
import services.AdminService;

@org.springframework.stereotype.Controller
public class AdminController {

 @Autowired
 private AdminService adminService;

 public Result index() {
 return play.mvc.Controller.ok("This is an admin-only resource:
" + adminService.getFoo());
 }
}
```

4. Create an admin service interface class in

`foo_java/app/services/AdminServices.java` and a mock admin service implementation class in `foo_java/app/services/AdminServicesImpl.java` with the following content:

```
// AdminService.java
package services;

public interface AdminService {
 String getFoo();
}

// AdminServiceImpl.java
package services;

import org.springframework.stereotype.Service;

@Service
public class AdminServiceImpl implements AdminService {
```

```

@Override
public String getFoo() {
 return "foo";
}
}

```

5. Add a new routes entry for the newly added action to foo\_java/conf/routes:

```
GET /admins @controllers.AdminController.index
```

6. Add a Global settings class to the foo\_java/app/Global.java file with the following content:

```

import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt;
import play.Application;
import play.GlobalSettings;

public class Global extends GlobalSettings {
private ApplicationContext ctx;

@Override
public void onStart(Application app) {
 ctx = new
AnnotationConfigApplicationContext(SpringConfig.class);
}

@Override
public <A> A getControllerInstance(Class<A> clazz) {
 return ctx.getBean(clazz);
}
}

```

7. Add the Spring configuration class to foo\_java/app/SpringConfig.java with the following content:

```

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan({"controllers", "services"})
public class SpringConfig {
}

```

8. Request our new route and examine the response body to confirm:

```

$ curl -v http://localhost:9000/admins
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admins HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>

```

```

< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
This is an admin-only resource: foo%

```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare Spring as a project dependency in `build.sbt`:

```
"org.springframework" % "spring-context" % "3.2.2.RELEASE"
```

3. Create a new admin controller in

`foo_scala/app/controllers/AdminController.scala` with the following content:

```

package controllers

import play.api.mvc.{Action, Controller}
import services.AdminService

class AdminController(implicit adminService: AdminService)
extends Controller {
 def index = Action {
 Ok("This is an admin-only resource:
%s".format(adminService.foo))
 }
}

```

4. Create an admin service class in `foo_scala/app/services/AdminServices.scala` with the following content:

```

package services

class AdminService {
 def foo = "foo"
}

```

5. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:

```

GET /admins
@controllers.AdminController.index

```

6. Add a Global settings class to the `foo_scala/app/Global.scala` with the following content:

```

import org.springframework.context.ApplicationContext
import
org.springframework.context.annotation.AnnotationConfigApplicationConte
xt

object Global extends play.api.GlobalSettings {

```

```

 private val ctx: ApplicationContext = new
AnnotationConfigApplicationContext(classOf[SpringConfig])

 override def getControllerInstance[A](clazz: Class[A]): A = {
 return ctx.getBean(clazz)
 }
}

```

7. Add the Spring configuration class to `foo_scala/app/SpringConfig.scala` with the following content:

```

import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Bean
import controllers._
import services._

@Configuration
class SpringConfig {
 @Bean
 implicit def adminService: AdminService = new AdminService

 @Bean
 def adminController: AdminController = new AdminController
}

```

8. Request our new route and examine the response body to confirm:

```

$ curl -v http://0.0.0.0:9000/admins
* Hostname was NOT found in DNS cache
* Trying 0.0.0.0...
* Connected to 0.0.0.0 (127.0.0.1) port 9000 (#0)
> GET /admins HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 0.0.0.0:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host 0.0.0.0 left intact
This is an admin-only resource: foo%

```

## How it works...

In this recipe, we configured our Play application to utilize Dependency injection in our controllers and service classes using Spring. We configured Spring in the `Global` settings file and loaded the `SpringConfig` class, which will contain our Spring-specific configurations.



# Dependency injection using Guice

For this recipe, we will explore how to integrate Google Guice with a Play application. We will use Guice for bean instantiation and injection using Play controllers and service classes.

# How to do it...

For Java, we need to take the following steps:

1. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare the guice module as a project dependency in build.sbt:

```
"com.google.inject" % "guice" % "3.0"
```

3. Configure Guice by modifying the contents of the Global settings class:

```
import com.google.inject.AbstractModule;
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.inject.Singleton;
import play.GlobalSettings;
import services.*;

public class Global extends GlobalSettings {
 private Injector injector = Guice.createInjector(new
AbstractModule() {
 @Override
 protected void configure() {
 bind(CategoryService.class).to(
CategoryServiceImpl.class).in(Singleton.class);
 }
 });

 @Override
 public <T> T getControllerInstance(Class<T> clazz) {
 return injector.getInstance(clazz);
 }
}
```

4. Create a category controller in

foo\_java/app/controllers/CategoryController.java by adding the following content:

```
package controllers;

import com.google.inject.Inject;
import play.libs.Json;
import play.mvc.Controller;
import play.mvc.Result;
import services.CategoryService;

public class CategoryController extends Controller {
 @Inject
 private CategoryService categoryService;

 public Result index() {
 return ok(Json.toJson(categoryService.list()));
 }
}
```

```
}
```

5. Create a category service interface in

`foo_java/app/services/CategoryService.java` by adding the following content:

```
package services;

import java.util.List;

public interface CategoryService {
 List<String> list();
}
```

6. Create a category service implementation class in

`foo_java/app/services/CategoryServicesImpl.java` by adding the following content:

```
package services;

import java.util.Arrays;
import java.util.List;

public class CategoryServiceImpl implements CategoryService {
 @Override
 public List<String> list() {
 return Arrays.asList(new String[] {"Manager", "Employee",
"Contractor"});
 }
}
```

7. Add a new routes entry for the newly added action to `foo_java/conf/routes`:

```
GET /categories @controllers.CategoryController.index
```

8. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/categories
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /categories HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
["Manager", "Employee", "Contractor"]%
```

For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare the securesocial module as a project dependency in build.sbt:

```
"com.google.inject" % "guice" % "3.0"
```

3. Configure Guice by modifying the contents of the Global settings class:

```
import com.google.inject.{Guice, AbstractModule}
import play.api.GlobalSettings
import services._

object Global extends GlobalSettings {
 val injector = Guice.createInjector(new AbstractModule {
 protected def configure() {

bind(classOf[CategoryService]).to(classOf[CategoryServiceImpl])
 }
})
}

override def getControllerInstance[A](controllerClass: Class[A]): A = {
 injector.getInstance(controllerClass)
}
```

4. Create a category controller in

foo\_scala/app/controllers/CategoryController.scala by adding the following content:

```
package controllers

import play.api.mvc._
import play.api.libs.json.Json._
import com.google.inject._
import services._

@Singleton
class CategoryController @Inject()(categoryService: CategoryService)
 extends Controller {

 def index = Action {
 Ok(toJson(categoryService.list))
 }
}
```

5. Create a category service in foo\_scala/app/services/CategoryService.scala by adding the following content:

```
package services

trait CategoryService {
 def list: Seq[String]
}

class CategoryServiceImpl extends CategoryService {
```

```
 override def list: Seq[String] = Seq("Manager", "Employee",
"Contractor")
}
```

6. Add a new routes entry for the newly added action to foo\_scala/conf/routes:

```
GET /categories @controllers.CategoryController.index
```

7. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/categories
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /categories HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 35
<
* Connection #0 to host localhost left intact
["Manager", "Employee", "Contractor"]%
```

## **How it works...**

In this recipe, we configured our Play application to utilize Dependency injection in our controllers and service classes using Google Guice. We configured Guice in the `Global` settings file, which will contain our Guice-specific configurations.



# Utilizing MongoDB

For this recipe, we will explore how to utilize MongoDB, the popular NoSQL library, within a Play application. MongoDB is one of the most widely-used NoSQL databases, and it most certainly has been a viable option as a datastore for many modern web applications. We will be using the Scala module, play-plugins-salat, which is an Object relation mapping tool that uses the official MongoDB Scala driver Casbah. This will be a Scala-only recipe.

For more information about Casbah, please refer to <https://github.com/mongodb/casbah>.

# How to do it...

Let's take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare `play-plugins-salat` as a project dependency in `build.sbt`:

```
"se.radley" %% "play-plugins-salat" % "1.5.0"
```

3. Add additional salat and MongoDB directives to `build.sbt`:

```
import play.PlayImport.PlayKeys._
import play.twirl.sbt.Import.TwirlKeys

routesImport += "se.radley.plugin.salat.Binders._"
TwirlKeys.templateImports += "org.bson.types.ObjectId"
```

4. Declare the salat plugin in `foo_scala/conf/play.plugins`:

```
500:se.radley.plugin.salat.SalatPlugin
```

5. Declare the MongoDB instance information in `foo_scala/conf/application.conf`:

```
mongodb.default.db = "cookbookdb"
```

6. Modify `foo_scala/app/controllers/WarehouseController.scala` by adding the following content:

```
package controllers

import models._
import play.api.libs.json._
import play.api.mvc.{BodyParsers, Action, Controller}
import se.radley.plugin.salat.Binders.ObjectId

object WarehouseController extends Controller {
 implicit val objectIdReads =
 se.radley.plugin.salat.Binders.objectIdReads
 implicit val objectIdWrites =
 se.radley.plugin.salat.Binders.objectIdWrites
 implicit val warehouseWrites = Json.writes[Warehouse]
 implicit val warehouseReads = Json.reads[Warehouse]

 def index = Action {
 val list = Warehouse.list
 Ok(Json.toJson(list))
 }

 def create = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Warehouse]
 post.fold(
 errors => {
 BadRequest(Json.obj("status" ->"error", "message" ->
JsError.toFlatJson(errors)))
 }
)
 }
}
```

```

 },
 warehouse => {
 Warehouse.create(warehouse)
 Created(Json.toJson(warehouse))
 }
)
 }
}

```

7. Add new routes for the newly added action to `foo_scala/conf/routes`:

```

GET /warehouses controllers.WarehouseController.index
POST /warehouses controllers.WarehouseController.create

```

8. Add the collection mapping for the warehouse model to `foo_scala/app/models/Warehouse.scala`:

```

package models

import play.api.Play.current
import com.mongodb.casbah.commons.MongoDBObject
import com.novus.salat.dao._
import se.radley.plugin.salat._
import se.radley.plugin.salat.Binders._
import mongoContext._

case class Warehouse(id: Option[ObjectId] = Some(new ObjectId),
name: String, location: String)

object Warehouse extends ModelCompanion[Warehouse, ObjectId] {
 val dao = new SalatDAO[Warehouse, ObjectId](collection =
mongoCollection("warehouses")) {}

 def list = dao.find(ref = MongoDBObject()).toList
 def create(w: Warehouse) = dao.save(w)
}

```

9. Add the Mongo context to `foo_scala/app/models/mongoContext.scala`:

```

package models

import com.novus.salat.dao._
import com.novus.salat.annotations._
import com.mongodb.casbah.Imports._

import play.api.Play
import play.api.Play.current

package object mongoContext {
 implicit val context = {
 val context = new Context {
 val name = "global"
 override val typeHintStrategy = StringTypeHintStrategy(when =
TypeHintFrequency.WhenNecessary, typeHint = "_t")
 }
 context.registerGlobalKeyOverride(remapThis = "id",
toThisInstead = "_id")
 }
}

```

```
 context.registerClassLoader(Play.classloader)
 context
 }
}
```

10. Add a new warehouse record by accessing the warehouse post endpoint using curl:

```
$ curl -v -X POST http://localhost:9000/warehouses --header
"Content-type: application/json" --data '{"name":"Warehouse A",
"location":"Springfield"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /warehouses HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 48
>
* upload completely sent off: 48 out of 48 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 47
<
* Connection #0 to host localhost left intact
{"name":"Warehouse A","location":"Springfield"}
```

11. View all warehouse records by accessing the warehouse index endpoint using curl:

```
$ curl -v http://localhost:9000/warehouses
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /warehouses HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 241
<
* Connection #0 to host localhost left intact
[{"id":"5490fde9e0820cf6df38584c", "name": "Warehouse
A", "location": "Springfield"}]%
```

# How it works...

In this recipe, we created a new URL route and action that will insert and retrieve warehouse records from a MongoDB instance. We used the Play module play-plugins-salat and configured the connection in `foo_scala/conf/application.conf`. We then mapped our Mongo collection in the warehouse model class:

```
case class Warehouse(id: Option[ObjectId] = Some(new ObjectId), name: String, location: String)
```

Next, we invoked the appropriate warehouse companion object methods from the warehouse controller:

```
val list = Warehouse.list
Warehouse.create(warehouse)
```

We also declared our JSON binders for the warehouse model and MongoDB's `ObjectId` in the warehouse controller:

```
implicit val objectIdReads =
se.radley.plugin.salat.Binders.objectIdReads
implicit val objectIdWrites =
se.radley.plugin.salat.Binders.objectIdWrites
implicit val warehouseWrites = Json.writes[Warehouse]
implicit val warehouseReads = Json.reads[Warehouse]
```



# Utilizing MongoDB and GridFS

For this recipe, we will explore how to store and deliver files with Play applications by using MongoDB and GridFS. We will continue by adding to the previous recipe. As with the previous recipe, this recipe will be Scala only.

# How to do it...

Let's take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_scala/app/controllers/WarehouseController.scala` by adding the following content:

```
import java.text.SimpleDateFormat
import play.api.libs.iteratee.Enumerator

def upload = Action(parse.multipartFormData) { request =>
 request.body.file("asset") match {
 case Some(asset) => {
 val gridFs = Warehouse.assets
 val uploadedAsset = gridFs.createFile(asset.ref.file)
 uploadedAsset.filename = asset.filename
 uploadedAsset.save()

 Ok("Asset is available at
http://localhost:9000/warehouses/assets/%s".format(uploadedAsset.id))
 }
 case None => {
 BadRequest
 }
 }
}

def retrieveFile(id: ObjectId) = Action {
 import com.mongodb.casbah.Implicits._
 import play.api.libs.concurrent.Execution.Implicits._

 val gridFs = Warehouse.assets

 gridFs.findOne(Map("_id" -> id)) match {
 case Some(f) => Result(
 ResponseHeader(OK, Map(
 CONTENT_LENGTH -> f.length.toString,
 CONTENT_TYPE -> f.contentType.getOrElse(BINARY),
 DATE -> new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss
'GMT'", java.util.Locale.US).format(f.uploadDate)
)),
 Enumerator.fromStream(f.inputStream)
)

 case None => NotFound
 }
}
```

3. Add new routes for the newly added action to `foo_scala/conf/routes`:

```
POST /warehouses/assets/upload
controllers.WarehouseController.upload
```

```
GET /warehouses/assets/:id
controllers.WarehouseController.retrieveFile(id: ObjectId)
```

4. Modify the collection mapping for the warehouse model in `foo_scala/app/models/Warehouse.scala`:

```
val assets = gridFS("assets")

def upload(asset: File) = {
 assets.createFile(asset)
}

def retrieve(filename: String) = {
 assets.find(filename)
}
```

5. Upload a new warehouse asset file by accessing the warehouse upload endpoint using curl:

```
$ curl -v http://localhost:9000/warehouses/assets/upload -F
"asset=@/tmp/1.jpg"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /warehouses/assets/upload HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-Length: 13583
> Expect: 100-continue
> Content-Type: multipart/form-data; boundary=-----
---4a001bdeff39c089
>
< HTTP/1.1 100 Continue
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 86
<
* Connection #0 to host localhost left intact
Asset is available at
http://localhost:9000/warehouses/assets/549121fbe082fc374fa6cb63%
```

6. Verify that our file delivery URL route is working by accessing the URL in a web browser that is part of the output of the previous step:

```
http://localhost:9000/warehouses/assets/549121fbe082fc374fa6cb63
```

## How it works...

In this recipe, we created new URL routes and actions that will be used to upload and retrieve warehouse asset files in a MongoDB instance using GridFS. We added the GridFS reference to our collection mapping file in `foo_scala/app/models/Warehouse.scala`:

```
val assets = gridFS("assets")
```

We then added the respective methods for file upload and retrieval:

```
def upload(asset: File) = {
 assets.createFile(asset)
}

def retrieve(filename: String) = {
 assets.find(filename)
}
```

Next, we created the actions in

`foo_scala/app/controllers/WarehouseController.scala`, which will handle the actual file upload and retrieval requests.



# Utilizing Redis

For this recipe, we will explore how Play applications integrate with Redis using Play cache. Redis is a widely used key-value database, usually utilized as an intermediary object cache for modern web applications. This recipe requires a running Redis instance that our Play 2 web application can interface with.

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare Redis as a project dependency in `build.sbt`:

```
"com.typesafe.play.plugins" %% "play-plugins-redis" % "2.3.1"
```

3. Declare the repository hosting Sedis, a library dependency of `play-plugins-redis` in `build.sbt`:

```
resolvers += "Sedis repository" at "http://pk11-scratch.googlecode.com/svn/trunk/"
```

4. Enable the `play-mailer` plugin by declaring it in `foo_java/conf/play.plugins`:

```
550:com.typesafe.plugin.RedisPlugin
```

5. Specify your Redis host information in `foo_java/conf/application.conf`:

```
ehcacheplugin=disabled
redis.uri="redis://127.0.0.1:6379"
```

6. Modify `foo_java/app/controllers/Application.java` by adding the following code:

```
import play.cache.*;

public static Result displayFromCache() {
 final String key = "myKey";
 String value = (String) Cache.get(key);

 if (value != null && value.trim().length() > 0) {
 return ok("Retrieved from Cache: " + value);
 } else {
 Cache.set(key, "Let's Play with Redis!");
 return ok("Setting key value in the cache");
 }
}
```

7. Add a new routes entry for the newly added action to `foo_java/conf/routes`:

```
GET /cache controllers.Application.displayFromCache
```

8. Request our new route and examine the response body to confirm that our `displayFromCache` action is setting the key value for the first time:

```
$ curl -v http://localhost:9000/cache
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /cache HTTP/1.1
```

```

> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Setting key value in the cache%

```

- Request the /cache route again to be able to view the value of the cache key:

```

$ curl -v http://localhost:9000/cache
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /cache HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 43
<
* Connection #0 to host localhost left intact
Retrieved from Cache: Let's Play with Redis!%

```

For Scala, we need to take the following steps:

- Run the foo\_scala application with Hot-Reloading enabled:

```
activator "~run"
```

- Declare Redis as a project dependency in build.sbt:

```
"com.typesafe.play.plugins" %% "play-plugins-redis" % "2.3.1"
```

- Declare the repository hosting Sedis, a library dependency of play-plugins-redis, in build.sbt:

```
resolvers += "Sedis repository" at "http://pk11-
scratch.googlecode.com/svn/trunk/"
```

- Enable the play-mailer plugin by declaring it in foo\_scala/conf/play.plugins:

```
550:com.typesafe.plugin.RedisPlugin
```

- Specify your Redis host information in foo\_scala/conf/application.conf:

```
ehcacheplugin=disabled
redis.uri="redis://127.0.0.1:6379"
```

- Modify foo\_scala/app/controllers/Application.scala by adding the following code:

```

import scala.util.Random
import play.api.cache._
import play.api.Play.current

def displayFromCache = Action {
 val key = "myKey"
 Cache.getAs[String](key) match {
 case Some(myKey) => {
 Ok("Retrieved from Cache: %s".format(myKey))
 }
 case None => {
 Cache.set(key, "Let's Play with Redis!")
 Ok("Setting key value in the cache")
 }
 }
}

```

7. Add a new routes entry for the newly added action to foo\_scala/conf/routes:

```
GET /cache controllers.Application.displayFromCache
```

8. Request our new route and examine the response body to confirm that our displayFromCache action is setting the key value for the first time:

```

$ curl -v http://localhost:9000/cache
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /cache HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Setting key value in the cache%

```

9. Request the /cache route again to be able to view the value of the cache key:

```

$ curl -v http://localhost:9000/cache
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /cache HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 43
<
* Connection #0 to host localhost left intact
Retrieved from Cache: Let's Play with Redis!%

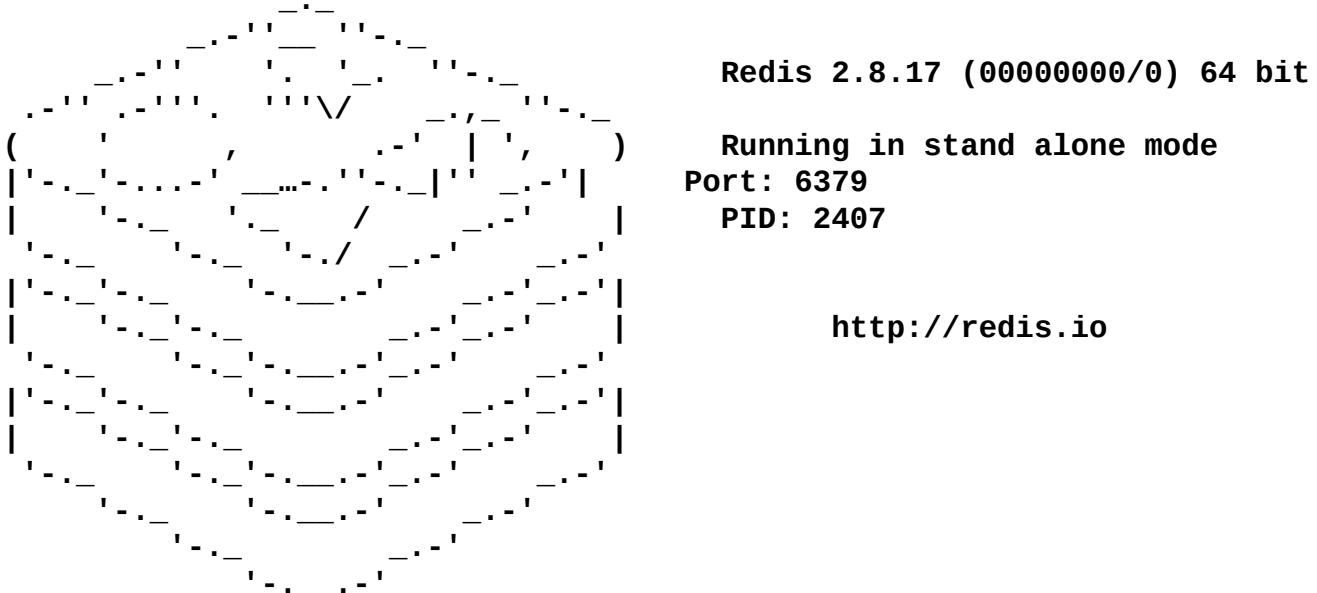
```



# How it works...

In this recipe, we created a new URL route and action that will interact with our Redis instance. To follow on with this recipe, you will need the following running Redis instance to connect to:

```
$ redis-server /usr/local/etc/redis.conf
[2407] 14 Apr 12:44:20.623 * Increased maximum number of open files to
10032 (it was originally set to 2560).
```



Redis 2.8.17 (00000000/0) 64 bit

Running in stand alone mode

Port: 6379

PID: 2407

<http://redis.io>

```
[2407] 14 Apr 12:44:20.631 # Server started, Redis version 2.8.17
[2407] 14 Apr 12:44:20.632 * DB loaded from disk: 0.001 seconds
[2407] 14 Apr 12:44:20.632 * The server is now ready to accept connections
on port 6379
```

For more information about installing and running a Redis server, please refer to <http://redis.io/topics/quickstart>.

We configured the Play Redis module by declaring the necessary dependencies and repository settings in `build.sbt`. We then configured the connection to the Redis instance in `conf/application.conf`. Finally, we loaded the Redis play-plugin in `conf/play.plugins`:

```
550:com.typesafe.plugin.RedisPlugin
```

The `displayFromCache` action, when invoked, has two distinct functions. First, it attempts to retrieve a value from the cache. If it is able to retrieve a value from the Redis cache, it prints the contents of the value in the response body. If it is unable to retrieve a value from the Redis cache, it sets a random string value to the key and prints a status message in the response body.

We then used `curl` to test out this new route and accessed the route twice; the action printed out two different messages in the response body.



# Integrating Play application with Amazon S3

For this recipe, we will explore how Play applications can upload files directly to **Amazon Web Services (AWS) S3**, a popular cloud storage solution.

For more information about S3, please refer to <http://aws.amazon.com/s3/>.

# How to do it...

For Java, we need to take the following steps:

1. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare play-s3 as a project dependency in build.sbt:

```
"com.amazonaws" % "aws-java-sdk" % "1.3.11"
```

3. Specify your AWS credentials in foo\_java/conf/application.conf:

```
aws.accessKeyId="YOUR S3 ACCESS KEY"
aws.secretKey="YOUR S3 SECRET KEY"
```

```
fooscala.s3.bucketName="YOUR S3 BUCKET NAME"
```

4. Modify foo\_java/app/controllers/Application.java by adding the following code:

```
import com.amazonaws.auth.*;
import com.amazonaws.services.s3.*;
import com.amazonaws.services.s3.model.*;

public static Result s3Upload() {
 return ok(views.html.s3.render());
}

public static Result submits3Upload() {
 Http.MultipartFormData body =
request().body().asMultipartFormData();
 Http.MultipartFormData.FilePart profileImage =
body.getFile("profile");

 if (profileImage != null) {
 try {
 File file = profileImage.getFile();
 String filename = profileImage.getFilename();

 String accessKey =
Play.application().configuration().getString("aws.accessKeyId");
 String secret =
Play.application().configuration().getString("aws.secretKey");
 String bucketName =
Play.application().configuration().getString("fooscala.s3.bucketName");

 try {
 AWSCredentials awsCredentials = new
BasicAWSCredentials(accessKey, secret);
 AmazonS3 s3Client = new
AmazonS3Client(awsCredentials);
 AccessControlList acl = new AccessControlList();
 acl.grantPermission(GroupGrantee.AllUsers,
Permission.Read);
```

```

 s3Client.createBucket(bucketName);
 s3Client.putObject(new PutObjectRequest(bucketName,
filename, file).withAccessControlList(acl));

 String img = "http://" + bucketName+
".s3.amazonaws.com/" + filename;
 return ok("Image uploaded: " + img);
 } catch (Exception e) {
 e.printStackTrace();
 return ok("Image was not uploaded");
 }

 } catch(Exception e) {
 return internalServerError(e.getMessage());
 }
} else {
 return badRequest();
}
}
}

```

- Add a new routes entry for the newly added action to foo\_java/conf/routes:

```

GET /s3_upload controllers.Application.s3Upload
POST /s3_upload controllers.Application.submitS3Upload

```

- Add the S3 file upload submission View template to foo\_java/app/views/s3.scala.html:

```

@helper.form(action = routes.Application.submitS3Upload, 'enctype -
> "multipart/form-data") {
 <input type="file" name="profile">
 <p>
 <input type="submit">
 </p>
}

```

For Scala, we need to take the following steps:

- Run the foo\_scala application with Hot-Reloading enabled:

```
activator ~run
```

- Declare play-s3 as a project dependency in build.sbt:

```
"nl.rhinofly" %% "play-s3" % "5.0.2",
```

- Declare the custom repository where the play-s3 module is hosted:

```
resolvers += "Rhinofly Internal Repository" at "http://maven-
repository.rhinofly.net:8081/artifactory/libs-release-local"
```

- Specify your AWS credentials in foo\_scala/conf/application.conf:

```
aws.accessKeyId="YOUR S3 ACCESS KEY"
aws.secretKey="YOUR S3 SECRET KEY"
```

```
fooscala.s3.bucketName="YOUR S3 BUCKET NAME"
```

5. Modify `foo_scala/app/controllers/Application.scala` by adding the following code:

```
import play.api.Play.current
import fly.play.s3._
def s3Upload = Action {
 Ok(s3())
}

def submitS3Upload = Action(parse.multipartFormData) { request =>
 import play.api.Play

 request.body.file("profile") match {
 case Some(profileImage) => {
 val bucketName =
 Play.current.configuration.getString("fooscala.s3.bucketName").get
 val bucket = S3(bucketName)

 val filename = profileImage.filename
 val contentType = profileImage.contentType
 val byteArray = Files.toByteArray(profileImage.ref.file)

 val result = bucket.add(BucketFile(filename,
 contentType.get, byteArray, Option(PUBLIC_READ), None))
 val future = Await.result(result, 10 seconds)
 Ok("Image uploaded to:
http://%s.s3.amazonaws.com/%s".format(bucketName, filename))
 }
 case None => {
 BadRequest
 }
 }
}
```

6. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:

```
GET /s3_upload controllers.Application.s3Upload
POST /s3_upload controllers.Application.submitS3Upload
```

Add the s3 file upload submission view template in `foo_scala/app/views/s3.scala.html`:

```
@helper.form(action = routes.Application.submitS3Upload, 'enctype -> "multipart/form-data") {
 <input type="file" name="profile">
 <p>
 <input type="submit">
 </p>
}
```

# How it works...

In this recipe, we created a new URL route and action that received the uploaded file. We then pushed this file to Amazon S3 using the RhinoFly S3 module by supplying the S3 access and secret keys in `conf/application.conf`. We also specified our S3 bucket name in `conf/application.conf` for future use. We are able to retrieve this value by using Play's configuration API:

```
val bucketName =
Play.current.configuration.getString("fooscala.s3.bucketName").get
```

We then printed the location of the uploaded file to the response body for easy verification:

```
Ok("Image uploaded to:
http://%s.s3.amazonaws.com/%s".format(bucketName, filename))
```

You should now see the response in the web browser with the following text:

```
Image uploaded to:
http://<YOUR_BUCKET_NAME>.s3.amazonaws.com/<FILENAME>
```

You can also use the AWS Management Console to verify the file upload in the S3 section:

The screenshot shows the AWS Management Console S3 Management Console interface. The browser address bar displays `https://console.aws.amazon.com/s3/home?region=ap-southeast-1`. The main content area shows a table of objects in the 'cookbook' bucket. The table has columns for Name, Storage Class, Size, and Last Modified. One object, 'a1ba628b-b611-4077-93c8-7a2f07f801f7', is selected, indicated by a blue border around its row. Other objects listed include various file names and folder names like 'raw', 'tb', and 'thumbnails'. The bottom of the page includes standard AWS footer links for Privacy Policy and Terms of Use, along with a Feedback button.

Name	Storage Class	Size	Last Modified
2b34caa2-39e6-41bf-9f59-c5962c489d42	Standard	13 KB	Wed Dec 17 18:08:57 GMT+800 2014
63c975d2-cf77-4ea1-9195-5e18e850fd35	Standard	13 KB	Wed Dec 17 18:19:03 GMT+800 2014
9487eae1-b898-4b57-93ca-e935467ac5a3	Standard	13 KB	Wed Dec 17 18:11:54 GMT+800 2014
a1ba628b-b611-4077-93c8-7a2f07f801f7	Standard	13 KB	Wed Dec 17 18:15:07 GMT+800 2014
e4d4b7b8-1dac-4090-8ac7-3616f96ada54	Standard	13 KB	Wed Dec 17 18:19:29 GMT+800 2014
ffeb3aff-33b6-4e5f-ba60-40a3c2929e01	Standard	13 KB	Wed Dec 17 18:11:06 GMT+800 2014
raw	--	--	--
tb	--	--	--
thumbnails	--	--	--



# Integrating with Play application Typesafe Slick

In this recipe, we will explore how we can integrate Typesafe Slick with Play applications using the `play-slick` module. Typesafe Slick is a relational mapping tool built on Scala and is handy for managing database objects like native Scala types.

# How to do it...

We need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare `play-slick` as a project dependency in `build.sbt`:

```
"com.typesafe.play" %% "play-slick" % "0.8.1"
```

3. Specify your database host information in `foo_scala/conf/application.conf`:

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
slick.default="models.*"
```

4. Create the new supplier controller in

`foo_scala/app/controllers/SupplierController.scala` with the following content:

```
package controllers

import play.api.mvc.{BodyParsers, Controller}
import play.api.db.slick._
import play.api.libs.json._
import play.api.Play.current
import models._

object SupplierController extends Controller {
 implicit val supplierWrites = Json.writes[Supplier]
 implicit val supplierReads = Json.reads[Supplier]

 def index = DBAction { implicit rs =>
 Ok(Json.toJson(Suppliers.list))
 }

 def create = DBAction(BodyParsers.parse.json) { implicit rs =>
 val post = rs.request.body.validate[Supplier]
 post.fold(
 errors => {
 BadRequest(Json.obj("status" ->"error", "message" -> JsError.toFlatJson(errors)))
 },
 supplier => {
 Suppliers.create(supplier)
 Created(Json.toJson(supplier))
 }
)
 }
}
```

5. Create a Slick mapping for suppliers in the

foo\_scala/app/models/Suppliers.scala file with the following content:

```
package models

import scala.slick.driver.H2Driver.simple._
import scala.slick.lifted.Tag

case class Supplier(id: Option[Int], name: String, contactNo: String)

class Suppliers(tag: Tag) extends Table[Supplier](tag, "SUPPLIERS") {
 def id = column[Int]("ID", O.PrimaryKey, O.AutoInc)
 def name = column[String]("NAME")
 def contactNo = column[String]("CONTACT_NO")

 def * = (id.?, name, contactNo) <=> (Supplier.tupled,
 Supplier.unapply)
}

object Suppliers {
 val suppliers = TableQuery[Suppliers]

 def list(implicit s: Session) = suppliers.sortBy(m =>
 m.name.asc).list
 def create(supplier: Supplier)(implicit s: Session) =
 suppliers.insert(supplier)
}
```

6. Add the new routes for the Supplier controller in foo\_scala/conf/routes:

```
GET /suppliers controllers.SupplierController.index
POST /suppliers controllers.SupplierController.create
```

7. Request the new Post route and examine the response headers and body to confirm that the record was inserted in the database:

```
$ curl -v -X POST http://localhost:9000/suppliers --header
"Content-type: application/json" --data '{"name":"Ned Flanders",
"contactNo":"555-1234"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /suppliers HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 47
>
* upload completely sent off: 47 out of 47 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 46
<
* Connection #0 to host localhost left intact
```

```
{"name": "Ned Flanders", "contactNo": "555-1234"}%
```

8. Request the listing route and verify that it is, in fact, returning records from the database:

```
$ curl -v http://localhost:9000/suppliers
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /suppliers HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 110
<
* Connection #0 to host localhost left intact
[{"id":1,"name":"Maud Flanders","contactNo":"555-1234"}, {"id":2,"name":"Ned Flanders","contactNo":"712-1234"}]%
```

## How it works...

In this recipe, we created a new URL route and action that will create and retrieve suppliers from the H2 database. We used Typesafe Slick as the relational mapping tool for creating queries and inserts. We started by declaring the required dependencies in `build.sbt`. Next, we defined the mapping properties for suppliers in `foo_scala/app/models/Supplier.scala`.

In the mapping file, we declared our case class `Supplier`. We also declared our Slick table mapping class. Lastly, we added our `Suppliers` object class that should ideally contain all the required functions for data insertion and querying. We added the appropriate routes to the `conf/routes` file and ran the database evolution. This allows Slick to automatically manage table creation and column syncing. To test our implementation, we used curl to request our POST and GET endpoints to be able to view the response headers and body.



# Utilizing play-mailer

For this recipe, we will explore how Play applications can send e-mails. We will use the Play module play-mailer to achieve this. We will be utilizing Mandrill, a cloud e-mailer service, to send out e-mails. For more information about Mandrill, please refer to <https://mandrill.com/>.

# How to do it...

For Java, we need to take the following steps:

1. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare play-mailer as a project dependency in build.sbt:

```
"com.typesafe.play.plugins" %% "play-plugins-mailer" % "2.3.1"
```

3. Enable the play-mailer plugin by declaring it in foo\_java/conf/play.plugins:

```
1500:com.typesafe.plugin.CommonsMailerPlugin
```

4. Specify your smtp host information in foo\_java/conf/application.conf:

```
smtp.host=smtp.mandrillapp.com
smtp.port=25
smtp.user="YOUR OWN USER HERE"
smtp.password="YOUR OWN PASSWORD HERE"
smtp.mock=true
```

5. Modify foo\_java/app/controllers/Application.java by adding the following code:

```
import play.libs.F;
import play.libs.F.Function;
import play.libs.F.Promise;
import com.typesafe.plugin.*;

public static Promise<Result> emailSender() {
 Promise<Boolean> emailResult = Promise.promise(
 new F.Function0<Boolean>() {
 @Override
 public Boolean apply() throws Throwable {
 try {
 MailerAPI mail =
play.Play.application().plugin(MailerPlugin.class).email();
 mail.setSubject("mailer");
 mail.setRecipient("ginduc@dynamicobjx.com");
 mail.setFrom("Play Cookbook
<noreply@email.com>");
 mail.send("text");

 return true;
 } catch (Exception e) {
 e.printStackTrace();
 return false;
 }
 }
);
);

 return emailResult.map(
 new Function<Boolean, Result>() {
```

```

 @Override
 public Result apply(Boolean sent) throws Throwable {
 if (sent) {
 return ok("Email sent!");
 } else {
 return ok("Email was not sent!");
 }
 }
);
}
}

```

- Add a new routes entry for the newly added action to `foo_java/conf/routes`:

```
POST /send_email controllers.Application.emailSender
```

- Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v -X POST http://localhost:9000/send_email
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /send_email HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 11
<
* Connection #0 to host localhost left intact
Email sent!%
```

For Scala, we need to take the following steps:

- Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

- Declare play-mailer as a project dependency in `build.sbt`:

```
"com.typesafe.play.plugins" %% "play-plugins-mailer" % "2.3.1"
```

- Enable the play-mailer plugin by declaring it in `foo_scala/conf/play.plugins`:

```
1500:com.typesafe.plugin.CommonsMailerPlugin
```

- Specify your smtp host information in `foo_scala/conf/application.conf`:

```
smtp.host=smtp.mandrillapp.com
smtp.port=25
smtp.user="YOUR OWN USER HERE"
smtp.password="YOUR OWN PASSWORD HERE"
smtp.mock=true
```

5. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
import scala.concurrent._
import com.typesafe.plugin._
import play.api.libs.concurrent.Execution.Implicits._
import play.api.Play.current

def emailSender = Action.async {
 sendEmail.map { messageId =>
 Ok("Sent email with Message ID: " + messageId)
 }
}

def sendEmail = Future {
 val mail = use[MailerPlugin].email

 mail.setSubject("Play mailer")
 mail.setRecipient("ginduc@dynamicobjx.com")
 mail.setFrom("Play Cookbook <noreply@email.com>")
 mail.send("text")
}
```

6. Add a new routes entry for the newly added action to `foo_scala/conf/routes`:

```
POST /send_email controllers.Application.emailSender
```

7. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v -X POST http://localhost:9000/send_email
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /send_email HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 30
<
* Connection #0 to host localhost left intact
Sent email with Message ID: ()%
```

## How it works...

In this recipe, we created a new URL route and action that will invoke our newly added `sendMail` function. We declared the module dependency in `foo_scala/build.sbt` and specified our `smtp` server settings in `foo_scala/conf/application.conf`. After this, we invoked the URL route using `curl` in the terminal to test out our e-mail sender. You should now receive the e-mail in your e-mail client software.



# Integrating Bootstrap and WebJars

For this recipe, we will explore how we can integrate and utilize the popular frontend framework Bootstrap with a Play 2 web application. We will integrate Bootstrap using WebJars, which is a tool to package frontend libraries into JAR files that can then be easily managed (in our case, by sbt).

# How to do it...

For Java, we need to take the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare Bootstrap and WebJars as a project dependency in `build.sbt`:

```
"org.webjars" % "bootstrap" % "3.3.1",
"org.webjars" %% "webjars-play" % "2.3.0"
```

3. Modify `foo_java/app/controllers/Application.java` by adding the following code:

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

4. Add the new route entries to `foo_java/conf/routes`:

GET	/webjars/*file	controllers.WebJarAssets.at(file)
GET	/bootstrapped	controllers.Application.bootstrapped

5. Create the new layout View template in `foo_java/app/views/mainLayout.scala.html` with the following content:

```
@(title: String)(content: Html)<!DOCTYPE html>

<html>
<head>
 <meta charset="utf-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-
scale=1">
 <meta name="description" content="">
 <meta name="author" content="">
 <title>@title</title>
 <link rel="shortcut icon" type="image/png"
href='@routes.Assets.at("images/favicon.png")'>
 <link rel="stylesheet" media="screen"
href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.cs
s"))' />
 <link rel="stylesheet" media="screen"
href='@routes.Assets.at("stylesheets/app.css")' />
 <style>
 body {
 padding-top: 50px;
 }
 </style>
</head>
<body>
<nav class="navbar navbar-inverse navbar-fixed-top" role="navigat
ion">
 <div class="container-fluid">
```

```

<div class="navbar-header">
 <button type="button" class="navbar-toggle collapsed" data-
 toggle="collapse" data-target="#navbar" aria-expanded="false" aria-
 controls="navbar">
 Toggle navigation

 </button>
 Admin
</div>
<div id="navbar" class="navbar-collapse collapse">
 <ul class="nav navbar-nav navbar-right">
 Dashboard
 Settings
 Profile
 Help

</div>
</div>
</nav>

<div class="container-fluid">
 <div class="row">
 <div class="col-sm-3 col-md-2 sidebar">
 <ul class="nav nav-sidebar">
 <li class="active">Overview (current)
 Reports
 Analytics
 Export

 <ul class="nav nav-sidebar">
 Users
 Audit Log

 <ul class="nav nav-sidebar">
 Sign out

 </div>
 <div class="col-sm-9 col-sm-offset-3 col-md-10 col-md-offset-2
main">
 @content
 </div>
 </div>
</div>

</body>
</html>

```

6. Create the Bootstrapped View template in  
`foo_java/app/views/bootstrapped.scala.html` with the following content:

```

@mainLayout("Bootstrapped") {
<div class="hero-unit">
 <h1>Hello, world!</h1>

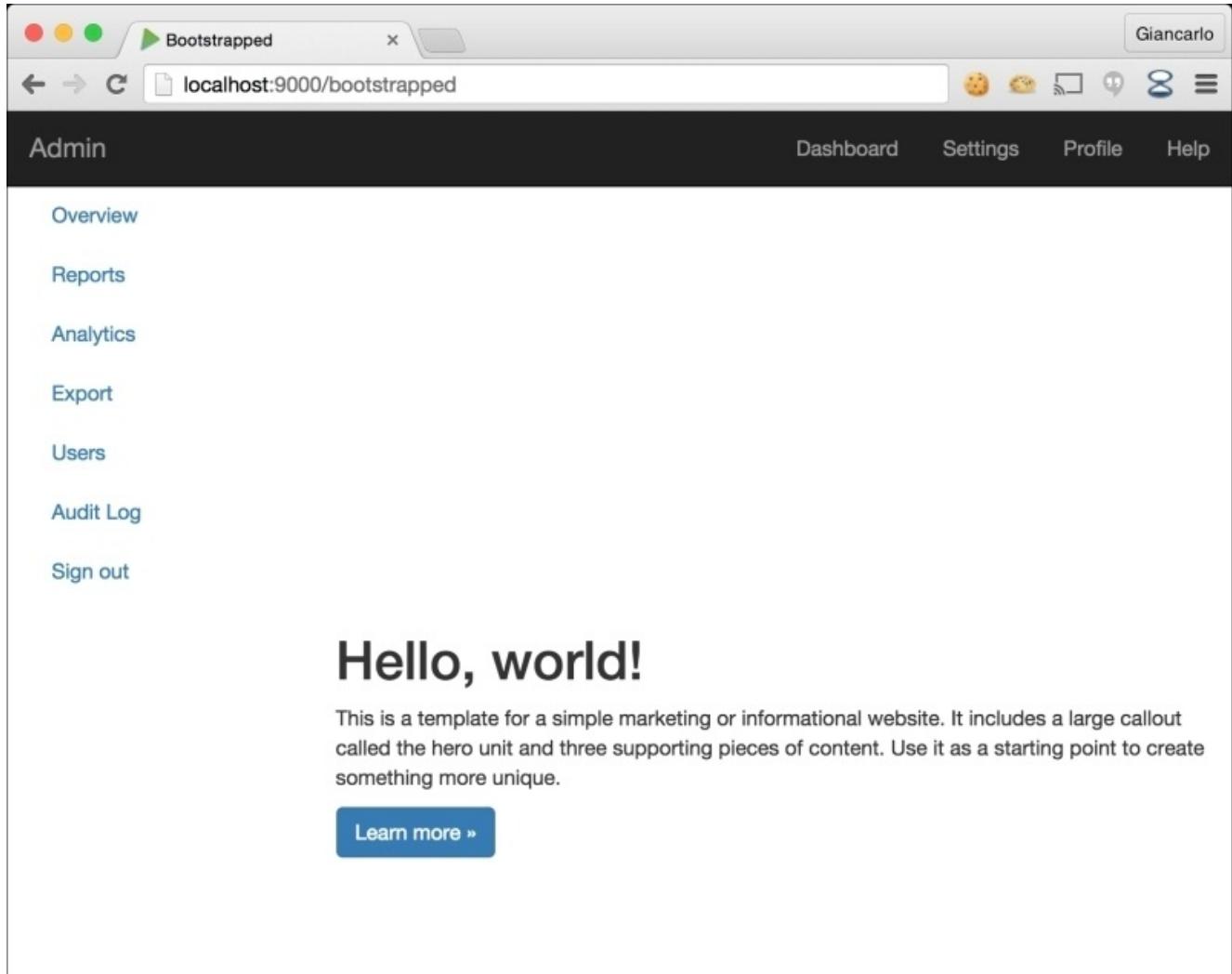
```

```

<p>This is a template for a simple marketing or informational website. It includes a large callout called the hero unit and three supporting pieces of content. Use it as a starting point to create something more unique.</p>
<p>Learn more »</p>
</div>
}

```

7. Request our new Bootstrapped route (<http://localhost:9000/bootstrapped>) using a web browser and examine the rendered page using a Bootstrap template:



For Scala, we need to take the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator ~run
```

2. Declare Bootstrap and WebJars as a project dependency in `build.sbt`:

```
"org.webjars" % "bootstrap" % "3.3.1",
"org.webjars" %% "webjars-play" % "2.3.0"
```

3. Modify `foo_scala/app/controllers/Application.scala` by adding the following

action:

```
def bootstrapped = Action {
 Ok(views.html.bootstrapped())
}
```

4. Add the new route entries to `foo_scala/conf/routes`:

```
GET /webjars/*file controllers.WebJarAssets.at(file)
GET /bootstrapped controllers.Application.bootstrapped
```

5. Create the new layout view template in `foo_scala/app/views/mainLayout.scala.html` with the following content:

```
@(title: String)(content: Html)<!DOCTYPE html>

<html>
<head>
 <meta charset="utf-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-
scale=1">
 <meta name="description" content="">
 <meta name="author" content="">
 <title>@title</title>
 <link rel="shortcut icon" type="image/png"
 href='@routes.Assets.at("images/favicon.png")'>
 <link rel="stylesheet" media="screen"
 href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.cs
s"))' />
 <link rel="stylesheet" media="screen"
 href='@routes.Assets.at("stylesheets/app.css")' />
 <style>
 body {
 padding-top: 50px;
 }
 </style>
</head>
<body>
<nav class="navbar navbar-inverse navbar-fixed-top" role="navigat
ion">
 <div class="container-fluid">
 <div class="navbar-header">
 <button type="button" class="navbar-toggle collapsed" data-
 toggle="collapse" data-target="#navbar" aria-expanded="false" aria-
 controls="navbar">
 Toggle navigation

 </button>
 Admin
 </div>
 <div id="navbar" class="navbar-collapse collapse">
 <ul class="nav navbar-nav navbar-right">
 Dashboard
```

```

 Settings
 Profile
 Help

</div>
</div>
</nav>

<div class="container-fluid">
 <div class="row">
 <div class="col-sm-3 col-md-2 sidebar">
 <ul class="nav nav-sidebar">
 <li class="active">Overview (current)
 Reports
 Analytics
 Export

 <ul class="nav nav-sidebar">
 Users
 Audit Log

 <ul class="nav nav-sidebar">
 Sign out

 </div>
 <div class="col-sm-9 col-sm-offset-3 col-md-10 col-md-offset-2
main">
 @content
 </div>
 </div>
</div>

</body>
</html>

```

## 6. Create the bootstrapped View template in

`foo_scala/app/views/bootstrapped.scala.html` with the following content:

```

@mainLayout("Bootstrapped") {
<div class="hero-unit">
 <h1>Hello, world!</h1>
 <p>This is a template for a simple marketing or informational
website. It includes a large callout called the hero unit and three
supporting pieces of content. Use it as a starting point to create
something more unique.</p>
 <p>Learn more »
</p>
</div>
}

```

## 7. Request our new bootstrapped route (`http://localhost:9000/bootstrapped`) using a web browser and examine the rendered page using a Bootstrap template:

A screenshot of a web browser window titled "Bootstrapped". The address bar shows "localhost:9000/bootstrapped". The user "Giancarlo" is logged in. The left sidebar is titled "Admin" and contains links: Overview, Reports, Analytics, Export, Users, Audit Log, and Sign out. The main content area features a large hero unit with the text "Hello, world!". Below it is a descriptive paragraph and a blue "Learn more »" button.

