Diego Pacheco

Building Applications with Scala

Write modern, scalable, and reactive applications with the power of Scala

Packt

Table of Contents

Building Applications with Scala
Credits
About the Author
Acknowledgments
About the Reviewer
www.PacktPub.com
    Why subscribe?
Preface
    What this book covers
    What you need for this book
    Who this book is for
    Conventions
    Reader feedback
    Customer support
        Downloading the example code
        Downloading the color images of this book
        Errata
        Piracy
        Questions
1. Introduction to FP, Reactive, and Scala
    Functional programming
    Principles of functional programming
        Immutability
        Disciplined state
        Pure functions and no side effects
        First-class functions and higher-order functions
        Type systems
        Referential transparency
    Installing Java 8 and Scala 2.11
    Read Eval Print and Loop - REPL
    Scala Hello World using the REPL
        Scala REPL Hello World program
        Scala object-oriented HelloWorld program

- Scala HelloWorld App in the Scala REPL
- Java HelloWorld application

Scala language - the basics

Scala variables - var and val

- Scala REPL var usage
- Scala val usage at the Scala REPL

Creating immutable variables

- Scala variable type in the Scala REPL
- Scala variables with explicit typing at the Scala REPL

Scala conditional and loops statements

- If statements in Scala REPL
- If statements in return statements in Scala REPL
- Basic for loop in Scala REPL
- For with List in Scala REPL
- For with if statements for filtering - Scala REPL
- Java code for filtering even numbers

For comprehensions

- For comprehension in Scala REPL
- Java code for performing filtering with collections

Scala collections

- Creating, removing, and getting an item from a mutable list in Scala REPL
- Scala tuples
- Scala immutable Map in Scala REPL
- Scala mutable Maps at Scala REPL

Monads

- Scala Map function in Scala REPL
- Option Monad in Scala
- A list of all methods using the Scala REPL

Scala class, traits, and OO programming

- A simple Scala class in Scala REPL
- Scala plain old Java object in Scala REPL
- Person class in Java

Traits and inheritance

- Scala inheritance code in Scala REPL
- Scala traits sample code in Scala REPL
- Scala traits using variable mixing technique at Scala REPL

Scala type alias sample in Scala REPL
Case classes
Scala case classes feature in Scala REPL
Pattern Matcher
Simple Pattern Matcher in Scala
Advanced pattern matcher in Scala REPL
Advanced complex pattern matcher in Scala REPL
Partial functions
Simple Partial function in Scala REPL
Scala PartialFunction without OO using case statements in Scala REPL
PartialFunction composition in Scala REPL
Package objects
package.scala
MainApp.scala
Functions
Partial application
Partial function in Scala REPL
Curried functions
Curried functions - Scala REPL
Curried transformation in Scala REPL
Operator overloading
Scala operator overloading in Scala REPL
Implicits
Scala Implicits in SCALA REPL
Implicit Parameter at Scala REPL
Futures
Simple Future code in Scala REPL
A complete Future sample at Scala REPL
Reactive Programing and RxScala
Simple Observables Scala with RxScala
Simple Observables Scala with RxScala - Execution in the console
Complex Scala with RxScala Observables
Summary
2. Creating Your App Architecture and Bootstrapping with SBT
Introducing SBT

- Installing SBT on Ubuntu Linux
- Getting started with SBT
- Adding dependencies
- Generating Eclipse project files from SBT
- Application distribution
- Hello world SBT / Scala App
- Bootstrapping our Play framework app with Activator
- Activator shell
- Activator - compiling, testing, and running
- Summary

3. Developing the UI with Play Framework
- Getting started
- Creating our models
- Creating routes
- Creating our controllers
- Working with services
- Configuring the Guice module
- Working with views(UI)
- Summary

4. Developing Reactive Backing Services
- Getting started with reactive programming
  - IPriceService - Scala trait
  - PriceService - RxScala PriceService implementation
  - Guice Injection - Module.scala
  - NGServiceEndpoint
- Play framework and high CPU usage
  - RndDoubleGeneratorController
  - IRndService.scala - Scala trait
  - RndService.scala - RndService implementation
  - Module.scala - Guice Injections
  - main.scala.html
  - product_details.scala.html
- Summary

5. Testing Your Application
- Unit testing principles
  - Making code testable
  - Isolation and self-contained tests

- Effective naming
- Levels of testing
- Testing with Junit
- Behavior-Driven Development - BDD
- MyFirstPlaySpec.scala - First BDD with ScalaTest and the Play framework
- Testing with Play framework support
  - ProductService.scala - FIX the code issue
  - ImageServiceTestSpec.scala - ImageService Test
  - ReviewServiceTestSpec.scala - ReviewService test
- Testing routes
  - RoutesTestingSpec.scala - Play framework route testing
- Controller testing
  - RndDoubleGeneratorControllerTestSpec.scala - RndDoubleGeneratorController tests
  - IntegrationSpec.scala
  - ProductControllerTestSpec.scala
  - product_index.scala.html
  - ImageControllerTestSpec.scala
  - image_index.scala.html
  - ReviewControllerTestSpec.scala
  - review_index.scala.html
  - ApplicationSpec.scala
  - NGServiceImplTestSpec.scala
  - NGServiceEndpointControllerTest.scala
- Summary

6. Persistence with Slick
- Introducing the Slick framework
- MySQL setup
- Configuring Slick in our Play framework app
  - Configure the database connection
- FPM Mapping
  - ProductDao
  - ReviewDAO
  - ImageDao
- Slick evolutions
- Refactoring services

- Refactoring controllers
- Configuring DAO packages in Guice
- Refactoring tests
- Generic mocks
- Service tests
- Controller tests
- Running the application
- Summary

7. Creating Reports
- Introducing JasperReports
- JasperReports workflow
- Jasper sessions
- Installing Jaspersoft Studio 6
- Configuring MySQL Data Adapter in Jaspersoft Studio
- Creating a product report
- Creating a review report
- Creating an image report
- Integrating JasperReports with Play framework
  - build.sbt
  - Generic report builder
  - Adding the report to the product controller
  - Adding the report to the review controller
  - Adding the report to the image controller
- Routes - adding new report routes
- New centralized reports UI
  - Adding the report button for each view
- Summary

8. Developing a Chat with Akka
- Adding the new UI introduction to Akka
- Introduction to the Actor model
  - What is an Actor?
- Message exchange and mailboxes
- Coding actors with Akka
- Actor routing
- Persistence
- Creating our chat application
- The chat protocol

- The chat controller
  - Implementing the chat controller
- Configuring the routes
- Working on the UI
- Adding Akka tests
  - Scala test for Akka Actor
  - Chat room Actor test
  - Chat Bot Admin Actor test
- Summary

9. Design Your REST API
- Introduction to REST
- REST API design
  - HTTP verbs design
  - Uniform API
  - Response with HTTP status codes
  - REST API patterns
  - API versioning
  - Some anti-patterns to be avoided
- Creating our API with REST and JSON
  - RestApiContoller
    - REST API Front Controller implementation
  - JSON mapping
  - Configuring new routes
  - Testing the API using the browser
- Creating a Scala client
  - Configuring plugins.sbt
  - Configuring build.sbt
  - Scala client code
- Creating our REST client proxies
  - Creating ScalaTest tests for the proxies
  - Adding back pressure
    - The leaky bucket algorithm
      - Scala leaky bucket implementation
      - Testing back pressure
  - Adding Swagger support
  - Swagger UI
    - Build and install Swagger Standalone

- Summary
- 10. Scaling up
  - Standalone deploy
  - Reports folder
  - Changing report builder
  - Defining the secret
  - Running the standalone deploy
  - Architecture principles
    - Service orientation (SOA/microservices)
    - Performance
    - Scalability/Resiliency
  - Scalability principles
    - Vertical and horizontal scaling (up and out)
    - Caching
    - Load balancer
    - Throttling
    - Database cluster
    - Cloud computing/containers
    - Auto Scaling
    - A note about automation
    - Don't forget about telemetry
    - Reactive Drivers and discoverability
  - Mid-Tier load balancer, timeouts, Back pressure, and caching
  - Scaling up microservices with an Akka cluster
  - Scaling up the infrastructure with Docker and AWS cloud
  - Summary