Admin

Overview

Reports

Analytics

Export

Users

Audit Log

Sign out

# Hello, world!

This is a template for a simple marketing or informational website. It includes a large callout called the hero unit and three supporting pieces of content. Use it as a starting point to create something more unique.

[Learn more »](#)

## How it works...

In this recipe, instead of downloading Bootstrap separately and managing different versions manually, we used the Play module and WebJars, declaring Bootstrap as a frontend dependency in `build.sbt`. We created the new View templates containing the Bootstrap template. We then created a new URL route that will utilize these new Bootstrap-based views.



# Chapter 4. Creating and Using Web APIs

In this chapter, we will cover the following recipes:

- Creating a `POST` API endpoint
- Creating a `GET` API endpoint
- Creating a `PUT` API endpoint
- Creating a `DELETE` API endpoint
- Securing API endpoints with HTTP basic authentication
- Consuming external web APIs
- Using the Twitter API with OAuth

# Introduction

In this chapter, we will look into creating REST APIs and interfacing with other external web-based APIs, in our case, the Twitter API.

With the increasing popularity of data exchanges between independent web services, REST APIs have become a popular approach not only to consume external data, but also to receive incoming data for further processing and persistence as well as exposing data to authorized clients. Based on the RESTful API spec, the HTTP method `POST` is used to insert new records and the HTTP method `GET` is used to retrieve data. The HTTP method `PUT` is used to update existing records and lastly, the HTTP method `DELETE` is used to remove records.

We will see how we can utilize different Play 2.0 libraries to build our own REST API endpoints and access other web-based APIs using the new Play WS library.



# Creating a POST API endpoint

In this recipe, we will explore how to use Play 2.0 to create a RESTful POST endpoint to add new records to our API.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create a new products controller in `foo_java/app/controllers/Products.java` with the following content:

```
package controllers;

import java.util.*;
import play.data.Form;
import play.mvc.*;
import models.Product;
import static play.libs.Json.toJson;

public class Products extends Controller {
 public static Map<String, Product> products = new
HashMap<String, Product>();

 @BodyParser.Of(BodyParser.Json.class)
 public static Result create() {
 try {
 Form<Product> form =
Form.form(Product.class).bindFromRequest();

 if (form.hasErrors()) {
 return badRequest(form.errorsAsJson());
 } else {
 Product product = form.get();
 products.put(product.getSku(), product);
 return created(toJson(product));
 }
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
 }
}
```

3. Create a product model class in `foo_java/app/models/Product.java`:

```
package models;

import play.data.validation.Constraints;

public class Product implements java.io.Serializable {
 @Constraints.Required
 private String sku;

 @Constraints.Required
 private String title;
```

```

 public String getSku() {
 return sku;
 }
 public void setSku(String sku) {
 this.sku = sku;
 }
 public String getTitle() {
 return title;
 }
 public void setTitle(String title) {
 this.title = title;
 }
 }
}

```

4. Add a new route entry for the newly added action in foo\_java/conf/routes:

```
POST /api/products controllers.Products.create
```

5. Request the new route and examine the response body to confirm:

```

$ curl -v -X POST http://localhost:9000/api/products --data
'{"sku":"abc", "title":"Macbook Pro Retina"}' --header "Content-type:
application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 43
>
* upload completely sent off: 43 out of 43 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 42
<
* Connection #0 to host localhost left intact
{"sku":"abc","title":"Macbook Pro Retina"}%

```

For Scala, we need to perform the following steps:

1. Run the foo\_scala application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create a new products controller in foo\_scala/app/controllers/Products.scala with the following content:

```

package controllers

import models.Product
import play.api.libs.json.{JsError, Json}
import play.api.libs.json.Json._
import play.api.mvc._
```

```

object Products extends Controller {
 implicit private val productWrites = Json.writes[Product]
 implicit private val productReads = Json.reads[Product]
 private val products =
 scala.collection.mutable.ListBuffer[Product] =
 scala.collection.mutable.ListBuffer[Product]()
}

def create = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Product]
 post.fold(
 errors => BadRequest(Json.obj("message" ->
 JsError.toFlatJson(errors))),
 p => {
 try {
 products += p
 Created(Json.toJson(p))
 } catch {
 case e: Exception => InternalServerError(e.getMessage)
 }
 }
)
}
}

```

3. Create a product model class in foo\_scala/app/models/Product.scala:

```

package models

case class Product(sku: String, title: String)

```

4. Add a new route entry for the newly added action in foo\_scala/conf/routes:

```
POST /api/products controllers.Products.create
```

5. Request the new route and examine the response body to confirm:

```

$ curl -v -X POST http://localhost:9000/api/products --data
'{"sku":"abc", "title":"Macbook Pro Retina"}' --header "Content-type:
application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 43
>
* upload completely sent off: 43 out of 43 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 42
<
* Connection #0 to host localhost left intact
{"sku":"abc","title":"Macbook Pro Retina"}%

```



# How it works...

In this recipe, we implemented a RESTful POST request using Play 2.0. The first step was to create our controller and model classes. For the model class, we declared two basic fields for a product. We annotated them as required fields. This allows Play to validate these two fields when a product is bound to a request body.

```
// Java
@Constraints.Required
private String sku;
```

As for a Scala equivalent of enforcing required request parameters, we declare optional parameters using the `scala.Option` class. In this recipe though, to keep the Java and Scala recipes consistent, it will be unnecessary to use `scala.Option` and we will enforce required fields in our case class like so:

```
case class Product(sku: String, title: String)
```

We then created an action method that will handle the product POST request in the controller class. We ensure that there aren't any validation errors encountered by the `play.data.Form` object during data binding; however, if it does encounter an issue, it will return an HTTP Status 400 wrapped by the `badRequest()` helper:

```
// Java
if (form.hasErrors()) {
 return badRequest(form.errorsAsJson());
}

// Scala
errors => BadRequest(Json.obj("message" ->
JsError.toJson(errors))),
```

If no errors are encountered, we proceed to persisting our new product and returning an HTTP Status 201 wrapped by the `created()` helper:

```
// Java
return created(toJson(product));

// Scala
Created(Json.toJson(p))
```

We then declared our new POST route in the `conf/routes` file. Finally, we used the command-line tool, `curl`, to simulate the HTTP POST request to test our route. To verify that our endpoint does execute the POST form field validations, omit the `title` parameter from the previous `curl` command and you will see the appropriate error message:

```
$ curl -v -X POST http://localhost:9000/api/products --data
'{"sku":"abc"}' --header "Content-type: application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /api/products HTTP/1.1
> User-Agent: curl/7.37.1
```

```
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 44
>
* upload completely sent off: 44 out of 44 bytes
< HTTP/1.1 400 Bad Request
< Content-Type: application/json; charset=utf-8
< Content-Length: 36
<
* Connection #0 to host localhost left intact
{"title":["This field is required"]}%
```



# Creating a GET API endpoint

For this recipe, we will create the RESTful GET endpoint, which will return a collection of JSON objects.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify the products controller in `foo_java/app/controllers/Products.java` by adding the following action method:

```
public static Result index() {
 return ok(toJson(products));
}
```

3. Add a new route entry for the newly added action in `foo_java/conf/routes`:

```
GET /api/products controllers.Products.index
```

4. Request the new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 50
<
* Connection #0 to host localhost left intact
{"abc":{"sku":"abc","title":"Macbook Pro Retina"},"def":
{"sku":"def","title":"iPad Air"}}%
```

For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify the product controller in `foo_scala/app/controllers/Products.scala` by adding the following action method:

```
def index = Action {
 Ok(toJson(products))
}
```

3. Add a new route entry for the newly added action in `foo_scala/conf/routes`:

```
GET /api/products controllers.Products.index
```

4. Request our new route and examine the response headers to confirm our modifications to the HTTP response header:

```
$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 50
<
* Connection #0 to host localhost left intact
{"abc":{"sku":"abc","title":"Macbook Pro Retina"},"def": {"sku":"def","title":"iPad Air"}}%
```

# How it works...

In this recipe, we implemented an API endpoint that returns a listing of product records. We were able to implement this by declaring a new action method that retrieves records from our data store, converts objects to JSON, and returns a JSON collection:

```
// Java
return ok(toJson(products));

// Scala
Ok(toJson(products))
```

We then declared a new route entry for the `GET` endpoint and used `curl` to verify the endpoints' functionality.

The endpoint will return an empty JSON array in the case of an empty data store:

```
Empty product list

$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 2
<
* Connection #0 to host localhost left intact
[]%
```



# Creating a PUT API endpoint

In this recipe, we will implement a RESTful `PUT` API endpoint using Play 2.0 to update an existing record in our data store.

# How to do it...

For Java, we need to perform the following steps:

1. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify foo\_java/app/controllers/Products.java by adding the following action:

```
@BodyParser.Of(BodyParser.Json.class)
public static Result edit(String id) {
 try {
 Product product = products.get(id);

 if (product != null) {
 Form<Product> form =
Form.form(Product.class).bindFromRequest();
 if (form.hasErrors()) {
 return badRequest(form.errorsAsJson());
 } else {
 Product productForm = form.get();
 product.setTitle(productForm.getTitle());
 products.put(product.getSKU(), product);

 return ok(toJSON(product));
 }
 } else {
 return notFound();
 }
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
}
```

3. Add a new route for the newly added action in foo\_java/conf/routes:

```
PUT /api/products/:id controllers.Products.edit(id: String)
```

4. Using curl, we will update an existing product in our data store:

```
$ curl -v -X PUT http://localhost:9000/api/products/def --data
'{"sku":"def", "title":"iPad 3 Air"}' --header "Content-type:
application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> PUT /api/products/def HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 35
>
* upload completely sent off: 35 out of 35 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
```

```

< Content-Length: 34
<
* Connection #0 to host localhost left intact
{"sku":"def","title":"iPad 3 Air"}%

```

For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_scala/app/controllers/Products.scala` by adding the following action:

```

def edit(id: String) = Action(BodyParsers.parse.json) { implicit
request =>
 val post = request.body.validate[Product]
 post.fold(
 errors => BadRequest(Json.obj("message" ->
JsError.toJson(errors))),
 p => {
 products.find(_.sku equals id) match {
 case Some(product) => {
 try {
 products -= product
 products += p

 Ok(Json.toJson(p))
 } catch {
 case e: Exception => InternalServerError(e.getMessage)
 }
 }
 case None => NotFound
 }
 }
)
}

```

3. Add a new route for the newly added action in `foo_scala/conf/routes`:

```
PUT /api/products/:id controllers.Products.edit(id: String)
```

4. Using curl, we will update an existing product in our data store:

```

$ curl -v -X PUT http://localhost:9000/api/products/def --data
'{"sku":"def", "title":"iPad 3 Air"}' --header "Content-type:
application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> PUT /api/products/def HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 35

```

```
>
* upload completely sent off: 35 out of 35 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 34
<
* Connection #0 to host localhost left intact
{"sku":"def","title":"iPad 3 Air"}%
```

# How it works...

In this recipe, we created a new URL route and action that will update an existing record in our data store. We added a new action to the products controller class and declared a new route for it in conf/routes. In our edit action, we declared that the action is to expect a request body in JSON format:

```
// Java
@BodyParser.Of(BodyParser.Json.class)

// Scala
def edit(id: String) = Action(BodyParsers.parse.json) { implicit
request =>
```

We check whether the ID value that is passed is valid by doing a lookup in our data store. We send an HTTP status 404 for invalid IDs:

```
// Java
return notFound();

// Scala
case None => NotFound
```

We also check for any form validation errors and will return the appropriate status code in the event of errors:

```
// Java
if (form.hasErrors()) {
 return badRequest(form.errorsAsJson());
}

// Scala
errors => BadRequest(Json.obj("message" ->
JsError.toJson(errors))),
```

Finally, we used curl to test the new product PUT action. We can further validate the PUT endpoint by testing how it handles invalid IDs and invalid request bodies:

# Passing an invalid Product ID:

```
$ curl -v -X PUT http://localhost:9000/api/products/XXXXXX
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> PUT /api/products/XXXXXX HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Length: 0
<
* Connection #0 to host localhost left intact

PUT requests with form validation error
```

```
$ curl -v -X PUT http://localhost:9000/api/products/def --data '{}' --
header "Content-type: application/json"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> PUT /api/products/def HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 2
>
* upload completely sent off: 2 out of 2 bytes
< HTTP/1.1 400 Bad Request
< Content-Type: application/json; charset=utf-8
< Content-Length: 69
<
* Connection #0 to host localhost left intact
{"title":["This field is required"],"sku":["This field is required"]}%
```



# Creating a DELETE API endpoint

In this recipe, we will implement a RESTful `DELETE` API endpoint to remove a record from our data store.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_java/app/controllers/Products.java` by adding the following action:

```
public static Result delete(String id) {
 try {
 Product product = products.get(id);

 if (product != null) {
 products.remove(product);

 return noContent();
 } else {
 return notFound();
 }
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
}
```

3. Add a new route for the newly added action in `foo_java/conf/routes`:

```
DELETE /api/products/:id controllers.Products.delete(id: String)
```

4. Using curl, remove an existing record, as follows:

```
$ curl -v -X DELETE http://localhost:9000/api/products/def
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> DELETE /api/products/def HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 204 No Content
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Modify `foo_scala/app/controllers/Products.scala` by adding the following action:

```
def delete(id: String) = BasicAuthAction {
```

```
products.find(_.sku equals id) match {
 case Some(product) => {
 try {
 products -= product
 NoContent
 } catch {
 case e: Exception => InternalServerError(e.getMessage)
 }
 }
 case None => NotFound
}
}
```

3. Add a new route for the newly added action in foo\_scala/conf/routes:

```
DELETE /api/products/:id controllers.Products.delete(id: String)
```

4. Using curl, remove an existing record as shown here:

```
$ curl -v -X DELETE http://localhost:9000/api/products/def
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> DELETE /api/products/def HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 204 No Content
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

# How it works...

In this recipe, we created a new URL route and action that is used to remove an existing product record. We declared the delete action to look up the record by the ID parameter passed in. We ensure that the appropriate HTTP status code is returned in the event of an invalid ID, in this case, HTTP status code 404:

```
// Java
return notFound();

// Scala
case None => NotFound
```

We then ensure that the appropriate HTTP status code for successful record removal is returned, in this case, HTTP status code 204:

```
// Java
return noContent();

// Scala
NoContent
```

We can also test the `DELETE` endpoint and verify that it handles invalid IDs correctly:

```
$ curl -v -X DELETE http://localhost:9000/api/products/XXXXXX
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> DELETE /api/products/asd HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 404 Not Found
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```



# Securing API endpoints with HTTP basic authentication

In this recipe, we will explore how to secure API endpoints using the HTTP basic authentication scheme with Play 2.0. We will use the Apache Commons Codec library for Base64 encoding and decoding for this recipe. This dependency is implicitly imported by Play and we will not need to explicitly declare it to our library dependencies in `build.sbt`.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create a new `play.mvc.Security.Authenticator` implementation class in `foo_java/app/controllers/BasicAuthenticator.java` with the following content:

```
package controllers;

import org.apache.commons.codec.binary.Base64;
import play.mvc.Http;
import play.mvc.Result;
import play.mvc.Security;

public class BasicAuthenticator extends Security.Authenticator {
 private static final String AUTHORIZATION = "authorization";
 private static final String WWW_AUTHENTICATE = "WWW-
Authenticate";
 private static final String REALM = "Basic realm=\"API
Realm\"";

 @Override
 public String getUsername(Http.Context ctx) {
 try {
 String authHeader =
ctx.request().getHeader(AUTHORIZATION);

 if (authHeader != null) {
 ctx.response().setHeader(WWW_AUTHENTICATE, REALM);
 String auth = authHeader.substring(6);
 byte[] decodedAuth = Base64.decodeBase64(auth);
 String[] credentials = new String(decodedAuth,
"UTF-8").split(":");

 if (credentials != null && credentials.length == 2)
{
 String username = credentials[0];
 String password = credentials[1];
 if (isAuthenticated(username, password)) {
 return username;
 } else {
 return null;
 }
 }
 }
 return null;
 } catch (Exception e) {
 return null;
 }
}
private boolean isAuthenticated(String username, String
```

```

password) {
 return username != null && username.equals("ned") &&
 password != null && password.equals("flanders");
}

@Override
public Result onUnauthorized(HttpContext context) {
 return unauthorized();
}
}

```

3. Modify `foo_java/app/controllers/Products.java` by adding the following annotation to the API actions:

```

@Security.Authenticated(BasicAuthenticator.class)
public static Result create()

@Security.Authenticated(BasicAuthenticator.class)
public static Result index()

@Security.Authenticated(BasicAuthenticator.class)
public static Result edit(String id)

@Security.Authenticated(BasicAuthenticator.class)
public static Result delete(String id)

```

4. Using curl, send a request to the existing RESTful GET endpoint as we did earlier; you will now see an unauthorized response:

```

$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Content-Length: 0
<
* Connection #0 to host localhost left intact

```

5. Using curl again, send another request to the existing RESTful GET endpoint, this time with the user credentials, ned (username) and flanders (password):

```

$ curl -v -u "ned:flanders" http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
* Server auth using Basic with user 'ned'
> GET /api/products HTTP/1.1
> Authorization: Basic bmVkOmZsYW5kZXJz
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*

```

```

>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< WWW-Authenticate: Basic realm="API Realm"
< Content-Length: 2
<
* Connection #0 to host localhost left intact
{}%

```

For Scala, we need to perform the following steps:

1. Run `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create a new `ActionBuilder` class in

`foo_scala/app/controllers/BasicAuthAction.scala` with the following content:

```

package controllers

import controllers.Products._
import org.apache.commons.codec.binary.Base64
import play.api.mvc._
import scala.concurrent.Future

object BasicAuthAction extends ActionBuilder[Request] {
 def invokeBlock[A](request: Request[A], block: (Request[A]) =>
Future[Result]) = {
 try {
 request.headers.get("authorization") match {
 case Some(headers) => {
 val auth = headers.substring(6)
 val decodedAuth = Base64.decodeBase64(auth)
 val credentials = new String(decodedAuth, "UTF-
8").split(":")

 if (credentials != null && credentials.length == 2 &&
 isAuthenticated(credentials(0), credentials(1))) {
 block(request)
 } else {
 unauthorized
 }
 }
 case None => unauthorized
 }
 } catch {
 case e: Exception =>
Future.successful(InternalServerError(e.getMessage))
 }
}

def unauthorized =
Future.successful(Unauthorized.withHeaders("WWW-Authenticate" -> "Basic
realm=\\"API Realm\\\""))

def isAuthenticated(username: String, password: String) =

```

```
username != null && username.equals("ned") && password != null &&
password.equals("flanders")
}
```

3. Modify `foo_scala/app/controllers/Products.scala` by adding the newly created `ActionBuilder` class with the API actions instead:

```
def index = BasicAuthAction

def create = BasicAuthAction(BodyParsers.parse.json)

def edit(id: String) = BasicAuthAction(BodyParsers.parse.json)

def delete(id: String) = BasicAuthAction
```

4. Using `curl`, send a request to the existing RESTful GET endpoint as we did earlier; you will now see an unauthorized response:

```
$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

5. Using `curl` again, send another request to the existing RESTful GET endpoint, this time with the user credentials, ned (username) and flanders (password):

```
$ curl -v -u "ned:flanders" http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 9000 (#0)
* Server auth using Basic with user 'ned'
> GET /api/products HTTP/1.1
> Authorization: Basic bmVkOmZsYW5kZXJz
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8

< Content-Length: 2
<
* Connection #0 to host localhost left intact
{ }%
```

# How it works...

In this recipe, we secured the RESTful API endpoints using the HTTP basic authentication scheme with Play 2.0. We created the respective security implementation class for both Java and Scala. For each security implementation class, `BasicAuthenticator.java` and `BasicAuthAction.scala`, we retrieved the authorization header and decoded the value string to decrypt the user credentials that we passed in:

```
// Java
String authHeader = ctx.request().getHeader(AUTHORIZATION);
if (authHeader != null) {
 ctx.response().setHeader(WWW_AUTHENTICATE, REALM);
 String auth = authHeader.substring(6);
 byte[] decodedAuth = Base64.decodeBase64(auth);
 String[] credentials = new String(decodedAuth, "UTF-8").split(":");
}

// Scala
request.headers.get("authorization") match {
 case Some(headers) => {
 val auth = headers.substring(6)
 val decodedAuth = Base64.decodeBase64(auth)
 val credentials = new String(decodedAuth, "UTF-8").split(":")
 }
}
```

Once we got the username and password, we invoked the `isAuthenticated` function to check the validity of the user credentials:

```
// Java
if (credentials != null && credentials.length == 2) {
 String username = credentials[0];
 String password = credentials[1];
 if (isAuthenticated(username, password)) {
 return username;
 } else {
 return null;
 }
}

// Scala
if (credentials != null && credentials.length == 2 &&
 isAuthenticated(credentials(0), credentials(1))) {
 block(request)
} else {
 unauthorized
}
```

We then utilized the security implementation classes by annotating the Java API actions and declaring it as the API action class:

```
// Java
@Security.Authenticated(BasicAuthenticator.class)
public static Result index() {
```

```
 return ok(toJson(products));
 }

// Scala
def index = BasicAuthAction {
 Ok(toJson(products))
}
```

Using curl, we can also check whether our secure API actions handle unauthenticated requests:

```
$ curl -v http://localhost:9000/api/products
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /api/products HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Basic realm="API Realm"
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```



# Consuming external web APIs

In this recipe, we will explore the Play WS API to consume external web services from a Play 2 web application. As web application requirements evolve, we become more dependent on external data services for data such as foreign exchange rates, real-time weather data, and so on. The Play WS library provides us with APIs to be able to interface with external web services.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare `playws` as a project dependency in `build.sbt`:

```
libraryDependencies ++= Seq(
 javaWs
)
```

3. Create a new controller in `foo_java/app/controllers/WebClient.java` and add the following content:

```
package controllers;

import com.fasterxml.jackson.databind.JsonNode;
import play.libs.F;
import play.libs.F.Promise;
import play.libs.ws.WS;
import play.mvc.Controller;
import play.mvc.Result;

public class WebClient extends Controller {
 public static Promise<Result> getTodos() {
 Promise<play.libs.ws.WSResponse> todos =
 WS.url("http://jsonplaceholder.typicode.com/todos").get();
 return todos.map(
 new F.Function<play.libs.ws.WSResponse, Result>() {
 public Result apply(play.libs.ws.WSResponse res) {
 JsonNode json = res.asJson();
 return ok("Todo Title: " +
 json.findValuesAsText("title"));
 }
 });
 }
}
```

4. Add a new route entry for the newly added action in `foo_java/conf/routes`:

```
GET /client/get.todos controllers.WebClient.getTodos
```

5. Using `curl`, we will be able to test how our new action is able to consume an external web API:

```
$ curl -v http://localhost:9000/client/get.todos
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /client/get.todos HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
```

```

> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 8699
<
Todo Title: [delectus aut autem, quis ut nam facilis et officia
qui, fugiat veniam minus, et porro tempora, laboriosam mollitia et enim
quasi adipisci quia provident illum, qui ullam ratione quibusdam
voluptatem quia omnis, illo expedita

```

For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Declare `playws` as a project dependency in `build.sbt`:

```
libraryDependencies ++= Seq(
 ws
)
```

3. Create a new controller in `foo_scala/app/controllers/WebClient.scala` and add the following content:

```

package controllers

import play.api.libs.concurrent.Execution.Implicits.defaultContext
import play.api.Play.current
import play.api.libs.ws.WS
import play.api.mvc.{Action, Controller}

object WebClient extends Controller {
 def getTodos = Action.async {
 WS.url("http://jsonplaceholder.typicode.com/todos").get().map {
 res =>
 Ok("Todo Title: " + (res.json \\" "title").map(_.as[String]))
 }
 }
}

```

4. Add a new route entry for the newly added action in `foo_scala/conf/routes`:

```
GET /client/get.todos controllers.WebClient.getTodos
```

5. Using `curl`, we will be able to test how our new action is able to consume an external web API:

```
$ curl -v http://localhost:9000/client/get.todos
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /client/get.todos HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
```

```
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 8699
<
Todo Title: [delectus aut autem, quis ut nam facilis et officia
qui, fugiat veniam minus, et porro tempora, laboriosam mollitia et enim
quasi adipisci quia provident illum, qui ullam ratione quibusdam
voluptatem quia omnis, illo expedita
```

# How it works...

In this recipe, we utilized the Play 2.0 plugin, WS, to consume an external web API. We created a new route and `AsynchronousAction` method. In the action, we passed the external API's URL into the WS api and specified that it will be a GET operation:

```
// Java
Promise<play.libs.ws.WSResponse> todos =
WS.url("http://jsonplaceholder.typicode.com/todos").get();
// Scala
WS.url("http://jsonplaceholder.typicode.com/todos").get()
```

We then parsed the JSON response and piped it into the resulting response of the newly created route, `/client/get_todos`:

```
// Java
return todos.map(
 new F.Function<play.libs.ws.WSResponse, Result>() {
 public Result apply(play.libs.ws.WSResponse res) {
 JsonNode json = res.asJson();
 return ok("Todo Title: " + json.findValuesAsText("title"));
 }
 }
);

// Scala
Ok("Todo Title: " + (res.json \\ "title").map(_.as[String]))
```



# Using the Twitter API and OAuth

In this recipe, we will explore how we can use the built-in support of Play 2.0 for OAuth to retrieve tweets from the Twitter API.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Specify your Twitter API information in `foo_java/conf/application.conf`:

```
tw.consumerKey="YOUR TWITTER DEV CONSUMER KEY HERE"
tw.consumerSecret="YOUR TWITTER DEV CONSUMER SECRET HERE"
tw.accessToken="YOUR TWITTER DEV ACCESS TOKEN HERE"
tw.accessTokenSecret="YOUR TWITTER DEV ACCESS TOKEN SECRET HERE"
```

3. Modify the `WebClient` controller in `foo_java/app/controllers/WebClient.java` with the following action:

```
// Add additional imports at the top section of the class file
import play.Play;
import play.libs.oauth.OAuth;
import play.libs.oauth.OAuth.OAuthCalculator;
import play.libs.ws.WSResponse;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

// Add the Action method
public static Promise<Result> getTweets(String hashtag) {
 final String url =
"https://api.twitter.com/1.1/search/tweets.json?q=%40" + hashtag;
 final OAuth.ConsumerKey consumerInfo = new OAuth.ConsumerKey(
 Play.application().configuration().getString("tw.consumerKey"),
 Play.application().configuration().getString("tw.consumerSecret"))
);
 final OAuth.RequestToken tokens = new OAuth.RequestToken(
 Play.application().configuration().getString("tw.accessToken"),
 Play.application().configuration().getString("tw.accessTokenSecret"))
);

 Promise<play.libs.ws.WSResponse> twRequest =
WS.url(url).sign(new OAuthCalculator(consumerInfo, tokens)).get();
 return twRequest.map(
 new F.Function<WSResponse, Result>(){
 @Override
 public Result apply(WSResponse res) throws Throwable {
 Map<String, String> map = new HashMap<String,
String>();
 JsonNode root = res.asJson();

 for (JsonNode json : root.get("statuses")) {
 map.put(

```

```

 json.findValue("user").findValue("screen_name").asText(),
 json.findValue("text").asText()
);
 }

 return ok(views.html.tweets.render(map));
 }
);
}
}

```

4. Add the new route for the `getTweets(hashtag: String)` action to `foo_java/conf/routes`:

```

GET /client/get_tweets/:hashtag
controllers.WebClient.getTweets(hashtag)

```

5. Add a new view template in `foo_java/app/views/tweets.scala.html` with the following content:

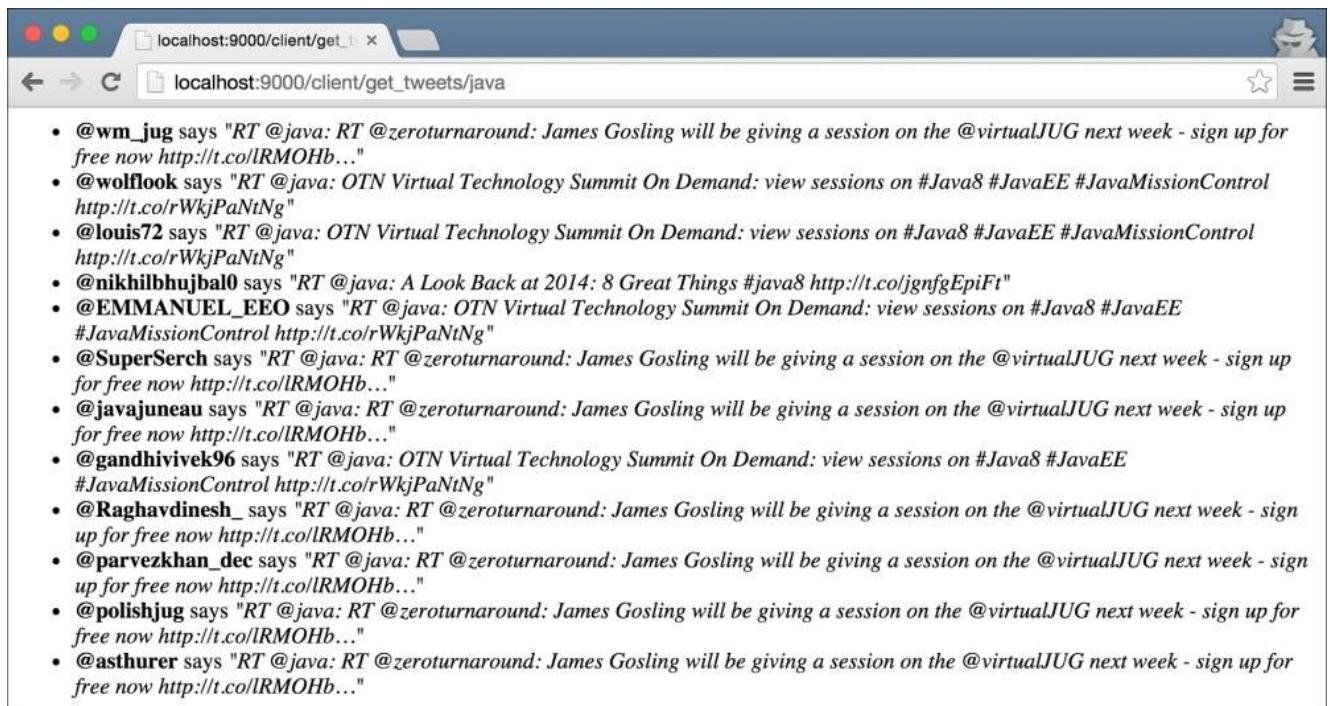
```

@(tweets: Map[String, String])

@tweets.map { tw =>
 @@@tw._1 says <i>"@tw._2"</i>
}


```

6. Using a web browser, access the `/client/get_tweets/:hashtag` route to view tweets retrieved from the Twitter API:



For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Specify your Twitter API information in `foo_scala/conf/application.conf`:

```
tw.consumerKey="YOUR TWITTER DEV CONSUMER KEY HERE"
tw.consumerSecret="YOUR TWITTER DEV CONSUMER SECRET HERE"
tw.accessToken="YOUR TWITTER DEV ACCESS TOKEN HERE"
tw.accessTokenSecret="YOUR TWITTER DEV ACCESS TOKEN SECRET HERE"
```

3. Modify the `WebClient` controller in `foo_scala/app/controllers/WebClient.scala` with the following action:

```
def getTweets(hashtag: String) = Action.async {
 import play.api.Play

 val consumerInfo = ConsumerKey(
 Play.application.configuration.getString("tw.consumerKey").get,
 Play.application.configuration.getString("tw.consumerSecret").get
)
 val tokens = RequestToken(
 Play.application.configuration.getString("tw.accessToken").get,
 Play.application.configuration.getString("tw.accessTokenSecret").get
)
 val url = "https://api.twitter.com/1.1/search/tweets.json?q=%40"
 + hashtag

 WS.url(url).sign(0AuthCalculator(consumerInfo, tokens)).get().map
 { res =>
 val tweets = ListBuffer[(String, String)]()
 (res.json \ "statuses").as[List[JsObject]].map { tweet =>
 tweets += (
 (tweet \ "user" \ "screen_name").as[String],
 (tweet \ "text").as[String]
)
 }
 Ok(views.html.tweets(tweets.toList))
 }
}
```

4. Add the new routes for the `getTweets(hashtag: String)` action in `foo_scala/conf/routes`:

```
GET /client/get_tweets/:hashtag
controllers.WebClient.getTweets(hashtag)
```

5. Add a new view template in `foo_scala/app/views/tweets.scala.html` with the following content:

```
@(tweets: List[(String, String)])

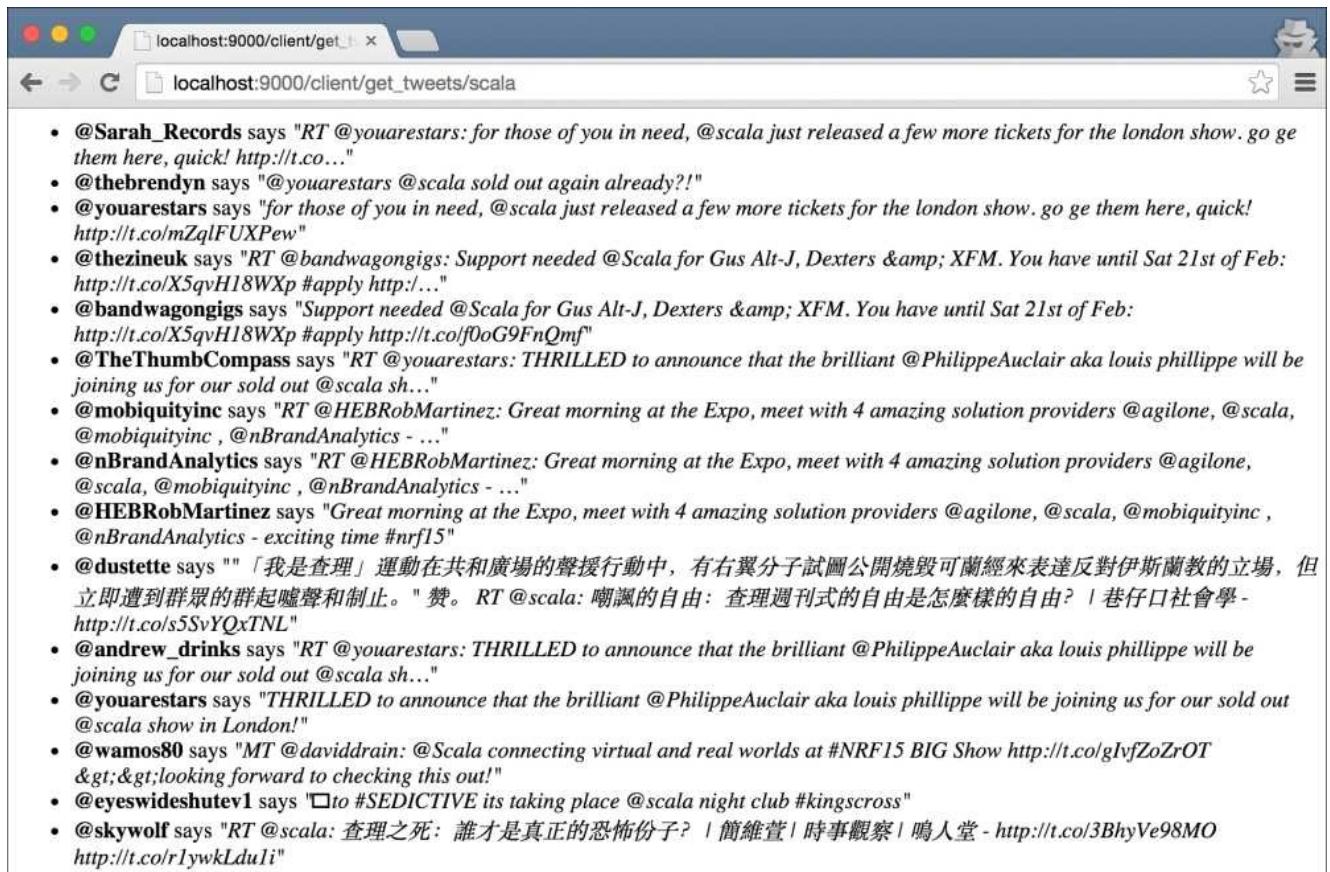

```

```

@tweets.map { tw =>
 @@@tw._1 says <i>"@tw._2"</i>
}


```

6. Using a web browser, access the /client/get\_tweets/:hashtag route to view tweets retrieved from the Twitter API, as shown in the following screenshot:



# How it works...

In this recipe, we created a new URL route and action to retrieve and display Tweets by the hashtag specified in the request route, /client/get\_tweets/:hashtag. We implemented the action method by retrieving the required Twitter API consumer and access token keys from conf/application.conf (remember to register for a Twitter Dev account at <http://dev.twitter.com> and generate your consumer and access tokens for this recipe):

```
// Java
final OAuth.ConsumerKey consumerInfo = new OAuth.ConsumerKey(
 Play.application().configuration().getString("tw.consumerKey"),
 Play.application().configuration().getString("tw.consumerSecret"))
);
final OAuth.RequestToken tokens = new OAuth.RequestToken(
 Play.application().configuration().getString("tw.accessToken"),
 Play.application().configuration()
.getString("tw.accessTokenSecret"))
);

// Scala
val consumerInfo = ConsumerKey(
 Play.application.configuration.getString("tw.consumerKey").get,
 Play.application.configuration.getString("tw.consumerSecret").get
)
val tokens = RequestToken(
 Play.application.configuration.getString("tw.accessToken").get,
 Play.application.configuration.getString("tw.accessTokenSecret").get
)
```

We then passed these credentials to the Play class OAuthCalculator as we accessed the Twitter search API endpoint:

```
// Java
Promise<play.libs.ws.WSResponse> twRequest =
WS.url(url).sign(new OAuthCalculator(consumerInfo, tokens)).get();

// Scala
WS.url(url).sign(OAuthCalculator(consumerInfo, tokens)).get()
```

Once the Twitter API response returns, we parse the response JSON and push it to a intermediate collection object, which we then passed on to our view template:

```
// Java
Map<String, String> map = new HashMap<String, String>();
JsonNode root = res.asJson();

for (JsonNode json : root.get("statuses")) {
 map.put(
 json.findValue("user")
 .findValue("screen_name").asText(),
 json.findValue("text").asText()
);
}
```

```
return ok(views.html.tweets.render(map));

// Scala
val tweets = ListBuffer[(String, String)]()
(res.json \ "statuses").as[List[JsObject]].map { tweet =>
 tweets += ((
 (tweet \ "user" \ "screen_name").as[String],
 (tweet \ "text").as[String]
))
}

Ok(views.html.tweets(tweets.toList))
```



# Chapter 5. Creating Plugins and Modules

In this chapter, we will cover the following recipes:

- Creating and using your own plugin
- Building a flexible registration module
- Using the same model for different applications
- Managing module dependencies
- Adding private module repositories using Amazon S3

# Introduction

In this chapter, we will look at how we can break down our Play 2.0 web applications into modular, reusable components. We will look at how we can create plugins and modules as a Play 2.0 subproject and as an independent module published in an internal module repository.

A Play 2.0 plugin can be useful when creating independent services and initializing shared resources such as database connections and Akka actor references. Other examples of useful Play plugins include the **play.i18n.MessagesPlugin**, which manages internationalization of text, and the **play.api.db.DBPlugin**, which abstracts how a Play web application connects and interfaces with databases.

A Play 2.0 module is useful to create smaller, logical subcomponents of a larger application; this promotes better code maintenance and isolation of tests.



# Creating and using your own plugin

In this recipe, we will explore how to use the Play 2.0 plugin that will monitor the filesystem for a specified file. We will initialize our plugin as part of the Play web application lifecycle, and the main plugin logic will be triggered on application startup.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create the modules directory inside `foo_java`:

```
mkdir modules
```

3. Generate the project directory for our first plugin inside `foo_java/modules`:

```
activator new filemon play-java
```

4. Remove the contents of the `modules/filemon/conf/application.conf` file, as these settings will conflict with the main configuration file that we have defined in the project root:

```
echo "" > modules/filemon/conf/application.conf
```

5. Remove the contents of the `modules/filemon/conf/routes` file and rename it to `filemon.routes`:

```
echo "" > modules/filemon/conf/routes && mv
modules/filemon/conf/routes modules/filemon/conf/filemon.routes
```

6. Remove the views directory from `modules/filemon/app`:

```
rm -rf modules/filemon/app/views
```

7. Remove the file `modules/filemon/app/controller/Application.java` using the following command:

```
rm modules/filemon/app/controllers/Application.java
```

8. Create a new package inside `modules/filemon/app/`:

```
mkdir modules/filemon/app/filemon
```

9. Create the `FileMonitor` plugin inside `modules/filemon/app/FileMonitor.java` with the following content:

```
package filemon;

import java.io.*;
import java.util.concurrent.TimeUnit;
import akka.actor.ActorSystem;
import play.Plugin;
import play.Application;
import play.libs.Akka;
import scala.concurrent.duration.Duration;

public class FileMonitor extends Plugin {
 private Application app;
 private ActorSystem actorSystem =
```

```

ActorSystem.create("filemon");
 private File file = new File("/var/tmp/foo");

 public FileMonitor(Application app) {
 this.app = app;
 }
 @Override
 public void onStart() {
 actorSystem.scheduler().schedule(
 Duration.create(0, TimeUnit.SECONDS),
 Duration.create(1, TimeUnit.SECONDS),
 new Runnable() {
 public void run() {
 if (file.exists()) {
 System.out.println(file.toString() + " exists..");
 } else {
 System.out.println(file.toString() + " does not exist..");
 }
 }
 },
 Akka.system().dispatcher()
);
 }
 @Override
 public void onStop() {
 actorSystem.shutdown();
 }
 @Override
 public boolean enabled() {
 return true;
 }
}

```

10. Enable the plugin from the `foo_java` application by creating the plugin's configuration file, `foo_java/conf/play.plugins`, and declaring our plugin there:

```
echo "1001:filemon.FileMonitor" > conf/play.plugins
```

11. Add the dependency between the root project (`foo_java`) and the module (`filemon`) in `build.sbt`, and add the `aggregate()` setting to ensure that activator tasks called from the project root, `foo_java`, are also invoked on the child module, `filemon`:

```

lazy val root = (project in file("."))
 .enablePlugins(PlayJava)
 .aggregate(filemon)
 .dependsOn(filemon)

lazy val filemon = (project in file("modules/filemon"))
 .enablePlugins(PlayJava)

```

12. Start the `foo_java` application:

```
activator clean "~run"
```

13. Request the default route and initialize our app:

```
$ curl -v http://localhost:9000
```

14. Confirm that the file monitor is running by looking at the console log of the foo\_java application:

```
(Server started, use Ctrl+D to stop and go back to the console...)
```

```
[info] Compiling 5 Scala sources and 1 Java source to...
[success] Compiled in 4s
Starting file mon
/var/tmp/foo not found..
[info] play - Application started (Dev)
/var/tmp/foo not found..
/var/tmp/foo not found..
/var/tmp/foo not found..
/var/tmp/foo not found..
```

For Scala, we need to perform the following steps:

1. Run the foo\_scala application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create the modules directory inside foo\_scala:

```
mkdir modules
```

3. Generate the project directory for our first plugin inside foo\_scala/modules:

```
activator new filemon play-scala
```

4. Remove the contents of the modules/filemon/conf/application.conf file:

```
echo "" > modules/filemon/conf/application.conf
```

5. Remove the contents of the modules/filemon/conf/routes file and rename it to filemon.routes:

```
echo "" > modules/filemon/conf/routes && mv
modules/filemon/conf/routes modules/filemon/conf/filemon.routes
```

6. Remove the views directory from modules/filemon/app:

```
rm -rf modules/filemon/app/views
```

7. Remove the file modules/filemon/app/controller/Application.scala using the following command:

```
rm modules/filemon/app/controllers/Application.scala
```

8. Create a new package inside modules/filemon/app/:

```
mkdir modules/filemon/app/filemon
```

9. Create the FileMonitor plugin inside modules/filemon/app/FileMonitor.scala

with the following contents:

```
package filemon

import java.io.File
import scala.concurrent.duration._
import akka.actor.ActorSystem
import play.api.{Plugin, Application}
import play.api.libs.concurrent.Execution.Implicits._

class FileMonitor(app: Application) extends Plugin {
 val system = ActorSystem("filemon")
 val file = new File("/var/tmp/foo")

 override def onStart() = {
 println("Starting file mon")
 system.scheduler.schedule(0 second, 1 second) {
 if (file.exists()) {
 println("%s exists..".format(file))
 } else {
 println("%s not found..".format(file))
 }
 }
 }

 override def onStop() = {
 println("Stopping file mon")
 system.shutdown()
 }

 override def enabled = true
}
```

10. Enable the plugin from the `foo_scala` application by creating the plugin's configuration file, `foo_scala/conf/play.plugins`, and declaring our plugin there:

```
echo "1001:filemon.FileMonitor" > conf/play.plugins
```

11. Add the dependency between the root project (`foo_scala`) and the module (`filemon`) in `build.sbt`, and add the `aggregate()` setting to ensure that activator tasks called from the project root, `foo_java`, are also invoked on the child module, `filemon`:

```
lazy val root = (project in file("."))
 .enablePlugins(PlayScala)
 .aggregate(filemon)
 .dependsOn(filemon)

lazy val filemon = (project in file("modules/filemon"))
 .enablePlugins(PlayScala)
```

12. Start the `foo_scala` application:

```
activator clean "~run"
```

13. Request our default route and initialize our app:

```
$ curl -v http://localhost:9000
```

14. Confirm that the file monitor is running by looking at the console log of the foo\_scala application:

```
(Server started, use Ctrl+D to stop and go back to the console...)
```

```
[info] Compiling 5 Scala sources and 1 Java source to...
[success] Compiled in 4s
Starting file mon
/var/tmp/foo not found..
[info] play - Application started (Dev)
/var/tmp/foo not found..
/var/tmp/foo not found..
/var/tmp/foo not found..
/var/tmp/foo not found..
```

# How it works...

In this recipe, we set up our first Play 2.0 plugin. The plugin simply checks for a file or directory in the local filesystem and logs in the console whether the file is found or not. We set up our plugin by creating the plugin project inside the newly created directory in the project root modules in `foo_java/modules` and `foo_scala/modules`:

```
$ ls
LICENSE conf
README logs
activator modules
activator-launch-1.2.10.jar project
activator.bat public
app target
build.sbt test

$ ls modules/filemon
LICENSE build.sbt
README conf
activator project
activator-launch-1.2.10.jar public
activator.bat target
app test
```

Once the plugin project is created, we need to remove some boilerplate files and configurations to ensure that the plugin does not conflict with the root projects, `foo_java` and `foo_scala`.

We then created the `FileMonitor` plugin in `modules/filemon/app/filemon/FileMonitor.scala`, extending the `play.api.Plugin` trait, which upon startup, creates a scheduled job which in turn checks for the existence of a local file every second:

```
override def onStart() = {
 println("Starting file mon")
 system.scheduler.schedule(0 second, 1 second) {
 if (file.exists()) {
 println("%s exists..".format(file))
 } else {
 println("%s not found..".format(file))
 }
 }
}
```

Once we had our plugin in place, we activated it by declaring it in the `conf/play.plugins` file in the root projects, `foo_java` and `foo_scala`, which follow the notation `<Priority Level>:<Plugin>`:

```
1001:filemon.FileMonitor
```

In our case, we used `1001` as the priority level to ensure that the Akka Play 2.0 plugin loads first. Refer to the official Play documentation for additional guidelines for declaring your plugins in the `play.plugins` configuration file:

<https://www.playframework.com/documentation/2.3.x/JavaPlugins>

<https://www.playframework.com/documentation/2.3.x/ScalaPlugins>

Finally, we ran the web application and confirmed our plugin to be running by watching the console log:

```
Starting file mon
/var/tmp/foo not found..
```

You can confirm the behavior of your plugin by creating or deleting the monitor file, in this example, /var/tmp/foo:

```
Create foo file then delete after 3 seconds
touch /var/tmp/foo && sleep 3 && rm /var/tmp/foo
```

You will see the output in the logs change accordingly:

```
/var/tmp/foo not found..
/var/tmp/foo not found..
/var/tmp/foo exists..
/var/tmp/foo exists..
/var/tmp/foo exists..
/var/tmp/foo not found..
/var/tmp/foo not found..
```



# Building a flexible registration module

In this recipe, we will create a new registration module that will manage user registration and authentication requests. Creating a module for this allows us to reuse a very common workflow in the modern web application.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Inside the modules directory, `foo_java/modules`, generate the registration module project using the activator:

```
activator new registration play-java
```

3. Add the dependency between the root project, `foo_java`, and the module, `registration`, in `foo_java/build.sbt`:

```
lazy val root = (project in file("."))
 .enablePlugins(PlayJava)
 .aggregate(filemon)
 .dependsOn(filemon)
 .aggregate(registration)
 .dependsOn(registration)

lazy val filemon = (project in file("modules/filemon"))
 .enablePlugins(PlayJava)

lazy val registration = (project in file("modules/registration"))
 .enablePlugins(PlayJava)
```

4. Remove all the unnecessary boilerplate files and configurations from the registration module:

```
rm -rf app/views app/controllers
rm conf/routes
mkdir app/registration
echo "" > conf/application.conf
```

5. Create the registration plugin in `module/registration/app/registration/RegistrationPlugin.java` with the following content:

```
package registration;

import play.Application;
import play.Plugin;

public class RegistrationPlugin extends Plugin {
 private Application app;
 private RegistrationService registrationService;

 public RegistrationPlugin(Application app) {
 this.app = app;
 }
 @Override
 public void onStart() {
 registrationService = new RegistrationServiceImpl();
 }
}
```

```

 registrationService.init();
 }

 @Override
 public void onStop() {
 registrationService.shutdown();
 }
 @Override
 public boolean enabled() {
 return true;
 }

 public RegistrationService getRegistrationService() {
 return registrationService;
 }
}

```

6. Next, create the RegistrationService interface and implementation class referred to by the Registration plugin:

```

// In
modules/registration/app/registration/RegistrationService.java

package registration;

public interface RegistrationService {
 void init();
 void shutdown();
 void create(User user);
 Boolean auth(String username, String password);
}

// In
modules/registration/app/registration/RegistrationServiceImpl.java
package registration;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.UUID;

public class RegistrationServiceImpl implements RegistrationService
{
 private Map<String, User> registrations;

 @Override
 public void create(User user) {
 final String id = UUID.randomUUID().toString();
 registrations.put(id, new User(id, user.getUsername(),
user.getPassword()));
 }

 @Override
 public Boolean auth(String username, String password) {
 for(Map.Entry<String, User> entry :
registrations.entrySet()) {
 if (entry.getValue().getUsername().equals(username) &&

```

```

 entry.getValue().getPassword().equals(password)) {
 return true;
 }
 }
 return false;
}

@Override
public void init() {
 registrations = new LinkedHashMap<String, User>();
}

@Override
public void shutdown() {
 registrations.clear();
}
}
}

```

## 7. Create the User model entity in

modules/registration/app/registration/User.java:

```

package registration;

public class User {
 private String id;
 private String username;
 private String password;

 public User() {}
 public User(String id, String username, String password) {
 this.id = id;
 this.username = username;
 this.password = password;
 }

 public String getId() {
 return id;
 }
 public void setId(String id) {
 this.id = id;
 }
 public String getUsername() {
 return username;
 }
 public void setUsername(String username) {
 this.username = username;
 }
 public String getPassword() {
 return password;
 }
 public void setPassword(String password) {
 this.password = password;
 }
}

```

## 8. Create the Registration controller and routes that will handle registration and login

requests in the project root, foo/java/app/controllers/Registrations.java:

```
package controllers;

import play.Play;
import play.data.Form;
import play.mvc.BodyParser;
import play.mvc.Controller;
import play.mvc.Result;
import registration.RegistrationPlugin;
import registration.RegistrationService;
import registration.User;
import static play.libs.Json.toJson;

public class Registrations extends Controller {
 private static RegistrationService registrationService =
Play.application().plugin(RegistrationPlugin.class).getRegistrationService();

 @BodyParser.Of(BodyParser.Json.class)
 public static Result register() {
 try {
 Form<User> form =
Form.form(User.class).bindFromRequest();
 User user = form.get();
 registrationService.create(user);
 return created(toJson(user));
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
 }

 @BodyParser.Of(BodyParser.Json.class)
 public static Result login() {
 try {
 Form<User> form =
Form.form(User.class).bindFromRequest();
 User user = form.get();
 if (registrationService.auth(user.getUsername(),
user.getPassword())) {
 return ok();
 } else {
 return forbidden();
 }
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
 }
}
```

9. Add the routes for the newly added Registration actions in the project root, foo\_java/conf/routes:

```
POST /register controllers.Registrations.register
POST /auth controllers.Registrations.login
```

- Finally, declare the registration plugin in the project root `foo_java/conf/play.plugins` file:

```
599:registration.RegistrationPlugin
```

- Using `curl`, submit a new registration and log in with the specified registration details; verify that our endpoint responds with an HTTP status 200 for successful operations by inspecting the response headers:

```
$ curl -v -X POST --header "Content-type: application/json"
http://localhost:9000/register -d '{"username":"ned@flanders.com",
"password":"password"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /register HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 54
>
* upload completely sent off: 54 out of 54 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 63
<
* Connection #0 to host localhost left intact
{"id":null,"username":"ned@flanders.com","password":"password"}%

$ curl -v -X POST --header "Content-type: application/json"
http://localhost:9000/auth -d '{"username":"ned@flanders.com",
"password":"password"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /auth HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 54
>
* upload completely sent off: 54 out of 54 bytes
< HTTP/1.1 200 OK
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

For Scala, we need to perform the following steps:

- Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Inside the modules directory (foo\_scala/modules), generate our registration module project using the activator:

```
activator new registration play-scala
```

3. Add the dependency between the root project, foo\_scala, and the module, registration, in build.sbt:

```
lazy val root = (project in file("."))
 .enablePlugins(PlayScala)
 .aggregate(filemon)
 .dependsOn(filemon)
 .aggregate(registration)
 .dependsOn(registration)

lazy val filemon = (project in file("modules/filemon"))
 .enablePlugins(PlayScala)

lazy val registration = (project in file("modules/registration"))
 .enablePlugins(PlayScala)
```

4. Remove all the unnecessary boilerplate files and configurations from the registration module:

```
rm -rf app/views app/controllers
rm conf/routes
mkdir app/registration
echo "" > conf/application.conf
```

5. Create the registration plugin in

module/registration/app/registration/RegistrationPlugin.scala with the following content:

```
package registration

import play.api.{Application, Plugin}

class RegistrationPlugin(app: Application) extends Plugin {
 val registrationService = new RegistrationService

 override def onStart() = {
 registrationService.init
 }

 override def onStop() = {
 registrationService.shutdown
 }

 override def enabled = true
}
```

6. Next, create the RegistrationService class referred to by the Registration plugin:

```
// In
modules/registration/app/registration/RegistrationService.scala
```

```

package registration

import java.util.UUID

class RegistrationService {
 type ID = String
 private val registrations = scala.collection.mutable.Map[ID, User]()
}

def init = {
 registrations.clear()
}

def create(user: User) = {
 val id: ID = UUID.randomUUID().toString
 registrations += (id -> user.copy(Some(id), user.username, user.password))
}

def auth(username: String, password: String) =
registrations.find(_.username.equals(username)) match {
 case Some(reg) => {
 if (reg.password.equals(password)) {
 Some(reg)
 } else {
 None
 }
 }
 case None => None
}

def shutdown = {
 registrations.clear()
}
}

```

7. Create the User model entity in

modules/registration/app/registration/User.scala:

```

package registration

case class User(id: Option[String], username: String, password: String)

```

8. Create the Registration controller and routes that will handle registration and login requests in the project root, foo\_scala/app/controllers/Registrations.scala:

```

package controllers

import play.api.Play.current
import play.api.Play
import play.api.libs.json.{JsError, Json}
import play.api.mvc.{BodyParsers, Action, Controller}
import registration.{RegistrationPlugin, User, RegistrationService}

object Registrations extends Controller {

```

```

implicit private val writes = Json.writes[User]
implicit private val reads = Json.reads[User]
private val registrationService: RegistrationService =
Play.application.plugin[RegistrationPlugin]
 .getOrElse(throw new IllegalStateException("RegistrationService is
required!"))
 .registrationService

def register = Action(BodyParsers.parse.json) { implicit request
=>
 val post = request.body.validate[User]

 post.fold(
 errors => BadRequest(JsonError.toJson(errors)),
 u => {
 registrationService.create(u)
 Created(Json.toJson(u))
 }
)
}

def login = Action(BodyParsers.parse.json) { implicit request =>
 val login = request.body.validate[User]

 login.fold(
 errors => BadRequest(JsonError.toJson(errors)),
 u => {
 registrationService.auth(u.username, u.password) match {
 case Some(user) => Ok
 case None => Forbidden
 }
 }
)
}
}

```

9. Add the routes for the newly added Registration actions:

```

POST /register controllers.Registrations.register
POST /auth controllers.Registrations.login

```

10. Finally, declare the registration plugin in the project root,  
`foo_scala/conf/play.plugins` file:

```
599:registration.RegistrationPlugin
```

11. Using curl, submit a new registration and login with the specified registration details; verify that our endpoint responds with an HTTP status 200 for successful operations by inspecting the response headers:

```

$ curl -v -X POST --header "Content-type: application/json"
http://localhost:9000/register -d '{"username":"ned@flanders.com",
"password":"password"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)

```

```
> POST /register HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 54
>
* upload completely sent off: 54 out of 54 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 63
<
* Connection #0 to host localhost left intact
{"id":null,"username":"ned@flanders.com","password":"password"}%
```

```
$ curl -v -X POST --header "Content-type: application/json"
http://localhost:9000/auth -d '{"username":"ned@flanders.com",
"password":"password"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /auth HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 54
>
* upload completely sent off: 54 out of 54 bytes
< HTTP/1.1 200 OK
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

# How it works...

In this recipe, we created a module that will handle registration functions, such as signup and login. We created it as a Play plugin so that it can not only be maintainable but also reusable in other applications. Another advantage of using modules is that when writing unit tests, we can isolate its execution in its enclosed subproject only and not the entire project.

We created the registration plugin inside the modules directory in our project root directory. We declared the dependency between the module and the main project in `build.sbt`:

```
lazy val root = (project in file("."))
 .enablePlugins(PlayScala)
 .aggregate(filemon)
 .dependsOn(filemon)
 .aggregate(registration)
 .dependsOn(registration)
```

We then enabled the plugin in `conf/play.plugins` using the priority level 599, within the 500-600 range for data-related plugins:

```
599:registration.RegistrationPlugin
```

We then grabbed a reference to the `RegistrationService` interface from the `Registration` plugin inside the controller:

```
// Java
private static RegistrationService registrationService =

Play.application().plugin(RegistrationPlugin.class).getRegistrationService()
;

// Scala
private val registrationService: RegistrationService =
Play.application.plugin[RegistrationPlugin]
 .getOrElse(throw new IllegalStateException("RegistrationService is
required!"))
 .registrationService
```

Once we established the reference, all registration functions were simply delegated to the `RegistrationService` interface from the controller:

```
// Java
registrationService.create(user);

// Scala
registrationService.create(u)
```

Using `curl`, we can also validate that our registration controller responds correctly to bad authentication:

```
$ curl -v -X POST --header "Content-type: application/json"
http://localhost:9000/auth -d '{"username": "ned@flanders.com",
```

```
"password":"passwordz"}'
 * Hostname was NOT found in DNS cache
 * Trying ::1...
 * Connected to localhost (::1) port 9000 (#0)
> POST /auth HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 55
>
* upload completely sent off: 55 out of 55 bytes
< HTTP/1.1 403 Forbidden
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```



# Using the same model for different applications

For this recipe, we will create a new standalone module that will contain product-related functions and data model classes, and we will publish it on a local repository:

# How to do it...

For Java, we need to perform the following steps:

1. Create a new Play 2 project in the same directory level as `foo_java`:

```
activator new product-contrib play-java
```

2. Create our default module package inside the app directory, `product-contrib/app`:

```
mkdir app/productcontrib
```

3. Create the `models` package, which will contain all the data model classes:

```
mkdir app/productcontrib/models
```

4. Remove the contents of the `conf/application.conf` file:

```
echo "" > conf/application.conf
```

5. Remove the contents of the `conf/routes` file and rename it to `productcontrib.routes`:

```
echo "" > conf/routes && mv conf/routes
modules/filemon/conf/productcontrib.routes
```

6. Remove the `views` directory from `modules/filemon/app`:

```
rm -rf app/views
```

7. Remove the file `app/controller/Application.java`:

```
rm app/controllers/Application.java
```

8. Create the product model in `app/productcontrib/models/Product.java` with the following contents:

```
package productcontrib.models;

import java.io.Serializable;

public class Product implements Serializable {
 private String sku;
 private String title;
 private Double price;

 public String getSku() {
 return sku;
 }
 public void setSku(String sku) {
 this.sku = sku;
 }
 public String getTitle() {
 return title;
 }
 public void setTitle(String title) {
 this.title = title;
 }
}
```

```

 }
 public Double getPrice() {
 return price;
 }
 public void setPrice(Double price) {
 this.price = price;
 }
}

```

9. Create the ProductService interface (`ProductService.java`) and In implementation class (`ProductServiceImpl.java`) in the package `app/productcontrib/services`:

```

// ProductService.java
package productcontrib.services;

public interface ProductService {
 String generateProductId();
}

// ProductServiceImpl.java
package productcontrib.services;

import java.util.UUID;

public class ProductServiceImpl implements ProductService {
 @Override
 public String generateProductId() {
 return UUID.randomUUID().toString();
 }
}

```

10. Insert additional module package settings in the `build.sbt` file:

```

name := """product-contrib"""
version := "1.0-SNAPSHOT"
organization := "foojava"

```

11. Using the activator, build the `contrib.jar` and publish it to the remote internal repository:

```
activator clean publish-local
```

12. You should be able to confirm in the console logs whether the upload was successful or not:

```

[info] published product-contrib_2.11 to
/.ivy2/local/foojava/product-contrib_2.11/1.0-SNAPSHOT/poms/product-
contrib_2.11.pom
[info] published product-contrib_2.11 to
/.ivy2/local/foojava/product-contrib_2.11/1.0-SNAPSHOT/jars/product-
contrib_2.11.jar
[info] published product-contrib_2.11 to
/.ivy2/local/foojava/product-contrib_2.11/1.0-SNAPSHOT/srcs/product-
contrib_2.11-sources.jar
[info] published product-contrib_2.11 to
/.ivy2/local/foojava/product-contrib_2.11/1.0-SNAPSHOT/docs/product-
contrib_2.11-javadoc.jar

```

```
[info] published ivy to /.ivy2/local/foojava/product-contrib_2.11/1.0-SNAPSHOT/ivys/ivy.xml
```

For Scala, we need to perform the following steps:

1. Create a new Play 2 project in the same directory level as foo\_scala:

```
activator new user-contrib play-scala
```

2. Create the default module package inside the app directory, user-contrib/app:

```
mkdir app/usercontrib
```

3. Create the models package that will contain all the data model classes:

```
mkdir app/usercontrib/models
```

4. Remove the contents of the conf/application.conf file:

```
echo "" > conf/application.conf
```

5. Remove the contents of the conf/routes file and rename it to usercontrib.routes:

```
echo "" > conf/routes && mv conf/routes
modules/filemon/conf/usercontrib.routes
```

6. Remove the views directory from modules/filemon/app:

```
rm -rf app/views
```

7. Remove the file app/controller/Application.scala:

```
rm app/controllers/Application.scala
```

8. Create the User model in app/usercontrib/models/User.scala with the following content:

```
package usercontrib.models

import java.util.UUID

case class User(id: Option[String], username: String, password: String)

object User {
 def generateId = UUID.randomUUID().toString
}
```

9. Insert additional module package settings in the build.sbt file:

```
name := """product-contrib"""
version := "1.0-SNAPSHOT"
organization := "foojava"
```

10. Using the activator, build the contrib.jar and publish it to the remote internal repository:

```
activator clean publish-local
```

11. You should be able to confirm in the console logs whether the upload was successful or not:

```
[info] published user-contrib_2.11 to ivy/fooscala/user-
contrib_2.11/1.0-SNAPSHOT/user-contrib_2.11-1.0-SNAPSHOT.pom
[info] published user-contrib_2.11 to ivy/fooscala/user-
contrib_2.11/1.0-SNAPSHOT/user-contrib_2.11-1.0-SNAPSHOT.jar
[info] published user-contrib_2.11 to ivy/fooscala/user-
contrib_2.11/1.0-SNAPSHOT/user-contrib_2.11-1.0-SNAPSHOT-sources.jar
[info] published user-contrib_2.11 to ivy/fooscala/user-
contrib_2.11/1.0-SNAPSHOT/user-contrib_2.11-1.0-SNAPSHOT-javadoc.jar
```

# How it works...

In this recipe, we created a new Play 2.0 module with the intention of packaging and publishing the module in our local repository. This makes the Play module available to the other Play web applications we will be working on. We created our model and service classes for a product that will be part of our module:

```
Java

$ find app/productcontrib
app/productcontrib
app/productcontrib/models
app/productcontrib/models/Product.java
app/productcontrib/services
app/productcontrib/services/ProductService.java
app/productcontrib/services/ProductServiceImpl.java

Scala
$ find app/usercontrib
app/usercontrib
app/usercontrib/models
app/usercontrib/models/User.scala
```

We built and published both modules into an internal repository using the activator publish command:

```
activator clean publish-local
```

Once these modules were published in the internal repository, we then declared them as dependencies to Maven-based Java projects, not limited to Play 2.0 applications, in our case, build.sbt:

```
// Java
"foojava" %% "product-contrib" % "1.0-SNAPSHOT"

// Scala
"fooscala" %% "user-contrib" % "1.0-SNAPSHOT"
```



# Managing module dependencies

In this recipe, we will tackle the topic of adding Play modules to your Play 2.0 application, which further demonstrates how powerful the Play 2.0 ecosystem is. This recipe requires the previous recipe to be run and assumes that you have followed on.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Add the `fooscala user-contrib` module as a project dependency in `build.sbt`:

```
"fooscala" %% "user-contrib" % "1.0-SNAPSHOT",
"foojava" %% "product-contrib" % "1.0-SNAPSHOT"
```

3. Modify `foo_java/app/controllers/Application.java` by adding the following action:

```
// Add the required imports
import productcontrib.services.ProductService;
import productcontrib.services.ProductServiceImpl;
import usercontrib.models.User;

// Add the necessary Action methods and helper
private static ProductService productService = new
ProductServiceImpl();

public static Result generateProductId() {
 return ok("Your generated product id: " +
productService.generateProductId());
}

public static Result generateUserId() {
 return ok("Your generated product id: " + User.generateId());
}
```

4. Add a new route for the newly added action in `foo_java/conf/routes`:

```
GET /generate-product-id
controllers.Application.generateProductId()
GET /generate-user-id
controllers.Application.generateUserId()
```

5. Using `curl`, we will be able to display the product and user Ids generated from the product and user contrib modules:

```
$ curl http://localhost:9000/generate-product-id
Your generated product id: 3acd3f36-6ee6-45ce-af07-faa257724b1e%
$ curl http://localhost:9000/generate-user-id
Your generated product id: ffca654e-35d8-48cd-9acd-9ea9fe567ba7%
```

For Scala, we need to perform the following steps:

1. Run the `foo_scala` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Add the `foojava productcontrib` module as a project dependency in `build.sbt`:

```
"foojava" %% "product-contrib" % "1.0-SNAPSHOT",
"fooscala" %% "user-contrib" % "1.0-SNAPSHOT"
```

3. Modify `foo_scala/app/controllers/Application.scala` by adding the following action:

```
import productcontrib.services.{ProductServiceImpl, ProductService}
import usercontrib.models.User

def productService: ProductService = new ProductServiceImpl

def generateProductId = Action {
 Ok("Your generated product id: " +
productService.generateProductId())
}

def generateUserId = Action {
 Ok("Your generated product id: " + User.generateId());
}
```

4. Add a new route for the newly added action in `foo_scala/conf/routes`:

```
GET /generate-product-id
controllers.Application.generateProductId()
GET /generate-user-id
controllers.Application.generateUserId()
```

5. Using `curl`, we will be able to display the product and user Ids generated from the product and user contrib modules:

```
$ curl http://localhost:9000/generate-product-id
Your generated product id: 3acd3f36-6ee6-45ce-af07-faa257724b1e%
$ curl http://localhost:9000/generate-user-id
Your generated product id: ffca654e-35d8-48cd-9acd-9ea9fe567ba7%
```

# How it works...

In this recipe, we explored how to include other modules into our Play 2.0 web application. With this recipe, we also displayed how Play Scala apps can work side by side with Play Java modules and vice versa.

We first declared that our root project will use both user and product contrib modules in `build.sbt`:

```
libraryDependencies += Seq(
 "foojava" %% "product-contrib" % "1.0-SNAPSHOT",
 "fooscala" %% "user-contrib" % "1.0-SNAPSHOT"
)
```

We then added the import statements to our controller so we could invoke their ID generation functions:

```
// Java
import productcontrib.services.ProductService;
import productcontrib.services.ProductServiceImpl;
import usercontrib.models.User;

// Invoke the contrib functions in foo_java
return ok("Your generated product id: " +
productService.generateProductId());
return ok("Your generated product id: " + User.generateId());

// Scala
import productcontrib.services.{ProductServiceImpl, ProductService}
import usercontrib.models.User

// Invoke the contrib functions in foo_scala:
Ok("Your generated product id: " + productService.generateProductId())
Ok("Your generated product id: " + User.generateId());
```

Finally, we used `curl` to request our new routes to see the generated Ids in action.



# Adding private module repositories using Amazon S3

In this recipe, we will explore how we can use an external module repository to publish and resolve internal modules in the interest of distributing our modules. In this recipe, we will use Amazon S3, a popular cloud storage service, to store our ivy-style repository assets. You will need a valid AWS account to follow this recipe, ensure that you sign up for one at <http://aws.amazon.com/>.

Please refer to S3's online documentation for more information:

<http://aws.amazon.com/s3/>

# How to do it...

We need to perform the following steps:

1. Open the product-contrib project and run the application with Hot-Reloading enabled:  

```
activator "~run"
```

2. Edit the plugins config file in project/plugins.sbt and add the following plugin and resolver:  

```
resolvers += "Era7 maven releases" at
"http://releases.era7.com.s3.amazonaws.com"
```

```
addSbtPlugin("ohnosequences" % "sbt-s3-resolver" % "0.12.0")
```

3. Edit the build config file in build.sbt to specify the settings we'll use for the S3 resolver plugin:  

```
S3Resolver.defaults
```

```
s3credentials := file(System.getProperty("user.home")) / ".sbt" /
.s3credentials"
```

```
publishMavenStyle := false
```

```
publishTo := {
 val prefix = if (isSnapshot.value) "snapshots" else "releases"
 Some(s3resolver.value(prefix+" S3 bucket",
 s3(prefix+".YOUR-
S3-BUCKET-NAME-HERE.amazonaws.com")) withIvyPatterns)
}
```

4. Specify your Amazon S3 API keys in the file ~/.sbt/.s3credentials:  

```
accessKey = <YOUR S3 API ACCESS KEY>
secretKey = <YOUR S3 API SECRET KEY>
```

5. Next, publish the product-contrib snapshot using the activator:  

```
activator clean publish
```

6. You will see the success status message of the upload in the console logs:  

```
[info] published ivy to
s3://snapshots.XXX.XXX.amazonaws.com/foojava/product-contrib_2.11/1.0-
SNAPSHOT/ivys/ivy.xml
[success] Total time: 58 s, completed 02 3, 15 9:39:44 PM
```

7. We will now use this repository in a new Play 2.0 application:  

```
activator new s3deps play-scala
```

8. Edit the plugins config file in s3deps/project/plugins.sbt and add the following plugin and resolver:  

```
resolvers += "Era7 maven releases" at
```

```
"http://releases.era7.com.s3.amazonaws.com"
```

```
"http://releases.era7.com.s3.amazonaws.com"
 addSbtPlugin("ohnosequences" % "sbt-s3-resolver" % "0.12.0")
```

9. Edit the build config file in build.sbt to specify the settings we'll use for the S3 resolver plugin:

```
S3Resolver.defaults

resolvers ++= Seq[Resolver](
 s3resolver.value("Snapshots resolver", s3("YOUR-S3-BUCKET-NAME-
HERE.amazonaws.com")) withIvyPatterns
)

libraryDependencies ++= Seq(
 "foojava" %% "product-contrib" % "1.0-SNAPSHOT"
)
```

10. Retrieve the product-contrib module using the activator:

```
activator clean dependencies
...
com.typesafe.play:play-java-ws_2.11:2.3.7
com.typesafe.play:play-java-jdbc_2.11:2.3.7
foojava:product-contrib_2.11:1.0-SNAPSHOT
```

# How it works...

In this recipe, we used the sbt-s3-resolver plugin to publish and resolve dependencies using Amazon S3. We included the sbt plugin in the file `project/plugins.sbt`:

```
resolvers += "Era7 maven releases" at
"http://releases.era7.com.s3.amazonaws.com"

addSbtPlugin("ohnosequences" % "sbt-s3-resolver" % "0.12.0")
```

We specify our Amazon S3 API keys in the `.s3credentials` files in `~/.sbt` directory:

```
accessKey = <ACCESS KEY>
secretKey = <SECRET KEY>
```

For publishing, we specify the resolver repository in `build.sbt` of the publishing project (`product-contrib`):

```
S3Resolver.defaults

s3credentials := file(System.getProperty("user.home")) / ".sbt" /
".s3credentials"

publishMavenStyle := false

publishTo := {
 val prefix = if (isSnapshot.value) "snapshots" else "releases"
 Some(s3resolver.value(prefix+" S3 bucket",
s3(prefix+".achiiva.devint.amazonaws.com")) withIvyPatterns)
}
```

To resolve dependencies, we specify the following in `build.sbt` of the consuming project (`s3deps`):

```
S3Resolver.defaults

resolvers ++= Seq[Resolver](
s3resolver.value("Snapshots resolver", s3("YOUR-S3-BUCKET-NAME-
HERE.amazonaws.com")) withIvyPatterns
)

libraryDependencies ++= Seq(
"foojava" %% "product-contrib" % "1.0-SNAPSHOT"
)
```



# Chapter 6. Practical Module Examples

In this chapter, we will cover the following recipes:

- Integrating a Play application with message queues
- Integrating a Play application with ElasticSearch
- Implementing token authentication using JWT

# Introduction

In this chapter, we will look further into integrating a Play application with other essential services and tools for the modern web application. Specifically, we will look into how we can integrate an external message queue service with a Play plugin. We will use the popular cloud service **IronMQ** for this.

We will also look into integrating a full text search engine service with a Play application using **ElasticSearch** and **Docker**.

Finally, we will implement our own Play wrappers for the integration of token authentication using **JSON Web Tokens (JWT)**.



# Integrating a Play application with message queues

In this recipe, we will explore how to use Play 2.0 to integrate with IronMQ, a popular cloud message queue service. We will use IronMQ's Java libraries, which can be found here:

[https://github.com/iron-io/iron\\_mq\\_java](https://github.com/iron-io/iron_mq_java)

We will use a Play plugin to initialize our IronMQ client and queue objects, and will expose helper methods to send and retrieve messages. We will then use this plugin in a Play controller that will allow clients to post messages and retrieve messages using the HTTP method, GET.

# How to do it...

For Java, we need to perform the following steps:

1. Run the `foo_java` application with Hot-Reloading enabled:

```
activator "~run"
```

2. Create an IronMQ account on <http://www.iron.io/> and create an IronMQ project; make a note of your project ID and token
3. Import the official IronMQ java libraries as app dependencies in `build.sbt`:

```
libraryDependencies ++= Seq(
 "io.iron.ironmq" % "ironmq" % "0.0.19"
)
```

4. Create the `plugins` package inside `foo_java/app`:

```
mkdir plugins
```

5. Create our plugin class, `MQPlugin`, in the `foo_java/app/plugins` directory:

```
package plugins;

import io.iron.ironmq.Client;
import io.iron.ironmq.Message;
import io.iron.ironmq.Messages;
import io.iron.ironmq.Queue;
import play.Application;
import play.Logger;
import play.Plugin;
import java.util.UUID;

public class MQPlugin extends Plugin {
 final private Integer messageSize = 10;
 private Client client;
 private Queue queue;

 public MQPlugin(Application app) {
 super();
 client = new Client(
 app.configuration().getString("mq.projectId"),
 app.configuration().getString("mq.token")
);
 }

 public void onStart() {
 queue = client.queue(UUID.randomUUID().toString());
 }

 public void onStop() {
 try {
 queue.clear();
 queue.destroy();
 client = null;
 } catch(Exception e) {
```

```

 Logger.error(e.toString());
 }
}

public void send(String msg) throws Exception {
 queue.push(msg);
}

public Message[] retrieve() throws Exception {
 Messages messages = queue.get(messageSize);
 if (messages.getSize() > 0) {
 Message[] msgArray = messages.getMessages();

 for(Message m : msgArray) {
 queue.deleteMessage(m);
 }

 return msgArray;
 }

 return new Message[] {};
}

public boolean enabled() {
 return true;
}
}

```

6. Modify conf/application.conf and add your IronMQ project ID and token:

```

mq.projectId="YOUR PROJECT ID"
mq.token="YOUR TOKEN"

```

7. Initialize the MQPlugin by declaring it in the conf/play.plugins file:

```
599:plugins.MQPlugin
```

8. Create a Messenger controller class in app/controllers:

```

package controllers;

import play.Logger;
import play.Play;
import play.data.Form;
import play.mvc.BodyParser;
import play.mvc.Controller;
import play.mvc.Result;
import plugins.MQPlugin;
import java.util.HashMap;
import java.util.Map;

import static play.libs.Json.toJson;

public class Messenger extends Controller {
 private static MQPlugin mqPlugin =
Play.application().plugin(MQPlugin.class);

```

```

 @BodyParser.Of(BodyParser.Json.class)
 public static Result sendMessage() {
 try {
 Form<Message> form =
Form.form(Message.class).bindFromRequest();
 Message m = form.get();
 mqPlugin.send(m.getBody());
 }
 Map<String, String> map = new HashMap<>();
 map.put("status", "Message sent.");
 return created(toJson(map));
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
}

public static Result getMessages() {
 try {
 return ok(toJson(mqPlugin.retrieve()));
 } catch (Exception e) {
 Logger.error(e.toString());
 return internalServerError();
 }
}
}

```

## 9. Add new routes for the Messenger controller actions:

POST	/messages/send	controllers.Messenger.sendMessage
GET	/messages	controllers.Messenger.getMessages

## 10. Using curl, let's send and retrieve messages:

```

$ curl -v -X POST http://localhost:9000/messages/send --header
"Content-type: application/json" -d '{"body":"Her mouth the mischief he
doth seek"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /messages/send HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 46
>
* upload completely sent off: 46 out of 46 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 26
<
* Connection #0 to host localhost left intact
{"status":"Message sent."}%

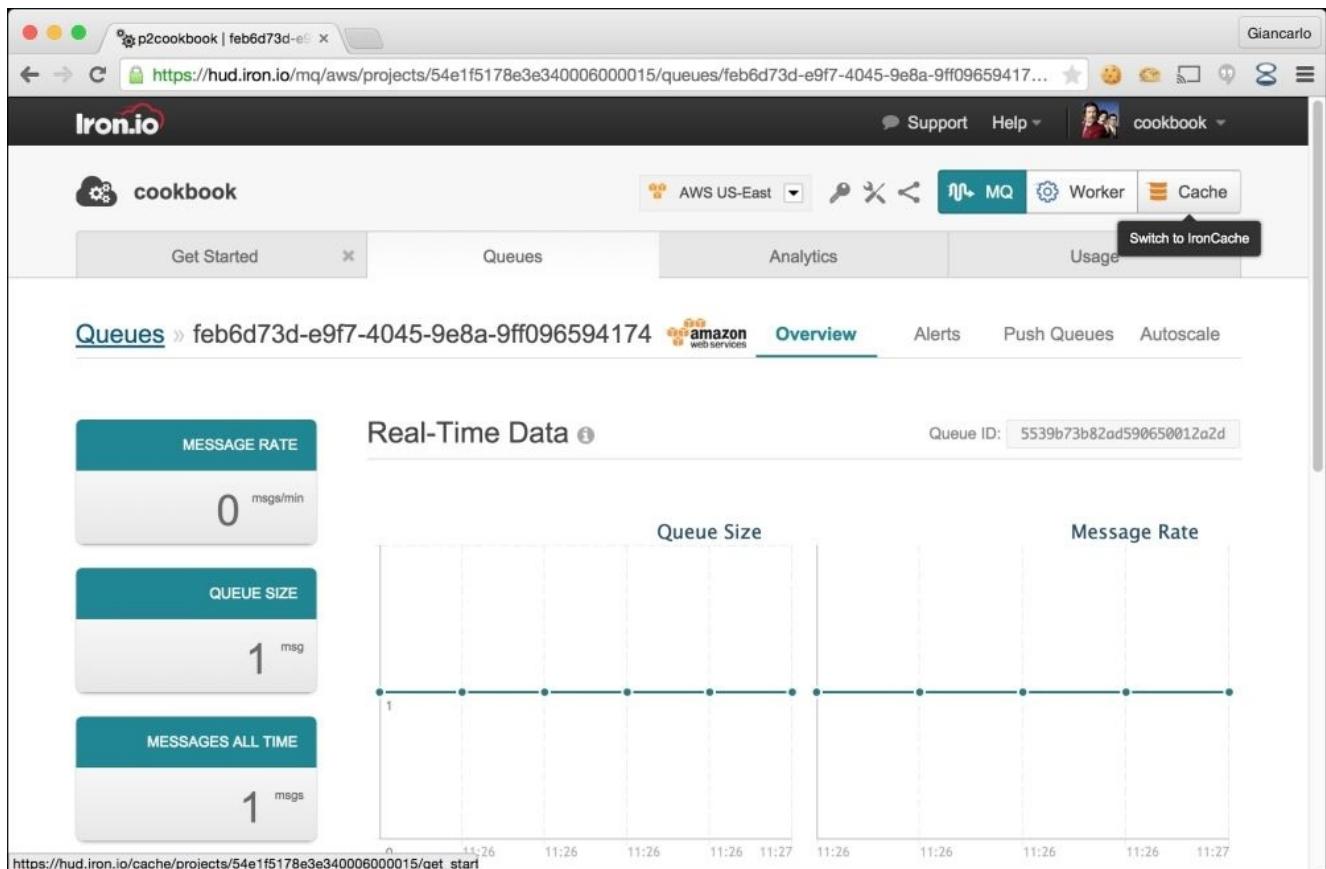
$ curl -v http://localhost:9000/messages
* Hostname was NOT found in DNS cache

```

```

* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /messages HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 95
<
* Connection #0 to host localhost left intact
{"messages": ["Her mouth the mischief he doth seek", "Her heart the
captive of which he speaks"]}%
```

- In the IronMQ Web Console, you can also confirm the queue message size to confirm that we were able to post a message:



For Scala, we need to perform the following steps:

- Run the `foo_scala` application with Hot-Reloading enabled:  
`activator "~run"`
- Create an IronMQ account on <http://www.iron.io/> and create an IronMQ project; make a note of your project ID and token.
- Import the official IronMQ java libraries as app dependencies in `build.sbt`:

```
libraryDependencies += Seq(
 "io.iron.ironmq" % "ironmq" % "0.0.19"
)
```

4. Create the plugins package in foo\_scala/app:

```
mkdir plugins
```

5. Create our plugin class, MQPlugin, in the foo\_scala/app/plugins directory:

```
package plugins

import java.util.UUID
import io.iron.ironmq.{Client, Queue}
import play.api.Play.current
import play.api.{Application, Play, Plugin}
import play.api.libs.concurrent.Execution.Implicits._
import scala.concurrent.Future

class MQPlugin(app: Application) extends Plugin {
 private val messageSize = 10
 private var client: Client = null
 private var queue: Queue = null

 override def onStart() = {
 client = new Client(
 Play.configuration.getString("mq.projectId").get,
 Play.configuration.getString("mq.token").get
)
 queue = client.queue(UUID.randomUUID().toString)
 }

 override def onStop() = {
 queue.clear()
 queue.destroy()
 client = null
 }

 def send(msg: String) = queue.push(msg)

 def retrieve = {
 val list = queue.get(messageSize)
 Future {
 list.getMessages.map(queue.deleteMessage(_))
 }
 list.getMessages.map(_.getBody)
 }
}

override def enabled = true
```

6. Modify conf/application.conf and add your IronMQ project ID and token:

```
mq.projectId="YOUR PROJECT ID"
mq.token="YOUR TOKEN"
```

7. Initialize the MQPlugin by declaring it in the conf/play.plugins file:

```
599:plugins.MQPlugin
```

## 8. Create a Messenger controller class in app/controllers:

```
package controllers

import play.api.Play.current
import play.api.Play
import play.api.libs.json.{JsError, Json}
import play.api.mvc.{BodyParsers, Action, Controller}
import plugins.MQPlugin

case class Message(body: String)

object Messenger extends Controller {
 implicit private val writes = Json.writes[Message]
 implicit private val reads = Json.reads[Message]
 private val mqPlugin = Play.application.plugin[MQPlugin].get

 def sendMessage = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Message]

 post.fold(
 errors => BadRequest(Json.Error.toJson(errors)),
 p => {
 mqPlugin.send(p.body)
 Created(Json.obj("status" -> "Message sent."))
 }
)
 }

 def getMessages = Action {
 Ok(Json.obj("messages" -> mqPlugin.retrieve))
 }
}
```

## 9. Add new routes for the Messenger controller actions:

POST	/messages/send	controllers.Messenger.sendMessage
GET	/messages	controllers.Messenger.getMessages

## 10. Using curl, let's send and retrieve messages:

```
$ curl -v -X POST http://localhost:9000/messages/send --header
"Content-type: application/json" -d '{"body":"Her mouth the mischief he
doth seek"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /messages/send HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: /*
> Content-type: application/json
> Content-Length: 46
>
```

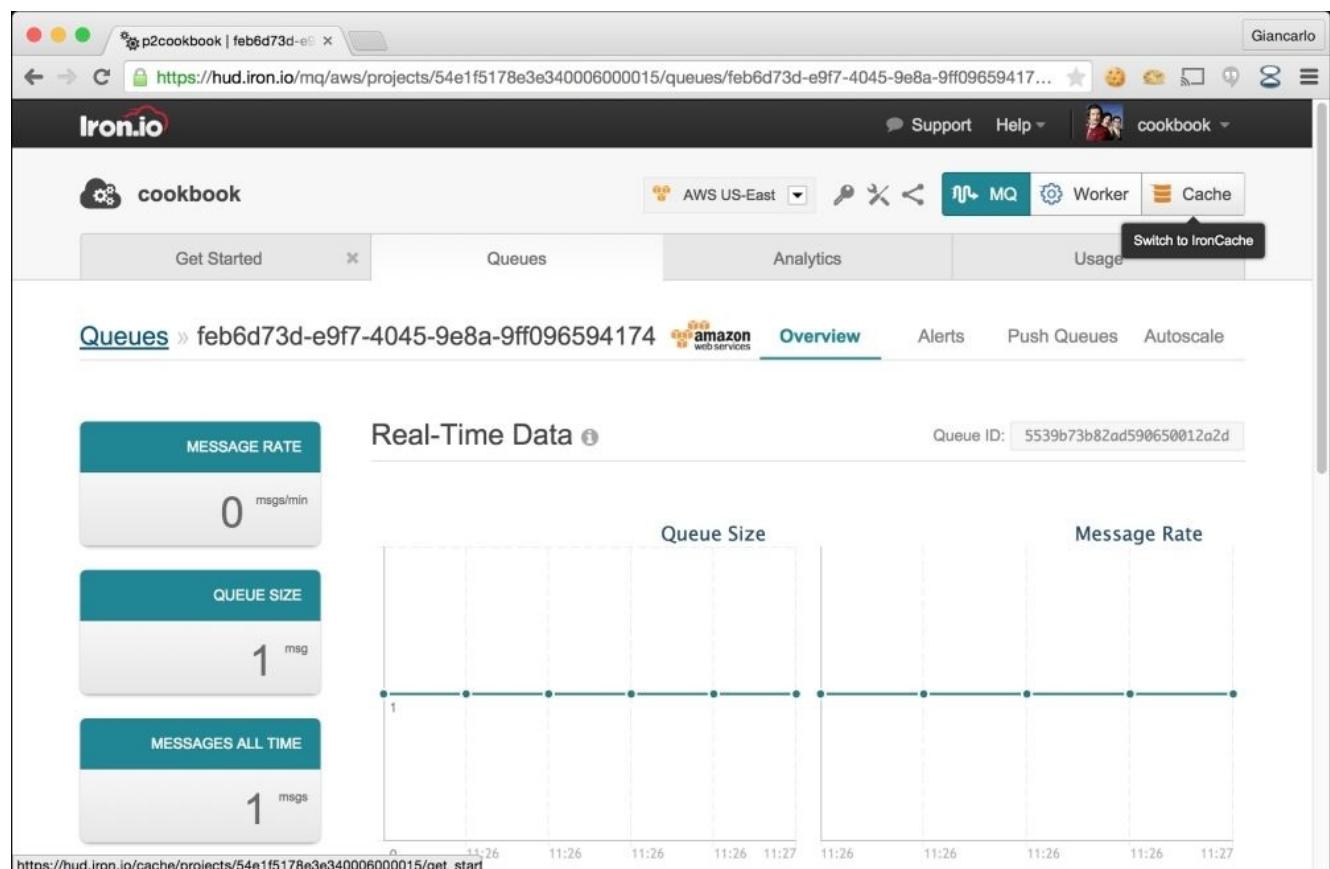
```

* upload completely sent off: 46 out of 46 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 26
<
* Connection #0 to host localhost left intact
{"status":"Message sent."}%

$ curl -v http://localhost:9000/messages
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /messages HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 95
<
* Connection #0 to host localhost left intact
{"messages":["Her mouth the mischief he doth seek", "Her heart the
captive of which he speaks"]}%

```

- In the IronMQ Web Console, you can also check the queue message size to confirm that we were able to post a message:



# How it works...

In this recipe, we set up the message queue plugin by first importing the official IronMQ Java library to `build.sbt`. We will also need to log in to IronMQ to create our IronMQ project and to retrieve our project tokens:

We configured the Play application by adding our IronMQ credentials to `conf/application.conf`:

```
mq.projectId="YOUR PROJECT ID"
mq.token="YOUR TOKEN"
```

We then implemented the `MQPlugin` class by retrieving the project ID and token from the config file and passing that to an instance of `io.iron.ironmq.Client`:

```
// Java
public MQPlugin(Application app) {
 super();
 client = new Client(
 app.configuration().getString("mq.projectId"),
 app.configuration().getString("mq.token")
);
}

// Scala
override def onStart() = {
 client = new Client(
 app.configuration().getString("mq.projectId"),
 app.configuration().getString("mq.token")
);
}
```

```

 Play.configuration.getString("mq.projectId").get,
 Play.configuration.getString("mq.token").get
)
}

```

We created our dynamic message queue in the `onStart` method as well, passing a UUID parameter as the default queue name:

```

// Java
public void onStart() {
 queue = client.queue(UUID.randomUUID().toString());
}

// Scala
queue = client.queue(UUID.randomUUID().toString)

```

We then declared the two methods that will facilitate message sending and retrieval:

```

// Java
public void send(String msg) throws Exception {
 queue.push(msg);
}

public Message[] retrieve() throws Exception {
 Messages messages = queue.get(messageSize);
 if (messages.getSize() > 0) {
 Message[] msgArray = messages.getMessages();

 for(Message m : msgArray) {
 queue.deleteMessage(m);
 }

 return msgArray;
 }

 return new Message[] {};
}

// Scala
def send(msg: String) = queue.push(msg)

def retrieve = {
 val list = queue.get(messageSize)
 Future {
 list.getMessages.map(queue.deleteMessage(_))
 }
 list.getMessages.map(_.getBody)
}

```

We wanted to be able to remove read messages from the queue and run an asynchronous function to delete them here:

```

// Scala
import play.api.libs.concurrent.Execution.Implicits._
import scala.concurrent.Future

```

```

Future {
 list.getMessages.map(queue.deleteMessage(_))
}

```

For Java, deleting the message will happen synchronously:

```

for(Message m : msgArray) {
 queue.deleteMessage(m);
}

```

Lastly, we implement the endpoints by creating the controller class, `Messenger`, which exposes two actions; one for message retrieval and another for messaging posting:

```

// Java
private static MQPlugin mqPlugin =
Play.application().plugin(MQPlugin.class);

@BodyParser.Of(BodyParser.Json.class)
public static Result sendMessage() {
 try {
 Form<Message> form =
Form.form(Message.class).bindFromRequest();
 Message m = form.get();
 mqPlugin.send(m.getBody());

 Map<String, String> map = new HashMap<>();
 map.put("status", "Message sent.");
 return created(toJson(map));
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
}

public static Result getMessages() {
 try {
 return ok(toJson(mqPlugin.retrieve()));
 } catch (Exception e) {
 Logger.error(e.toString());
 return internalServerError();
 }
}

// Scala
implicit private val writes = Json.writes[Message]
implicit private val reads = Json.reads[Message]
private val mqPlugin = Play.application.plugin[MQPlugin].get

def sendMessage = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Message]

 post.fold(
 errors => BadRequest(JsonError.toFlatJson(errors)),
 p => {
 mqPlugin.send(p.body)
 Created(Json.obj("status" -> "Message sent."))
 }
)
}

```

```
)
}

def getMessages = Action {
 Ok(Json.obj("messages" -> mqPlugin.retrieve))
}
```

And finally, add the respective routes to conf/routes:

POST	/messages/send	controllers.Messenger.sendMessage
GET	/messages	controllers.Messenger.getMessages



# Integrating a Play application with ElasticSearch

In this recipe, we will create a very common web application functionality to create, index, and search, in our case, products. We will use ElasticSearch as our search service. We will use Docker to create our local ElasticSearch container and to run all search operations.

A prerequisite for this recipe is having access to an ElasticSearch instance, either local or remote, in our recipe, as well as having Docker installed in the local development machine:

```
$ docker -v
Docker version 1.3.3, build d344625
```

We used Docker to deploy our local ElasticSearch instance using the command:

```
docker run -d -p 9200:9200 -p 9300:9300 dockerfile/elasticsearch
```

The preceding command instructs docker to run the elasticsearch container as a detached service and that ports 9200 and 9300 in the container should be accessible from the corresponding ports in the host.

We will use an open source Play module, **play2-elasticsearch**, to wrap our calls to the ElasticSearch instance. This recipe assumes some familiarity in Docker and full-text searching services. More information about play2-elasticsearch can be found at <https://github.com/cleverage/play2-elasticsearch>.

## Note

For more information regarding Docker and how to install it, please refer to their online documentation at <https://docs.docker.com/installation/>.

# How to do it...

For Java, we need to perform the following steps:

1. First, let's fire up a local ElasticSearch container using Docker:

```
$ docker run -d -p 9200:9200 -p 9300:9300 dockerfile/elasticsearch
```

2. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

3. Add the play2-elasticsearch dependency in build.sbt. It is important to note that as of writing this, support for Play 2.3.x has not been released for play2-elasticsearch, hence, the need to exclude dependencies to older Play libraries:

```
resolvers += "Sonatype OSS Snapshots" at
"https://oss.sonatype.org/content/repositories/snapshots"

libraryDependencies ++= Seq(
 ("com.clever-age" % "play2-elasticsearch" % "1.4-SNAPSHOT")
 .exclude("com.typesafe.play", "play-functional_2.10")
 .exclude("com.typesafe.akka", "akka-actor_2.10")
 .exclude("com.typesafe.play", "play-json_2.10")
 .exclude("com.typesafe.play", "play_2.10")
 .exclude("com.typesafe.play", "play-iteratees_2.10")
 .exclude("com.typesafe.akka", "akka-slf4j_2.10")
 .exclude("org.scala-stm", "scala-stm_2.10")
 .exclude("com.typesafe.play", "play-datacommons_2.10")
 .exclude("com.typesafe.play", "play-java_2.10")
)
```

4. Declare the play2-elasticsearch plugin in conf/play.plugins:

```
9000:com.github.cleverage.elasticsearch.plugin.IndexPlugin
```

5. Add play2-elasticsearch configuration parameters to conf/application.conf:

```
elasticsearch.local=false
elasticsearch.client=<YOUR ELASTIC SEARCH HOST HERE>:9300"
elasticsearch.sniff=false
elasticsearch.index.name="test"
elasticsearch.index.settings="{ analysis: { analyzer: {
 my_analyzer: { type: \"custom\", tokenizer: \"standard\" } } } }"
elasticsearch.index.clazzs="models.*"
elasticsearch.index.show_request=true
elasticsearch.cluster.name=elasticsearch
```

6. Create the product model in app/models/Product.java:

```
package models;

import com.github.cleverage.elasticsearch.Index;
import com.github.cleverage.elasticsearch.IndexQuery;
import com.github.cleverage.elasticsearch.IndexResults;
import com.github.cleverage.elasticsearch.Indexable;
import com.github.cleverage.elasticsearch.annotations.IndexType;
```

```
import org.elasticsearch.index.query.QueryBuilders;
import java.util.HashMap;
import java.util.Map;

@IndexType(name = "product")
public class Product extends Index {
 private String id;
 private String title;
 private String shortDesc;

 public Product() {}

 public Product(String id, String title, String shortDesc) {
 this.id = id;
 this.title = title;
 this.shortDesc = shortDesc;
 }

 public String getId() {
 return id;
 }

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

 public String getTitle() {
 return title;
 }

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

 public String getShortDesc() {
 return shortDesc;
 }

 public void setShortDesc(String shortDesc) {
 this.shortDesc = shortDesc;
 }

 public static Finder<Product> find = new Finder<>(
 Product.class);

 @Override
 public Map toIndex() {
 Map<String, Object> map = new HashMap<>();
 map.put("id", this.id);
 map.put("title", this.title);
 map.put("description", this.getShortDesc());
 return map;
 }

 @Override
 public Indexable fromIndex(Map map) {
```

```

 Product p = new Product();
 p.setId((String) map.get("id"));
 p.setTitle((String) map.get("title"));
 p.setShortDesc((String) map.get("description")));
 return p;
 }

 public static IndexResults<Product> doSearch(String keyword) {
 IndexQuery<Product> indexQuery = Product.find.query();

 indexQuery.setBuilder(QueryBuilders.multiMatchQuery(keyword, "title",
 "description"));
 return Product.find.search(indexQuery);
 }
}

```

7. Next, create the products endpoint that will serve the creation and searching of products in app/controllers/Products.java:

```

package controllers;

import com.github.cleverage.elasticsearch.IndexResults;
import models.Product;
import play.data.Form;
import play.mvc.BodyParser;
import play.mvc.Controller;
import play.mvc.Result;
import java.util.HashMap;
import java.util.Map;

import static play.libs.Json.toJson;

public class Products extends Controller {
 @BodyParser.Of(BodyParser.Json.class)
 public static Result create() {
 try {
 Form<Product> form =
Form.form(Product.class).bindFromRequest();
 Product product = form.get();
 product.index();

 return created(toJson(product));
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
 }

 @BodyParser.Of(BodyParser.Json.class)
 public static Result search() {
 try {
 Form<Search> form =
Form.form(Search.class).bindFromRequest();
 Search search = form.get();

 IndexResults<Product> results =

```

```

Product.doSearch(search.getKeyword());
 Map<String, Object> map = new HashMap<>();
 map.put("total", results.getTotalCount());
 map.put("products", results.getResults());

 return ok(toJson(map));

} catch (Exception e) {
 return internalServerError(e.getMessage());
}
}
}
}
}

```

8. Let's also add the Search class from the helper class to app/controllers/Search.java:

```

package controllers;

public class Search {
 private String keyword;

 public String getKeyword() {
 return keyword;
 }
 public void setKeyword(String keyword) {
 this.keyword = keyword;
 }
}

```

9. Finally, let's add the routes to the product controller actions to conf/routes:

POST	/products	controllers.Products.create
GET	/products/search	controllers.Products.search

10. Using curl, we can test product creation and indexing as follows:

# Let's insert 2 products:

```

curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" -d '{"id":"1001", "title":"Intel Core i7-4790K Processor", "shortDesc": "New Unlocked 4th Gen Intel Core Processors deliver 4 cores of up to 4 GHz base frequency providing blazing-fast computing performance for the most demanding users"}'

curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" -d '{"id":"1002", "title": "AMD FD6300WMHKBOX FX-6300 6-Core Processor", "shortDesc": "AMD FX 6-Core Processor Unlocked Black Edition. AMD's next-generation architecture takes 8-core processing to a new level. Get up to 24% better frame rates in some of the most demanding games at stunning resolutions. Get faster audio encoding so you can enjoy your music sooner. Go up to 5.0 GHz with aggressive cooling solutions from AMD."}'

```

11. We can also use curl to execute product searches:

```
$ curl -X GET http://localhost:9000/products/search --header "Content-type: application/json" -d '{"keyword": "processor"}'
```

```
{"total":2,"products":[{"id":"1001","title":"Intel Core i7-4790K Processor","shortDesc":"New Unlocked 4th Gen Intel Core Processors deliver 4 cores of up to 4 GHz base frequency providing blazing-fast computing performance for the most demanding users"}, {"id":"1002","title":"AMD FD6300WMHKBOX FX-6300 6-Core Processor","shortDesc":"AMD FX 6-Core Processor Unlocked Black Edition. AMDs next-generation architecture takes 8-core processing to a new level. Get up to 24% better frame rates in some of the most demanding games at stunning resolutions. Get faster audio encoding so you can enjoy your music sooner. Go up to 5.0 GHz with aggressive cooling solutions from AMD."}]}%
```

```
$ curl -X GET http://localhost:9000/products/search --header "Content-type: application/json" -d '{"keyword": "amd"}'
{"total":1,"products":[{"id":"1002","title":"AMD FD6300WMHKBOX FX-6300 6-Core Processor","shortDesc":"AMD FX 6-Core Processor Unlocked Black Edition. AMDs next-generation architecture takes 8-core processing to a new level. Get up to 24% better frame rates in some of the most demanding games at stunning resolutions. Get faster audio encoding so you can enjoy your music sooner. Go up to 5.0 GHz with aggressive cooling solutions from AMD."}]}%
```

For Scala, we need to perform the following steps:

1. First, let's fire up a local ElasticSearch container using Docker:

```
$ docker run -d -p 9200:9200 -p 9300:9300 dockerfile/elasticsearch
```

2. Run the foo\_scala application with Hot-Reloading enabled:

```
activator "~run"
```

3. Add the play2-elasticsearch dependency in build.sbt. It is important to note that as of writing this, support for Play 2.3.x has not been released for play2-elasticsearch, hence the need to exclude dependencies from older Play libraries:

```
resolvers += "Sonatype OSS Snapshots" at
"https://oss.sonatype.org/content/repositories/snapshots"

libraryDependencies ++= Seq(
 ("com.clever-age" % "play2-elasticsearch" % "1.4-SNAPSHOT")
 .exclude("com.typesafe.play", "play-functional_2.10")
 .exclude("com.typesafe.akka", "akka-actor_2.10")
 .exclude("com.typesafe.play", "play-json_2.10")
 .exclude("com.typesafe.play", "play_2.10")
 .exclude("com.typesafe.play", "play-iteratees_2.10")
 .exclude("com.typesafe.akka", "akka-sl4j_2.10")
 .exclude("org.scala-stm", "scala-stm_2.10")
 .exclude("com.typesafe.play", "play-datacommons_2.10")
)
```

4. Declare the play2-elasticsearch plugin in conf/play.plugins:

```
9000:com.github.cleverage.elasticsearch.plugin.IndexPlugin
```

5. Add play2-elasticsearch configuration parameters to conf/application.conf:

```

elasticsearch.local=false
elasticsearch.client="<YOUR_ELASTICSEARCH_HOST_HERE>:9300"
elasticsearch.sniff=false
elasticsearch.index.name="test"
elasticsearch.index.settings="{ analysis: { analyzer: {
my_analyzer: { type: \"custom\", tokenizer: \"standard\" } } } }"
elasticsearch.index.clazzes="models.*"
elasticsearch.index.show_request=true
elasticsearch.cluster.name=elasticsearch

```

6. Create our product model and the ProductManager class in app/models/Product.scala:

```

package models

import com.github.cleverage.elasticsearch.ScalaHelpers.{IndexQuery,
IndexableManager, Indexable}
import org.elasticsearch.index.query.QueryBuilders
import play.api.libs.json.{Writes, Json, Reads}

case class Product(id: String, title: String, shortDesc: String)
extends Indexable

object ProductManager extends IndexableManager[Product] {
 override val indexType: String = "string"
 override val reads: Reads[Product] = Json.reads[Product]
 override val writes: Writes[Product] = Json.writes[Product]

 def doSearch(keyword: String) = {
 val indexQuery = new IndexQuery[Product]()
 .withBuilder(QueryBuilders.multiMatchQuery(keyword, "title",
"description"))
 search(indexQuery)
 }
}

```

7. Next, create the products endpoint that will serve the creation and searching of products in app/controllers/Products.scala:

```

package controllers

import models.{Product, ProductManager}
import play.api.libs.json.{JsError, Json}
import play.api.mvc.{BodyParsers, Action, Controller}

case class Search(keyword: String)

object Products extends Controller {
 implicit private val productWrites = Json.writes[Product]
 implicit private val productReads = Json.reads[Product]
 implicit private val searchWrites = Json.writes[Search]
 implicit private val searchReads = Json.reads[Search]

 def create = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Product]
 }
}

```

```

post.fold(
 errors => BadRequest(JsonError.toJson(errors)),
 p => {
 ProductManager.index(p)
 Created(Json.toJson(p))
 }
)
}

def search = Action(BodyParsers.parse.json) { implicit request =>
 request.body.validate[Search].fold(
 errors => BadRequest(JsonError.toJson(errors)),
 search => {
 val results = ProductManager.doSearch(search.keyword)
 Ok(Json.obj(
 "total" -> results.totalCount,
 "products" -> results.results
))
 }
)
}
}
}

```

- Finally, let's add the routes to the product controller actions to conf/routes:

POST	/products	controllers.Products.create
GET	/products/search	controllers.Products.search

- Using curl, we can test product creation and indexing as follows:

```
Let's insert 2 products:
```

```
curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" -d '{"id":"1001", "title":"Intel Core i7-4790K Processor", "shortDesc": "New Unlocked 4th Gen Intel Core Processors deliver 4 cores of up to 4 GHz base frequency providing blazing-fast computing performance for the most demanding users"}'
```

```
curl -v -X POST http://localhost:9000/products --header "Content-type: application/json" -d '{"id":"1002", "title": "AMD FD6300WMHKBOX FX-6300 6-Core Processor", "shortDesc": "AMD FX 6-Core Processor Unlocked Black Edition. AMDs next-generation architecture takes 8-core processing to a new level. Get up to 24% better frame rates in some of the most demanding games at stunning resolutions. Get faster audio encoding so you can enjoy your music sooner. Go up to 5.0 GHz with aggressive cooling solutions from AMD."}'
```

- We can also use curl to execute product searches:

```
$ curl -X GET http://localhost:9000/products/search --header "Content-type: application/json" -d '{"keyword":"processor"}'
{"total":2,"products":[{"id":"1001","title":"Intel Core i7-4790K Processor","shortDesc": "New Unlocked 4th Gen Intel Core Processors deliver 4 cores of up to 4 GHz base frequency providing blazing-fast computing performance for the most demanding users"}, {"id":"1002","title": "AMD FD6300WMHKBOX FX-6300 6-Core
```

Processor", "shortDesc": "AMD FX 6-Core Processor Unlocked Black Edition. AMDs next-generation architecture takes 8-core processing to a new level. Get up to 24% better frame rates in some of the most demanding games at stunning resolutions. Get faster audio encoding so you can enjoy your music sooner. Go up to 5.0 GHz with aggressive cooling solutions from AMD."}]}%

```
$ curl -X GET http://localhost:9000/products/search --header
"Content-type: application/json" -d '{"keyword": "amd"}'
{"total":1,"products":[{"id": "1002","title": "AMD FD6300WMHKBOX FX-6300
6-Core Processor", "shortDesc": "AMD FX 6-Core Processor Unlocked Black
Edition. AMDs next-generation architecture takes 8-core processing to a
new level. Get up to 24% better frame rates in some of the most
demanding games at stunning resolutions. Get faster audio encoding so
you can enjoy your music sooner. Go up to 5.0 GHz with aggressive
cooling solutions from AMD."}]}%
```

# How it works...

In this recipe, we created endpoints for product creation and product search utilizing ElasticSearch as the underlying search service. A prerequisite for this recipe is having access to an ElasticSearch instance, either local or remote, in our recipe, as well as having Docker installed in the local development machine:

```
$ docker -v
Docker version 1.3.3, build d344625
```

We used Docker to deploy our local ElasticSearch instance using the following command:

```
docker run -d -p 9200:9200 -p 9300:9300 dockerfile/elasticsearch
```

The preceding command instructs docker to run the elasticsearch container as a detached service and that ports 9200 and 9300 in the container should be accessible from the corresponding ports in the host.

We start by importing an open source play module, play2-elasticsearch, by declaring it in the build.sbt file:

```
resolvers += "Sonatype OSS Snapshots" at
"https://oss.sonatype.org/content/repositories/snapshots"

libraryDependencies ++= Seq(
 "io.iron.ironmq" % "ironmq" % "0.0.19",
 ("com.clever-age" % "play2-elasticsearch" % "1.4-SNAPSHOT")
 .exclude("com.typesafe.play", "play-functional_2.10")
 .exclude("com.typesafe.akka", "akka-actor_2.10")
 .exclude("com.typesafe.play", "play-json_2.10")
 .exclude("com.typesafe.play", "play_2.10")
 .exclude("com.typesafe.play", "play-iteratees_2.10")
 .exclude("com.typesafe.akka", "akka-slf4j_2.10")
 .exclude("org.scala-stm", "scala-stm_2.10")
 .exclude("com.typesafe.play", "play-datacommons_2.10")
)
```

We activate the plugin in conf/play.plugins and specify the configuration parameters in conf/application.conf:

```
// conf/play.plugins
9000:com.github.cleverage.elasticsearch.plugin.IndexPlugin
// conf/application.conf
elasticsearch.local=false
elasticsearch.client="192.168.59.103:9300"
elasticsearch.sniff=false
elasticsearch.index.name="test"
elasticsearch.index.settings="{ analysis: { analyzer: { my_analyzer: { type: \"custom\", tokenizer: \"standard\" } } } }"
elasticsearch.index.clazzs="models.*"
elasticsearch.index.show_request=true
elasticsearch.cluster.name=elasticsearch
```

We then created our product model, extending the play2-elasticsearch class, Indexable, and a search manager class, which extends the play2-elasticsearch IndexableManager

class. We wrote a helper method to execute a multifield query to search the title and description fields by keyword:

```
// Java
IndexQuery<Product> indexQuery = Product.find.query();
indexQuery.setBuilder(QueryBuilders.multiMatchQuery(keyword, "title",
"description"));

// Scala
val indexQuery = new IndexQuery[Product]()
 .withBuilder(QueryBuilders.multiMatchQuery(keyword, "title",
"description"))
```

Finally, in our controller class, Products, we invoked the appropriate product manager methods for the creation and search actions:

```
// Java
@BodyParser.Of(BodyParser.Json.class)
public static Result create() {
 ...
 product.index();
 ...
}

@BodyParser.Of(BodyParser.Json.class)
public static Result search() {
 ...
 Product.doSearch(search.getKeyword());
 ...
}

// Scala
def create = Action(BodyParsers.parse.json) { implicit request =>
 ...
 ProductManager.index(p)
 ...
}

def search = Action(BodyParsers.parse.json) { implicit request =>
 ...
 val results = ProductManager.doSearch(search.keyword)
 ...
}
```

Upon the web application's start up or reload, you will be able to see logging information regarding the initialization of the play2-elasticsearch plugin:

```
[info] application - Elasticsearch : Settings
{client.transport.sniff=false, cluster.name=elasticsearch}
[info] application - ElasticSearch : Starting in Client Mode
[info] application - ElasticSearch : Client - Host: 192.168.59.103 Port:
9300
[info] application - ElasticSearch : Started in Client Mode
[info] application - ElasticSearch : Plugin has started
[info] play - Application started (Dev)
```



# Implementing token authentication using JWT

In this recipe, we integrate the widely-used user authentication strategy, Token authentication, with authenticate requests to protected Play actions and endpoints. We will use the open source library, **nimbus-jose-jwt**, by Connect2Id to sign-in and verify JWT for successful user logins.

Subsequent requests to other protected endpoints and actions will now only require the JWT to be added to the request header using the authorization header. Signed JWTs will, however, have a prescribed expiration date and we will ensure that we validate this for each JWT signed request.

## Note

More information about Connect2id and nimbus-jose-jwt can be found here:

<http://connect2id.com/products/nimbus-jose-jwt>

More information about JWT is available here:

<http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html>

# How to do it...

For Java, we need to perform the following steps:

1. Run the foo\_java application with Hot-Reloading enabled:

```
activator "~run"
```

2. Add the nimbus-jose-jwt dependency in build.sbt:

```
libraryDependencies ++= Seq(
 "com.nimbusds" % "nimbus-jose-jwt" % "3.8.2"
)
```

3. Using activator, generate a new application secret, as follows:

```
$ activator play-generate-secret
[info] Loading project definition from
/Users/ginduc/Developer/workspace/bitbucket/Play2.0CookbookRecipes/Ch06
/foo_java/project
[info] Set current project to foo_java (in build
file:/Users/ginduc/Developer/workspace/bitbucket/Play2.0CookbookRecipes
/Ch06/foo_java/)
[info] Generated new secret:
DDqEUKPssmdHOL=U`XMANZAPYG4fUYA5QwGtK49[PmUh2kAH/IpHuHuLIuNgv_o_
[success] Total time: 0 s, completed 02 24, 15 11:44:42 AM
```

4. Add the required configuration parameters to conf/application.conf using the generated secret from the previous step as the value for jwt.sharedSecret. As for issuer and audience, based on the JWT spec, the issuer is the principal issuing the token and the audience is the intended recipient of the token:

```
jwt.sharedSecret =
"DDqEUKPssmdHOL=U`XMANZAPYG4fUYA5QwGtK49[PmUh2kAH/IpHuHuLIuNgv_o_"
 jwt.issuer=<YOUR_ISSUER>
 jwt.expiryInSecs=600
 jwt.audience=<YOUR_AUDIENCE>
```

5. Create the JWT Plugin class in app/plugins/JWTPlugin.java:

```
package plugins;

import com.nimbusds.jose.JWSAlgorithm;
import com.nimbusds.jose.JWSHeader;
import com.nimbusds.jose.crypto.MACSigner;
import com.nimbusds.jose.crypto.MACVerifier;
import com.nimbusds.jwt.JWTClaimsSet;
import com.nimbusds.jwt.SignedJWT;
import play.Application;
import play.Logger;
import play.Plugin;
import java.util.Date;

public class JWTPlugin extends Plugin {
 final private String tokenPrefix = "Bearer ";
```

```

private String issuer;
private String sharedSecret;
private Integer expiryTime;
private String audience;
private JWSHeader algorithm;
private MACSigner signer;
private MACVerifier verifier;

public JWTPlugin(Application app) {
 super();

 issuer = app.configuration().getString("jwt.issuer");
 sharedSecret =
app.configuration().getString("jwt.sharedSecret");
 expiryTime =
app.configuration().getInt("jwt.expiryInSecs");
 audience = app.configuration().getString("jwt.audience");
}

public void onStart() {
 algorithm = new JWSHeader(JwsAlgorithm.HS256);
 signer = new MACSigner(sharedSecret);
 verifier = new MACVerifier(sharedSecret);
}

public void onStop() {
 algorithm = null;
 signer = null;
 verifier = null;
}

public boolean verify(String token) {
 try {
 final JWTClaimsSet payload = decode(token);

 // Check expiration date
 if (!new Date().before(payload.getExpirationTime())) {
 Logger.error("Token expired: " +
payload.getExpirationTime());
 return false;
 }

 // Match Issuer
 if (!payload.getIssuer().equals(issuer)) {
 Logger.error("Issuer mismatch: " +
payload.getIssuer());
 return false;
 }

 // Match Audience
 if (payload.getAudience() != null &&
payload.getAudience().size() > 0) {
 if (!payload.getAudience().get(0).equals(audience))
{
 Logger.error("Audience mismatch: " +
payload.getAudience().get(0));
 }
 }
 }
}

```

```

 return false;
 }
} else {
 Logger.error("Audience is required");
 return false;
}

return true;
} catch(Exception e) {
 return false;
}
}

public JWTClaimsSet decode(String token) throws Exception {
 Logger.debug("Verifying: " +
token.substring(tokenPrefix.length()));
 SignedJWT signedJWT =
SignedJWT.parse(token.substring(tokenPrefix.length()));

 if (!signedJWT.verify(verifier)) {
 throw new IllegalArgumentException("Json Web Token
cannot be verified!");
 }

 return (JWTClaimsSet) signedJWT.getJWTClaimsSet();
}

public String sign(String userInfo) throws Exception {
 final JWTClaimsSet claimsSet = new JWTClaimsSet();
 claimsSet.setSubject(userInfo);
 claimsSet.setIssueTime(new Date());
 claimsSet.setIssuer(issuer);
 claimsSet.setAudience(audience);
 claimsSet.setExpirationTime(
 new Date(claimsSet.getIssueTime().getTime() +
(expiryTime * 1000))
);

 SignedJWT signedJWT = new SignedJWT(algorithm, claimsSet);
 signedJWT.sign(signer);
 return signedJWT.serialize();
}

public boolean enabled() {
 return true;
}
}

```

## 6. Initialize the JWT Plugin in conf/play.plugins:

```
10099:plugins.JWTPlugin
```

## 7. Create an action class inheriting from the Simple Action class that we will use to secure actions with JWT in app/controllers/JWTSigned.java:

```
package controllers;
```

```

import play.*;
import play.mvc.*;
import play.libs.*;
import play.libs.F.*;
import plugins.JWTPlugin;

public class JWTSigned extends play.mvc.Action.Simple {
 private static final String AUTHORIZATION = "Authorization";
 private static final String WWW_AUTHENTICATE = "WWW-
Authenticate";
 private static final String APP_REALM = "Protected Realm";
 private static final String AUTH_HEADER_PREFIX = "Bearer ";
 private static JWTPlugin jwt =
Play.application().plugin(JWTPlugin.class);

 public F.Promise<Result> call(Http.Context ctx) throws
Throwable {
 try {
 final String authHeader =
ctx.request().getHeader(AUTHORIZATION);

 if (authHeader != null &&
authHeader.startsWith(AUTH_HEADER_PREFIX)) {
 if (jwt.verify(authHeader)) {
 return delegate.call(ctx);
 }
 } else {
 return Promise.pure((Result) unauthorized());
 }
 } catch (Exception e) {
 Logger.error("Error during session authentication: " +
e);
 }

 ctx.response().setHeader(WWW_AUTHENTICATE, APP_REALM);
 return Promise.pure((Result) forbidden());
 }
}

```

8. Create our test actions for logging in and token signing, and another action to be secured with `JWTSigned` in `app/controllers/Application.java`:

```

package controllers;

import play.*;
import play.data.Form;
import play.mvc.*;

import plugins.JWTPlugin;
import views.html.*;

import java.util.HashMap;
import java.util.Map;

import static play.libs.Json.toJson;

```

```

public class Application extends Controller {
 private static JWTPlugin jwt =
Play.application().plugin(JWTPlugin.class);

 public static Result index() {
 return ok(index.render("Your new application is ready."));
 }

 @With(JWTSigned.class)
 public static Result adminOnly() {
 return ok("");
 }

 @BodyParser.Of(BodyParser.Json.class)
 public static Result auth() {
 try {
 Form<Login> form =
Form.form(Login.class).bindFromRequest();
 Login login = form.get();
 if (login.getUsername().equals("ned") &&
login.getPassword().equals("flanders")) {
 final String token = jwt.sign(login.getUsername());
 final Map<String, String> map = new HashMap<>();
 map.put("token", token);
 return ok(toJson(map));
 } else {
 return forbidden();
 }
 } catch (Exception e) {
 return internalServerError(e.getMessage());
 }
 }
}

```

9. We also need to create the `Login` model used during user authentication in `app/controllers/Login.java`:

```

package controllers;

public class Login {
 private String username;
 private String password;

 public Login() {}
 public Login(String username, String password) {
 this.username = username;
 this.password = password;
 }

 public String getUsername() {
 return username;
 }

 public void setUsername(String username) {

```

```

 this.username = username;
 }

 public String getPassword() {
 return password;
 }

 public void setPassword(String password) {
 this.password = password;
 }
}

```

10. Finally, we add the necessary entries to conf/routes for our new actions:

POST	/user/auth	controllers.Application.auth
GET	/admin	controllers.Application.adminOnly

11. Using curl, let's verify that the /admin route is gated and secured by JWTSigned:

```

$ curl -v http://localhost:9000/admin
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Basic realm="Protected Realm"
< Content-Length: 0
<
* Connection #0 to host localhost left intact

```

12. Next, let's sign in and make a note of the returned token in the response body:

```

$ curl -v http://localhost:9000/user/auth --header "Content-type: application/json" -d '{"username":"ned", "password":"flanders"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /user/auth HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 41
>
* upload completely sent off: 41 out of 41 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 181
<
* Connection #0 to host localhost left intact
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJleHAiOiJE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6yEMJV8KiRddZsZEcW5BAH2vw99Xc"}%

```

13. Finally, let's request the /admin route again, but this time, by adding the signed token using the authorization header, prefixed with Bearer:

```
$ curl -v http://localhost:9000/admin --header "Authorization:
Bearer
eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6Im
FwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6y
EMJV8KiRddZsZEcW5BAH2vw99Xc"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6Im
FwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6y
EMJV8KiRddZsZEcW5BAH2vw99Xc
>
< HTTP/1.1 200 OK
< Content-Type: text/plain; charset=utf-8
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

14. We can also verify that the JWTPlugin handles token expiration correctly by running the previous request again after the expiration is set in the token, and it should result in something like this:

```
$ curl -v http://localhost:9000/admin --header "Authorization:
Bearer
eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6Im
FwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6y
EMJV8KiRddZsZEcW5BAH2vw99Xc"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6Im
FwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6y
EMJV8KiRddZsZEcW5BAH2vw99Xc
>
< HTTP/1.1 403 Forbidden
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

For Scala, we need to perform the following steps:

1. Run the foo\_scala application with Hot-Reloading enabled:

```
activator "~run"
```

2. Add the nimbus-jose-jwt dependency to build.sbt:

```
libraryDependencies ++= Seq(
 "com.nimbusds" % "nimbus-jose-jwt" % "3.8.2"
)
```

3. Using activator, generate a new application secret as follows:

```
$ activator play-generate-secret
[info] Loading project definition from
/Users/ginduc/Developer/workspace/bitbucket/Play2.0CookbookRecipes/Ch06
/foo_scala/project
[info] Set current project to foo_scala (in build
file:/Users/ginduc/Developer/workspace/bitbucket/Play2.0CookbookRecipes
/Ch06/foo_scala/)
[info] Generated new secret:
LKQhArck[KCAFcEplWDeBSV^e@a1o6X>2SI_D3Q^X0h`eigla5ywm^k6E9z7Nx=p
[success] Total time: 0 s, completed 02 23, 15 10:32:56 PM
```

4. Add the required configuration parameters to conf/application.conf using the generated secret from the previous step as the value for jwt.sharedSecret. As for the issuer and audience, based on the JWT spec, the issuer is the principal issuing the token and audience is the intended recipient of the token:

```
jwt.sharedSecret =
"LKQhArck[KCAFcEplWDeBSV^e@a1o6X>2SI_D3Q^X0h`eigla5ywm^k6E9z7Nx=p"
 jwt.issuer=<YOUR_ISSUER>
 jwt.expiryInSecs=600
 jwt.audience=<YOUR_AUDIENCE>
```

5. Create the JWT Plugin class in app/plugins/JWTPlugin.scala:

```
package plugins

import java.util.Date
import com.nimbusds.jose.crypto.{MACVerifier, MACSigner}
import com.nimbusds.jose.{JWSAlgorithm, JWSHeader}
import com.nimbusds.jwt.{JWTClaimsSet, SignedJWT}
import play.api.{Logger, Play, Application, Plugin}
import play.api.Play.current

class JWTPlugin(app: Application) extends Plugin {
 val tokenPrefix = "Bearer "

 private val issuer =
 Play.application.configuration.getString("jwt.issuer").getOrElse("jwt")
 private val sharedSecret =
 Play.application.configuration.getString("jwt.sharedSecret")
 .getOrElse(throw new IllegalStateException("JWT Shared Secret is
required!"))
 private val expiryTime =
 Play.application.configuration.getInt("jwt.expiryInSecs").getOrElse(60
 * 60 * 24)
 private val audience =
```

```

Play.application.configuration.getString("jwt.audience").getOrElse("jwt")
)
 private val algorithm = new JWSHeader(JWSAlgorithm.HS256)

 private lazy val signer: MACSigner = new MACSigner(sharedSecret)
 private lazy val verifier: MACVerifier = new
MACVerifier(sharedSecret)

 override def onStart() = {

 signer
 verifier
 }

 override def onStop() = {
 Logger.info("Shutting down plugin")
 }

 def verify(token: String): Boolean = {
 val payload = decode(token)

 // Check expiration date
 if (!new Date().before(payload.getExpirationTime)) {
 Logger.error("Token expired: " + payload.getExpirationTime)
 return false
 }

 // Match Issuer
 if (!payload.getIssuer.equals(issuer)) {
 Logger.error("Issuer mismatch: " + payload.getIssuer)
 return false
 }

 // Match Audience
 if (payload.getAudience != null && payload.getAudience.size() >
0) {
 if (!payload.getAudience.get(0).equals(audience)) {
 Logger.error("Audience mismatch: " +
payload.getAudience.get(0))
 return false
 }
 } else {
 Logger.error("Audience is required")
 return false
 }
 return true
 }

 def decode(token: String) = {
 val signedJWT =
SignedJWT.parse(token.substring(tokenPrefix.length))

 if (!signedJWT.verify(verifier)) {
 throw new IllegalArgumentException("Json Web Token cannot be
verified!")
 }
 }
}

```

```

 signedJWT.getJWTClaimsSet
 }

 def sign(userInfo: String): String = {
 val claimsSet = new JWTClaimsSet()
 claimsSet.setSubject(userInfo)
 claimsSet.setIssueTime(new Date())
 claimsSet.setIssuer(issuer)
 claimsSet.setAudience(audience)
 claimsSet.setExpirationTime(
 new Date(claimsSet.getIssueTime.getTime + (expiryTime *
1000)))
 }

 val signedJWT = new SignedJWT(algorithm, claimsSet)
 signedJWT.sign(signer)
 signedJWT.serialize()
}

override def enabled = true
}

```

## 6. Initialize the JWTPlugin in conf/play.plugins:

```
10099:plugins.JWTPlugin
```

## 7. Create an ActionBuilder class that we will use to secure actions with JWT in app/controllers/JWTSigned.scala:

```

package controllers

import play.api.Play
import play.api.mvc.{Result, WrappedRequest, Request,
ActionBuilder}
import play.api.http.HeaderNames._
import play.api.mvc.Results._
import play.api.Play.current
import plugins.JWTPlugin
import scala.concurrent.Future

class JWTSignedRequest[A](val jwt: String, request: Request[A])
extends WrappedRequest[A](request)

object JWTSigned extends ActionBuilder[JWTSignedRequest] {
 private val jwt = Play.application.plugin[JWTPlugin].get

 def invokeBlock[A](req: Request[A], block: (JWTSignedRequest[A]) => Future[Result]) = {
 req.headers.get(AUTHORIZATION) map { token =>
 if (jwt.verify(token)) {
 block(new JWTSignedRequest(token, req))
 } else {
 Future.successful(Forbidden)
 }
 } getOrElse {
 }
}

```

```

 Future.successful(Unauthorized.withHeaders(WWW_AUTHENTICATE =
> """Basic realm="Protected Realm"""))
 }
}
}

```

8. Create test actions for logging in and token signing, and another action to be secured with JWTSigned in app/controllers/Application.scala:

```

case class Login(username: String, password: String)

private val jwt = Play.application.plugin[JWTPlugin].get
implicit private val productWrites = Json.writes[Login]
implicit private val productReads = Json.reads[Login]

def adminOnly = JWTSigned {
 Ok("")
}

def auth = Action(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Login]

 post.fold(
 errors => Unauthorized,
 u => {
 if (u.username.equals("ned") &&
u.password.equals("flanders")) {

 Ok(Json.obj("token" -> jwt.sign(u.username)))
 } else {
 Forbidden
 }
 }
)
}

```

9. Finally, we add the necessary entries to conf/routes for our new actions:

POST	/user/auth	controllers.Application.auth
GET	/admin	controllers.Application.adminOnly

10. Using curl, let's verify that the /admin route is gated and secured by JWTSigned:

```

$ curl -v http://localhost:9000/admin
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Basic realm="Protected Realm"
< Content-Length: 0
<
* Connection #0 to host localhost left intact

```

11. Next, let's sign in and make a note of the returned token in the response body:

```
$ curl -v http://localhost:9000/user/auth --header "Content-type: application/json" -d '{"username":"ned", "password":"flanders"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /user/auth HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 41
>
* upload completely sent off: 41 out of 41 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 181
<
* Connection #0 to host localhost left intact
{"token":"eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsawVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6yEMJV8KiRddZsZEcW5BAH2vw99Xc"}%
```

12. Finally, let's request the /admin route again, but this time, adding the signed token using the authorization header, prefixed with Bearer:

```
$ curl -v http://localhost:9000/admin --header "Authorization: Bearer eyJhbGciOiJIUzI1NiJ9eyJleHAiOjE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsawVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6yEMJV8KiRddZsZEcW5BAH2vw99Xc"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1

```

13. We can also verify that the JWTPlugin handles token expiration correctly by running the previous request again after the expiration set in the token, and should result in something like this:

```
$ curl -v http://localhost:9000/admin --header "Authorization: Bearer
```

```
eyJhbGciOiJIUzI1NiJ9eyJleHAiOiE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6yEMJV8KiRddZsZEcW5BAH2vw99Xc"
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9eyJleHAiOiE0MjQ3MDM0NjAsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsaWVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDM0MDB9.No2skaVfGeERDY6yEMJV8KiRddZsZEcW5BAH2vw99Xc
>
< HTTP/1.1 403 Forbidden
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

# How it works...

In this recipe, we implemented the signing and verification of JWT for the purpose of securing Play actions with token authentication.

We created a Play plugin, `JWTPlugin`, which will load the configurations from the `conf/application.conf` file and also contain method definitions for signing, decoding, and verifying JWTs using Connect2id's `nimbus-jose-jwt` library:

```
// Java
public JWTPlugin(Application app) {
 super();
 issuer = app.configuration().getString("jwt.issuer");
 sharedSecret = app.configuration().getString("jwt.sharedSecret");
 expiryTime = app.configuration().getInt("jwt.expiryInSecs");
 audience = app.configuration().getString("jwt.audience");
}

public void onStart() {
 algorithm = new JWSHeader(JWSAlgorithm.HS256);
 signer = new MACSigner(sharedSecret);
 verifier = new MACVerifier(sharedSecret);
}

// Scala
private val issuer =
Play.application.configuration.getString("jwt.issuer").getOrElse("jwt")
private val sharedSecret =
Play.application.configuration.getString("jwt.sharedSecret")
 .getOrElse(throw new IllegalStateException("JWT Shared Secret is
required!"))
private val expiryTime =
Play.application.configuration.getInt("jwt.expiryInSecs").getOrElse(60 * 60
* 24)
private val audience =
Play.application.configuration.getString("jwt.audience").getOrElse("jwt")
private val algorithm = new JWSHeader(JWSAlgorithm.HS256)
private var signer: MACSigner = null
private var verifier: MACVerifier = null

override def onStart() = {
 signer = new MACSigner(sharedSecret)
 verifier = new MACVerifier(sharedSecret)
}
```

In the preceding code, you will notice that we are utilizing HMAC using SHA-256 as the default hash algorithm.

For Java, add the following code:

```
// Java
algorithm = new JWSHeader(JWSAlgorithm.HS256);
```

For Scala, add the following code:

```
// Scala
private val algorithm = new JWSHeader(JWSAlgorithm.HS256)
```

For signing, we build the Claim Set, which is the standard set of token metadata according to the JWT spec: you can refer to the following link:

<http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html#Claims>

With the appropriate parameter values and adding the user information as the claim subject, we then sign and serialize it to a String:

```
// Java
public String sign(String userInfo) throws Exception {
 final JWTClaimsSet claimsSet = new JWTClaimsSet();
 claimsSet.setSubject(userInfo);
 claimsSet.setIssueTime(new Date());
 claimsSet.setIssuer(issuer);
 claimsSet.setAudience(audience);
 claimsSet.setExpirationTime(
 new Date(claimsSet.getIssueTime().getTime() + (expiryTime *
1000)))
);

 SignedJWT signedJWT = new SignedJWT(algorithm, claimsSet);
 signedJWT.sign(signer);
 return signedJWT.serialize();
}

// Scala
def sign(userInfo: String): String = {
 val claimsSet = new JWTClaimsSet()
 claimsSet.setSubject(userInfo)
 claimsSet.setIssueTime(new Date)
 claimsSet.setIssuer(issuer)
 claimsSet.setAudience(audience)
 claimsSet.setExpirationTime(
 new Date(claimsSet.getIssueTime.getTime + (expiryTime * 1000)))
}

val signedJWT = new SignedJWT(algorithm, claimsSet)
signedJWT.sign(signer)
signedJWT.serialize()
```

To verify, we get the passed token decoded and verified and then proceed to verifying parts of the Claim Set. It returns the Boolean value `true` only when the token has passed all verification tests:

```
// Java
public boolean verify(String token) {
 try {
 final JWTClaimsSet payload = decode(token);

 // Check expiration date
 if (!new Date().before(payload.getExpirationTime())) {
 Logger.error("Token expired: " +
```

```

payload.getExpirationTime());
 return false;
}

// Match Issuer
if (!payload.getIssuer().equals(issuer)) {
 Logger.error("Issuer mismatch: " + payload.getIssuer());
 return false;
}

// Match Audience
if (payload.getAudience() != null &&
payload.getAudience().size() > 0) {
 if (!payload.getAudience().get(0).equals(audience)) {
 Logger.error("Audience mismatch: " +
payload.getAudience().get(0));
 return false;
 }
} else {
 Logger.error("Audience is required");
 return false;
}

return true;
} catch(Exception e) {
 return false;
}
}

// Scala
def verify(token: String): Boolean = {
 val payload = decode(token)

 // Check expiration date
 if (!new Date().before(payload.getExpirationTime)) {
 Logger.error("Token expired: " + payload.getExpirationTime)
 return false
 }

 // Match Issuer
 if (!payload.getIssuer.equals(issuer)) {
 Logger.error("Issuer mismatch: " + payload.getIssuer)
 return false
 }

 // Match Audience
 if (payload.getAudience != null && payload.getAudience.size() > 0) {
 if (!payload.getAudience.get(0).equals(audience)) {
 Logger.error("Audience mismatch: " + payload.getAudience.get(0))
 return false
 }
 } else {
 Logger.error("Audience is required")
 return false
 }
}

```

```
 return true
 }
```

We then created a Simple Action / Action Builder class that will do the actual JWT verification:

```
// Java
public F.Promise<Result> call(HttpContext ctx) throws Throwable {
 try {
 final String authHeader =
ctx.request().getHeader(AUTHORIZATION);

 if (authHeader != null &&
authHeader.startsWith(AUTH_HEADER_PREFIX)) {
 if (jwt.verify(authHeader)) {
 return delegate.call(ctx);
 }
 }
 } catch (Exception e) {
 Logger.error("Error during session authentication: " + e);
 }

 ctx.response().setHeader(WWW_AUTHENTICATE, APP_REALM);
 return Promise.pure((Result) forbidden());
}

// Scala
def invokeBlock[A](req: Request[A], block: (JWTSignedRequest[A]) =>
Future[Result]) = {
 req.headers.get(AUTHORIZATION) map { token =>
 if (jwt.verify(token)) {
 block(new JWTSignedRequest(token, req))
 } else {
 Future.successful(Forbidden)
 }
 } getOrElse {
 Future.successful(Unauthorized.withHeaders(WWW_AUTHENTICATE ->
"""Basic realm="Protected Realm"""))
 }
}
```

The preceding code will only invoke the next request block if the token passes the verification test. It will return an **Http Status Forbidden** error for unsuccessful verifications and an **Http Status Unauthorized** error for requests that do not have the authorization header set.