Building Applications with Scala

Building Applications with Scala

Copyright © 2016 Packt Publishing

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

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

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

First published: December 2016

Production reference: 1021216

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78646-148-3

www.packtpub.com

Credits

Author Diego Pacheco	**Copy Editor** Sonia Mathur
Reviewer Yuanhang Wang	**Project Coordinator** Suzanne Coutinho
Commissioning Editor Kunal Parikh	**Proofreader** Safis Editing
Acquisition Editor Denim Pinto	**Indexer** Tejal Daruwale Soni
Content Development Editor Rohit Singh	**Graphics** Jason Monteiro
Technical Editor Jijo Maliyekal	**Production Coordinator** Melwyn Dsa

About the Author

Diego Pacheco is an experienced software architect and DevOps practitioner with over 10 years of solid experience. He has led architecture teams using open source solutions such as Java, Scala, Amazon Web Services (AWS), Akka, Apache Cassandra, Redis, ActiveMQ, NetflixOSS Stack - Simian Army, RxJava, Karyon, Eureka, and Ribbon on cig customers in Brazil, London, Barcelona, India, and the USA. Diego has a passion for functional programming and is currently working as a software architect/agile coach with Scala, Akka, and NetflixOSS. During his free time, he enjoys gaming, blogging, and playing wicked tunes on his guitar. You can check out his blog at http://diego-pacheco.blogspot.in/.

Some of his core skills include the following:

- Architecture design and architecture coding for high scalable systems
- Distributed systems using SOA and microservices principles, tools, and techniques
- Performance tuning and DevOps engineering
- Functional programming and Scala
- Agile coaching and servant leadership for architecture teams
- Consultancy on development practices with XP/Kanban

More about him can be found at the following:

- Linkedin: https://www.linkedin.com/in/diegopachecors
- Blog: http://diego-pacheco.blogspot.in/
- Github: https://github.com/diegopacheco
- Slideshare: http://www.slideshare.net/diego.pacheco/presentations
- Presentations: https://gist.github.com/diegopacheco/ad3e3804a5071ef219d1

His recent lectures include Netflix (https://www.youtube.com/watch?v=Z4_rzsZd70o&feature=youtu.be), QCon (http://qconsp.com/sp2016/speaker/diego-pacheco), and Amazon (http://www.meetup.com/Sao-Paulo-Amazon-Web-Services-AWS-Meetup/events/229283010/).

Acknowledgments

First of all, I'm very thankful for everything God has given to me in life. So, I need to say thank you to God at least three times. Thank you God, thank you God, thank you God. I’m very glad to have finished this book, and I also need to say a big thank you to all my family and supportive friends, especially Andressa Bicca, my true love; my mother, Denise Maris; my grandmother, Walkyria; and my dear friends, Margarida Avila, Adão Avila, Israel Prestes, and Tais da Rosa, for all their love and support. I need to say thanks to Packt, especially to Kirk D'costa and Rohit Kumar Singh for being great editors. Also, I need to say thank you to ilegra.com and especially to Ivã Boesing and Romulo Dornelles for all the space, trust, and support. Also, I cannot forget to say thank you to my coworkers, customers, and friends, who are great people to work with, and who I’ve learned a lot from: Sam Sgro, Daniel Wildt, Anibal Rojas, Alexandre Poletto, Jeferson Machado, Nilseu Padilha, Jackson Santos, Christophe Marchal, Joel Correa, and Rafael Souza. Finally, thank you to all of you who bought this book and have read it--you are awesome!

About the Reviewer

Yuanhang Wang describes himself as an enthusiast of purely functional programming and neural networks, with a primary focus on Domain Specific Language (DSL) design, and he has dabbled in several functional programming languages. He is currently a data scientist at China Mobile Research Center, working on a typed data processing engine and optimizer built on top of several big data platforms.

www.PacktPub.com

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://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Preface

Functional programming started in academia and ended up in the IT industry. The Scala language is a multi-paradigm language used by big players and large organizations that helps you get the correct (in the sense of pure functional programming) software and, at the same time, software that is practical and scalable.

Scala has a very rich ecosystem, including Play Framework, Akka, Slick, Gatling, Finable, and more. In this book, we will start right from the basic principles and ideas on functional and ReactiveX programming, and through practical examples, you will learn how to code with the most important frameworks of the Scala ecosystem, such as Play, Akka, and Slick.