We are now able to secure Play controller actions using the `JWTSigned ActionBuilder` class:

```
// Java
@With(JWTSigned.class)
public static Result adminOnly() {
 return ok("");
}

// Scala
```

```
def adminOnly = JWTSigned {
 Ok("")
}
```

Finally, what we have is a Play action returning serialized, signed JWTs, and an action utilizing the `JWTSigned ActionBuilder` class that secures the action from unauthenticated and unauthorized requests:

### # Signing

```
$ curl http://localhost:9000/user/auth --header "Content-type: application/json" -d '{"username":"ned", "password":"flanders"}'
{"token":"eyJhbGciOiJIUzI1NiJ9eyJleHAiOiE0MjQ3MDUzOTgsInN1YiI6Im5lZCIsImF1ZCI6ImFwaWNsawVudHMiLCJpc3MiOiJwMmMiLCJpYXQiOjE0MjQ3MDUzMzh9.uE5GNQv2uXQh29sHhy_Jbg9omDhQMrnW1tjqFBrUwSs"}%
```

### # Verifying

```
$ curl -v http://localhost:9000/admin
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /admin HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Basic realm="Protected Realm"
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```



# Chapter 7. Deploying Play 2 Web Apps

In this chapter, we will cover the following recipes:

- Deploying a Play application on Heroku
- Deploying a Play application on AWS Elastic Beanstalk
- Deploying a Play application with CoreOS and Docker
- Deploying a Play application with Dokku
- Deploying a Play application with Nginx

# Introduction

In this chapter, we will explore various deployment options for a Play 2.0 web application. In the advent of numerous cloud-based services, such as **Infrastructure as a Service (IaaS)** and **Platform as a Service (PaaS)** service offerings, we are given multiple options on how we can deploy our Play 2.0 into different environments, be it for development, testing, integration, or production.

We explore this using popular developer services such as Heroku and Amazon web services. We will also look into using popular developer tools such as **Docker** and **Dokku** to deploy Play 2 web applications.

We will also use two different operating systems as our base operating system in **CentOS** and **CoreOS**.

In this chapter, we aim to show the contrast in the efforts required to deploy web apps with cloud services and manually deploying web apps with a raw virtual machine. We will also look at how modern tools allow developers to be more efficient by focusing more time on development instead of doing what to the infrastructure.



# Deploying a Play application on Heroku

In this recipe, we will deploy a Play webapp on **Heroku**, specifically, the Heroku Cedar stack that supports the Play Framework. Deploying on Heroku requires some knowledge in Git, the popular source control management software.

# How to do it...

1. First, sign up for a Heroku account by visiting the following link:  
<https://signup.heroku.com/>
2. Install the Heroku CLI tools (more information at  
<https://devcenter.heroku.com/articles/heroku-command>). Once installed, you should now be able to login to Heroku using the CLI tools as follows:

```
$ heroku login
Enter your Heroku credentials.
Email: <YOUR_HEROKU_USER>
Password (typing will be hidden):
Authentication successful.
```

3. Create our Play webapp using the activator template, computer-database-scala:

```
activator new play2-heroku computer-database-scala
```

4. Once our webapp has been generated, change into the project root of play2-heroku and initialize Git on the project:

```
cd play2-heroku
git init
```

5. Create a new heroku app. The default app name will be assigned by Heroku, for this recipe, shielded-dusk-8715:

```
$ heroku create --stack cedar-14
Creating shielded-dusk-8715... done, stack is cedar-14
https://shielded-dusk-8715.herokuapp.com/
https://git.heroku.com/shielded-dusk-8715.git
Git remote heroku added
```

6. Add Heroku-specific configs in the project root directory. Optionally, you can create the file, system.properties, with the following contents, to specify that we will require JDK version 8:

```
java.runtime.version=1.8
```

7. Create the Heroku file, Procfile, which is the standard Heroku text-based file to declare your apps and to run commands and other environment variables, in the project root with the following contents:

```
web: target/universal/stage/bin/play2-heroku -Dhttp.port=${PORT} -
DapplyEvolutions.default=true
```

8. Verify the location of our new Heroku git remote with the following command:

```
$ git remote show heroku
* remote heroku
 Fetch URL: https://git.heroku.com/shielded-dusk-8715.git
 Push URL: https://git.heroku.com/shielded-dusk-8715.git
 HEAD branch: unknown
 Remote branch:
```

```
 master tracked
 Local branch configured for 'git pull':
 master merges with remote master
 Local ref configured for 'git push':
 master pushes to master (up to date)
```

9. Add and commit our webapp boilerplate code to git:

```
git add --all && git commit -am "Initial commit"
```

10. Finally, let's deploy our webapp to Heroku by pushing our local repository to the git Heroku origin:

```
$ git push heroku master
Counting objects: 40, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (35/35), done.
Writing objects: 100% (40/40), 1.01 MiB | 0 bytes/s, done.
Total 40 (delta 1), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Play 2.x - Scala app detected
remote: -----> Installing OpenJDK 1.7... done
remote: -----> Priming Ivy cache (Scala-2.11, Play-2.3)... done
remote: -----> Running: sbt compile stage
..
..
remote: Default types for Play 2.x - Scala -> web
remote:
remote: -----> Compressing... done, 95.0MB
remote: -----> Warning: This app's git repository is large.
remote: Large repositories can cause problems.
remote: See: https://devcenter.heroku.com/articles/git#repo-size
remote: -----> Launching... done, v6
remote: https://shielded-dusk-8715.herokuapp.com/ deployed
to Heroku
remote:
remote: Verifying deploy.... done.
To https://git.heroku.com/shielded-dusk-8715.git
```

11. Using a web browser, we are now able to access the computer-database-scala webapp deployed in Heroku:

Play sample application — Computer database

**574 computers found**

Filter by computer name... Filter by name Add a new computer

Computer name	Introduced	Discontinued	Company
ACE	—	—	—
Acer Extensa	—	—	—
Acer Extensa 5220	—	—	—
Acer Iconia	—	—	—
Acorn Archimedes	—	—	Acorn computer
Acorn Electron	—	—	—
Acorn System 2	—	—	—
Alex eReader	—	—	—
Altair 8800	19 Dec 1974	—	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	—	Amiga Corporation

← Previous Displaying 1 to 10 of 574 Next →

12. The actual Heroku URL will differ; ensure that you replace your Heroku-generated app name on the URL, but for this recipe it should be deployed at:

<https://shielded-dusk-8715.herokuapp.com>

# How it works...

In this recipe, we deployed a Play 2.0 web application in the popular PaaS, Heroku. Deploying to Heroku requires very minimal customization and configuration, which makes it very easy for developers to quickly deploy the Play 2.0 webapps in a solid platform such as Heroku's.

For this recipe, we used the activator template `computer-database-scala` and used this as our sample web application, which we in turn, deployed to Heroku's Cedar stack. After setting up our web application with the template boilerplate code, we added two files that serve as configuration files for Heroku:

## Procfile

In **Procfile**, we specify the entry point for the web application, which Heroku uses to initialize and run the web app, in this recipe, `bin/play2-heroku`. We also specify here, the port number to be used by the web app, which is dictated by the Heroku runtime, specified in the environment variable,  `${PORT}`:

```
web: target/universal/stage/bin/play2-heroku -Dhttp.port=${PORT} -DapplyEvolutions.default=true
```

Finally, we set a jvm property used by Play to determine whether to execute database evolution scripts or not.

```
-DapplyEvolutions.default=true
```

## system.properties

```
java.runtime.version=1.8
```

As an optional step, in `system.properties`, we simply specify the version of the JDK to utilize for our Play web application.

## There's more...

Refer to the official Git documentation for more information about Git and post-commit hooks:

- <http://git-scm.com/documentation/>
- <http://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks>



# Deploying a Play application on AWS Elastic Beanstalk

In this recipe, we will deploy a Play 2 web application on Amazon web services' Elastic Beanstalk. **Elastic Beanstalk** is Amazon's Platform-as-a-Service offering and allows developers to deploy web applications with the same ease as other PAAS offerings, such as Heroku.

We will package our Play 2 web application as a Docker image and upload it to Elastic Beanstalk as a ZIP package. We will mostly be interacting with the AWS management console using a web browser.

Ensure that you have signed up for an account with AWS here:

<http://aws.amazon.com>

# How to do it...

1. Create our Play 2 web application using the activator template, computer-database-scala:

```
activator new play2-deploy-72 computer-database-scala
```

2. Edit conf/application.conf to enable automatic database evolutions:

```
applyEvolutions.default=true
```

3. Edit build.sbt to specify Docker settings for the web app, making note of the maintainer and dockerExposedPorts settings, using your own Docker Hub username and web app port, respectively:

```
import NativePackagerKeys._
import com.typesafe.sbt.SbtNativePackager._

name := """play2-deploy-72"""

version := "0.0.1-SNAPSHOT"

scalaVersion := "2.11.4"

maintainer := "ginduc"

dockerExposedPorts in Docker := Seq(9000)

libraryDependencies ++= Seq(
 jdbc,
 anorm,
 "org.webjars" % "jquery" % "2.1.1",
 "org.webjars" % "bootstrap" % "3.3.1"
)

lazy val root = (project in file(".")).enablePlugins(PlayScala)
```

4. Create a new JSON config file specific to Elastic Beanstalk and Docker apps in the project root, Dockerrun.aws.json.template, with the following contents:

```
{
 "AwSEBDockerrunVersion": "1",
 "Ports": [
 {
 "ContainerPort": "9000"
 }
]
}
```

5. Generate a Docker image using activator:

```
$ activator clean docker:stage
```

6. Copy the Dockerrun.aws.json.template file to the Docker root in target/docker:

```
$ cp Dockerrun.aws.json.template target/docker/Dockerrun.aws.json
```

7. Zip the target/docker directory in preparation for uploading to Elastic Beanstalk. Make a note of the location of the output ZIP file:

```
$ cd target/docker && zip -r ../play2-deploy-72.zip .
```

8. Once the docker package is prepped up, head to the AWS management console and create an Elastic Beanstalk (EB) application, ensuring that we pick Docker as the platform:

The screenshot shows the AWS Elastic Beanstalk Management Console. On the left, there is a sidebar with options: Dashboard, Configuration, Logs, Monitoring (which is selected), Alarms, and Events. The main area displays monitoring data for an environment named 'MobileBackend'. Key metrics shown include Average Latency (53.6 ms), Sum Requests (148K), CPU Utilization (65%), Max Network In (354KB), and Maximum DiskReadLatency (12KB). Below this is a 'Monitoring' section with two line graphs: 'Average Latency (ms)' and 'Sum Requests by count'. A 'Launch Now' button is visible at the bottom right of the main content area.

9. Once the initial EB application has been initialized, we will need to navigate to the **Configuration** tab and click on the cogwheel icon next to **instances** to edit the environment and select a different EC2 instance type, selecting **t2.medium** instead of **t2.micro** (selecting an instance type other than **t2.micro** means that you will be using a non-free EC2 instance):

My First Elastic Beanstalk Application > Default-Environment ([Default-Environment-rweywrtpc.elasticbeanstalk.com](https://console.aws.amazon.com/elasticbeanstalk/home?region=us-east-1#/environment/configuration?applicationName=Default-Environment&environmentName=rweywrtpc.elasticbeanstalk.com))

Dashboard Server

Configuration The following settings let you configure the environment servers. [Learn more.](#)

Logs Instance type: t2.medium Determines the processing power of the servers in your environment.

Monitoring EC2 security groups: awseb-e-mpcfyt5jfs-stack-AWSEBSecurity The names of the security groups (comma separated) that define firewall access to the launched EC2 instances.

Alarms

Events

Tags

Instance profile: aws-elasticbeanstalk-ec2-role Refresh The instance profile grants your environment specific permissions under your AWS account. [Learn more.](#)

Monitoring interval: 5 minute The time interval between when metrics are reported from the EC2 instances.

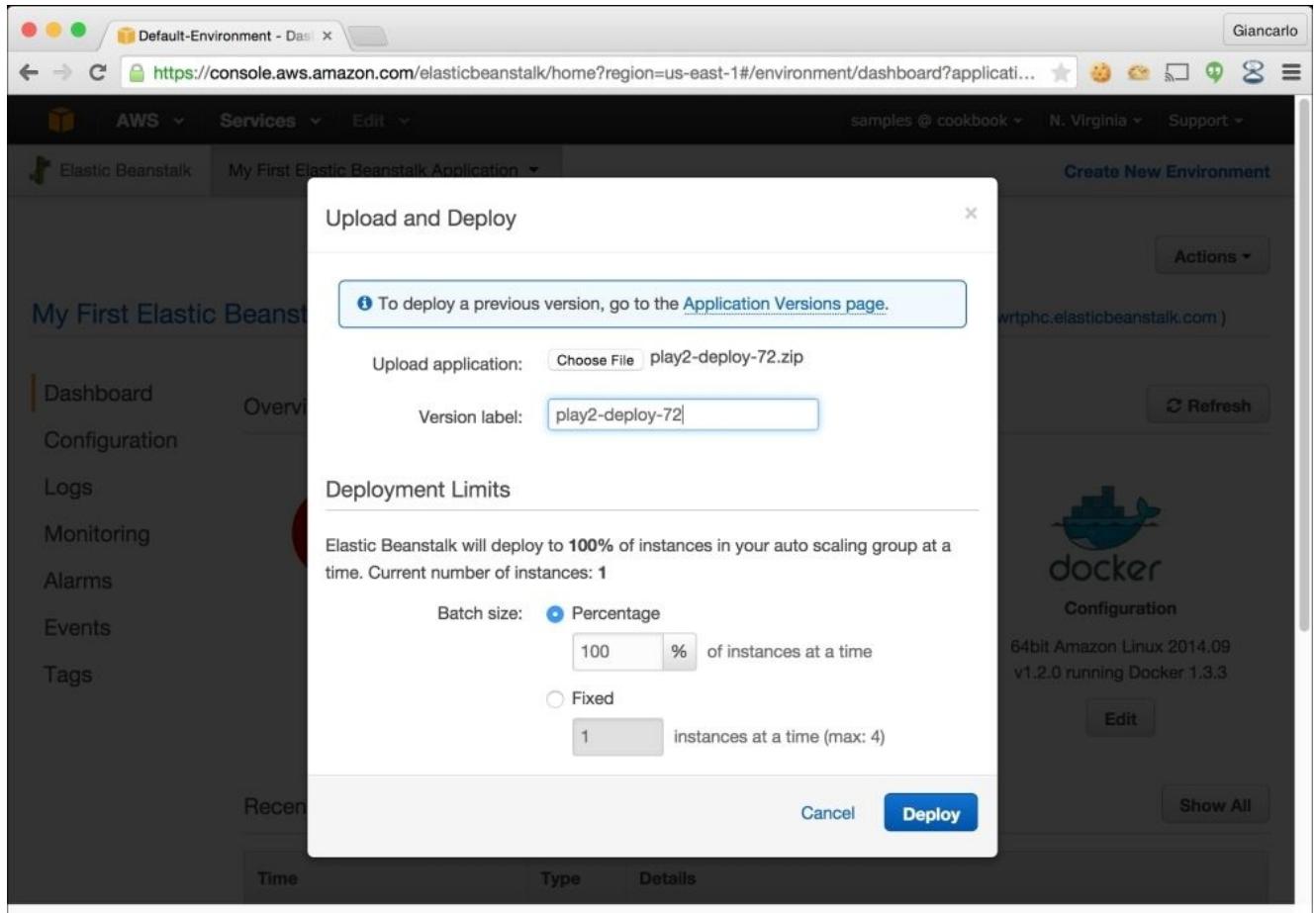
Custom AMI ID: ami-a66723ce The AMI to use for launched instances.

Root Volume (Boot Device)

The following settings let you configure the root volume for the auto scaling launched EC2 instances. [Learn more.](#)

Root volume type: (Container default) Refresh Determines the type of storage volume to attach to instances.

- Once this environment change has been applied, navigate back to the dashboard page so we can proceed to uploading the docker package that we built earlier:



11. Click on **Deploy**.
12. Specify the version label and deployment limits; for our recipe, we will keep the defaults. It will take a few minutes to upload and initialize but once the Play 2 web app has been deployed, we can access it from a web browser using the link next to **Default Environment** under the dashboard tab:

Computers database

Giancarlo

default-environment-rweywrphc.elasticbeanstalk.com/computers

## Play sample application — Computer database

### 574 computers found

Filter by computer name... Filter by name Add a new computer

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

← Previous Displaying 1 to 10 of 574 Next →

# How it works...

In this recipe, we deployed a Play 2 web application to AWS Elastic Beanstalk. We packaged our Play 2 web app as a Docker image and zipped it before uploading on the AWS Management Console using a web browser. More information regarding sbt and Docker integration can be found here:

<https://github.com/sbt/sbt-native-packager>

We made very minimal modifications to the `build.sbt` to specify the Docker-specific settings required during the build phase:

```
import NativePackagerKeys._
import com.typesafe.sbt.SbtNativePackager._
/* ... */

maintainer := "ginduc"

dockerExposedPorts in Docker := Seq(9000)
```

We imported the necessary packages that contain the sbt and Docker integrations, and placed them at the top of the file, `build.sbt`. We then specified the maintainer and the port number to be used by the docker container, in this recipe, port number 9000.

We added an Elastic Beanstalk-Docker configuration file in the project root, `Dockerrun.aws.json`, which specifies the run version and docker container port that we will be utilizing, in JSON format:

```
{
 "AWSEBDockerrunVersion": "1",
 "Ports": [
 {
 "ContainerPort": "9000"
 }
]
}
```

A final deployment configuration step worth noting is modifying the EC2 instance type set by default by Elastic Beanstalk from a `t2.micro` to a `t2.medium` instance type. A `t2.medium` instance is a non-free EC2 instance, which means you will incur costs for using this EC2 instance type. This is necessary as we may encounter JVM issues when using an instance type with less than 2 GB of RAM. Applying this environment change will require Elastic Beanstalk to reinitialize the environment and should take a few minutes to complete. We can then proceed to uploading and deploying our prepackaged Docker image using the AWS Management Console.

Screenshot of the AWS Elastic Beanstalk Default Environment dashboard.

The dashboard shows the following details:

- Health:** Green
- Running Version:** Sample Application
- Upload and Deploy** button (highlighted with a red box)
- Configuration:** Docker (64bit Amazon Linux 2014.09 v1.2.0 running Docker 1.3.3)
- Recent Events:**

Time	Type	Details
2015-03-24 12:27:50 UTC+0800	INFO	Adding instance 'i-692ae2be' to your environment.
2015-03-24 12:27:30 UTC+0800	INFO	Successfully launched environment: Default-Environment
2015-03-24 12:27:30 UTC+0800	INFO	Application available at Default-Environment-46qu2kp9vi.elasticbeanstalk.com.

Once deployment on the AWS Management Console is completed, we are able to access the computer-database-scala web application from its Elastic Beanstalk URL by using a web browser.

## **There's more...**

Refer to the online documentation for more information about AWS Elastic Beanstalk:

<http://aws.amazon.com/elasticbeanstalk/developer-resources/>



# Deploying a Play application on CoreOS and Docker

In this recipe, we will deploy a Play 2 web application using CoreOS and Docker. **CoreOS** is a new, lightweight operating system ideal for modern application stacks. Together with Docker, a software container management system, this forms a formidable deployment environment for Play 2 web applications that boasts of simplified deployments, isolation of processes, ease in scalability, and so on.

For this recipe, we will utilize the popular cloud IaaS, Digital Ocean. Ensure that you sign up for an account here:

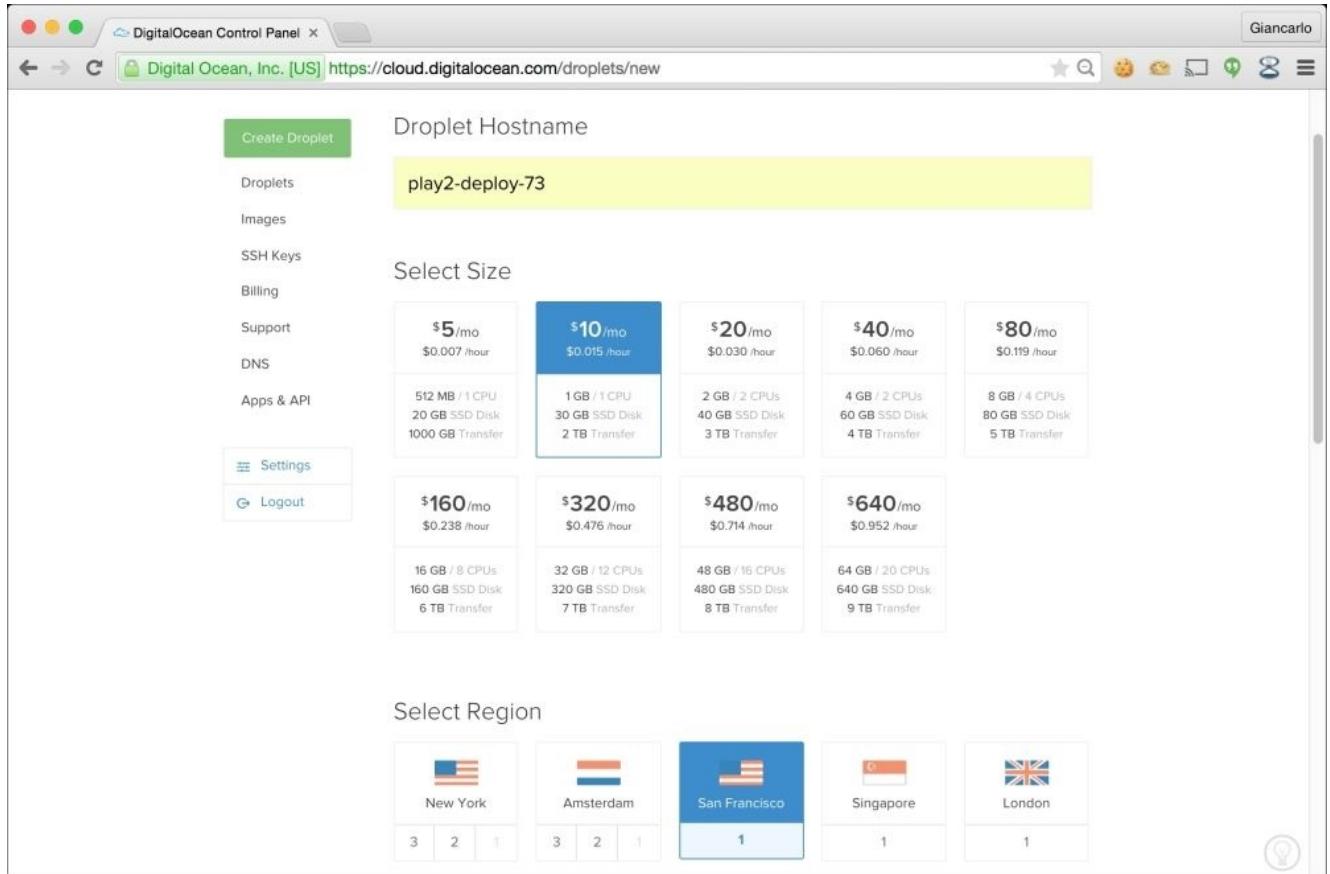
<https://cloud.digitalocean.com/registrations/new>

This recipe also requires Docker to be installed in the developer's machine. Refer to the official Docker documentation regarding installation:

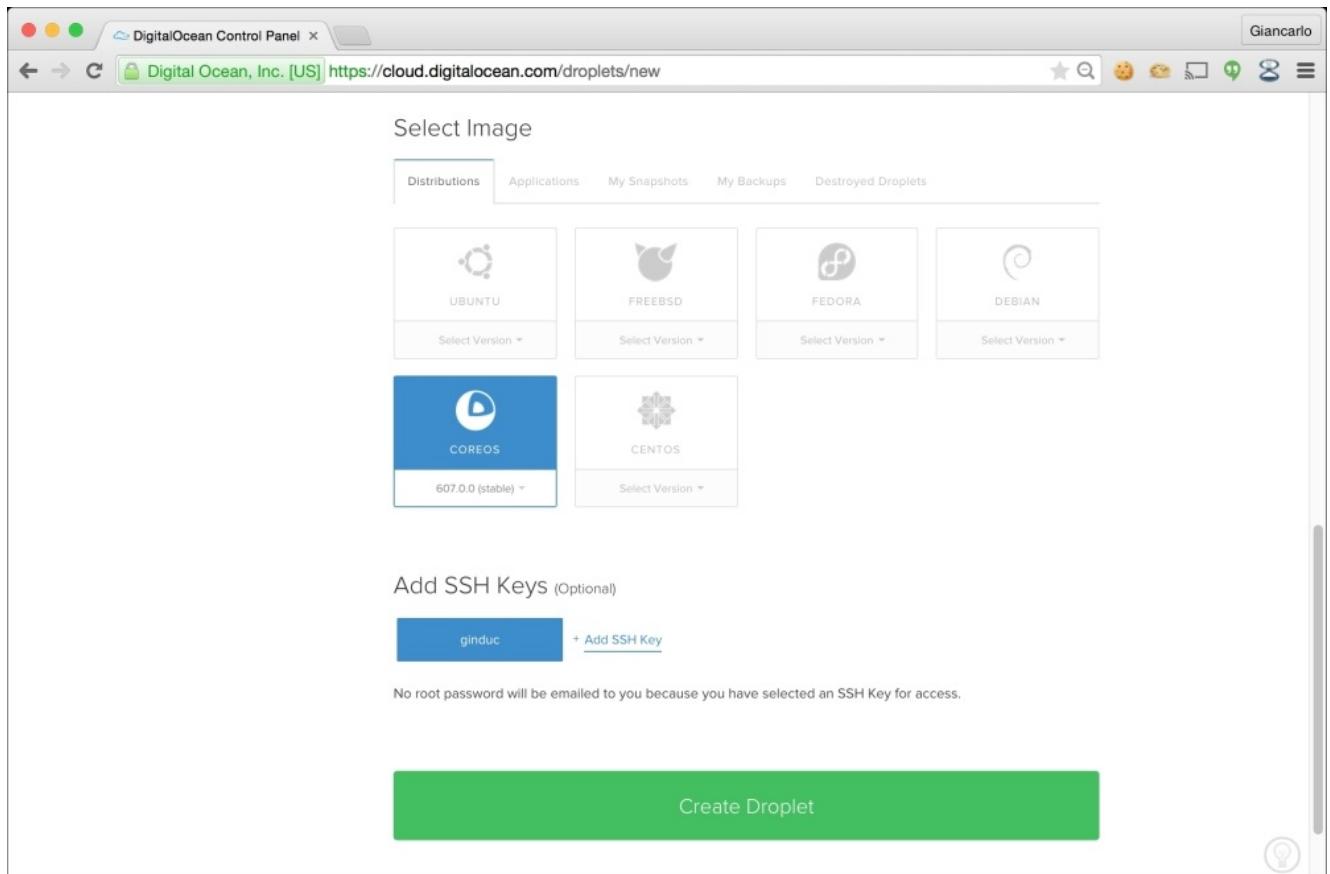
<https://docs.docker.com/installation/>

# How to do it...

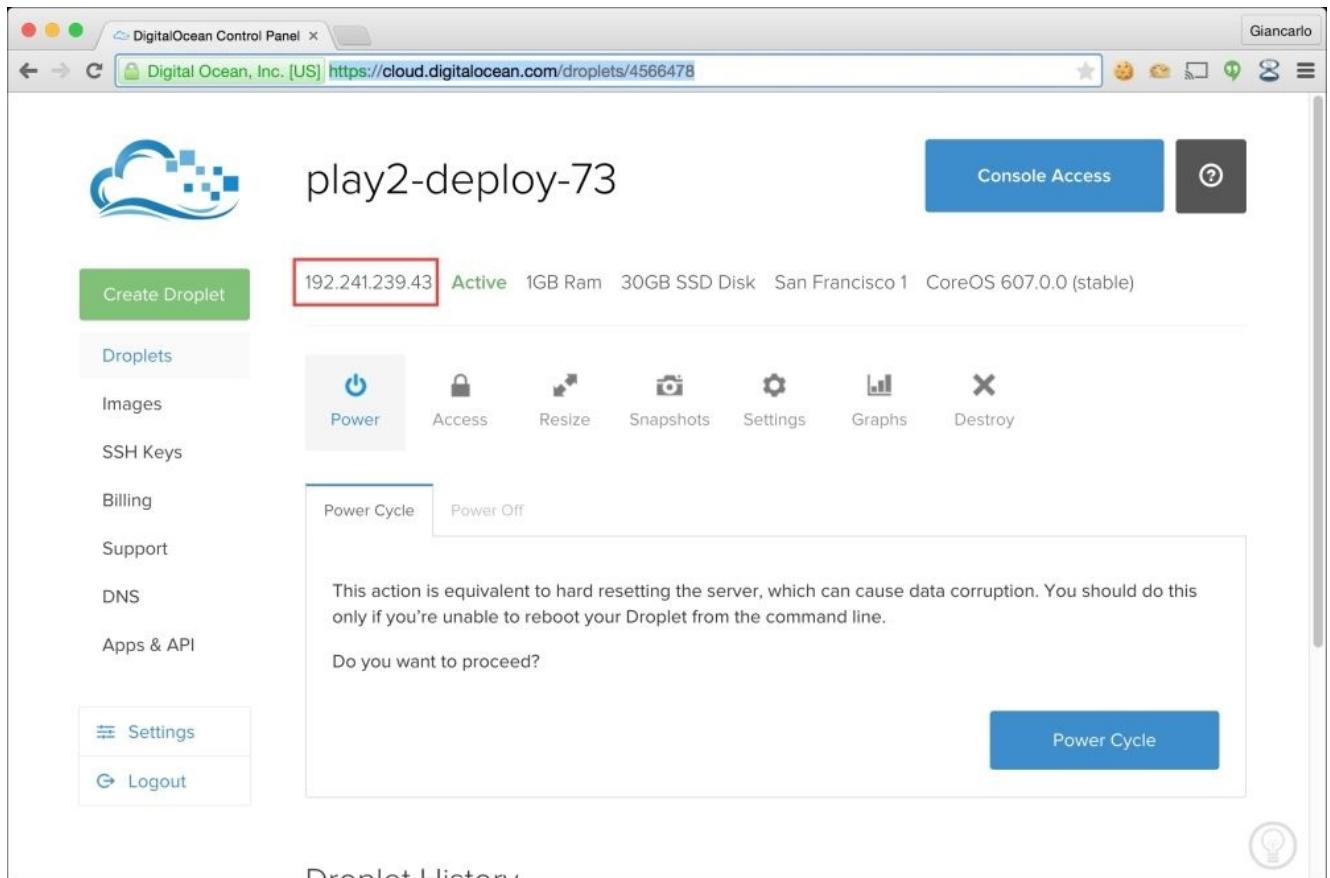
1. Create a new Digital Ocean droplet using CoreOS as the base operating system. Ensure that you use a droplet with at least 1 GB of RAM for the recipe to work. note that Digital Ocean does not have a free tier and are all paid instances:



2. Ensure that you select the appropriate droplet region:



3. Select **CoreOS 607.0.0** and specify a **SSH key** to use. Visit the following link if you need more information regarding SSH key generation:  
<https://www.digitalocean.com/community/tutorials/how-to-set-up-ssh-keys--2>:
4. Once the Droplet is created, make a special note of the Droplet's IP address which we will use to log in to the Droplet:



5. Next, create a [Docker.com](https://hub.docker.com/account/signup/) account at <https://hub.docker.com/account/signup/>
6. Create a new repository to house the **play2-deploy-73** docker image that we will use for deployment:

The screenshot shows a web browser displaying a Docker repository page. The URL in the address bar is <https://registry.hub.docker.com/u/ginduc/play2-deploy-73/>. The page header includes the Docker logo and links for 'What is Docker?', 'Use Cases', 'Try It!', 'Browse', 'Install & Docs', 'Log In', and 'Sign Up'. A message 'Updated an hour ago' is displayed above the repository name 'ginduc / play2-deploy-73'. Below the repository name, it says 'No description set' and shows statistics: 0 stars, 0 comments, and 0 forks. To the right, there are buttons for 'Pull this repository' and 'docker pull ginduc/play2-deploy-73'. The main content area is divided into three tabs: 'Information' (selected), 'Tags', and 'Properties'. The 'Information' tab contains the message 'No description set'. The 'Tags' tab shows a single tag entry: '2015-03-23 23:13:51' by user 'ginduc'. The 'Properties' tab shows 'Webhooks'. On the left side, there is a 'Comments' section with a note 'Please login to comment.'

7. Create a new Play 2 webapp using the activator template, computer-database-scala, and change into the project root:

```
activator new play2-deploy-73 computer-database-scala && cd play2-deploy-73
```

8. Edit conf/application.conf to enable automatic database evolutions:

```
applyEvolutions.default=true
```

9. Edit build.sbt to specify Docker settings for the web app:

```
import NativePackagerKeys._
import com.typesafe.sbt.SbtNativePackager._

name := """play2-deploy-73"""

version := "0.0.1-SNAPSHOT"

scalaVersion := "2.11.4"

maintainer := "<YOUR_DOCKERHUB_USERNAME HERE>"

dockerExposedPorts in Docker := Seq(9000)

dockerRepository := Some("YOUR_DOCKERHUB_USERNAME HERE ")
```

```

libraryDependencies ++= Seq(
 jdbc,
 anorm,
 "org.webjars" % "jquery" % "2.1.1",
 "org.webjars" % "bootstrap" % "3.3.1"
)

lazy val root = (project in file(".")).enablePlugins(PlayScala)

```

10. Next, we build the Docker image and publish it to Docker Hub:

```

$ activator clean docker:stage docker:publish
..
[info] Step 0 : FROM dockerfile/java
[info] --> 68987d7b6df0
[info] Step 1 : MAINTAINER ginduc
[info] --> Using cache
[info] --> 9f856752af9e
[info] Step 2 : EXPOSE 9000
[info] --> Using cache
[info] --> 834eb5a7daec
[info] Step 3 : ADD files /
[info] --> c3c67f0db512
[info] Removing intermediate container 3b8d9c18545e
[info] Step 4 : WORKDIR /opt/docker
[info] --> Running in 1b150e98f4db
[info] --> ae6716cd4643
[info] Removing intermediate container 1b150e98f4db
[info] Step 5 : RUN chown -R daemon .
[info] --> Running in 9299421b321e
[info] --> 8e15664b6012
[info] Removing intermediate container 9299421b321e
[info] Step 6 : USER daemon
[info] --> Running in ea44f3cc8e11
[info] --> 5fd0c8a22cc7
[info] Removing intermediate container ea44f3cc8e11
[info] Step 7 : ENTRYPOINT bin/play2-deploy-73
[info] --> Running in 7905c6e2d155
[info] --> 47fded583dd7
[info] Removing intermediate container 7905c6e2d155
[info] Step 8 : CMD
[info] --> Running in b807e6360631
[info] --> c3e1999cfbfd
[info] Removing intermediate container b807e6360631
[info] Successfully built c3e1999cfbfd
[info] Built image ginduc/play2-deploy-73:0.0.2-SNAPSHOT
[info] The push refers to a repository [ginduc/play2-deploy-73]
(len: 1)
[info] Sending image list
[info] Pushing repository ginduc/play2-deploy-73 (1 tags)
[info] Pushing tag for rev [c3e1999cfbfd] on {https://cdn-registry-1.docker.io/v1/repositories/ginduc/play2-deploy-73/tags/0.0.2-SNAPSHOT}
[info] Published image ginduc/play2-deploy-73:0.0.2-SNAPSHOT

```

11. Once the Docker image has been published, log in to the Digital Ocean droplet using SSH to pull the uploaded docker image. You will need to use the core user for your

CoreOS Droplet:

```
ssh core@<DROPLET_IP_ADDRESS_HERE>
core@play2-deploy-73 ~ $ docker pull <YOUR_DOCKERHUB_USERNAME
HERE>/play2-deploy-73:0.0.1-SNAPSHOT
Pulling repository ginduc/play2-deploy-73
6045dfea237d: Download complete
511136ea3c5a: Download complete
f3c84ac3a053: Download complete
a1a958a24818: Download complete
709d157e1738: Download complete
d68e2305f8ed: Download complete
b87155bee962: Download complete
2097f889870b: Download complete
5d2fb9a140e9: Download complete
c5bdb4623fac: Download complete
68987d7b6df0: Download complete
9f856752af9e: Download complete
834eb5a7daec: Download complete
fae5f7dab7bb: Download complete
ee5ccc9a9477: Download complete
74b51b6dcfe7: Download complete
41791a2546ab: Download complete
8096c6beaae7: Download complete
Status: Downloaded newer image for <YOUR_DOCKERHUB_USERNAME
HERE>/play2-deploy-73:0.0.2-SNAPSHOT
```

12. We are now ready to run our Docker image using the following docker command:

```
core@play2-deploy-73 ~ $ docker run -p 9000:9000
<YOUR_DOCKERHUB_USERNAME_HERE>/play2-deploy-73:0.0.1-SNAPSHOT
```

13. Using a web browser, access the computer-database webapp using the IP address we made note of in an earlier step of this recipe (<http://192.241.239.43:9000/computers>):

Computers database

192.241.239.43:9000/computers

## Play sample application — Computer database

### 574 computers found

Filter by computer name... [Filter by name](#)

[Add a new computer](#)

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

[← Previous](#) Displaying 1 to 10 of 574 [Next →](#)

# How it works...

In this recipe, we deployed a Play 2 web application by packaging it as a Docker image and then installing and running the same Docker image in a Digital Ocean Droplet. Firstly, we will need an account on [DigitalOcean.com](#) and [Docker.com](#).

Once our accounts are ready and verified, we create a CoreOS-based droplet. CoreOS has Docker installed by default, so all we need to install in the droplet is the Play 2 web app Docker image.

The Play 2 web app Docker image is based on the activator template, computer-database-scala, which we named play2-deploy-73.

We make two modifications to the boilerplate code. The first modification in `conf/application.conf`:

```
applyEvolutions.default=true
```

This setting enables database evolutions by default. The other modification is to be made in `build.sbt`. We import the required packages that contain the Docker-specific settings:

```
import NativePackagerKeys._
import com.typesafe.sbt.SbtNativePackager._
```

The next settings are to specify the repository maintainer, the exposed Docker ports, and the Docker repository in [Docker.com](#); in this case, supply your own Docker Hub username as the maintainer and Docker repository values:

```
maintainer := "<YOUR DOCKERHUB_USERNAME>"
dockerExposedPorts in Docker := Seq(9000)
dockerRepository := Some("<YOUR_DOCKERHUB_USERNAME>")
```

We can now build Docker images using the activator command, which will generate all the necessary files for building a Docker image:

```
activator clean docker:stage
```

Now, we will use the activator docker command to upload and publish to your specified [Docker.com](#) repository:

```
activator clean docker:publish
```

To install the Docker image in our Digital Ocean Droplet, we first log in to the droplet using the `core` user:

```
ssh core@<DROPLET_IP_ADDRESS>
```

We then use the docker command, `docker pull`, to download the `play2-deploy-73` image from [Docker.com](#), specifying the tag:

```
docker pull <YOUR_DOCKERHUB_USERNAME>/play2-deploy-73:0.0.1-SNAPSHOT
```

Finally, we can run the Docker image using the `docker run` command, exposing the

container port 9000:

```
docker run -p 9000:9000 <YOUR_DOCKERHUB_USERNAME>/play2-deploy-
73:0.0.1-SNAPSHOT
```

## **There's more...**

Refer to the following links for more information on Docker and Digital Ocean:

- <https://www.docker.com/whatisdocker/>
- <https://www.digitalocean.com/community/tags/docker>



# Deploying a Play application with Dokku

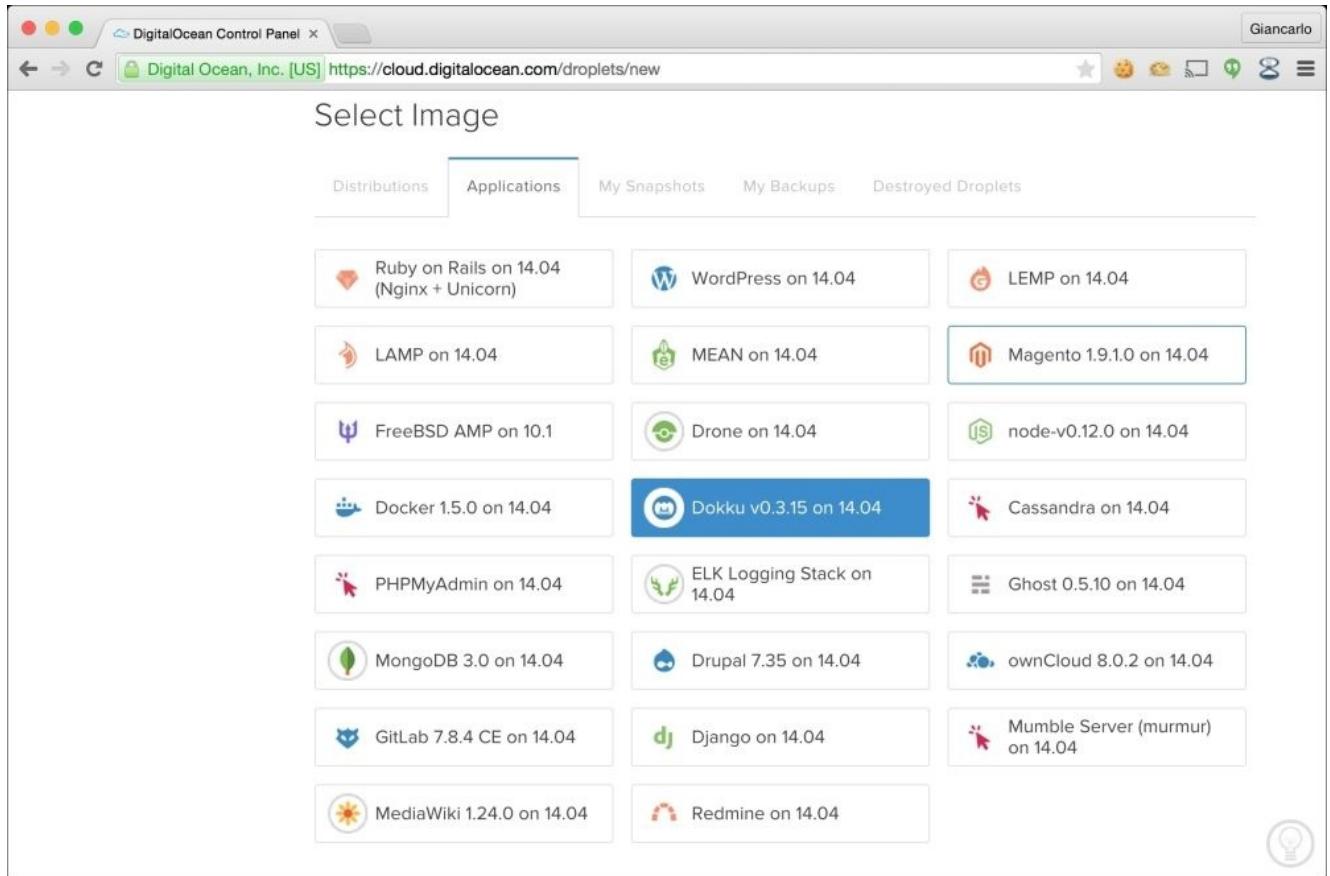
In this recipe, we will use the Docker-based tool, Dokku, to manage our Play 2 web application deployment. **Dokku** provides a very straightforward deployment interface, quite similar to Heroku's deployment interface and allows developers to quickly deploy Play 2 web apps.

We will have Dokku running and will deploy our Play 2 web application in a Digital Ocean Droplet. We will need a Droplet with at least 2GB of RAM to run our sample webapp. Ensure that you sign up for a Digital Ocean account to follow this recipe:

<https://cloud.digitalocean.com/registrations/new>

# How to do it...

1. Prepare your deployment droplet by creating a new 2GB RAM Droplet with **Dokku v0.3.15** on **14.04** or higher application option pre-installed:



2. Once droplet creation is completed, you can verify the Droplet settings on the dashboard, ensuring that it is configured with 2GB of RAM and making note of the assigned IP address for the droplet:

DigitalOcean Control Panel

Digital Ocean, Inc. [US] https://cloud.digitalocean.com/droplets/4575229

Giancarlo

play2-deploy-74

Create Droplet

192.241.239.43 Active 2GB Ram 40GB SSD Disk San Francisco 1 Ubuntu Dokku v0.3.15 on 14.04

Droplets

Images

SSH Keys

Billing

Support

DNS

Apps & API

Power Access Resize Snapshots Settings Graphs Destroy

Power Cycle Power Off

This action is equivalent to hard resetting the server, which can cause data corruption. You should do this only if you're unable to reboot your Droplet from the command line.

Do you want to proceed?

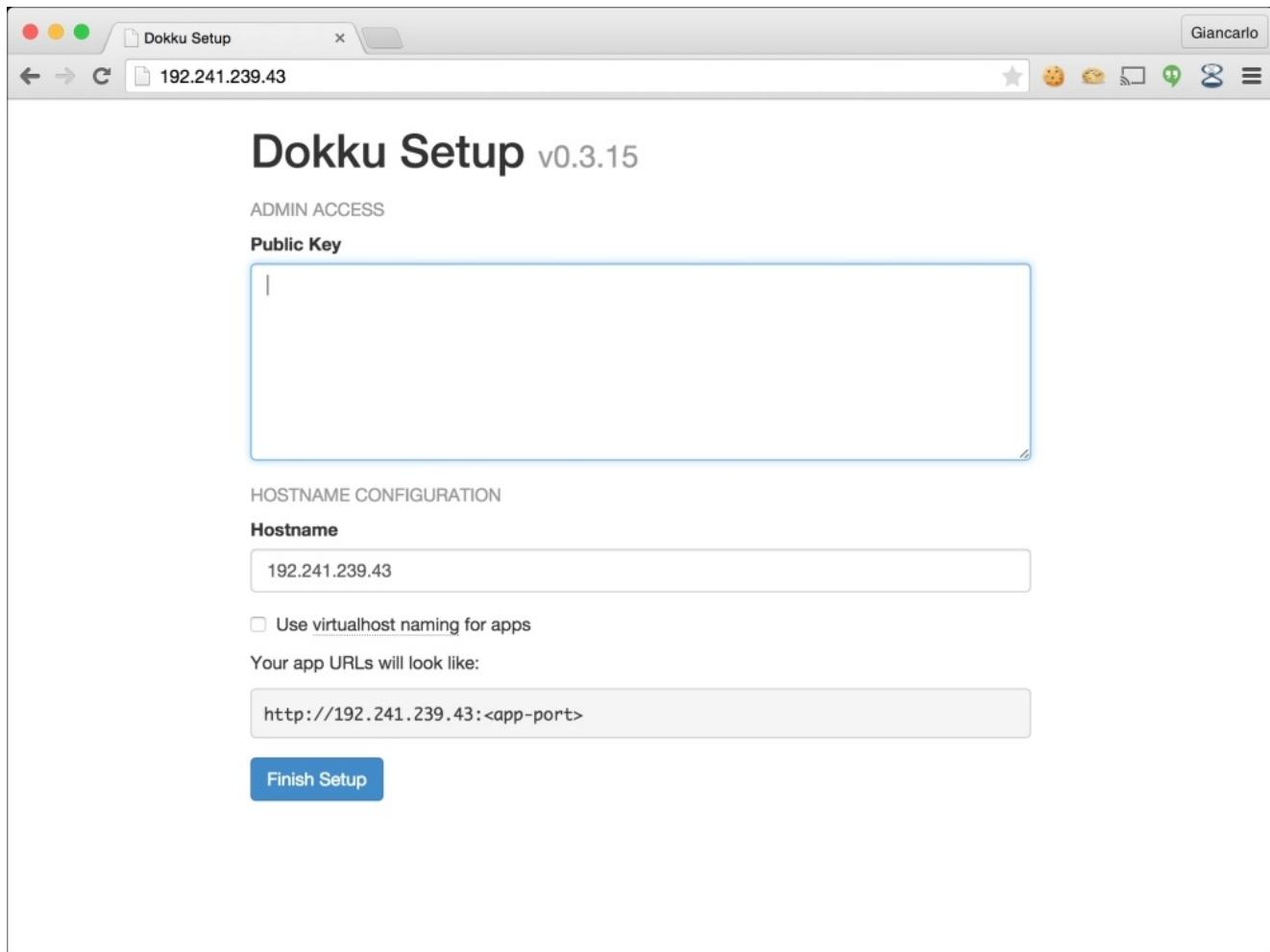
Power Cycle

Droplet History

Event	Initiated	Execution Time
Create	about 1 hour ago	1 minute 45 seconds

Settings Logout

3. Next, we will need to complete the Dokku installation by accessing the web port of our newly created droplet; this can be done by accessing the Droplet's assigned IP address using a web browser to confirm the Dokku settings, such as your public key, hostname, and so on:



4. Once our Droplet is set up, we need to prep our Play 2 web app using the activator template, computer-database-scala, in our local development machine and changing into the project root:

```
$ activator new play2-deploy-74 computer-database-scala && cd play2-deploy-74
```

5. Edit conf/application.conf to enable automatic database evolutions:

```
applyEvolutions.default=true
```

6. We will need to initialize git on the project root:

```
git init && git add --all && git commit -am "initial"
```

7. We now add a new git remote to the codebase pointed at our Dokku Droplet:

```
git remote add dokku dokku@<YOUR_DOKKU_IP_ADDRESS>:play2-deploy-74
```

8. As the last deployment step, we push our commit to our Dokku remote. Dokku will then automatically deploy the web app using git post-commit hooks:

```
$ git push dokku master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
```

```
Writing objects: 100% (3/3), 293 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
-----> Cleaning up...
-----> Building play2-deploy-74 from buildstep...
-----> Adding BUILD_ENV to build environment...
-----> Play 2.x - Scala app detected
-----> Installing OpenJDK 1.6...done
-----> Running: sbt compile stage
..
-----> Dropping ivy cache from the slug
-----> Dropping compilation artifacts from the slug
-----> Discovering process types
 Default process types for Play 2.x - Scala -> web
-----> Releasing play2-deploy-74...
-----> Deploying play2-deploy-74...
-----> Running pre-flight checks
 check-deploy: /home/dokku/play2-deploy-74/CHECKS not found.
attempting to retrieve it from container...
 CHECKS file not found in container. skipping checks.
-----> Running post-deploy
-----> NO_VHOST config detected
-----> Shutting down old container in 60 seconds
===== Application deployed:
http://<YOUR_DOKKU_IP>:<YOUR_DOKKU_ASSIGNED_PORT>

To dokku@192.241.239.43:play2-deploy-74
77a951d..a007b6b master -> master
```

We can now use a web browser to access our Dokku-deployed Play 2 web application:  
[http://<YOUR\\_DOKKU\\_IP\\_ADDRESS>:<YOUR\\_DOKKU\\_PORT>](http://<YOUR_DOKKU_IP_ADDRESS>:<YOUR_DOKKU_PORT>)

Computers database

192.241.239.43:49154/computers

Giancarlo

## Play sample application — Computer database

### 574 computers found

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

← Previous      Displaying 1 to 10 of 574      Next →

# How it works...

In this recipe, we looked into using Dokku to deploy our Play 2 web app on a Digital Ocean Droplet. After some very straightforward initializations and configurations, we were able to deploy our Play 2 web app with minimal friction and with ease.

Essential to this recipe is having a virtual machine pre-installed with Dokku. Dokku, the Docker-based deployment tool, provides developers with a very simple deployment process, which ultimately boils down to a `git push` command:

```
$ git push dokku master
```

Dokku has git post-commit hooks configured to detect commits to the codebase, and subsequently, runs a deployment script for every detected code commit.