You will learn how to bootstrap a Scala application with SBT and Activator, how to build a Play and Akka application step by step, and we cover the theory of how to scale massive Scala applications with cloud and the NetflixOSS stack. This book will help you to go from the basic subjects to the most advanced ones in order to make you a Scala expert.

What this book covers

Chapter 1, *Introduction to FP, Reactive, and Scala*, looks at how to set up a Scala development environment, the difference between functional programming and object-oriented programming, and the concepts of functional programming.

Chapter 2, *Creating Your App Architecture and Bootstrapping with SBT*, discusses the overall architecture, SBT basics, and how to create your own application.

Chapter 3, *Developing the UI with Play Framework*, covers the principles of web development in Scala, creating our models, creating our views, and adding validations.

Chapter 4, *Developing Reactive Backing Services*, introduces you to reactive programming principles, refactoring our controllers, and adding Rx Scala to our services.

Chapter 5, *Testing Your Application*, looks into testing principles with Scala and JUnit, behavior-driven development principles, using ScalaTest specs and DSL in our tests, and running our tests with SBT.

Chapter 6, *Persistence with Slick*, covers principles of database persistence with Slick, working with Functional Relational Mapping in your application, creating the queries you need with SQL support, and improving the code with async database operations.

Chapter 7, *Creating Reports*, helps you understand Jasper reports and add database reports to your application.

Chapter 8, *Developing a Chat with Akka*, discusses the actor model, actor systems, actor routing, and dispatchers.

Chapter 9, *Design Your REST API*, looks into REST and API design, creating our API with REST and JSON, adding validations, adding

backpressure, and creating a Scala client.

Chapter 10, *Scaling Up*, touches upon the architecture principles and scaling up the UI, reactive drivers, and discoverability. It also covers middle-tier load balancers, timeouts, back pressure, and caching, and guides you through scaling up microservices with an Akka cluster and scaling up the infrastructure with Docker and AWS cloud.

What you need for this book

For this book, you will need the following:

- Ubuntu Linux 14 or superior
- Java 8 update 48 or superior
- Scala 2.11.7
- Typesafe Activator 1.3.9
- Jasper Reports Designer
- Windows fonts for Linux
- Eclipse IDE

Who this book is for

This book is for professionals who want learn Scala, as well as functional and reactive techniques. This book is mainly focused on software developers, engineers, and architects. This is a practical book with practical code; however, we also have theory about functional and reactive programming.

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: "The next step is to create the environment variable called `SCALA_HOME`, and to put the Scala binaries in the `PATH` variable."

A block of code is set as follows:

```
package scalabook.javacode.chap1;

    public class HelloWorld {
      public static void main(String args[]){
        System.out.println("Hellow World");
      }
    }
```

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

```
export JAVA_HOME=~/jdk1.8.0_77
export PATH=$PATH:$JAVA_HOME/bin
```

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: "The **Actor** behavior is the code you will have inside your **Actor**."

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

You can download the code files by following these steps:

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

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

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

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/Building-Applications-with-Scala. We also have other code bundles from our rich catalog of books and

videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

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

Errata

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

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

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright

and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

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

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

Questions

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

Chapter 1. Introduction to FP, Reactive, and Scala

In our first chapter, we will learn the basic concepts of **Functional Programing** (**FP**), reactive programming, and the Scala language. These concepts are listed as follows:

- Setting up a Scala development environment with Eclipse Scala IDE.
- Basic constructs of the language like var, val, for, if, switch, and operator overload.
- The difference between FP and object-oriented programming.
- Principles of pure FP: immutability, no side effects, state discipline, composition, and higher order functions.
- Concepts of FP such as lambda, recursion, for comprehensions, partial functions, Monads, currying, and functions.
- Pattern Matcher, recursion, reflection, package objects, and concurrency.

Let's get going!