```
----> Discovering process types
 Default process types for Play 2.x - Scala -> web
----> Releasing play2-deploy-74...
----> Deploying play2-deploy-74...
----> Running pre-flight checks
 check-deploy: /home/dokku/play2-deploy-74/CHECKS not found.
attempting to retrieve it from container...
 CHECKS file not found in container. skipping checks.
----> Running post-deploy
----> NO_VHOST config detected
----> Shutting down old container in 60 seconds
===== Application deployed:
 http://192.241.239.43:49154
```

This ease of use allows developers and non-developers alike to spin up dev and test instances quickly.

However, to get to this point in the development lifecycle, we will need to follow some steps to initialize and configure our Dokku setup. For this recipe, we relied on Digital Ocean, a popular cloud-based virtual machine provider, and its predefined Dokku application instance to spin up a ready-to-use Dokku VM instance.

The next necessary steps would be configuring our Play 2 web app to be git-enabled by initializing the codebase, adding all codebase files to the git repository, and committing the initial state of the web app:

```
$ git init && git add -all && git commit -am "initial"
```

Also, add some necessary application configurations for our sample application, specifically the `conf/application.conf` file with the following settings:

```
applyEvolutions.default=true
```

We will need to commit this change to the local repo as well:

```
$ git commit -am "enabled automatic db evolutions"
```

The final step now will be to push our commits to the Dokku instance we just created, which should then trigger a deployment:

```
$ git push dokku master
```

## There's more...

Refer to the following links for more information on Dokku:

- <http://progrium.viewdocs.io/dokku/index>
- <https://www.digitalocean.com/community/tags/dokku>



# Deploying a Play application with Nginx

In this recipe, we will manually deploy a Play 2 web application using a CentOS 6.5-based virtual machine with **Nginx** as the frontend server for our Play 2 web applications. We will be utilizing Digital Ocean; ensure that you sign up for an account here:

<https://cloud.digitalocean.com/registrations/new>

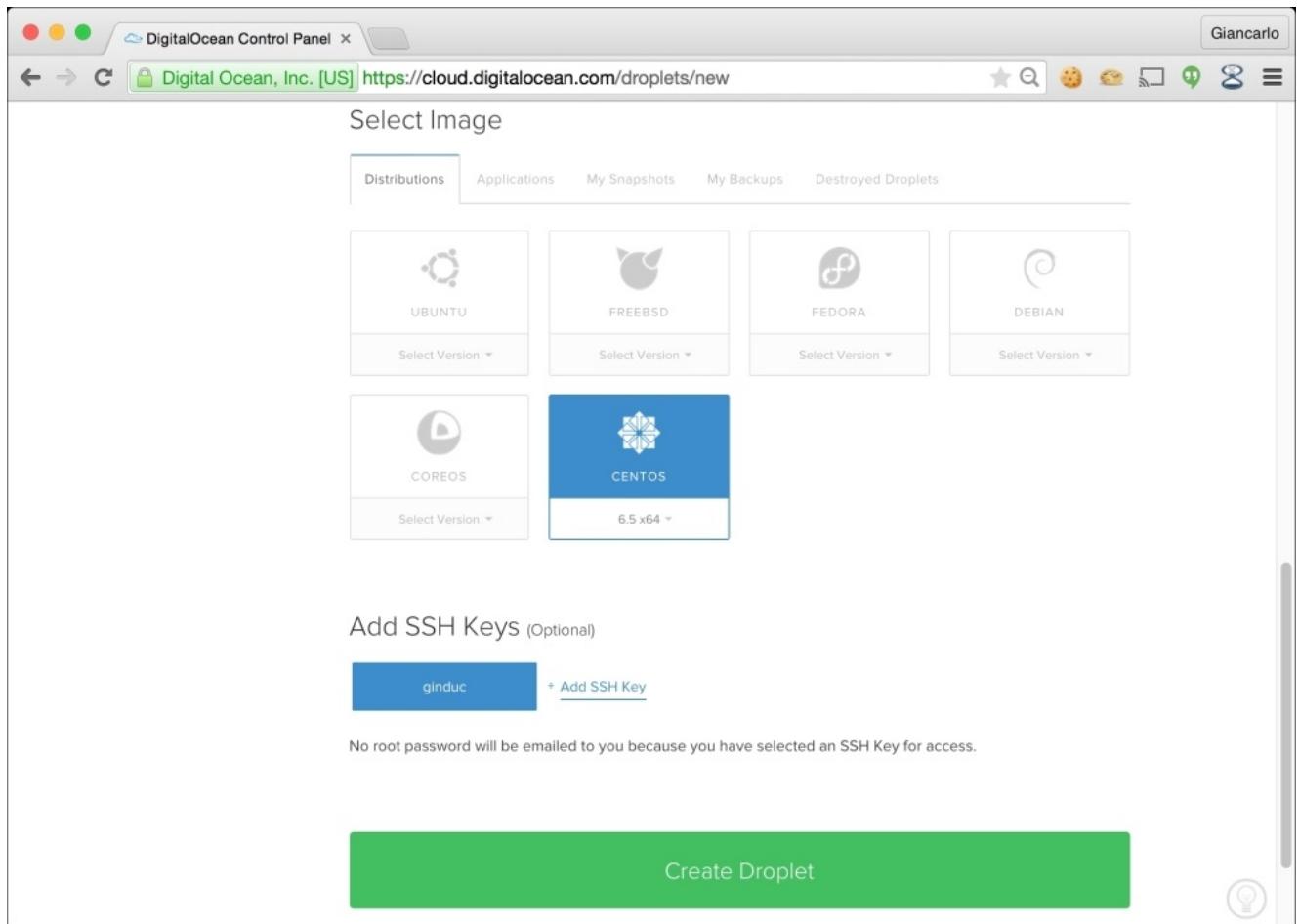
# How to do it...

1. Log in to Digital Ocean and create a new Droplet, selecting CentOS as the base operating system and with at least 2GB of RAM:

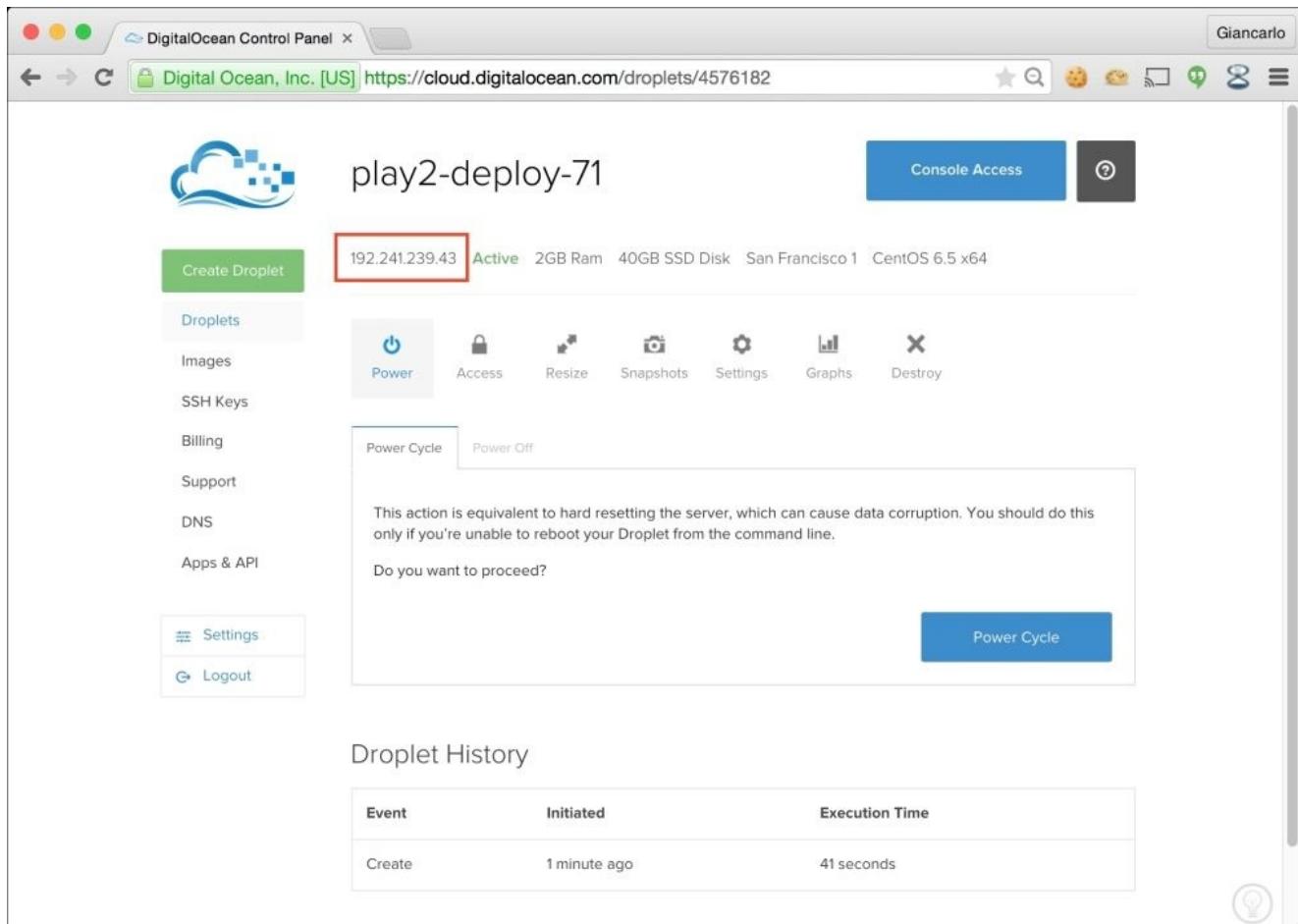
The screenshot shows the DigitalOcean Control Panel with the title 'Create Droplet'. On the left, there's a sidebar with links: 'Create Droplet' (highlighted in green), 'Droplets', 'Images', 'SSH Keys', 'Billing', 'Support', 'DNS', and 'Apps & API'. Below the sidebar, there are two buttons: 'Settings' and 'Logout'. The main area has a heading 'Droplet Hostname' with the value 'play2-deploy-71'. Underneath is a section titled 'Select Size' containing a grid of 12 droplet options. The first row contains four options: \$5/mo (\$0.007/hour), \$10/mo (\$0.015/hour), \$20/mo (\$0.030/hour) which is highlighted in blue, and \$40/mo (\$0.060/hour). The second row contains four options: \$80/mo (\$0.119/hour), 4 GB / 2 CPUs (60 GB SSD Disk, 4 TB Transfer), 8 GB / 4 CPUs (80 GB SSD Disk, 5 TB Transfer), and another \$80/mo (\$0.119/hour) option. The third row contains four options: \$160/mo (\$0.238/hour), \$320/mo (\$0.476/hour), \$480/mo (\$0.714/hour), and \$640/mo (\$0.952/hour). The fourth row contains two options: 16 GB / 8 CPUs (160 GB SSD Disk, 6 TB Transfer) and 32 GB / 12 CPUs (320 GB SSD Disk, 7 TB Transfer).

Size	Price	RAM	CPU	Disk	Transfer
512 MB / 1 CPU 20 GB SSD Disk 1000 GB Transfer	\$5/mo \$0.007/hour	512 MB	1 CPU	20 GB SSD Disk	1000 GB Transfer
1 GB / 1 CPU 30 GB SSD Disk 2 TB Transfer	\$10/mo \$0.015/hour	1 GB	1 CPU	30 GB SSD Disk	2 TB Transfer
2 GB / 2 CPUs 40 GB SSD Disk 3 TB Transfer	\$20/mo \$0.030/hour	2 GB	2 CPUs	40 GB SSD Disk	3 TB Transfer
4 GB / 2 CPUs 60 GB SSD Disk 4 TB Transfer	\$40/mo \$0.060/hour	4 GB	2 CPUs	60 GB SSD Disk	4 TB Transfer
8 GB / 4 CPUs 80 GB SSD Disk 5 TB Transfer	\$80/mo \$0.119/hour	8 GB	4 CPUs	80 GB SSD Disk	5 TB Transfer
16 GB / 8 CPUs 160 GB SSD Disk 6 TB Transfer	\$160/mo \$0.238/hour	16 GB	8 CPUs	160 GB SSD Disk	6 TB Transfer
32 GB / 12 CPUs 320 GB SSD Disk 7 TB Transfer	\$320/mo \$0.476/hour	32 GB	12 CPUs	320 GB SSD Disk	7 TB Transfer
48 GB / 16 CPUs 480 GB SSD Disk 8 TB Transfer	\$480/mo \$0.714/hour	48 GB	16 CPUs	480 GB SSD Disk	8 TB Transfer
64 GB / 20 CPUs 640 GB SSD Disk 9 TB Transfer	\$640/mo \$0.952/hour	64 GB	20 CPUs	640 GB SSD Disk	9 TB Transfer

2. Select **CentOS 6.5 x64** as the Droplet image and specify your **SSH Key**:



3. Once the Droplet has been created, make a special note of the VM's IP address:



#### 4. Log in to our newly created Centos-based Droplet using SSH:

```
ssh root@<YOUR_DROPLET_IP_ADDRESS>
```

#### 5. Create a non-root user using the adduser command, assigning it with a password using the passwd command, and finally adding our new user to the sudoers group, which is a list of users with special system privileges:

```
$ adduser deploy
$ passwd deploy
$ echo "deploy ALL=(ALL) ALL" >> /etc/sudoers
```

#### 6. Relog in to our Droplet using the non-root user:

```
ssh deploy@<YOUR_DROPLET_IP_ADDRESS>
```

#### 7. Install the necessary tools using Yum, the CentOS Package Manager, to proceed with our VM setup:

```
sudo yum install -y yum-plugin-fastestmirror
sudo yum install -y git unzip wget
```

#### 8. Install a JDK using the following commands:

```
curl -LO 'http://download.oracle.com/otn-pub/java/jdk/7u51-b13/jdk-
7u51-linux-x64.rpm' -H 'Cookie: oraclelicense=accept-securebackup-
cookie'
sudo rpm -i jdk-7u51-linux-x64.rpm
```

```
sudo /usr/sbin/alternatives --install /usr/bin/java java
/usr/java/default/bin/java 200000
```

9. Verify that we have the Oracle JDK installed and that it is accessible:

```
Verify installed jdk
$ java -version
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
```

10. Install activator using the following commands:

```
cd ~
wget http://downloads.typesafe.com/typesafe-
activator/1.3.2/typesafe-activator-1.3.2-minimal.zip
unzip typesafe-activator-1.3.2-minimal.zip
chmod u+x ~/activator-1.3.2-minimal/activator
export PATH=$PATH:~/activator-1.3.2-minimal/
```

Add the official Nginx yum repository with the following command:

```
echo -e "[nginx]\nname=nginx
repo\nbaseurl=http://nginx.org/packages/centos/\$releasever/\$basearch/\ngp
gcheck=0\nenabled=1" > /tmp/nginx.repo && sudo mv /tmp/nginx.repo
/etc/yum.repos.d/nginx.repo
```

1. Install Nginx using yum:

```
sudo yum install -y nginx
```

2. Add the custom configuration file to /etc/nginx/conf.d/computer-database-scala.conf with the following content:

```
upstream playapp1 {
 server 127.0.0.1:9000;
}

server {
 listen 80;
 server_name computer-database-scala.com;
 location / {
 proxy_pass http://playapp1;
 }
}
```

3. Restart Nginx to load our new configs:

```
sudo service nginx restart
```

4. Configure iptables to allow access to a minimum set of ports only (port 22 and 80):

```
sudo /sbin/iptables -P INPUT ACCEPT
sudo /sbin/iptables -F
sudo /sbin/iptables -A INPUT -i lo -j ACCEPT
sudo /sbin/iptables -A INPUT -m state --state ESTABLISHED,RELATED -
j ACCEPT
```

```
sudo /sbin/iptables -A INPUT -p tcp --dport 22 -j ACCEPT
sudo /sbin/iptables -A INPUT -p tcp --dport 80 -j ACCEPT
sudo /sbin/iptables -P INPUT DROP
sudo /sbin/iptables -P FORWARD DROP
sudo /sbin/iptables -P OUTPUT ACCEPT
sudo /sbin/iptables -L -v
sudo /sbin/service iptables save
sudo /sbin/service iptables restart
sudo /sbin/service iptables status
```

- Clone the computer-database-scala Play2 web application in the directory ~/apps:

```
$ mkdir ~/apps && cd $_ && git clone
https://github.com/typesafehub/activator-computer-database-scala.git
```

- Edit conf/application.conf to enable automatic database evolutions:

```
applyEvolutions.default=true
```

- We can now start the web app using activator:

```
$ activator start
[info] Loading project definition from /home/deploy/apps/activator-
computer-database-scala/project
[info] Set current project to computer-database-scala (in build
file:/home/deploy/apps/activator-computer-database-scala/)
[info] Packaging /home/deploy/apps/activator-computer-database-
scala/target/scala-2.11/computer-database-scala_2.11-0.0.1-SNAPSHOT-
sources.jar...
[info] Done packaging.
[warn] There may be incompatibilities among your library
dependencies.
[warn] Here are some of the libraries that were evicted:
[warn] * org.webjars:jquery:1.11.1 -> 2.1.1
[warn] Run 'evicted' to see detailed eviction warnings
[info] Wrote /home/deploy/apps/activator-computer-database-
scala/target/scala-2.11/computer-database-scala_2.11-0.0.1-SNAPSHOT.pom
[info] Packaging /home/deploy/apps/activator-computer-database-
scala/target/scala-2.11/computer-database-scala_2.11-0.0.1-
SNAPSHOT.jar...
[info] Done packaging.

(Starting server. Type Ctrl+D to exit logs, the server will remain
in background)
```

```
Play server process ID is 3094
[info] play - database [default] connected at jdbc:h2:mem:play
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

- Using a web browser, we will now be able to access the Play 2 web application using the droplet's IP address, in our case, http://<YOUR\_DROPLET\_IP  
ADDRESS>/computers:

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

We can verify that Nginx is indeed serving our HTTP requests using curl and verifying with the response headers:

```
$ curl -v http://192.241.239.43/computers
* Hostname was NOT found in DNS cache
* Trying 192.241.239.43...
* Connected to 192.241.239.43 (192.241.239.43) port 80 (#0)
> GET /computers HTTP/1.1
> User-Agent: curl/7.37.1
> Host: 192.241.239.43
> Accept: */*
>
< HTTP/1.1 200 OK
* Server nginx/1.6.2 is not blacklisted
< Server: nginx/1.6.2
< Date: Tue, 24 Mar 2015 08:23:04 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 7371
< Connection: keep-alive
```

# How it works...

In this recipe, we manually deployed a Play 2 web application on a remote CentOS-based virtual machine, which we initialized using Digital Ocean. We then proceeded to install and configure various software components, such as the Java Development Kit, Nginx, IPTables, and so on:

```
$ java -version
java version "1.7.0_51"

$ activator --version
sbt launcher version 0.13.8-M5

$ nginx -v
nginx version: nginx/1.6.2

$ iptables --version
iptables v1.4.7
$ curl --version
curl 7.19.7

$ git --version
git version 1.9.5
```

Once we have the necessary services installed and configured, we need to start our Play 2 web application:

```
$ cd apps/activator-computer-database-scala/ && activator clean start
Play server process ID is 3917
[info] play - database [default] connected at jdbc:h2:mem:play
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

We then successfully accessed our deployed Play 2 web application using a web browser and curl in the appropriate port, port 80. We can also verify that we can only access the Play 2 web app via Nginx by attempting to access port 9000 directly using curl:

```
$ curl -v http://192.241.239.43:9000/computers
* Hostname was NOT found in DNS cache
* Trying 192.241.239.43...
* connect to 192.241.239.43 port 9000 failed: Operation timed out
* Failed to connect to 192.241.239.43 port 9000: Operation timed out
* Closing connection 0
curl: (7) Failed to connect to 192.241.239.43 port 9000: Operation
timed out
```

In this chapter, we demonstrated how remarkably involved it is to manually deploy Play 2 applications, and, in contrast, how convenient and easy it is to use cloud services such as Heroku, AWS Beanstalk, and tools such as Docker and Dokku in deploying Play 2 applications in the cloud. While there could be merits to being able to deploy Play web applications manually, it is without question that Cloud PAAS services greatly boost a developer's productivity and efficiency, and allow developers to focus on actual software development.



# Chapter 8. Additional Play Information

In this chapter, we will cover the following recipes:

- Testing with Travis CI
- Monitoring with New Relic
- Integrating a Play application with AngularJS
- Integrating a Play application with Parse
- Creating a Play development environment using Vagrant
- Coding Play 2 web apps with IntelliJ IDEA 14

# Introduction

In this chapter, we will explore additional recipes for Play that developers will find handy and useful in their toolbox. We will touch on automated testing and monitoring tools for Play 2.0 web applications that are essential auxiliary tools for the modern web application. We will also look at integrating an AngularJS frontend and integrating [Parse.com](#), a **Backend-as-a-Service (BaaS)**, to manage our data in a Play web application.

Finally, we will look into automating the creation of a Play development environment using the popular tool **Vagrant**, allowing developers to create shareable and more portable development environments.



# Testing with Travis CI

For this recipe, we will explore how to use **Travis CI** to build and run automated tests for a Play 2.0 web app. We need to sign up for an account on Travis CI together with a Github account. We will also configure our Travis account in such a way that it is connected to a Github repository to conduct automatic tests on code commits.

# How to do it...

For this recipe, you need to perform the following steps:

1. Create a Github account at <https://github.com/join>:

The screenshot shows the 'Join GitHub' page on a web browser. The URL in the address bar is <https://github.com/join>. The page title is 'Join GitHub'. At the top right are 'Sign up' and 'Sign in' buttons. Below the title, there's a heading 'Join GitHub' and a subtext 'The best way to design, build, and ship software.' A progress bar at the top indicates three steps: 'Step 1: Set up a personal account', 'Step 2: Choose your plan', and 'Step 3: Go to your dashboard'. The main form area is titled 'Create your personal account' and contains fields for 'Username', 'Email Address', 'Password', and 'Confirm your password'. To the right, a sidebar titled 'You'll love GitHub' lists 'Unlimited collaborators', 'Unlimited public repositories', and three benefits: 'Great communication', 'Friction-less development', and 'Open source community'. At the bottom of the form is a 'Create an account' button.

2. Create a new public Github repository called play2-travis at:  
<https://github.com/new>
3. On your development machine, create a new Play 2.0 web app using the activator template play-slick-angular-test-example:  

```
activator new play2-travis play-slick-angular-test-example
```
4. Edit .travis.yml to trigger our test script:

```
language: scala
scala:
- 2.11.2
```

```
script:
- sbt test
```

5. Commit and push to the Github remote repository (please make special note of your Github username and specify it in the command below):

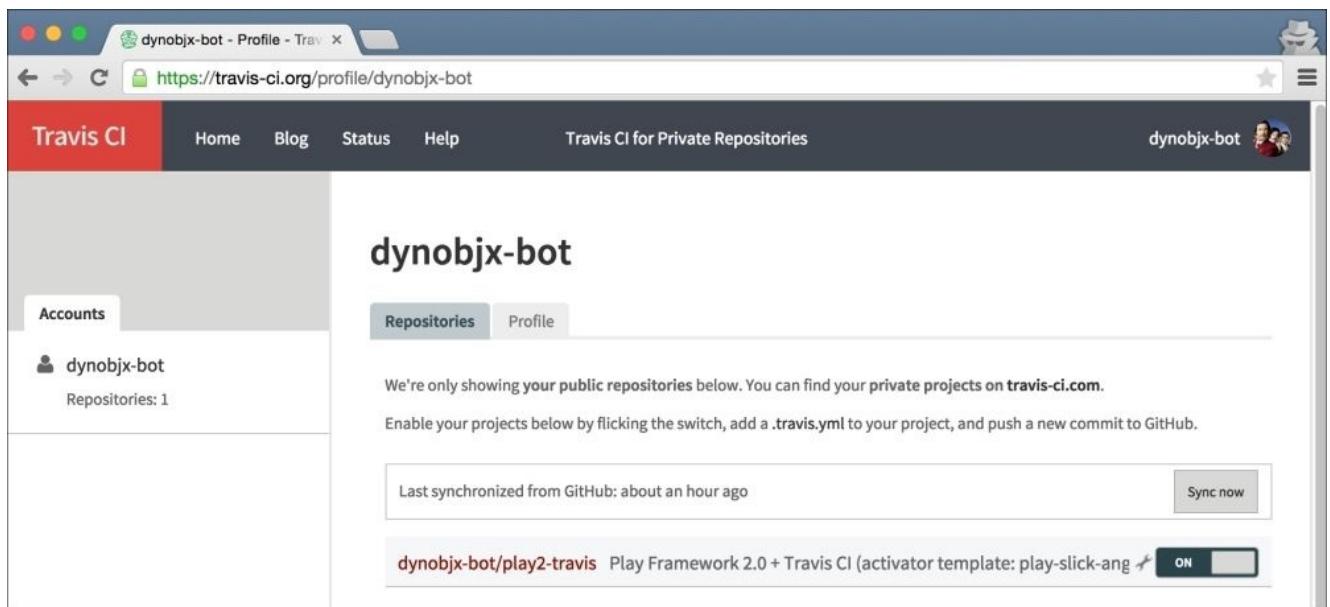
```
git add --all && git commit -am "Initial" && git remote add origin
https://github.com/<YOUR_GITHUB_USER>/play2-travis.git && git push
origin master
```

6. Sign up for a Travis account by using your Github account at:

<https://travis-ci.org>

7. After Travis has synced your Github repositories, enable Travis builds for the play2-travis repository at

[https://travis-ci.org/profile/<YOUR\\_GITHUB\\_USER>](https://travis-ci.org/profile/<YOUR_GITHUB_USER>):



8. Next, modify test/controllers/ReportSpec.scala by adding a sample test failure:

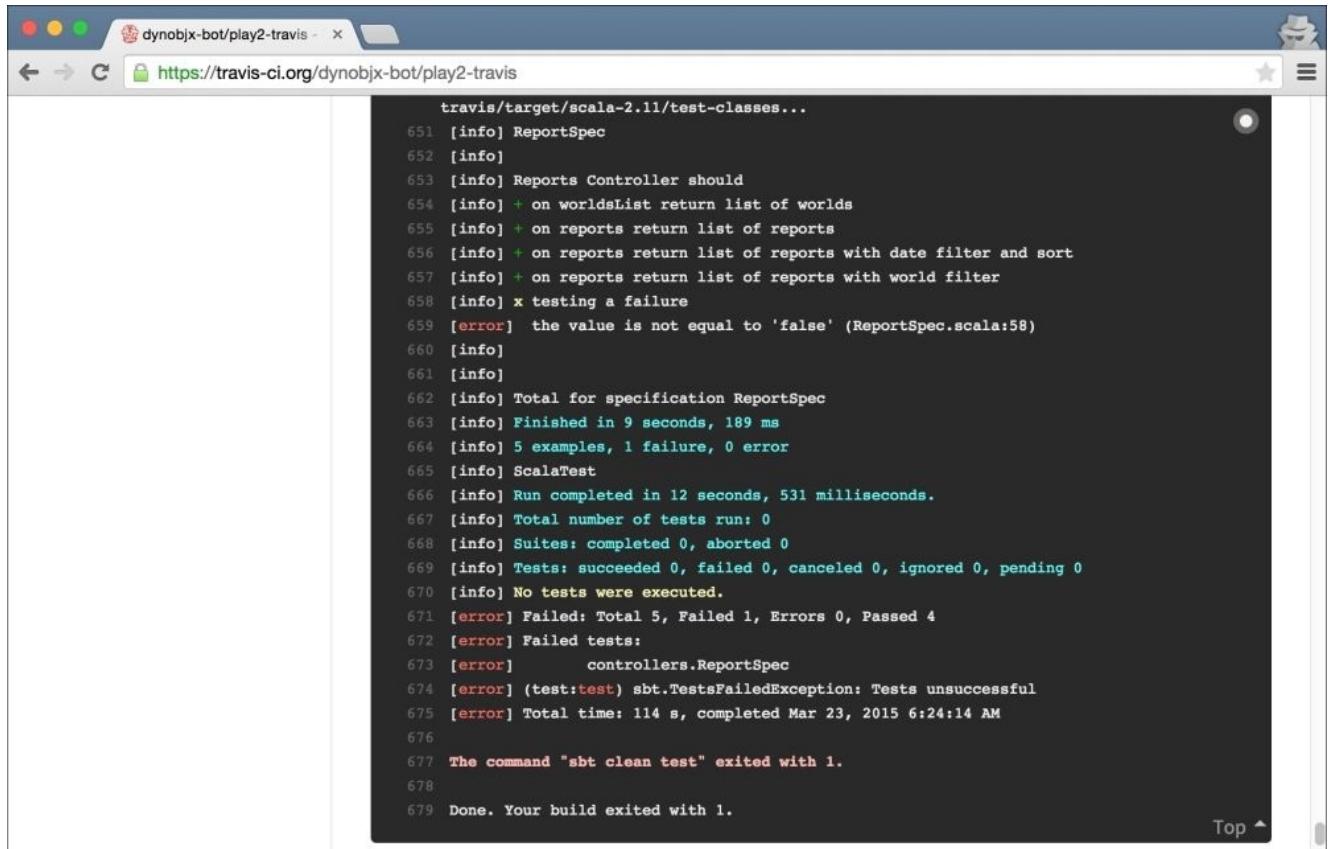
```
"testing a failure" in {
 true must equalTo(false)
}
```

9. Commit and push to trigger a Travis build with the expected test failure:

```
git commit -am "With expected test failure" && git push origin
master
```

10. This should trigger our build in Travis after a few seconds. We expect our very first build to fail and should see a result similar to the following:

```
[info] x testing a failure
[error] the value is not equal to 'false' (ReportSpec.scala:58)
```



```
travis@target:scala-2.11/test-classes...
651 [info] ReportSpec
652 [info]
653 [info] Reports Controller should
654 [info] + on worldsList return list of worlds
655 [info] + on reports return list of reports
656 [info] + on reports return list of reports with date filter and sort
657 [info] + on reports return list of reports with world filter
658 [info] x testing a failure
659 [error] the value is not equal to 'false' (ReportSpec.scala:58)
660 [info]
661 [info]
662 [info] Total for specification ReportSpec
663 [info] Finished in 9 seconds, 189 ms
664 [info] 5 examples, 1 failure, 0 error
665 [info] ScalaTest
666 [info] Run completed in 12 seconds, 531 milliseconds.
667 [info] Total number of tests run: 0
668 [info] Suites: completed 0, aborted 0
669 [info] Tests: succeeded 0, failed 0, canceled 0, ignored 0, pending 0
670 [info] No tests were executed.
671 [error] Failed: Total 5, Failed 1, Errors 0, Passed 4
672 [error] Failed tests:
673 [error] controllers.ReportSpec
674 [error] (test:test) sbt.TestsFailedException: Tests unsuccessful
675 [error] Total time: 114 s, completed Mar 23, 2015 6:24:14 AM
676
677 The command "sbt clean test" exited with 1.
678
679 Done. Your build exited with 1.
```

11. Now, comment out the sample test failure in `test/controllers/ReportSpec.scala`:

```
/*"testing a failure" in {
 true must equalTo(false)
}*/
```

12. Let's commit and push these latest changes, this time expecting our Travis build to pass:

```
git commit -am "disabling failing test" && git push origin master
```

13. This commit should again trigger a build in Travis, and this time around, we should see all of the tests passing in our Travis dashboard.

Build #2 - dynobjx-bot/play2-travis

https://travis-ci.org/dynobjx-bot/play2-travis/builds/55446315

Travis CI Home Blog Status Help Travis CI for Private Repositories dynobjx-bot

Search all repositories

My Repositories Recent +

dynobjx-bot/play2-travis #3

Duration: 3 min 24 sec Finished: 34 minutes ago

master disabling sample test failure

# 2 passed Commit 2dcfc52 Compare 4df034f..2dc ran for 5 min 38 sec about an hour ago

Gian Inductivo authored and committed

Remove Log Download Log

```
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-18
2
3 Build system information
4
5
6 git clone --depth=50 --branch=master
7 jdk_switcher use default
8 Switching to Oracle JDK7 (java-7-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-7-oracle
```

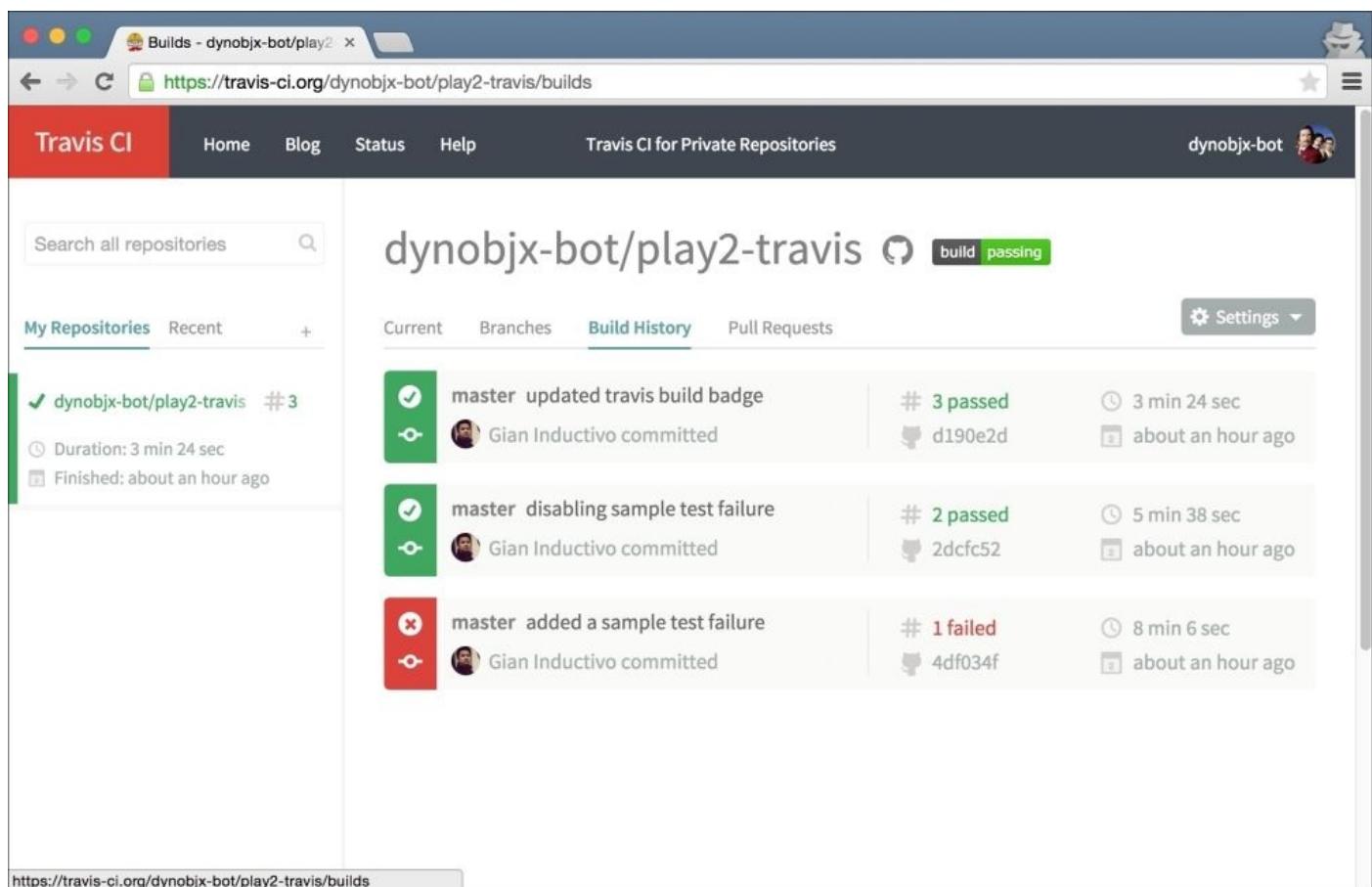
# How it works...

In this recipe, we utilized Travis CI to build and execute tests in our linked Github repository. This setup enables us to establish a **development-commit-test** process. This setup requires user accounts for Travis CI and Github.

Once we have identified a Github repository, we want to integrate with Travis. We need to update the Travis configuration (`.travis.yml`) in the project root, specifying a run script to execute our webapp tests:

```
script:
- sbt test
```

This is the command that Travis executes when running our tests. Travis will configure builds based on the settings in the `.travis.yml` configuration file (in our recipe, running the `sbt task test`) to execute our webapp tests. Build results are displayed on the repository dashboard in Travis:



The screenshot shows the Travis CI dashboard for the repository `dynobjx-bot/play2-travis`. The build status is **passing**. The build history section displays three recent builds:

Build	Status	Duration	Commit	Time
master updated travis build badge	3 passed	3 min 24 sec	d190e2d	about an hour ago
master disabling sample test failure	2 passed	5 min 38 sec	2dcfc52	about an hour ago
master added a sample test failure	1 failed	8 min 6 sec	4df034f	about an hour ago

Once we have our Github repository linked and enabled in Travis, we observe that a build is triggered after every code commit and pushed to the Github repository. This is a great development tool and process for developers to be made aware of any regression issues to recently checked-in code, with support for other build tooling such as artifact publishing and notifications.



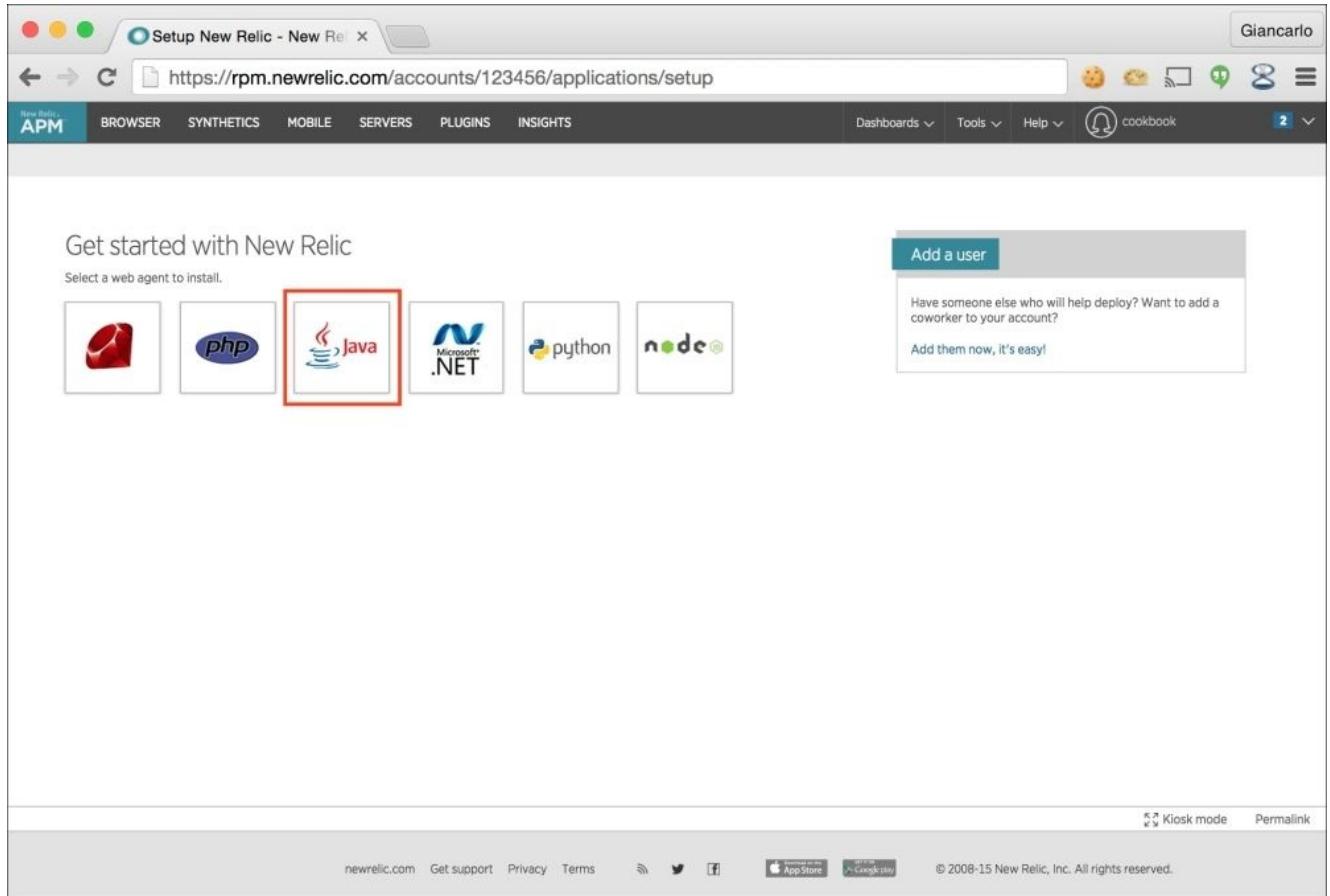
# Monitoring with New Relic

For this recipe, we will deploy a Play 2.0 web app using Docker and Digital Ocean, and monitor the web app using **New Relic**. We will deploy the web app as a Docker container and detail out how to instrument the computer-database-scala sample web application using a New Relic JAR file integrated with our activator build script.

# How to do it...

For this recipe, you need to perform the following steps:

1. First, sign up for a New Relic account at <https://newrelic.com/signup>
2. Create a new Java application in the New Relic dashboard:



3. During the creation of the Java application, please make note of your New Relic license key:

The screenshot shows the New Relic Java agent setup page. At the top, there's a navigation bar with links for BROWSER, SYNTHETICS, MOBILE, SERVERS, PLUGINS, and INSIGHTS. On the right side of the header, there are links for Dashboards, Tools, Help, and a user profile for 'Giancarlo'. Below the header, the main content area has a title 'Install the Java agent' and a section titled 'Before you begin' with a list of requirements:

- 1. Administrator access to the computer on which you will install.
- 2. Ability to configure any firewalls or proxies to allow the agent to report data to New Relic.
- 3. Access to your Windows Azure Administration Portal.

On the right side of the main content area, there's a sidebar titled 'Installing the Java agent' with a play button icon. Below the main content, there are three numbered steps:

- 1 Get your license key ([Java agent docs](#))  
A box highlights the license key '11111112222223333333344444555555'.
- 2 Download the Java agent  
[Download the Java agent](#)
- 3 Install the agent  
Select your environment:  
 Linux or Mac  
 Windows

At the bottom of the sidebar, there's a note for Linux or Mac users: 'Unzip the file into your app's home directory.'

4. Next, download the New Relic Java agent ZIP and make note of the download location. This ZIP file should contain the Java agent library, license key file, API documentation, and other useful New Relic resources.

The screenshot shows the New Relic Java agent setup page. It includes a sidebar with links like 'Dashboards', 'Tools', 'Help', and 'cookbook'. The main content area has three steps: 1. Get your license key (with a placeholder key '11111122222233333333444445555555'), 2. Download the Java agent (with a red 'Download the Java agent' button), and 3. Install the agent (with options for Linux or Mac or Windows). A sidebar on the right provides helpful support docs and a video player.

5. Unzip the Java agent ZIP file and make note of two important files that we will need, `newrelic.yml` and `newrelic.jar`:

```
$ ls ~/Downloads/newrelic/
CHANGELOG
extension-example.xml
newrelic-api-sources.jar
newrelic.yml
LICENSE
extension.xsd
newrelic-api.jar
nrcerts
README.txt
newrelic-api-javadoc.jar
newrelic.jar
```

6. Edit the `newrelic.yml` file by adding a relevant name to the setting parameter `app_name`, For this recipe, we will name our `app_name`, `computer-database-scala`:

```
common: &default_settings
```

```
license_key: '111112222233333444445555556666666'
agent_enabled: true
```

```
app_name: computer-database-scala
..
```

7. Create a new Play web application using the activator template computer-database-scala, and change it into the project root directory:

```
activator new play2-deploy-81 computer-database-scala
cd play2-deploy-81
```

8. Create an instrument directory in the conf directory:

```
mkdir conf/instrument
```

9. Copy our two New Relic config files to conf/instrument:

```
cp ~/Downloads/newrelic/newrelic.yml conf/instrument
cp ~/Downloads/newrelic/newrelic.jar conf/instrument
```

10. Edit conf/application.conf to enable automatic database evolution:

```
applyEvolutions.default=true
```

11. Add a newer version of the native Docker packager sbt plugin in project/plugins.sbt, which has additional native support for Docker:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.0.0-M3")
```

12. Edit build.sbt to specify docker-specific settings for the web app:

```
$ vi build.sbt
import com.typesafe.sbt.SbtNativePackager._

name := """play2-deploy-81"""

version := "0.0.1-SNAPSHOT"

scalaVersion := "2.11.4"

dockerRepository := Some("ginduc")

dockerExposedPorts := Seq(9000)

dockerEntrypoint := Seq("bin/play2-deploy-81", "-J-
javaagent:conf/instrument/newrelic.jar")

libraryDependencies ++= Seq(
 jdbc,
 anorm,
 "org.webjars" % "jquery" % "2.1.1",
 "org.webjars" % "bootstrap" % "3.3.1"
)

lazy val root = (project in file(".")).enablePlugins(PlayScala)
```

13. Create the Docker image using activator:

```
activator clean docker:stage
```

14. Log in to Docker from your local development machine using your Docker Hub credentials:

```
$ docker login
```

15. Build the image and upload to [hub.docker.com](https://hub.docker.com) using activator:

```
activator docker:publish
```

16. Pull the play2-deploy-81 Docker image from [hub.docker.com](https://hub.docker.com) into the virtual machine we will be deploying the web application on:

```
docker pull <YOUR_DOCKERHUB_USERNAME>/play2-deploy-81:0.0.1-SNAPSHOT
```

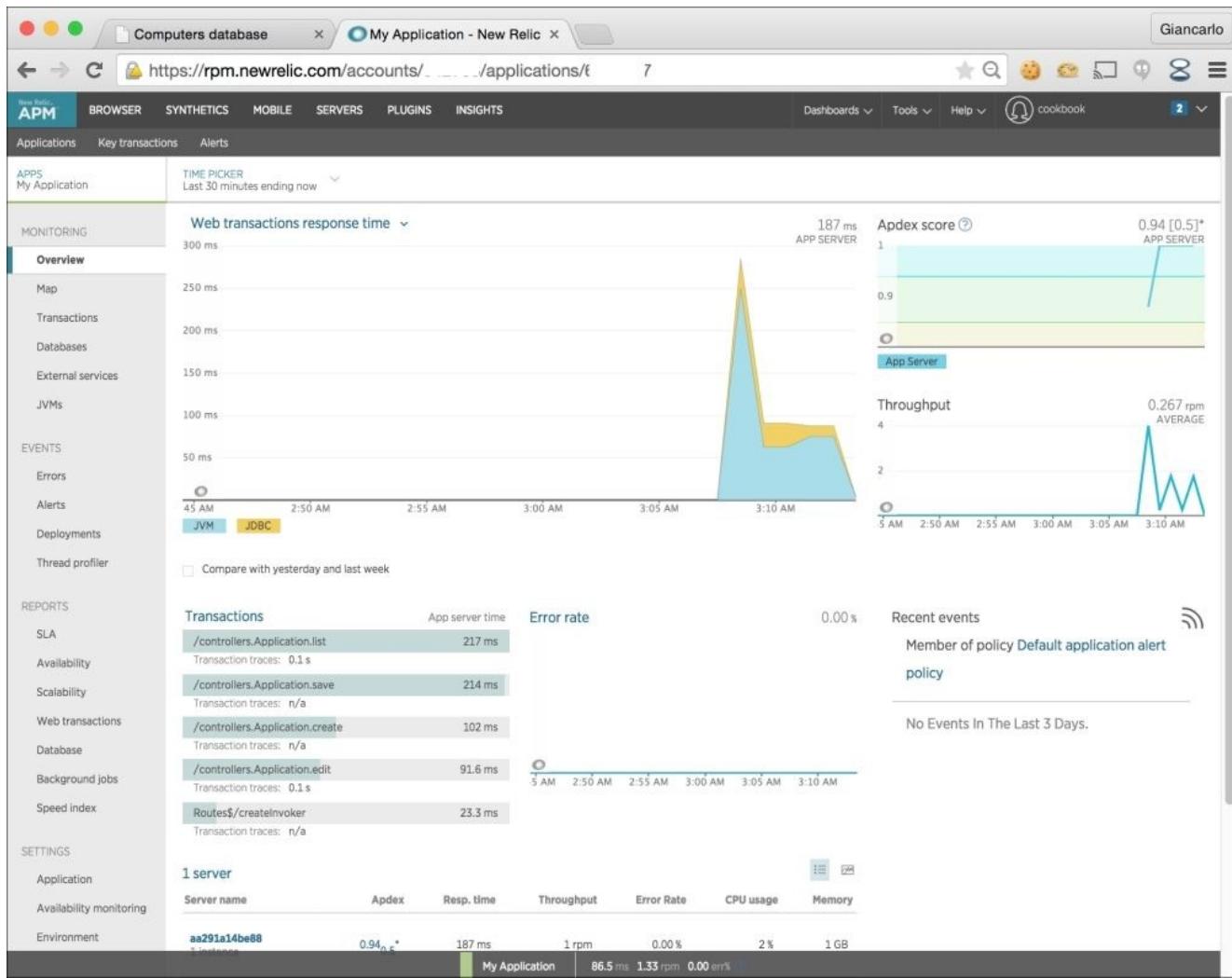
17. Run the play2-deploy-81 Docker container in the same remote virtual machine:

```
docker run -d -p 9000:9000 ginduc/play2-deploy-81:0.0.1-SNAPSHOT
```

18. Using a web browser, you should now be able to access our just-deployed computer database web application:

Computer name	Introduced	Discontinued	Company
ACE	-	-	-
Acer Extensa	-	-	-
Acer Extensa 5220	-	-	-
Acer Iconia	-	-	-
Acorn Archimedes	-	-	Acorn computer
Acorn Electron	-	-	-
Acorn System 2	-	-	-
Alex eReader	-	-	-
Altair 8800	19 Dec 1974	-	Micro Instrumentation and Telemetry Systems
Amiga	01 Jan 1985	-	Amiga Corporation

19. Now, log in to your New Relic account and navigate to your application's dashboard. You should be able to see some relevant application statistics in the form of charts and graphs:



# How it works...

In this recipe, we deployed a Play 2.0 web application in a remote **virtual machine (vm)**. For the virtual machine, we used CoreOS version 607.0.0 as the base operating system, which should automatically install Docker:

```
$ docker -v
Docker version 1.5.0, build a8a31ef-dirty
```

Once the deployment VM was set up, we turned our attention to setting up our New Relic account. After creating the account, we downloaded the New Relic Java agent and made a special note of our account license key with New Relic. We will use both next, as we integrate the New Relic java agent with our Play web application:

```
$ unzip newrelic-java-3.15.0.zip
Archive: newrelic-java-3.15.0.zip
 creating: newrelic/
 inflating: newrelic/newrelic.jar
 inflating: newrelic/LICENSE
 inflating: newrelic/README.txt
 inflating: newrelic/extension.xsd
 inflating: newrelic/nrcerts
 inflating: newrelic/extension-example.xml
 inflating: newrelic/CHANGELOG
 inflating: newrelic/newrelic.yml
 inflating: newrelic/newrelic-api.jar
 inflating: newrelic/newrelic-api-sources.jar
 inflating: newrelic/newrelic-api-javadoc.jar
```

We used the activator template computer-database-scala for this recipe as our sample Play web application.

Once we have our web project generated, we will place the two New Relic config files inside the conf/instrument directory in our project root:

```
$ ls conf/instrument
newrelic.jar
newrelic.yml
```

To load the native Docker packager, we need to add the sbt-native-packager plugin to our build plugins file in project/plugins.sbt:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.0.0-M3")
```

The last step is to configure our main build file build.sbt, to specify the docker-specific settings to create the image:

```
maintainer := "ginduc <ginduc@dynamicobjx.com>"

dockerRepository := Some("dynobjx")

dockerExposedPorts := Seq(9000)

dockerEntrypoint := Seq("bin/imapi", "-J-
javaagent:conf/instrument/newrelic.jar")
```

With the preceding settings, we specify the default repository in [hub.docker.com](https://hub.docker.com) and the main port number we will be exposing our app in (in this recipe, port number 9000). The final setting is where we specify the `entrypoint` command. We had to modify it to pass in the necessary settings to specify the New Relic java agent:

```
dockerEntrypoint := Seq("bin/imapi", "-J-
javaagent:conf/instrument/newrelic.jar")
```

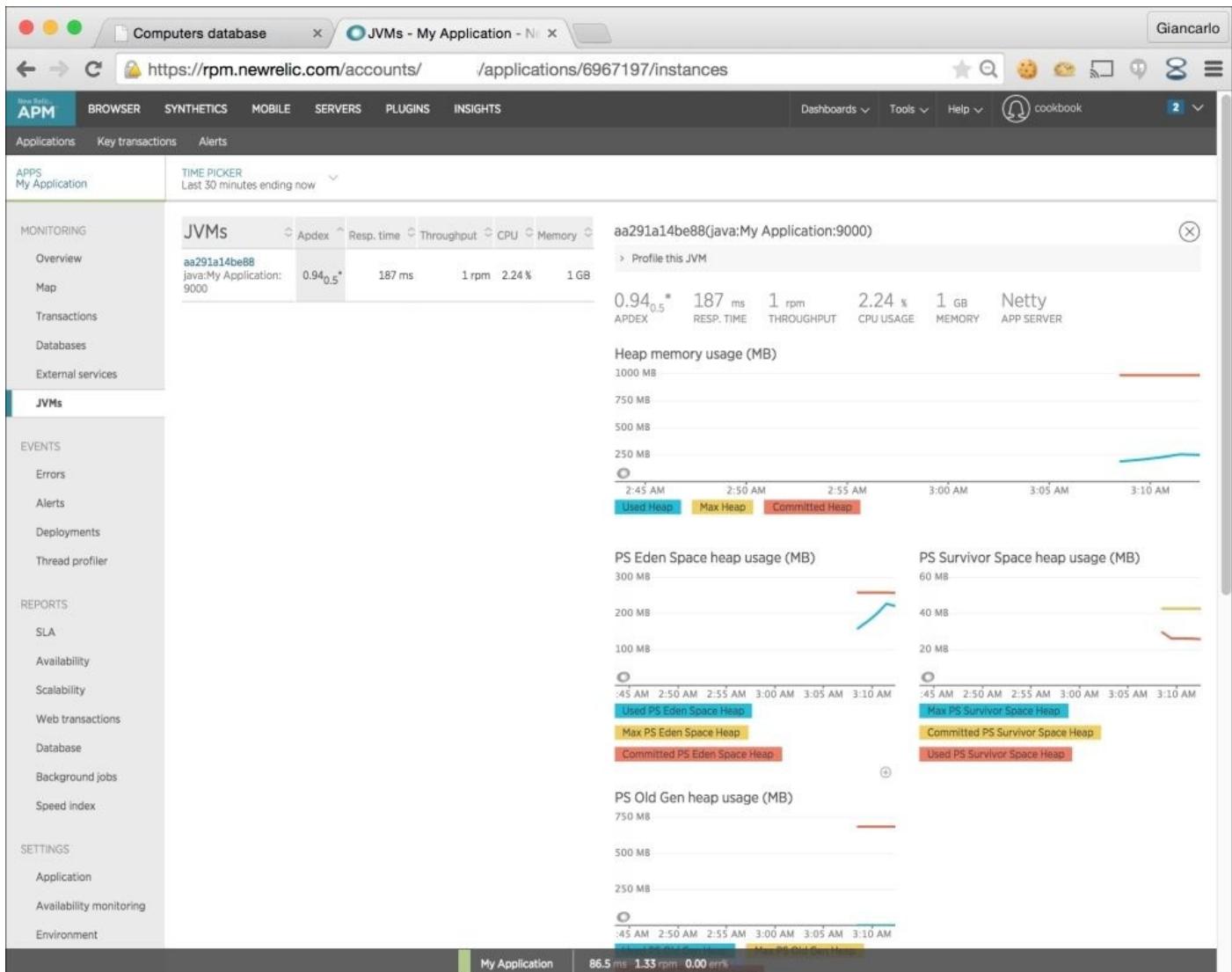
The generated Dockerfile from the preceding settings should look like this:

```
$ cat target/docker/Dockerfile
FROM dockerfile/java:latest
MAINTAINER ginduc <ginduc@dynamicobjx.com>
EXPOSE 9000
ADD files /
WORKDIR /opt/docker
RUN ["chown", "-R", "daemon", "."]
USER daemon
ENTRYPOINT ["bin/imapi", "-J-javaagent:conf/instrument/newrelic.jar"]
CMD []
```

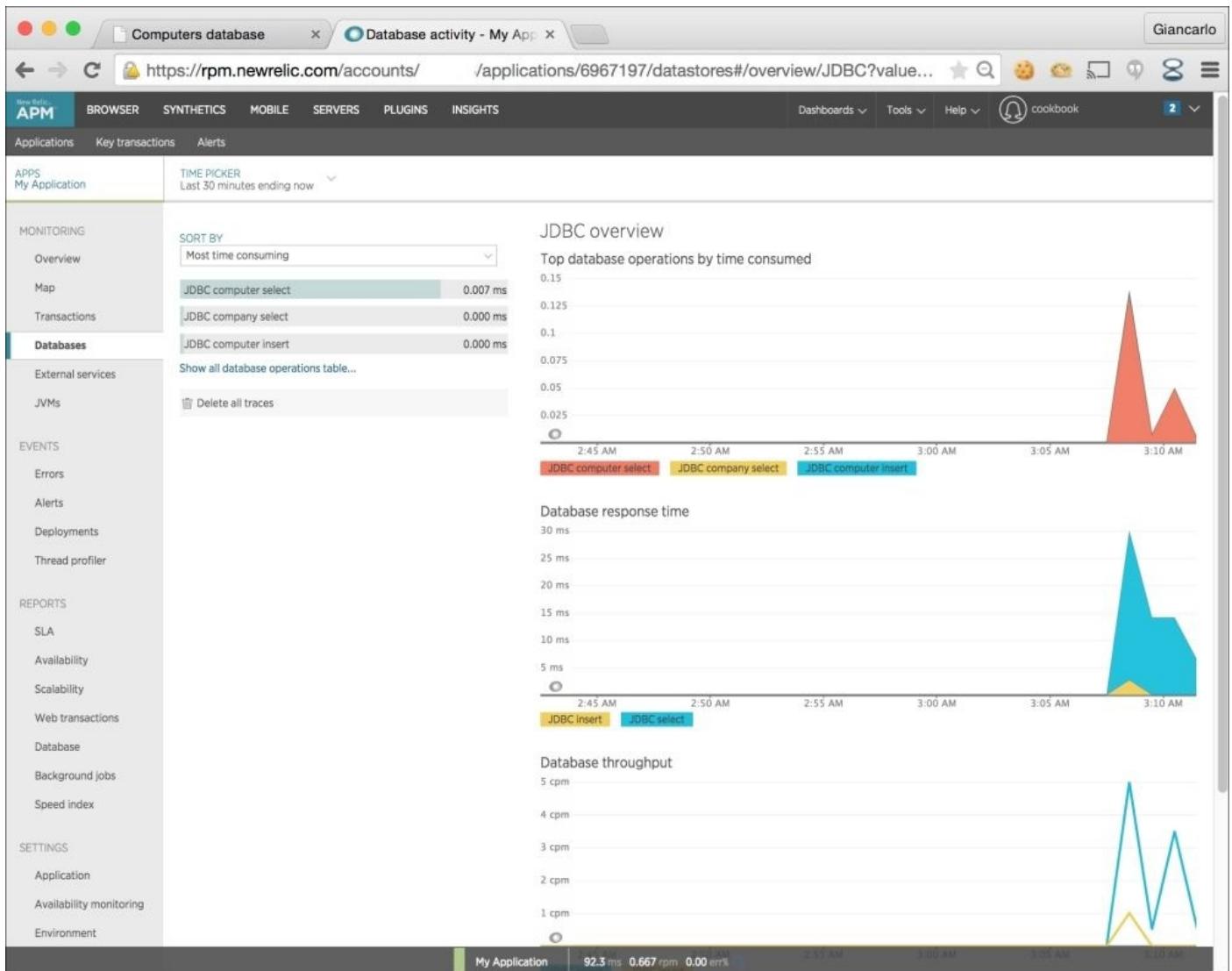
We can verify that the New Relic java agent is loaded currently, by viewing the Docker logs as we run our Docker container:

```
$ docker logs
9790caf8046c7da1d561dcc6e221169d64fa125cdd0a689222fe31637c7bc234
Mar 30, 2015 13:54:11 +0000 [1 1] com.newrelic INFO: New Relic Agent:
Loading configuration file "/opt/docker/conf/instrument/.newrelic.yml"
Mar 30, 2015 13:54:11 +0000 [1 1] com.newrelic INFO: New Relic Agent:
Writing to log file: /opt/docker/lib/logs/newrelic_agent.log
Play server process ID is 1
[info] play - database [default] connected at jdbc:h2:mem:play
[info] play - Application started (Prod)
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
```

Once we have installed and deployed the Docker container in our deployment VM, we can access the computer-database-scala web application using a web browser. We can then view all relevant instrumentation data points, such as JVM and database metrics, with charts and graphs, using the New Relic dashboard:



Here, we can view how in-depth New Relic's application metrics are, with reporting views for the JVM, database connections, and so on:





# Integrating a Play application with AngularJS

For this recipe, we will integrate a Play web application with an AngularJS-based frontend. AngularJS is a popular JavaScript framework and provides developers with tools to build powerful interactive UIs with ease. Some familiarity with AngularJS is assumed for this recipe.

More information about AngularJS can be found here at <https://angularjs.org/>.

In this recipe, we will also use WebJars, a Play-friendly dependency management repository, to manage our AngularJS libraries.

More information about WebJars can be found at <http://www.webjars.org/>.

We will also be using RequireJS, which is a JavaScript module script loader, to manage the AngularJS module and `public/javascripts/main.js`, our main application JavaScript module. For more information about RequireJS, refer to their online documentation at <http://requirejs.org/>.

# How to do it...

For this recipe, you need to perform the following steps:

1. Create a new Play 2 web application project by using the activator template play-scala:

```
activator new play2-angular-83 play-scala && cd play2-angular-83
```

2. Edit the build.sbt build file to import RequireJS, AngularJS, and Bootstrap:

```
libraryDependencies ++= Seq(
 "org.webjars" %% "webjars-play" % "2.3.0",
 "org.webjars" % "angularjs" % "1.3.4",
 "org.webjars" % "bootstrap" % "3.3.1" exclude("org.webjars",
"jquery"),
 "org.webjars" % "requirejs" % "2.1.15" exclude("org.webjars",
"jquery")
)
```

3. Edit the contents of the default application controller file app/controllers/Application.scala, and replace the contents with the following snippet:

```
package controllers

import play.api._
import play.api.libs.json.Json
import play.api.mvc._

case class Product(sku: String, title: String)

object Application extends Controller {
 implicit val productWrites = Json.writes[Product]

 def index = Action {
 Ok(views.html.index())
 }

 val products = Seq(
 Product("733733-421", "HP ProLiant DL360p Gen8"),
 Product("7147H2G", "IBM System x x3690 X5"),
 Product("R630-3552", "DELL PowerEdge R630"),
 Product("RX-2280I", "Supermicro RTG RX-2280I"),
 Product("MB449D/A", "Apple Xserve")
)

 def listProducts = Action {
 Ok(Json.toJson(products))
 }
}
```

4. Edit the contents of the routes file conf/routes, and replace with the following snippet:

```

Routes
GET / controllers.Application.index
GET /assets/*file
controllers.Assets.versioned(path="/public", file: Asset)
 GET /api/products
controllers.Application.listProducts
 GET /webjars/*file
controllers.WebJarAssets.at(file)

```

5. Edit the contents of the default index template HTML file

app/views/index.scala.html and replace with the following snippet:

```

@main("Product Catalogue") {
 <nav class="navbar navbar-inverse navbar-fixed-top"
role="navigation">
 <div class="container-fluid">
 <div class="navbar-header">
 <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#navbar" aria-expanded="false" aria-
controls="navbar">
 Toggle navigation

 </button>
 Product Catalogue
 </div>
 </div>
</nav>

<div class="container-fluid">
 <div class="row">
 <ng-view />
 </div>
</div>
}

```

6. Edit the contents of the default layout template file app/views/main.scala.html and replace with the following snippets:

```

@(title: String)(content: Html)<!DOCTYPE html>

<html>
<head>
 <meta charset="utf-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-
scale=1">
 <meta name="description" content="">
 <meta name="author" content="">
 <title>@title</title>
 <link rel="shortcut icon" type="image/png"
href='@routes.Assets.versioned("images/favicon.png")'>
 <link rel="stylesheet" media="screen"
href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.cs
s"))' />

```

```

<style>
 body {
 padding-top: 50px;
 }
</style>
</head>
<body>
 @content
 <script data-
main='@routes.Assets.versioned("javascripts/main.js").url'
src='@routes.WebJarAssets.at(WebJarAssets.locate("require.min.js")).url
'></script>
</body>
</html>

```

- Add the main JavaScript file for our web application in public/javascripts/main.js and add the following snippet:

```

'use strict';

requirejs.config({
 paths: {
 'angular': ['../lib/angularjs/angular'],
 'angular-route': ['../lib/angularjs/angular-route'],
 'angular-resource': ['../lib/angularjs/angular-resource.min']
 },
 shim: {
 'angular': {
 exports : 'angular'
 },
 'angular-route': {
 deps: ['angular'],
 exports : 'angular'
 },
 'angular-resource': {
 deps: ['angular'],
 exports : 'angular'
 }
 }
});

require([
 'angular',
 'angular-route',
 'angular-resource',
 './services',
 './controllers'
],
function(angular) {
 angular.module('azApp', [
 'ngRoute',
 'ngResource',
 'azApp.services',
 'azApp.controllers'
])
})

```

```

 .config(['$routeProvider', '$locationProvider', '$httpProvider',
function($routeProvider, $locationProvider, $httpProvider) {
 $routeProvider
 .when('/', {
 templateUrl: 'assets/javascripts/partials/products.html',
 controller: 'ProductsCtrl'
 })
 }]);
}

angular.bootstrap(document, ['azApp']);
});

```

8. Next, we add the Angular controller JavaScript file in public/javascripts/controllers.js with the following contents:

```

'use strict';

define(['angular'], function(angular) {
 angular.module('azApp.controllers', [])

 .controller('ProductsCtrl', ['$scope', 'Products', function
($scope, Products) {
 $scope.products = Products.list().query();
 }])
;

});

```

9. After adding the angular controllers file, we add the Angular factory JavaScript file in public/javascript/services.js with the following contents:

```

'use strict';

define(['angular'], function(angular) {
 angular.module('azApp.services', [])
 .factory('Products', ['$resource', '$http', function
Contacts($resource, $http) {
 var endpointURI = '/api/products';

 return {
 list: function(options) {
 return $resource(endpointURI);
 }
 }
 }])
;
});

```

10. Finally, we add the products partial HTML file in public/javascripts/partials/products.html with the following contents:

```

<div class="table-responsive">
 <table class="table table-striped table-hover">
 <thead>
 <th>Product Title</th>
 <th>SKU</th>

```

```

 </thead>
 <tbody>
 <tr ng-repeat="p in products | orderBy:'title'">
 <td ng-bind="p.title"></td>
 <td ng-bind="p.sku"></td>
 </tr>
 </tbody>
</table>
</div>

```

11. We can now execute the activator command run to start the Play 2 web application:

```

$ activator ~run
[info] Loading project definition
[info] Set current project to play2-angular-83
--- (Running the application, auto-reloading is enabled) ---
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000
(Server started, use Ctrl+D to stop and go back to the console...)

[success] Compiled in 354ms

```

12. Using curl, we can verify that our products API endpoint is working correctly:

```

$ curl http://localhost:9000/api/products
[{"sku":"733733-421","title":"HP ProLiant DL360p Gen8"}, {"sku":"7147H2G","title":"IBM System x x3690 X5"}, {"sku":"R630-3552","title":"DELL PowerEdge R630"}, {"sku":"RX-2280I","title":"Supermicro RTG RX-2280I"}, {"sku":"MB449D/A","title":"Apple Xserve"}]%

```

13. We can now access our Angular-driven products listing page with the Play 2 backed API endpoint by loading the URL <http://localhost:9000> in a web browser:

Product Title	SKU
Apple Xserve	MB449D/A
DELL PowerEdge R630	R630-3552
HP ProLiant DL360p Gen8	733733-421
IBM System x x3690 X5	7147H2G
Supermicro RTG RX-2280I	RX-2280I

# How it works...

In this recipe, we created a Play 2 web application that used AngularJS and Bootstrap to display a listing of products. The list of products was served by a Play 2-based Rest API endpoint, which returns a set of products that contain a product title and SKU.

To wire up everything together, we had to modify a few configuration settings to the base play-scala activator template and add new JavaScript files that contained our main AngularJS script.

1. First, we had to declare that our web application required AngularJS, RequireJS, and Bootstrap by modifying our library dependencies in the build.sbt file:

```
libraryDependencies ++= Seq(
 "org.webjars" %% "webjars-play" % "2.3.0",
 "org.webjars" % "angularjs" % "1.3.4",
 "org.webjars" % "bootstrap" % "3.3.1" exclude("org.webjars",
"jquery"),
 "org.webjars" % "requirejs" % "2.1.15" exclude("org.webjars",
"jquery")
)
```

2. Next, we modified the application controller to add a product case class and the listProducts action, which will serve our products API endpoint:

```
case class Product(sku: String, title: String)

def listProducts = Action {
 Ok(Json.toJson(products))
}
```

3. Next, we need to modify our routes file to declare new routes and reconfigure an existing route:

```
GET /assets/*file
controllers.Assets.versioned(path="/public", file: Asset)
GET /api/products
controllers.Application.listProducts
GET /webjars/*file
controllers.WebJarAssets.at(file)
```

4. In the preceding snippet, we reconfigured the existing /assets/\*file route by using the versioned action instead. We then added the products API endpoint route and the route entry for the WebJars assets.
5. Next, we need to make modifications to the existing app/views/index.scala.html template to insert the Angular View tag to render partial HTMLs:

```
@main("Product Catalogue") {

 <div class="container-fluid">
 <div class="row">
 <ng-view />
 </div>
 </div>
```

```
}
```

6. The next step would be to modify the layouts template file

app/views/main.scala.html to load our main JavaScript file and its dependencies:

```
@(title: String)(content: Html)<!DOCTYPE html>

<html>
<head>
 <meta charset="utf-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-
scale=1">
 <meta name="description" content="">
 <meta name="author" content="">
 <title>@title</title>
 <link rel="shortcut icon" type="image/png"
href='@routes.Assets.versioned("images/favicon.png")'>
 <link rel="stylesheet" media="screen"
href='@routes.WebJarAssets.at(WebJarAssets.locate("css/bootstrap.min.cs
s"))' />
 <style>
 body {
 padding-top: 50px;
 }
 </style>
</head>
<body>
 @content
 <script data-
main='@routes.Assets.versioned("javascripts/main.js").url'
src='@routes.WebJarAssets.at(WebJarAssets.locate("require.min.js")).url
'></script>
</body>
</html>
```

7. We then need to add our main JavaScript file in public/javascripts/main.js to configure our main Angular app:

```
'use strict';

requirejs.config({
 paths: {
 'angular': ['../lib/angularjs/angular'],
 'angular-route': ['../lib/angularjs/angular-route'],
 'angular-resource': ['../lib/angularjs/angular-resource.min']
 },
 shim: {
 'angular': {
 exports : 'angular'
 },
 'angular-route': {
 deps: ['angular'],
 exports : 'angular'
 },
 'angular-resource': {
```

```

 deps: ['angular'],
 exports : 'angular'
 }
}
});

require([
 'angular',
 'angular-route',
 'angular-resource',
 './services',
 './controllers'
],
function(angular) {
 angular.module('azApp', [
 'ngRoute',
 'ngResource',
 'azApp.services',
 'azApp.controllers'
])
 .config(['$routeProvider', '$locationProvider', '$httpProvider',
 function($routeProvider, $locationProvider, $httpProvider) {
 $routeProvider
 .when('/', {
 templateUrl: 'assets/javascripts/partials/products.html',
 controller: 'ProductsCtrl'
 })
 }]);
 angular.bootstrap(document, ['azApp']);
});

```

8. In the preceding snippet, we initialized Angular and two other Angular plugins, angular-routes and angular-resources, which will handle request routes and manage API calls respectively. We also loaded and initialized our Angular controllers and services script files:

```

require([
 'angular',
 'angular-route',
 'angular-resource',
 './services',
 './controllers'
])

```

9. Lastly, we configured our Angular app routes using the \$routeProvider directive. For this recipe, the base URL loads the products controller by default, using the template partial public/javascripts/partials/products.html:

```

$routeProvider
 .when('/', {
 templateUrl: 'assets/javascripts/partials/products.html',
 controller: 'ProductsCtrl'
 })

```

})])

For this recipe, we were successfully able to integrate AngularJS with our Play 2 web application using the WebJars repository to manage all frontend libraries (for this recipe, Angular, RequireJS, and Bootstrap). We were able to access a products API endpoint and display its contents in an Angular template.



# Integrating a Play application with Parse.com

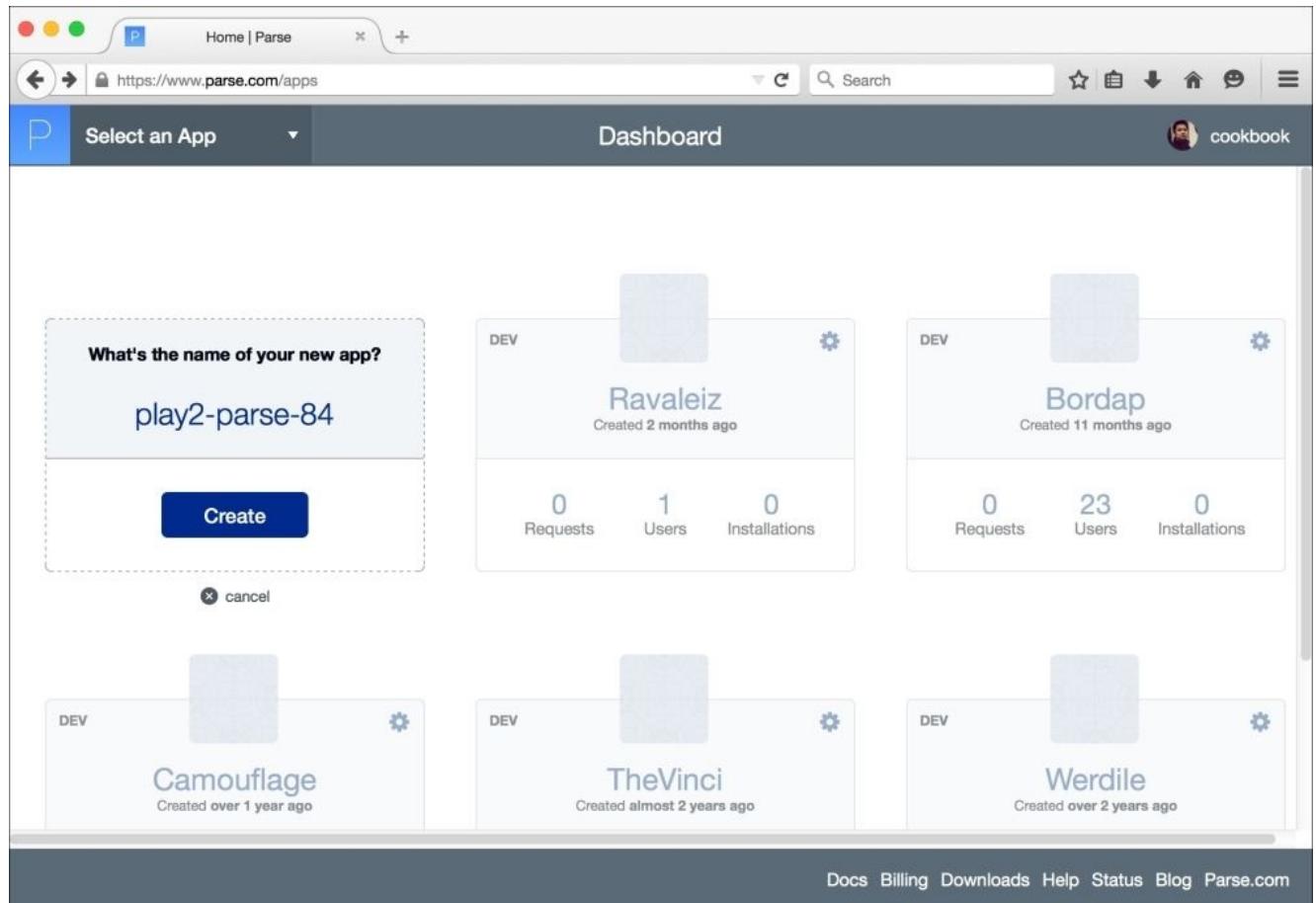
For this recipe, we will integrate a Play 2 Rest API with a BaaS such as [Parse.com](#)'s Parse Core, which is a cloud service that allows developers to store data in the cloud. In this recipe, we want to be able to see how we can use Play to integrate other external web services into our web application. It should not be uncommon for modern web applications to have more than one data source. We will use a Parse Core application to mimic this. We will use the Play WS library to connect to the Parse API, particularly using HTTP headers to send application credentials and JSON data to the Parse API web service. We will also be able to use [Parse.com](#)'s own core data browser to view the data we have stored in our Parse Core application.

For this recipe, we will need a [Parse.com](#) account. You can sign up for one at <https://parse.com/#signup>.

# How to do it...

For this recipe, you need to perform the following steps:

1. Sign in to your [Parse.com](https://www.parse.com) account and create a new Parse Core application:



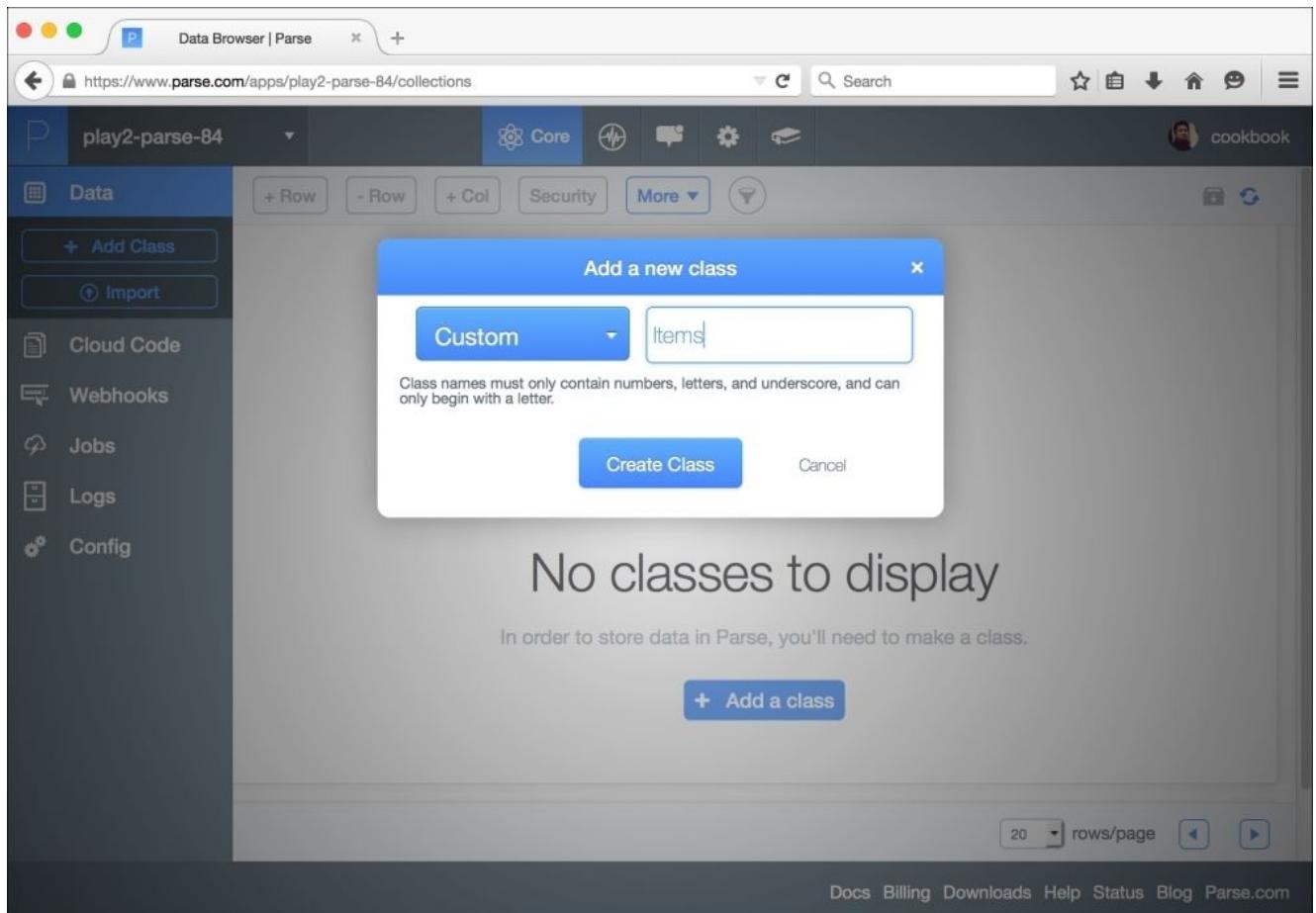
2. Once a new Parse app has been created, navigate to the **settings** section to retrieve your **application ID** and **Rest API key**:

The screenshot shows the 'Edit Your App | Parse' interface for the app 'play2-parse-84'. The left sidebar has 'General' selected under 'Keys'. The main area displays 'Application Keys' with five entries: Application ID (abc123), Client Key (def456), JavaScript Key (ghi789), .NET Key (jk1123), and REST API Key (pqr789). The 'Application ID' and 'REST API Key' fields are highlighted with red boxes. A 'Copy' button is next to each key value.

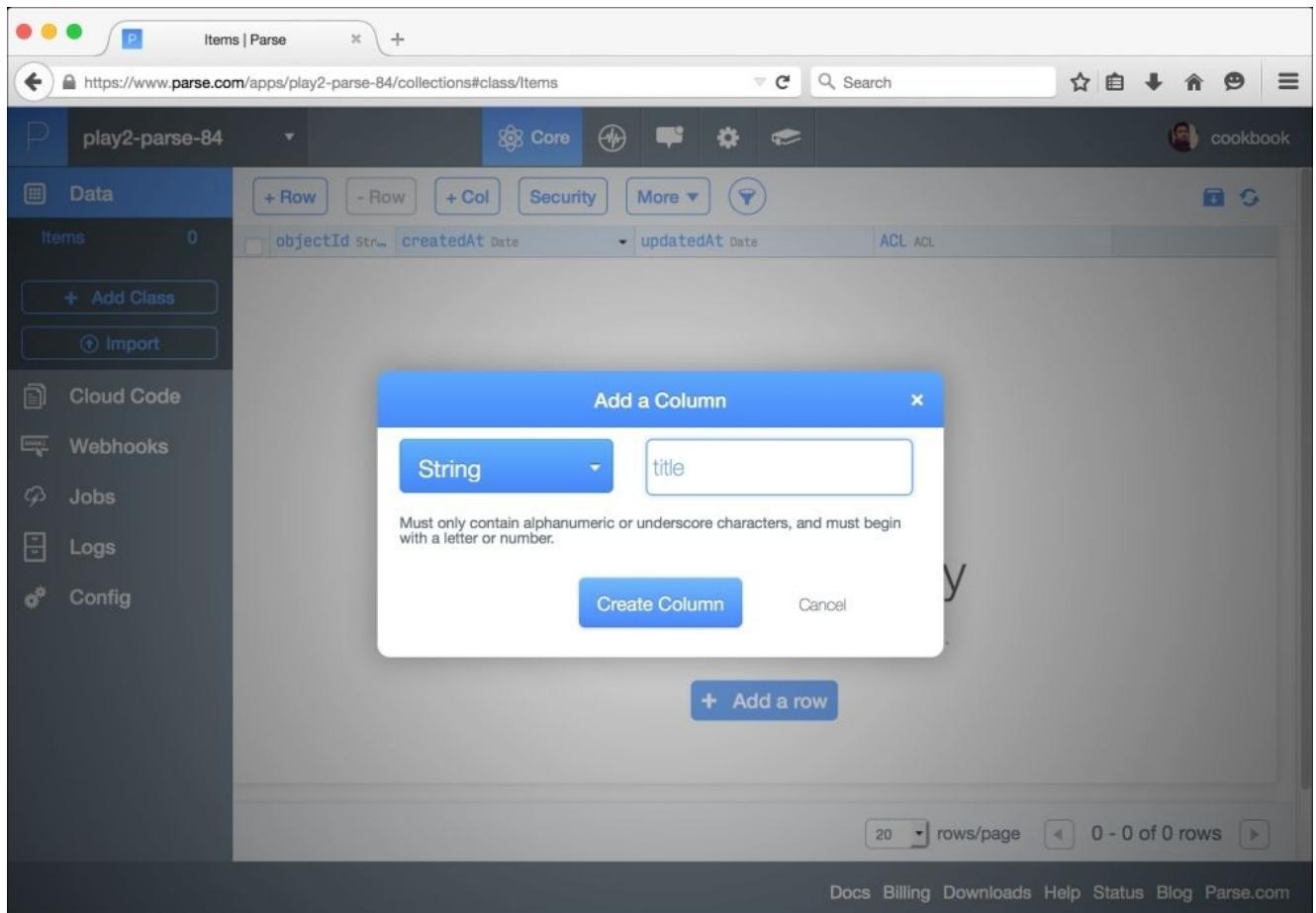
Key Type	Value	Action
Application ID	abc123	Copy
Client Key	def456	Copy
JavaScript Key	ghi789	Copy
.NET Key	jk1123	Copy
REST API Key	pqr789	Copy

Docs Billing Downloads Help Status Blog Parse.com

3. Next, create a new Parse class (analogous to a database table) in the Parse Core section. For this recipe, we will store **Item** records:



4. After creating the Parse class, add the necessary columns for our items. For this recipe, we will add the **title** and **SKU** columns:



5. Next, we will work on our Play 2 web application that will interface with Parse Core. Generate a new Play 2 web application based on the activator template play-scala:

```
activator new play2-parse-84 play-scala && cd play2-parse-84
```

6. Next, create a new plugin in app/plugins/ParsePlugin.scala, with the following content:

```
package plugins

import play.api.{Application, Plugin}

class ParsePlugin(app: Application) extends Plugin {
 lazy val parseAPI: ParseAPI = {
 new ParseAPI(
 app.configuration.getString("parse.core.appId").get,
 app.configuration.getString("parse.core.restKey").get
)
 }

 override def onStart() = {
 parseAPI
 }
}
```

7. Next, add the Parse Core keys, which we made note of in the previous step, in the conf/application.conf file. Make sure to replace the placeholders with your

actual Parse app ID and Rest key:

```
parse.core.appId=<YOUR_PARSE_APP_ID>
parse.core.restKey=<YOUR_PARSE_REST_KEY>
```

8. Next, create our Parse helper class in `app/plugins/ParseAPI.scala`, with the following content:

```
package plugins

import play.api.Play.current
import play.api.libs.json.JsValue
import play.api.libs.ws.WS

class ParseAPI(appId: String, restKey: String) {
 private val PARSE_API_HEADER_APP_ID = "X-Parse-Application-Id"
 private val PARSE_API_HEADER_REST_API_KEY = "X-Parse-REST-API-Key"
 private val PARSE_API_URL = "https://api.parse.com"
 private val PARSE_API_URL_CLASSES = "/1/classes/"
 private val PARSE_API_HEADER_CONTENT_TYPE = "Content-Type"
 private val CONTENT_TYPE_JSON = "application/json; charset=utf-8"

 private val parseBaseUrl = "%s%s".format(PARSE_API_URL,
PARSE_API_URL_CLASSES)

 def list(className: String) = parseWS(className).get()

 def create(className: String, json: JsValue) = {
 parseWS(className)
 .withHeaders(PARSE_API_HEADER_CONTENT_TYPE ->
CONTENT_TYPE_JSON)
 .post(json)
 }

 private def parseWS(className: String) =
WS.url("%s%s".format(parseBaseUrl, className))
 .withHeaders(PARSE_API_HEADER_APP_ID -> appId)
 .withHeaders(PARSE_API_HEADER_REST_API_KEY -> restKey)
}
```

9. Next, initialize the plugin on the app startup by creating the Play plugins configuration file in `conf/play.plugins` with the following content:

```
799:plugins.ParsePlugin
```

10. Finally, let's add our `Items` controller in the `app/controllers/Items.scala` file, with the following contents, which should add two action methods, `index()`, for returning items from Parse Core, and `create()`, which will persist items on Parse Core:

```
package controllers

import play.api.Play
import play.api.Play.current
import play.api.libs.json.{JsError, Json, JsObject}
```

```

import play.api.mvc.{BodyParsers, Action, Controller}
import play.api.libs.concurrent.Execution.Implicits._
import plugins.ParsePlugin
import scala.collection.mutable.ListBuffer
import scala.concurrent.Future

case class Item(objectId: Option[String], title: String, sku: String)

object Items extends Controller {
 private val parseAPI =
 Play.application.plugin[ParsePlugin].get.parseAPI
 implicit val itemWrites = Json.writes[Item]
 implicit val itemReads = Json.reads[Item]

 val `Items` = "Items"

 def index = Action.async { implicit request =>
 parseAPI.list(`Items`).map { res =>
 val list = ListBuffer[Item]()
 (res.json \ "results").as[List[JsObject]].map { itemJson =>
 list += itemJson.as[Item]
 }
 Ok(Json.toJson(list))
 }
 }

 def create = Action.async(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Item]
 post.fold(
 errors => Future.successful {
 BadRequest(Json.obj("error" -> JsError.toJson(errors)))
 },
 item => {
 parseAPI.create(`Items`, Json.toJson(item)).map { res =>
 if (res.status == CREATED) {
 Created(Json.toJson(res.json))
 } else {
 BadGateway("Please try again later")
 }
 }
 }
)
 }
}

```

11. Add the necessary routes entry in the conf/routes file for our Items actions:

GET	/api/items	controllers.Items.index
POST	/api/items	controllers.Items.create

12. To run our web application, we will use the activator command run, with the tilde character (~) to signify that we want Hot-Reloading enabled for this web application:

```
$ activator "~run"
```

```

[info] Loading project definition
[info] Set current project to play2-parse-84

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

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)

[success] Compiled in 400ms
[info] play - Application started (Dev)

```

13. Using curl, we can now insert new records into our Parse Core application:

```

$ curl -v -X POST http://localhost:9000/api/items --header
"Content-type: application/json" -d '{"title":"Supermicro RTG RX-
2280I", "sku":"RX-2280I"}'
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> POST /api/items HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
> Content-type: application/json
> Content-Length: 53
>
* upload completely sent off: 53 out of 53 bytes
< HTTP/1.1 201 Created
< Content-Type: application/json; charset=utf-8
< Content-Length: 64
<
* Connection #0 to host localhost left intact
{"createdAt":"2015-04-08T06:13:52.103Z", "objectId": "K7Q2JEXxmI"}%

```

14. Now, we can also retrieve items stored on Parse Core by using curl:

```

$ curl -v http://localhost:9000/api/items
* Hostname was NOT found in DNS cache
* Trying ::1...
* Connected to localhost (::1) port 9000 (#0)
> GET /api/items HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:9000
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Content-Length: 231
<
* Connection #0 to host localhost left intact
[{"objectId": "0aTdEVaWaF", "title": "DELL PowerEdge
R630", "sku": "R630-3552"}, {"objectId": "T3TqVdi9a2", "title": "HP
ProLiant DL360p Gen8", "sku": "733733-421"}, {"objectId": "K7Q2JEXxmI", "title": "Supermicro RTG RX-2280I", "sku": "RX-
2280I"}]%

```

# How it works...

In this recipe, we looked into a more practical example of integrating an external web service with a Play 2 web application using the Play WS library. The Play WS library saves developers from a lot of boilerplate code in setting up and connecting to a remote HTTP host, it also provides convenient methods to set up headers, request parameters, and so on.

Parse Core is a very popular and solid backend-as-a-service provider, which offers other developer services, such as mobile push notifications and mobile analytics, all of which are important additions to any developer toolchain.

Essential to this recipe is signing up for a [Parse.com](#) account and creating a Parse Core application. Once we have that set up, we can proceed to creating a Play plugin, which will take care of the initialization and setting up of our connection to the Parse API:

```
// app/plugins/ParsePlugin.scala
class ParsePlugin(app: Application) extends Plugin {
 lazy val parseAPI: ParseAPI = {
 new ParseAPI(
 app.configuration.getString("parse.core.appId").get,
 app.configuration.getString("parse.core.restKey").get
)
 }

 override def onStart() = {
 parseAPI
 }
}

// conf/application.conf
parse.core.appId=<YOUR_PARSE_APP_ID>
parse.core.restKey=<YOUR_PARSE_REST_KEY>
```

Next, we create our Parse Core delegate class `conf/plugins/ParseAPI.scala`, which will encapsulate all interfacing with the Parse API:

```
package plugins

import play.api.Play.current
import play.api.libs.json.JsValue
import play.api.libs.ws.WS

class ParseAPI(appId: String, restKey: String) {
 private val PARSE_API_HEADER_APP_ID = "X-Parse-Application-Id"
 private val PARSE_API_HEADER_REST_API_KEY = "X-Parse-REST-API-Key"
 private val PARSE_API_URL = "https://api.parse.com"
 private val PARSE_API_URL_CLASSES = "/1/classes/"
 private val PARSE_API_HEADER_CONTENT_TYPE = "Content-Type"
 private val CONTENT_TYPE_JSON = "application/json; charset=utf-8"

 private val parseBaseUrl = "%s%s".format(PARSE_API_URL,
PARSE_API_URL_CLASSES)
```

```

def list(className: String) = parseWS(className).get()

def create(className: String, json: JsValue) = {
 parseWS(className)
 .withHeaders(PARSE_API_HEADER_CONTENT_TYPE -> CONTENT_TYPE_JSON)
 .post(json)
}

private def parseWS(className: String) =
WS.url("%s%s".format(parseBaseUrl, className))
 .withHeaders(PARSE_API_HEADER_APP_ID -> appId)
 .withHeaders(PARSE_API_HEADER_REST_API_KEY -> restKey)
}

```

In the preceding class, we created two public methods that should have data retrieval and record creation. We include the required Parse API headers for authentication whenever we do a GET or POST request:

```

private def parseWS(className: String) =
WS.url("%s%s".format(parseBaseUrl, className))
 .withHeaders(PARSE_API_HEADER_APP_ID -> appId)
 .withHeaders(PARSE_API_HEADER_REST_API_KEY -> restKey)

```

For the POST request, we add the required additional header to set our content type to application/json:

```

def create(className: String, json: JsValue) = {
 parseWS(className)
 .withHeaders(PARSE_API_HEADER_CONTENT_TYPE -> CONTENT_TYPE_JSON)
 .post(json)
}

```

Once the Parse plugin is all set up, we create the `Items` controller, which will receive the requests for items and item creation and will be responsible for the delegation of these requests to the Parse API helper:

```

def index = Action.async { implicit request =>
 parseAPI.list(`Items`).map { res =>
 val list = ListBuffer[Item]()
 (res.json \ "results").as[List[JsObject]].map { itemJson =>
 list += itemJson.as[Item]
 }
 Ok(Json.toJson(list))
 }
}

def create = Action.async(BodyParsers.parse.json) { implicit request =>
 val post = request.body.validate[Item]
 post.fold(
 errors => Future.successful {
 BadRequest(Json.obj("error" -> JsError.toJson(errors)))
 },
 item => {
 parseAPI.create(`Items`, Json.toJson(item)).map { res =>
 if (res.status == CREATED) {
 Created(Json.toJson(res.json))
 }
 }
 }
)
}

```

```

 } else {
 BadGateway("Please try again later")
 }
 }
}
}

```

Do not forget to add the subsequent routes in the conf/routes config file:

GET	/api/items	controllers.Items.index
POST	/api/items	controllers.Items.create

We can use the Parse Core dashboard to view all data created via the Parse API:

The screenshot shows the Parse Core dashboard interface. The top navigation bar includes tabs for 'Core', 'Cloud Code', 'Webhooks', 'Jobs', 'Logs', and 'Config'. The main area is titled 'play2-parse-84' and displays the 'Data' section for the 'Items' collection. There are three items listed in a table:

	objectId	String	title	String	createdAt	updatedAt	ACL	sku	String
<input type="checkbox"/>	0aTdEVawaF	DELL	PowerEdge R630		Apr 08, 2014	Apr 08, 2014	Public	R630-3552	
<input type="checkbox"/>	T3TqVdi9a2	HP	ProLiant DL360p Gen8		Apr 08, 2014	Apr 08, 2014	Public	733733-421	
<input type="checkbox"/>	K7Q2JEXxmI	Supermicro	RTG RX-2280I		Apr 08, 2014	Apr 08, 2014	Public	RX-2280I	

Below the table, there are buttons for '+ Row', '- Row', '+ Col', 'Security', and 'More'. The bottom right corner of the dashboard includes links for 'Docs', 'Billing', 'Downloads', 'Help', 'Status', 'Blog', and 'Parse.com'.

## **There's more...**

For more information about Parse Core, please refer to their online documentation found at <https://parse.com/docs>.



# Creating a Play development environment using Vagrant

We will explore how to create a portable development environment for Play 2 development using **Vagrant**, a powerful addition to any developer's toolchain. Vagrant allows developers to automate the creation of a development environment, from installing the required development kits for **Read-Eval-Print Loop (REPL)** tools, to installing other services such as MySQL and Redis. This setup is useful for multimember development teams or developers who work off of multiple workstations, where a consistent, identical development environment is necessary and ideal.

For this recipe, we will create our Vagrant instance from the ground up, installing the required libraries to run our sample Play web application, and running a MySQL service using Docker and the actual sample Play web application using the activator template `play-slick-angular-test-example`.

# How to do it...

For this recipe, you need to perform the following steps:

1. Install Vagrant by following the installation instructions at the following link:

<https://docs.vagrantup.com/v2/installation/index.html>

2. You should now have a version of Vagrant installed locally:

```
$ vagrant -v
Vagrant 1.6.3
```

3. Create a workspace directory and change into the newly created directory:

```
mkdir play2-vagrant-85 && cd $_
```

4. Create a `Vagrantfile` in the project root with the following content:

```
-*- mode: ruby -*-
vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
 config.vm.box = "Centos-6.5"
 config.vm.box_url = "https://github.com/2creatives/vagrant-centos/releases/download/v6.5.1/centos65-x86_64-20131205.box"
 config.vm.hostname = "VG-play2"

 config.vm.provision :shell, :path => "bootstrap.sh"

 config.vm.network "forwarded_port", guest: 9000, host: 9000
 config.vm.network "forwarded_port", guest: 3306, host: 3306

 config.vm.provider :virtualbox do |vb|
 vb.name = "VG-play2"
 vb.gui = false
 vb.customize ["modifyvm", :id, "--memory", "4096", "--cpus", "2", "--ioapic", "on"]
 end
end
```

5. Create a Bootstrap bash script file `bootstrap.sh`, with the following content:

```
#!/usr/bin/env bash

set -x

SCALA_VER=2.11.2
ACTIVATOR_VER=1.3.2
MYSQL_ROOT_PW="cookbook"

if [! -e "/home/vagrant/.firstboot"]; then

 # tools etc
 yum -y install yum-plugin-fastestmirror
```

```

yum -y install git wget curl rpm-build

Pre-docker install: http://stackoverflow.com/a/27216873
sudo yum-config-manager --enable public_ol6_latest
sudo yum install -y device-mapper-event-libs

Docker
yum -y install docker-io
service docker start
chkconfig docker on
usermod -a -G docker vagrant

Install the JDK
curl -LO 'http://download.oracle.com/otn-pub/java/jdk/7u51-
b13/jdk-7u51-linux-x64.rpm' -H 'Cookie: oraclelicense=accept-
securebackup-cookie'
rpm -i jdk-7u51-linux-x64.rpm
/usr/sbin/alternatives --install /usr/bin/java java
/usr/java/default/bin/java 200000
rm jdk-7u51-linux-x64.rpm

Install scala repl
rpm -ivh http://scala-lang.org/files/archive/scala-$SCALA_VER.rpm

Install Activator
cd /opt
wget http://downloads.typesafe.com/typesafe-
activator/$ACTIVATOR_VER/typesafe-activator-$ACTIVATOR_VER.zip
unzip typesafe-activator-$ACTIVATOR_VER.zip
chown -R vagrant:vagrant /opt/activator-$ACTIVATOR_VER

Set Path
echo "export JAVA_HOME=/usr/java/default/" >>
/home/vagrant/.bash_profile
echo "export
PATH=$PATH:$JAVA_HOME/bin:/home/vagrant/bin:/opt/activator-
$ACTIVATOR_VER" >> /home/vagrant/.bash_profile

touch /home/vagrant/.firstboot
fi

Start MySQL
docker run -e MYSQL_PASS=$MYSQL_ROOT_PW -i -d -p 3306:3306 --name
mysqld -t tutum/mysql

```

6. Create a git ignore file .gitignore, to exclude the Vagrant directory workspace from git:

```
$ cat .gitignore
.vagrant
```

7. Once Vagrant is installed and our Vagrantfile has been configured properly, we can initialize our Vagrant instance:

```
vagrant up
```

8. We can log in to the Vagrant instance using the following command:

```
vagrant ssh
```

9. Once logged in to the Vagrant instance, change into the Vagrant workspace directory with the following command:

```
cd /vagrant
```

10. List the contents of the /vagrant directory to verify that you are in the correct directory:

```
$ ls -ltra
-rw-r--r-- 1 vagrant vagrant 9 Mar 31 12:48 .gitignore
-rw-r--r-- 1 vagrant vagrant 683 Mar 31 12:50 Vagrantfile
drwxr-xr-x 1 vagrant vagrant 102 Mar 31 12:57 .vagrant
-rwxr-xr-x 1 vagrant vagrant 1632 Apr 1 12:44 bootstrap.sh
```

11. For this recipe, we will use the activator template play-slick-angular-test-example and generate a new Play 2 project based on this:

```
$ activator new play-slick-angular-test-example play-slick-angular-test-example
```

12. Edit the configuration file in conf/application.conf by modifying the following lines of code:

```
db.default.driver=com.mysql.jdbc.Driver
db.default.url="jdbc:mysql://localhost/report?
createDatabaseIfNotExist=true"
db.default.user=admin
db.default.password="cookbook"

applyEvolutions.default=true
```

13. Run the Play web application using activator:

```
$ activator run
```

14. You should now be able to access the Play web application by using a web browser:

Play Slick Angular Test Example x Giancarlo

localhost:9000/#/

Play Slick Angular Test Example Report Old Home

# Report

From:  To:  World:

TOTAL: Detected: 0 Banned: 0 Deleted: 0

World	Date	Detected	Banned	Deleted
-------	------	----------	--------	---------

10 25 50 100

The screenshot shows a web browser window with a title bar 'Play Slick Angular Test Example x' and a user 'Giancarlo'. The address bar shows 'localhost:9000/#/'. Below the title bar, there are links for 'Report' and 'Old Home'. The main content area has a large heading 'Report'. Below it are three input fields: 'From' with value '23.03.2015', 'To' with value '06.04.2015', and 'World' with value 'all'. Underneath these is a summary row with the text 'TOTAL: Detected: 0 Banned: 0 Deleted: 0'. Below this is a table with five columns: 'World', 'Date', 'Detected', 'Banned', and 'Deleted'. Each column has a dropdown arrow indicating it can be sorted. At the bottom right of the table are buttons for page size: '10', '25', '50', and '100'.

# How it works...

In this recipe, we installed Vagrant, a popular developer tool that automates the initialization and setup of developer environments, as per the following steps:

1. In the `Vagrantfile` config file, we declared that we will be using a Centos 6.5 Vagrant box as our base OS:

```
config.vm.box_url = "https://github.com/2creatives/vagrant-centos/releases/download/v6.5.1/centos65-x86_64-20131205.box"
```

2. We declared that we want our `bootstrap.sh` script file to be run during the Vagrant instance provisioning:

```
config.vm.provision :shell, :path => "bootstrap.sh"
```

3. Next, we declared which ports to forward from our vagrant instance to our host machine, port 9000 for Play and port 3306 for MySQL:

```
config.vm.network "forwarded_port", guest: 9000, host: 9000
config.vm.network "forwarded_port", guest: 3306, host: 3306
```

4. Finally, we optionally configured our Vagrant instance to have a 4-GB RAM and utilize two CPUs:

```
vb.customize ["modifyvm", :id, "--memory", "4096", "--cpus", "2",
"--ioapic", "on"]
```

5. We installed relevant tools for Play development, which we specified in the `bootstrap.sh` script file. We declared the version of Scala and Activator at the top of the `bootstrap.sh` file:

```
SCALA_VER=2.11.2
ACTIVATOR_VER=1.3.2
```

6. We also declared the default MySQL password to use for our MySQL instance:

```
MYSQL_ROOT_PW="cookbook"
```

7. Next, we installed the required and necessary CentOS packages:

```
tools etc
yum -y install yum-plugin-fastestmirror
yum -y install git wget curl rpm-build
```

8. The next packages to install are Docker and its required libraries:

```
Pre-docker install: http://stackoverflow.com/a/27216873
sudo yum-config-manager --enable public_ol6_latest
sudo yum install -y device-mapper-event-libs

Docker
yum -y install docker-io
service docker start
chkconfig docker on
usermod -a -G docker vagrant
```

9. Next, we installed the JDK, a Scala binary, and Activator:

```

Install the JDK
curl -LO 'http://download.oracle.com/otn-pub/java/jdk/7u51-b13/jdk-
7u51-linux-x64.rpm' -H 'Cookie: oraclelicense=accept-securebackup-
cookie'
rpm -i jdk-7u51-linux-x64.rpm
/usr/sbin/alternatives --install /usr/bin/java java
/usr/java/default/bin/java 200000
rm jdk-7u51-linux-x64.rpm

Install scala repl
rpm -ivh http://scala-lang.org/files/archive/scala-$SCALA_VER.rpm

Install Activator
cd /opt
wget http://downloads.typesafe.com/typesafe-
activator/$ACTIVATOR_VER/typesafe-activator-$ACTIVATOR_VER.zip
unzip typesafe-activator-$ACTIVATOR_VER.zip
chown -R vagrant:vagrant /opt/activator-$ACTIVATOR_VER

Set Path
echo "export JAVA_HOME=/usr/java/default/" >>
/home/vagrant/.bash_profile
echo "export
PATH=$PATH:$JAVA_HOME/bin:/home/vagrant/bin:/opt/activator-
$ACTIVATOR_VER" >> /home/vagrant/.bash_profile

```

10. Finally, we ran a MySQL Docker container on the instance start up:

```

Start MySQL
docker run -e MYSQL_PASS=$MYSQL_ROOT_PW -i -d -p 3306:3306 --name
mysqld -t tutum/mysql

```

11. We run the Vagrant command `vagrant up`, to initialize the Vagrant instance from scratch. After a short while, our Play 2 development environment should be ready. Log in to the Vagrant instance using the command `vagrant ssh`. You should be able to verify whether all the required binaries have been installed:

```

[vagrant@VG-play2 ~]$ java -version
java version "1.7.0_51"
Java(TM) SE Runtime Environment (build 1.7.0_51-b13)
Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)
[vagrant@VG-play2 ~]$ activator --version
sbt launcher version 0.13.8-M5
[vagrant@VG-play2 ~]$ scala -version
Scala code runner version 2.11.2-Copyright 2002-2013, LAMP/EPFL
[vagrant@VG-play2 ~]$ docker ps
CONTAINER ID IMAGE COMMAND
CREATED STATUS PORTS
NAMES
08ecd4a3d98c tutum/mysql:latest "/run.sh" 21
minutes ago Up 21 minutes 0.0.0.0:3306->3306/tcp
mysqld

```

12. Once the Vagrant instance is up and running, we can build and run a Play web

application; in this recipe, the play-slick-angular-test-example activator template that we installed in the /vagrant directory:

```
cd /vagrant/play-slick-angular-test-example
activator run
[info] Loading project definition from /vagrant/play-slick-angular-
test-example/project
[info] Set current project to
PlaySlickAngularBootstrapH2TestsExample (in build file:/vagrant/play-
slick-angular-test-example/)
[success] Total time: 4 s, completed Apr 6, 2015 2:55:54 PM
[info] Updating {file:/vagrant/play-slick-angular-test-
example/}root...
[info] Resolving jline#jline;2.12...
[info] Done updating.

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

[info] play - Listening for HTTP on /0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

## There's more...

Remember to pause your Vagrant instance whenever you want to shut down the instance for a while and return to it later:

```
vagrant halt
```

This allows the Vagrant instance to retain its current state without having to reinitialize the Vagrant instance later. For more information about Vagrant, refer to the documentation at <https://docs.vagrantup.com/v2/>.



# **Coding Play 2 web apps with IntelliJ IDEA 14**

For this recipe, we will explore how to use the popular IDE, IntelliJ IDEA 14 to code Play 2 web applications. We will be using the community edition:

# How to do it...

For this recipe, you need to perform the following steps:

1. Download and install IntelliJ IDEA 14 from the Jetbrains website:

<https://www.jetbrains.com/idea/download/>

2. Navigate to the Play 2 web application with which you will want to use IDEA 14; in this recipe, play2-parse-84:

```
cd play2-parse-84
```

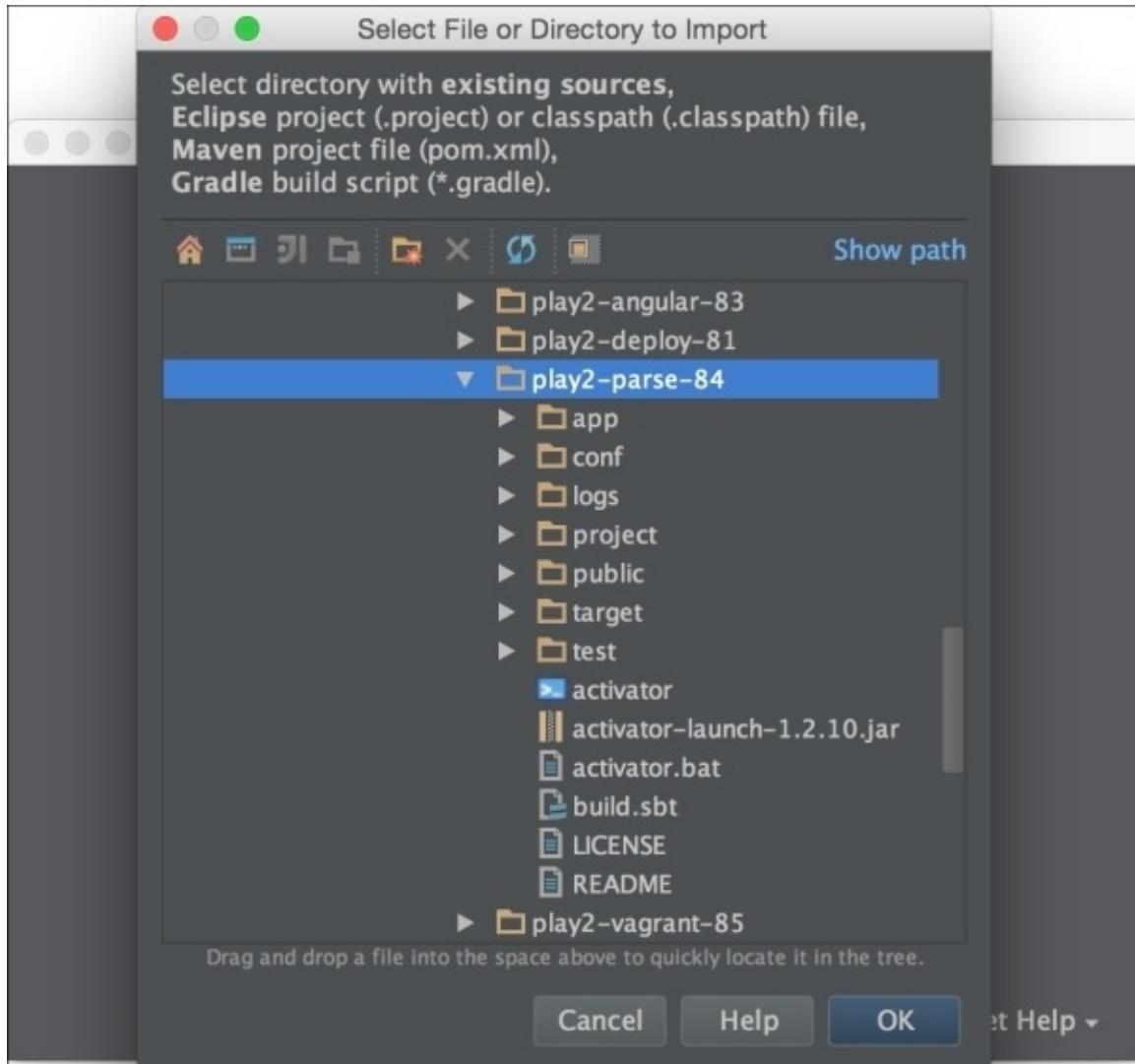
3. Using activator, generate the base IDEA 14 project files:

```
$ activator idea
[info] Creating IDEA module for project 'play2-parse-84' ...
[info] Running compile:managedSources...
[info] Running test:managedSources...
[info] Created /Users/cookbook/play2-parse-84/.idea/IdeaProject.iml
[info] Created /Users/cookbook/play2-parse-84/.idea
[info] Created /Users/cookbook/play2-parse-84.iml
[info] Created /Users/cookbook/play2-parse-84/.idea_modules/play2-
parse-84-build.iml
```

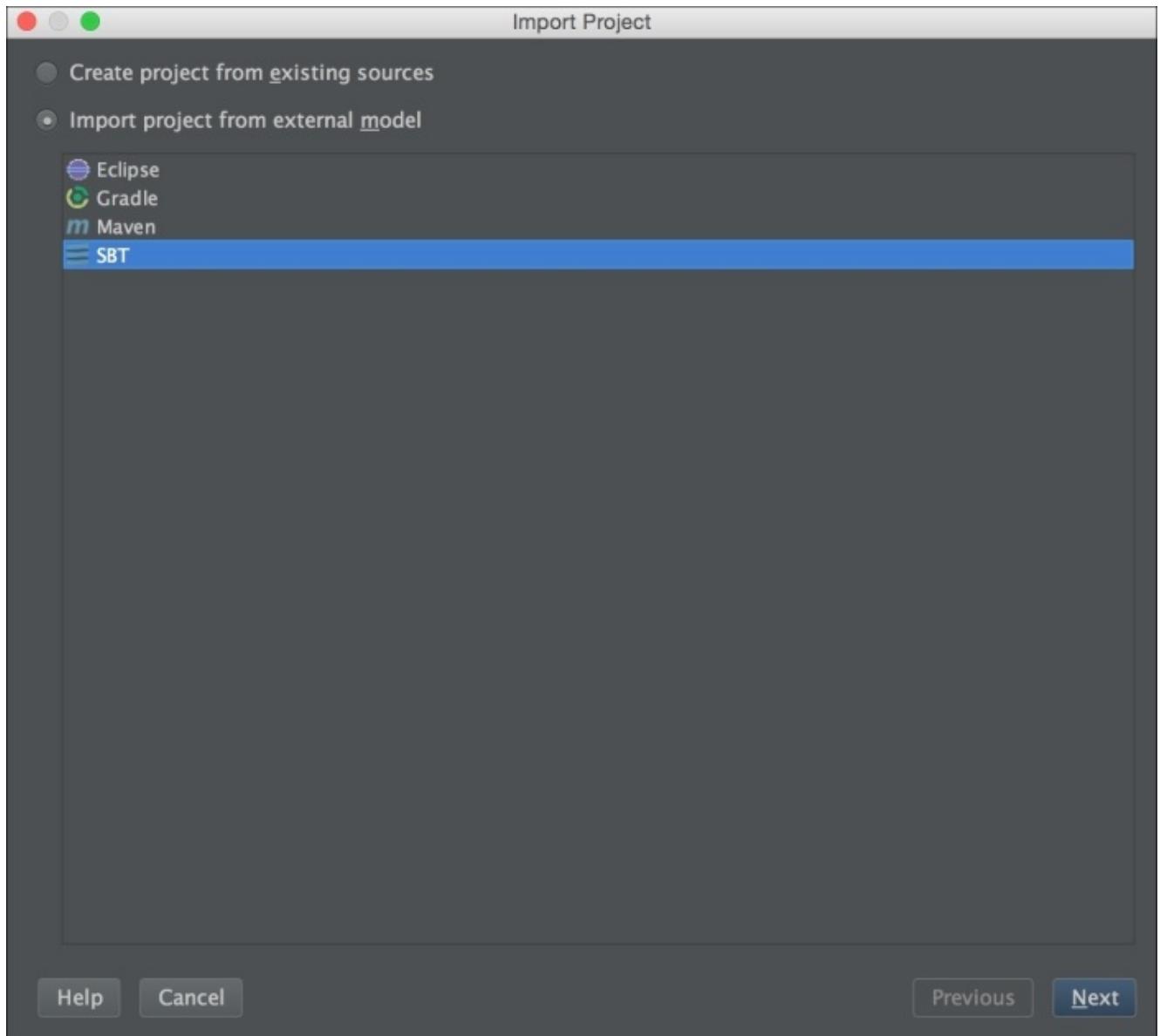
4. Start IntelliJ IDEA 14 and click on **Import Project**:



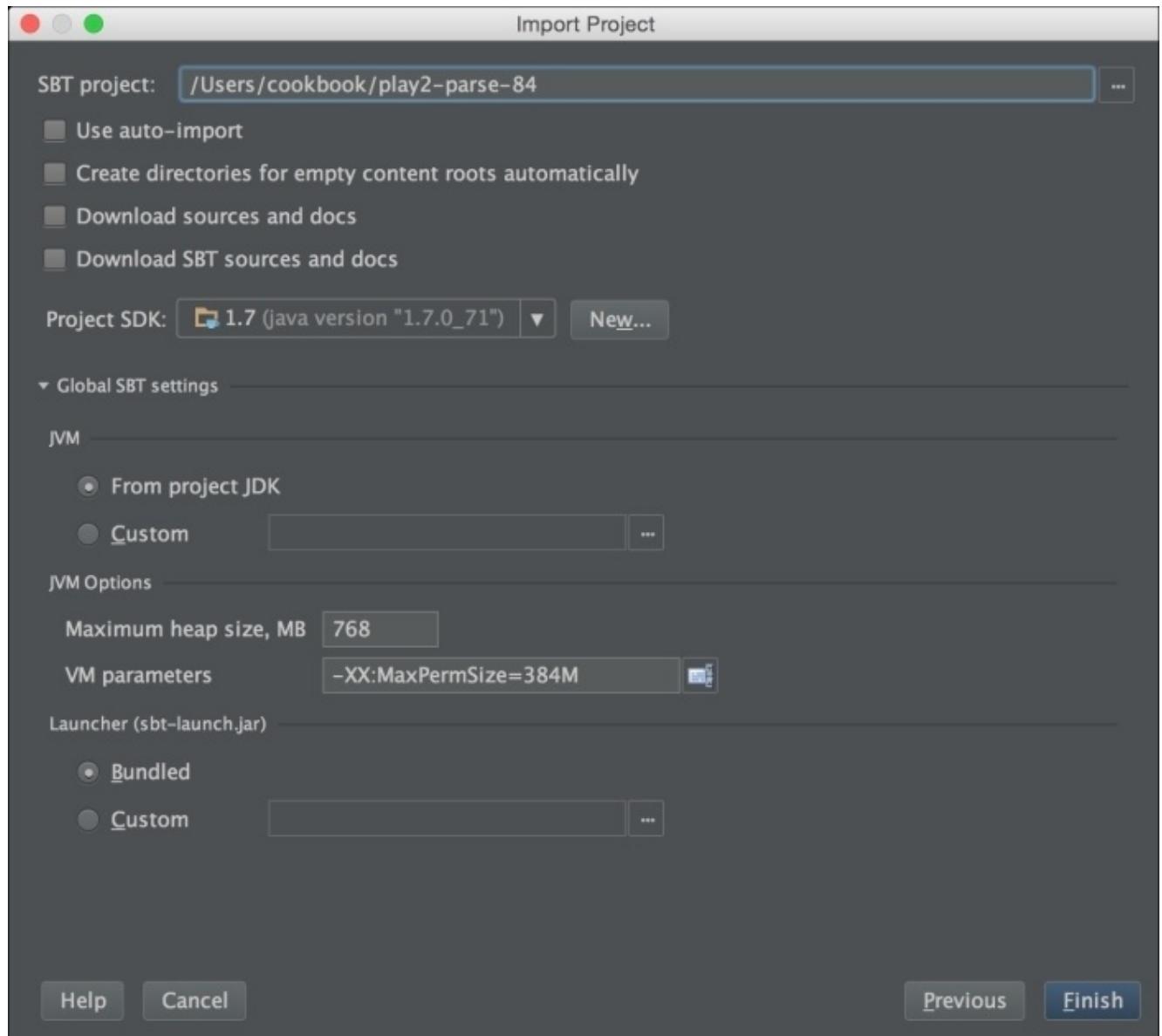
5. Using IDEA 14, navigate to the project directory of the project you want to work on; in this recipe, **play2-parse-84**:



6. On the next screen, select **SBT** as the external model for the project:



7. Next, select additional project settings, such as the installed JDK version to use:



8. After clicking on **Finish**, you should have your Play 2 web application loaded correctly on IntelliJ IDEA 14:

The screenshot shows the IntelliJ IDEA interface with a Scala project named "play2-parse-84". The left sidebar displays the project structure, including the "app" directory which contains "controllers", "models", "views", and "plugins" (with "ParseAPI" and "ParsePlugin" sub-directories). The "conf" directory contains "application.conf", "play.plugins", and "routes". The "test" directory includes ".gitignore", "activator", "activator.bat", "activator-launch-1.2.10.jar", "build.sbt", "LICENSE", and "README". The bottom navigation bar shows tabs for "TODO", "Terminal", and "Changes". The status bar at the bottom right indicates the time as 1:82, encoding as LF, file encoding as UTF-8, and the current branch as Git: master.

```
1 package plugins
2
3 import play.api.Play.current
4 import play.api.libs.json.JsValue
5 import play.api.libs.ws.WS
6
7 class ParseAPI(appId: String, restKey: String) {
8 private val PARSE_API_HEADER_APP_ID = "X-Parse-Application-Id"
9 private val PARSE_API_HEADER_REST_API_KEY = "X-Parse-REST-API-Key"
10 private val PARSE_API_URL = "https://api.parse.com"
11 private val PARSE_API_URL_CLASSES = "/1/classes/"
12 private val PARSE_API_HEADER_CONTENT_TYPE = "Content-Type"
13 private val CONTENT_TYPE_JSON = "application/json; charset=utf-8"
14
15 private val parseBaseUrl = "%s%s".format(PARSE_API_URL, PARSE_API_URL_CLASSES)
16
17 def list(className: String) = parseWS(className).get()
18
19 def create(className: String, json: JsValue) = {
20 parseWS(className)
21 .withHeaders(PARSE_API_HEADER_CONTENT_TYPE -> CONTENT_TYPE_JSON)
22 .post(json)
23 }
24
25 private def parseWS(className: String) = WS.url("%s%s".format(parseBaseUrl, className))
26 .withHeaders(PARSE_API_HEADER_APP_ID -> appId)
27 .withHeaders(PARSE_API_HEADER_REST_API_KEY -> restKey)
28 }
29
```

## How it works...

In this recipe, we simply used Activator's built-in support for IntelliJ IDEA to generate our Play 2 web applications IDEA project files using the command `activator idea`. Once we generated the IDEA project files from our current code base, all we needed to do was import it into IntelliJ IDEA and follow the project settings screens. We should now be able to work on our Play 2 web applications using IntelliJ IDEA.

# Index

## A

- Akka actors
  - using / [Using futures with Akka actors](#), [How to do it...](#), [How it works...](#)
- Amazon
  - URL / [Adding private module repositories using Amazon S3](#)
- Amazon S3
  - URL / [Integrating Play application with Amazon S3](#), [Adding private module repositories using Amazon S3](#)
  - Play application, integrating / [How to do it...](#), [How it works...](#)
  - used, for adding private module repositories / [Adding private module repositories using Amazon S3](#), [How to do it...](#), [How it works...](#)
- Amazon Web Services (AWS) / [Integrating Play application with Amazon S3](#)
- AngularJS
  - integrating with / [Integrating a Play application with AngularJS](#), [How to do it...](#), [How it works...](#)
  - URL / [Integrating a Play application with AngularJS](#)
- Anorm (Scala)
  - using, with MySQL / [Using Anorm \(Scala\) and database evolutions with MySQL](#)
  - record, creating / [Creating a new record](#)
  - record, updating / [Updating a record](#)
  - record, deleting / [Deleting a record](#)
- API endpoints
  - securing, with HTTP basic authentication / [Securing API endpoints with HTTP basic authentication](#), [How to do it...](#), [How it works...](#)
- AWS Elastic Beanstalk
  - deploying / [Deploying a Play application on AWS Elastic Beanstalk](#), [How to do it...](#), [How it works...](#)

# B

- Backend-as-a-Service (BaaS) / [Introduction](#)
- Bootstrap
  - integrating, WebJars used / [Integrating Bootstrap and WebJars](#), [How to do it...](#), [How it works...](#)

# C

- Casbah
  - about / [Utilizing MongoDB](#)
  - URL / [Utilizing MongoDB](#)
- Connect2id
  - URL / [Implementing token authentication using JWT](#)
- controllers
  - about / [Working with controllers and routes](#)
  - using / [How to do it...](#)
  - Action parameters, using / [Using Action parameters in controllers](#), [How to do it...](#)
  - testing / [Testing controllers](#), [How to do it...](#)
- CoreOS
  - deploying on / [Deploying a Play application on CoreOS and Docker](#)
- custom actions
  - using / [Using custom actions](#), [How to do it...](#), [How it works...](#)
- custom plugin
  - creating / [Creating and using your own plugin](#), [How to do it...](#), [How it works...](#)
  - using / [Creating and using your own plugin](#), [How to do it...](#), [How it works...](#)
- Cygwin
  - URL / [Using HTTP headers](#)

# D

- database evolutions
  - URL / [Using Ebean \(Java\) with MySQL](#)
  - using, with MySQL / [Using Anorm \(Scala\) and database evolutions with MySQL](#)
- DELETE API endpoint
  - creating / [Creating a DELETE API endpoint](#), [How to do it...](#), [How it works...](#)
- dependency injection
  - with Spring / [Dependency injection with Spring](#), [How to do it...](#)
  - with Guice / [Dependency injection using Guice](#), [How to do it...](#), [How it works...](#)
- Digital Ocean
  - URL / [There's more...](#), [Deploying a Play application with Dokku](#), [Deploying a Play application with Nginx](#)
- dispatchers, Akka
  - reference link / [How it works...](#)
- Docker
  - URL / [Integrating a Play application with ElasticSearch](#), [How it works...](#),  
[Deploying a Play application on CoreOS and Docker](#), [There's more...](#)
  - deploying on / [Deploying a Play application on CoreOS and Docker](#), [How to do it...](#), [How it works...](#)
- Dokku
  - deploying with / [Deploying a Play application with Dokku](#), [How to do it...](#),  
[How it works...](#)
  - URL / [There's more...](#)

# E

- Ebean (Java)
  - using, with MySQL / [Using Ebean \(Java\) with MySQL](#), [How to do it...](#)
  - about / [Using Ebean \(Java\) with MySQL](#)
  - record, creating / [Creating a record](#)
  - record, updating / [Updating a record](#)
  - record, querying / [Querying a record](#)
  - record, retrieving / [Retrieving a record](#)
- ElasticSearch
  - used, for integrating Play application / [Integrating a Play application with ElasticSearch](#), [How to do it...](#), [How it works...](#)
- environment variables, Oracle
  - URL / [Getting ready](#)
- external web APIs
  - consuming / [Consuming external web APIs](#), [How to do it...](#), [How it works...](#)

# F

- files
  - uploading / [Uploading files, How to do it...](#)
- filters
  - using / [Using filters, How to do it...](#)
- flexible registration module
  - building / [Building a flexible registration module, How to do it..., How it works...](#)
- form submission
  - securing / [Securing form submission, How to do it...](#)
- form template
  - using / [Using a form template and web action, How to do it...](#)
- form validation
  - using / [Using form validation, How to do it..., How it works...](#)
- futures
  - using / [Using futures with Akka actors, How to do it..., How it works...](#)

# G

- GET API endpoint
  - creating / [Creating a GET API endpoint](#), [How to do it...](#), [How it works...](#)
- Git
  - URL / [There's more...](#)
- GridFS
  - utilizing / [Utilizing MongoDB and GridFS](#), [How to do it...](#), [How it works...](#)
- Guice
  - dependency injection / [Dependency injection using Guice](#), [How to do it...](#), [How it works...](#)

# H

- helper tags
  - about / [Using helper tags](#)
  - using / [Using helper tags](#), [How to do it...](#), [How it works...](#)
- Heroku
  - Play application, deploying / [Deploying a Play application on Heroku](#), [How to do it...](#)
  - URL / [How to do it...](#)
- Heroku CLI tools
  - URL / [How to do it...](#)
- HTTP basic authentication
  - API endpoints, securing with / [Securing API endpoints with HTTP basic authentication](#), [How to do it...](#), [How it works...](#)
- HTTP cookies
  - using / [Using HTTP cookies](#), [How to do it...](#), [How it works...](#)
- HTTP headers
  - using / [Using HTTP headers](#), [How to do it...](#), [How it works...](#)

# I

- Infrastructure as a Service (IaaS) / [Introduction](#)
- IntelliJ IDEA 14
  - Play 2 web apps, coding with / [Coding Play 2 web apps with IntelliJ IDEA 14](#),  
[How to do it...](#), [How it works...](#)
- IronMQ / [Introduction](#)
  - URL / [Integrating a Play application with message queues](#), [How to do it...](#)

# J

- Java Database Connectivity (JDBC) module
  - about / [How it works...](#)
- Java Development Kit (JDK)
  - about / [Installing Play Framework](#)
  - URL / [Getting ready](#)
- Jetbrains
  - URL / [How to do it...](#)
- JSON
  - serving / [Serving JSON](#), [How to do it...](#), [How it works...](#)
  - receiving / [Receiving JSON](#), [How to do it...](#), [How it works...](#)
- JUnit (Java)
  - used, for testing / [Testing with JUnit \(Java\) and specs2 \(Scala\)](#), [How to do it...](#)
- JWT
  - used, for implementing token authentication / [Implementing token authentication using JWT](#), [How to do it...](#), [How it works...](#)
  - URL / [Implementing token authentication using JWT](#), [How it works...](#)

# M

- Mandrill
  - about / [Utilizing play-mailer](#)
  - URL / [Utilizing play-mailer](#)
- message queues
  - used, for integrating Play application / [Integrating a Play application with message queues](#), [How to do it...](#), [How it works...](#)
- Model-View-Controller (MVC)
  - about / [Introduction](#)
- models
  - testing / [Testing models](#), [How to do it...](#)
- module dependencies
  - managing / [Managing module dependencies](#), [How to do it...](#), [How it works...](#)
- modules
  - using / [Working with modules](#)
  - declaring / [How to do it...](#), [How it works...](#)
  - reference link / [There's more...](#)
- MongoDB
  - utilizing / [Utilizing MongoDB](#), [How to do it...](#), [How it works...](#), [Utilizing MongoDB and GridFS](#), [How to do it...](#), [How it works...](#)
- MySQL
  - Ebean (Java), using / [Using Ebean \(Java\) with MySQL](#), [How to do it...](#)
  - Anorm (Scala), using / [Using Anorm \(Scala\) and database evolutions with MySQL](#)
  - database evolutions, using / [Using Anorm \(Scala\) and database evolutions with MySQL](#)

# N

- New Relic
  - monitoring with / [Monitoring with New Relic](#), [How to do it...](#), [How it works...](#)
  - URL / [How to do it...](#)
- Nginx
  - deploying with / [Deploying a Play application with Nginx](#), [How to do it...](#), [How it works...](#)
  - URL / [Deploying a Play application with Nginx](#)
- nimbus-jose-jwt
  - URL / [Implementing token authentication using JWT](#)

# P

- Parse.com
  - integrating with / [Integrating a Play application with Parse.com](#), [How to do it...](#), [How it works...](#)
  - URL / [Integrating a Play application with Parse.com](#), [There's more...](#)
- path binders
  - using / [Using path binders](#), [How to do it...](#), [How it works...](#)
- Platform as a Service (PaaS) / [Introduction](#)
- play-mailer
  - utilizing / [Utilizing play-mailer](#), [How to do it...](#), [How it works...](#)
- play.api.db.DBPlugin / [Introduction](#)
- play.i18n.MessagesPlugin / [Introduction](#)
- Play 2 web apps
  - coding, IntelliJ IDEA 14 used / [Coding Play 2 web apps with IntelliJ IDEA 14](#), [How to do it...](#), [How it works...](#)
- play2-elasticsearch
  - URL / [Integrating a Play application with ElasticSearch](#)
- Play 2.0 module
  - using, for different applications / [Using the same model for different applications](#), [How to do it...](#), [How it works...](#)
- Play application
  - creating, Typesafe Activator used / [Creating a Play application using Typesafe Activator](#), [How it works...](#)
  - integrating, with Amazon S3 / [How to do it...](#), [How it works...](#)
  - integrating, with Typesafe Slick / [Integrating with Play application Typesafe Slick](#), [How to do it...](#), [How it works...](#)
  - integrating, with Bootstrap / [Integrating Bootstrap and WebJars](#), [How to do it...](#), [How it works...](#)
  - integrating, with message queues / [Integrating a Play application with message queues](#), [How to do it...](#), [How it works...](#)
  - integrating, with ElasticSearch / [Integrating a Play application with ElasticSearch](#), [How to do it...](#), [How it works...](#)
  - deploying, on Heroku / [Deploying a Play application on Heroku](#), [How to do it...](#)
  - Procfile / [Procfile](#)
  - system.properties / [system.properties](#)
- Play console
  - using / [Using the Play console](#), [How to do it...](#), [There's more...](#)
  - about / [Using the Play console](#)
- Play development environment
  - Vagrant used / [Creating a Play development environment using Vagrant](#), [How to do it...](#), [How it works...](#), [There's more...](#)
- Play Framework
  - about / [Introduction](#)

- installing / [Installing Play Framework](#), [Getting ready](#)
- post-commit hooks
  - URL / [There's more...](#)
- POST API endpoint
  - creating / [Creating a POST API endpoint](#), [How to do it...](#), [How it works...](#)
- private module repositories
  - adding, Amazon S3 used / [Adding private module repositories using Amazon S3](#), [How to do it...](#), [How it works...](#)
- Procfile / [Procfile](#)
- PUT API endpoint
  - creating / [Creating a PUT API endpoint](#), [How to do it...](#), [How it works...](#)

# R

- Read-Eval-Print Loop (REPL) tools / [Creating a Play development environment using Vagrant](#)
- redirects
  - using / [Using reverse routing and redirects](#), [How to do it...](#), [How it works...](#)
- Redis
  - utilizing / [Utilizing Redis](#), [How to do it...](#), [How it works...](#)
  - about / [Utilizing Redis](#)
  - URL / [How it works...](#)
- RequireJS
  - URL / [Integrating a Play application with AngularJS](#)
- reverse routing
  - using / [Using reverse routing and redirects](#), [How to do it...](#), [How it works...](#)
- routes
  - about / [Working with controllers and routes](#)
  - using / [How to do it...](#)

# S

- session
  - about / [Using the session](#)
  - using / [How to do it...](#), [How it works...](#)
- specs2 (Scala)
  - used, for testing / [Testing with JUnit \(Java\) and specs2 \(Scala\)](#), [How to do it...](#)
- Spring
  - dependency injection / [Dependency injection with Spring](#), [How to do it...](#)
- SSH key generation
  - URL / [How to do it...](#)
- Structured Query Language (SQL)
  - about / [Introduction](#)

# T

- text file
  - working with / [Working with XML and text files](#), [How to do it...](#)
- token authentication
  - implementing, JWT used / [Implementing token authentication using JWT](#), [How to do it...](#), [How it works...](#)
- Travis CI
  - testing with / [Testing with Travis CI](#), [How to do it...](#), [How it works...](#)
  - Github, URL / [How to do it...](#)
  - development-commit-test process / [How it works...](#)
- Twitter API
  - and OAuth, using / [Using the Twitter API and OAuth](#), [How to do it...](#), [How it works...](#)
- Typesafe Activator
  - about / [How to do it...](#)
  - URL / [How to do it...](#)
  - used, for creating Play application / [Creating a Play application using Typesafe Activator](#), [How it works...](#)
  - reference link / [There's more...](#)
- Typesafe Slick
  - Play application, integrating / [Integrating with Play application Typesafe Slick](#), [How to do it...](#), [How it works...](#)

# V

- Vagrant
  - used, for creating Play development environment used / [Creating a Play development environment using Vagrant](#), [How to do it...](#), [How it works...](#), [There's more...](#)
  - URL / [How to do it...](#), [There's more...](#)
- View layouts
  - using / [Using View layouts and Includes](#), [How to do it...](#)
  - defined view, including / [Using View layouts and Includes](#), [How to do it...](#)
- View template
  - about / [Working with View templates](#)
  - using / [How to do it...](#)
- virtual machine (vm) / [How it works...](#)

# W

- web action
  - using / [Using a form template and web action, How to do it...](#)
- WebJars
  - used, for integrating Bootstrap / [Integrating Bootstrap and WebJars, How to do it..., How it works...](#)
  - URL / [Integrating a Play application with AngularJS](#)

# X

- XML file
  - working with / [Working with XML and text files, How to do it...](#)