Functional programming

FP is not new at all. The very first implementation of FP is Lisp and is dated from the 1950s. Currently, we are living in a post-functional programming era, where we have the strong math principles and ideas from the 50s mixed with the most modern and beautiful piece of engineering, also know as the **Java Virtual Machine** (**JVM**). Scala is a post-functional programming language built on top of the JVM. Being on top of the JVM gives us a lot of benefits such as the following: Scala is a post-functional programming language built on top of the JVM. Being on top of the JVM gives us a lot of benefits such as the following:

- **Reliability and performance**: Java is used by 10 out of 10 top websites we have currently, like Netflix, Apple, Uber, Twitter, Yahoo, eBay, Yelp, LinkedIn, Google, Amazon, and many others. JVM is the best solution at scale and is battle-tested by these web-scale companies.
- **Native JVM eco-system**: Full access to all of the Java ecosystem including frameworks, libraries, servers, and tools.
- **Operations leverage**: Your operation team can run Scala in the same way they run Java.
- **Legacy code leverage**: Scala allows you to easily integrate Scala code with Java code. This feature is great because it enables Java legacy system integration inside the box.
- **Java interoperability**: A code written in Scala can be accessed in Java.

Scala was created in 2001 at EPFL by Martin Odersky. Scala is a strong static-typed language, and was inspired by another functional language called **Haskell**. Scala addresses several criticisms of the Java language, and delivers a better developer experience through less code and more concise programs, without losing performance.

Scala and Java share the same infrastructure as the JVM, but in terms of design, Scala is a different language in comparison with Java. Java is an imperative object-oriented language and Scala is a post-functional, multiparadigm programing language. FP works with different principles than **object-oriented programing** (**OOP**). OOP got very popular and well established in enterprise thanks to languages like Java, C#, Ruby, and Python. However, languages like Scala, Clojure, F#, and Swift are gaining a huge momentum, and FP has grown a lot in the last 10 years. Most of the new languages are pure functional, post-functional, or hybrid (like Java 8). In this book, you will see Scala code compared with Java code so you can see by yourself how Scala is way more compact, objective, and direct than Java and imperative OOP languages.

FP started at academia and spread to the world; FP is everywhere. Big Data and Stream processing solutions like Hadoop and Spark

(built on top of Scala and Akka) are built on top of FP ideas and principles. FP spread to UI with **RxJavaScript** - you can even find FP in a database with Datomic (Clojure). Languages like Clojure and Scala made FP more practical and attractive to enterprise and professional developers. In this book, we will be exploring both principles and practical aspects of the Scala language.

Principles of functional programming

FP is a way of thinking, a specific style of constructing and building programs. Having an FP language helps a lot in terms of syntax, but at the end of the day, it's all about ideas and developer mindset. FP favors disciplined state management and immutability in a declarative programming way rather than the imperative programming mostly used by OOP languages such as Java, Python, and Ruby.

FP has roots in math back to **Lambda calculus** - a formal system developed in the 1930s. Lambda calculus is a mathematical abstraction and not a programming language, but it is easy to see its concepts in programming languages nowadays.

Imperative programming uses statements to change the program state. In other words, this means you give commands to the program to perform actions. This way of thinking describes a sequence of steps on how the program needs to operate. What you need to keep in mind is the kind of style focus on how FP works in a different way, focusing on what the program should accomplish without telling the program how to do it. When you are coding in FP, you tend to use fewer variables, for loops, and IFS, and write more functions and make function composition.

The following are the CORE principles of FP:

- Immutability
- Disciplined state
- Pure functions and no side effects/disciplined states
- First class functions and high order functions
- Type systems
- Referential transparency

Let's understand these principles in detail.

Immutability

The concept of immutability is the CORE of FP, and it means that once you assign a value to something, that value won't change. This is very important, because it eliminates side effects (anything outside of the local function scope), for instance, changing other variables outside the function. Immutability makes it easier to read code, because you know the function that you are using is a pure function. Since your function has a disciplined state and does not change other variables outside of the function, you don't need to look at the code outside the function definition. This sounds like you're not working with state at all, so how would it be possible to write professional applications this way? Well, you will change state but in a very disciplined way. You will create another instance or another pointer to that instance, but you won't change that variable's value. Having immutability is the key to having better, faster, and more correct programs, because you don't need to use locks and your code is parallel by nature.

Disciplined state

Shared mutable state is evil, because it is much harder to scale and to run it concurrently. What is shared mutable state? A simple way to see it is as a global variable that all your functions have access to. Why is this bad? First of all, because it is hard to keep this state correct since there are many functions that have direct access to this state. Second, if you are performing refactoring, this kind of code is often the hardest to refactor as well. It's also hard to read this code. This is because you can never trust the local method, since your local method is just one part of the program. And with mutable state, you need to look up for all the functions that use that variable, in order to understand the logic. It's hard to debug for the very same reason. When you are coding with FP principles in mind, you avoid, as much as possible, having a shared mutable state. Of course you

can have state, but you should keep it local, which means inside your function. This is the state discipline: you use state, but in a very disciplined way. This is simple, but it could be hard especially if you are a professional developer, because this aspect is now usual to see in enterprise languages such as Java, .NET, Ruby, and Python.

Pure functions and no side effects

Pure functions are the ones with no side effects. Side effects are bad, because they are unpredictable and make your software hard to test. Let's say you have a method that receives no parameters and returns nothing--this is one of the worst things we could have, because how do you test it? How can you reuse this code? This is not what we call a pure function. What are the possible side effects? Database call, global variables, IO call, and so on. This makes sense, but you cannot have a program with just pure functions, because it won't be practical.

First-class functions and higher-order functions

First-class means that the language treats functions as first-class citizens. In other words, it means having language support to pass functions as arguments to other functions and to return values as functions. First-class function also implies that the language allows you to store functions as variables or any other data structure.

Higher-order functions are related to First-class functions, but they are not the same thing. Higher-order functions often means language support for partial functional application and Currying. Higher-order functions are a mathematical concept where functions operate with other functions.

Partial functions are when you can fix a value (argument) to a particular function, which you may or may not change later on. This is great for function composition.

Currying is a technique to transform a function with multiple parameters in a sequence of functions with each function having a single argument. Scala language does not force currying, however, languages like ML and Haskell almost always use this kind of technique.

Type systems

Type system is all about the compiler. The idea is simple: you create a type system, and by doing so, you leverage the compiler to avoid all kinds of mistakes and errors. This is because the compiler helps in making sure that you only have the right types as arguments, turn statements, function composition, and so on. The compiler will not allow you do make any basic mistakes. Scala and Haskell are examples of languages that are Strong-type. Meanwhile, Common Lisp, Scheme, and Clojure are dynamic languages that may accept wrong values during compilation time. One of the biggest benefits of the strong type system is that you have to write fewer tests, because the compiler will take care of several issues for you. For instance, if you have a function that receives a string, it could be dangerous, because you can pass pretty much anything in a string. However, if you have a function that receives a type called salesman, then you don't write a validation to check if it is a salesman. All this may sound silly, but in a real application, this saves lots of lines of code and makes you program better. Another great benefit of strong typing is that you have better documentation, as your code becomes your documentation, and it's way more clear what you can or can't do.

Referential transparency

Referential transparency is a concept which works close with pure functions and immutability since your program has fewer assignment statements, and often when you have it, you tend to never change that value. This is great because you eliminate side effects with this technique. During program execution, any variable can be replaced since there are no side effects, and the program becomes

referentially transparent. Scala language makes this concept very clear the moment you declare a variable.

Installing Java 8 and Scala 2.11

Scala requires JVM to work, so we need get the JDK 8 before installing Scala. Go to the Oracle website, and download and install JDK 8 from http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html.

Once you've downloaded Java, we need to add Java to the `PATH` variable; otherwise, you can use the terminal. We do this as follows:

```
$ cd ~/
$ wget --no-cookies --no-check-certificate --
header "Cookie:
gpw_e24=http%3A%2F%2Fwww.oracle.com%2F;
oraclelicense=accept-securebackup-cookie" "
http://download.oracle.com/otn-pub/java/jdk/8u77-
b03/jdk-8u77-linux-i586.tar.gz"
$ tar -xzvf $ jdk-8u77-linux-x64.tar.gz
$ rm -f jdk-8u77-linux-x64.tar.gz
```

The next step is to create the environment variable called `JAVA_HOME`, and to put the Java 8 binaries in the `PATH` variable. In Linux, we need to edit the `~/.bashrc` file, and export the variables we need, like in the following:

```
export JAVA_HOME=~/jdk1.8.0_77
export PATH=$PATH:$JAVA_HOME/bin
```

Save the file, and then on the same terminal we need to source the file via `$ source ~/.bashrc`

Now we can test our Java 8 installation. Just type in `$ java -version`. You should see something like the following:

```
$ java -version
java version "1.8.0_77"
Java(TM) SE Runtime Environment (build 1.8.0_77-
```

```
b03)
Java HotSpot(TM) Server VM (build 25.77-b03, mixed
mode)
```

Let's get started. We will be using the latest Scala version 2.11.8. However, the code inside this book should work with any Scala 2.11.x version. First of all, let's download Scala from http://www.scala-lang.org/.

Scala works on Windows, Mac, and Linux. For this book, I will show how to use Scala on Ubuntu Linux(Debian-based). Open your browser and go to http://www.scala-lang.org/download/.

Download `scala 2.11.8`: it will be a TGZ file. Extract it and add it to your path; otherwise, you can use the terminal. Do this as follows:

```
$ cd ~/
$ wget
http://downloads.lightbend.com/scala/2.11.8/scala-
2.11.8.tgz
$ tar -xzvf scala-2.11.8.tgz
$ rm -rf scala-2.11.8.tgz
```

The next step is to create the environment variable called `SCALA_HOME`, and to put the Scala binaries in the `PATH` variable. In Linux, we need to edit the `~/.bashrc` file and export the variables we need, like in the following:

```
export SCALA_HOME=~/scala-2.11.8/
export PATH=$PATH:$SCALA_HOME/bin
```

Save the file, and then, on the same terminal, we need to source the file via `$ source ~/.bashrc`.

Now we can test our Scala installation. Just type in `$ scala -version`. You should see something like the following:

```
$ scala -version
Scala code runner version 2.11.8 -- Copyright
```

```
2002-2016, LAMP/EPFL
```

You have successfully installed Java 8 and Scala 2.11. Now we are ready to start learning the FP principles in Scala. For this, we will be using the Scala REPL in the beginning. Scala REPL is bundled with the default Scala installation, and you just need to type `$ scala` in your terminal as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) Server
VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala>
Scala REPL
```

Congratulations! You have installed Java 8 and Scala 2.11 successfully.

Read Eval Print and Loop - REPL

Read Eval Print and Loop (**REPL**) is also know as a language shell. Many other languages have shells, like Lisp, Python, and Ruby for instance. The REPL is a simple environment to experiment the language in. It's possible to write very complex programs using REPL, but this is not the REPL goal. Using REPL does not invalidate the usage of an IDE like Eclipse or IntelliJ IDEA. REPL is ideal for testing simple commands and programs without having to spend much time configuring projects like you do with an IDE. The Scala REPL allows you to create a variable, functions, classes, and complex functions as well. There is a history of every command you perform; there is some level of autocomplete too. As a REPL user, you can print variable values and call functions.

Scala Hello World using the REPL

Let's get started. Go ahead, open your terminal, and type `$ scala` in order to open the Scala REPL. Once the REPL is open, you can just type `"Hello World"`. By doing this, you perform two operations: eval and print. The Scala REPL will create a variable called `res0`, and store your String there. Then it will print the content of the `res0` variable.

Scala REPL Hello World program

We will see how to create Hello World program in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> "Hello World" res0: String = Hello World scala>**

Scala is a hybrid language, which means it is object-oriented and functional as well. You can create classes and objects in Scala. Next we will create a complete Hello World application using classes.

Scala object-oriented HelloWorld program

We will see how to create object-oriented HelloWorld program in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> object HelloWorld { | def main(args:Array[String]) = println("Hello World") | } defined object HelloWorld scala> HelloWorld.main(null) Hello World scala>**

The first thing you need to realize is that we use the word object instead of class. The Scala language has different constructs compared to Java. Object is a singleton in Scala. It's the same as coding the singleton pattern in Java.

Next we see the word `def` that is used in Scala to create functions. In the preceding program, we create the main function similar to the way we do it in Java, and we call the built-in function `println` in order to print the String Hello World. Scala imports some Java objects and packages by default. Coding in Scala does not require you to type, for instance, `System.out.println("Hello World")`, but you can if you want. Let's take a look at it in the following code: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> System.out.println("Hello World") Hello World scala>**

We can and we will do better. Scala has some abstractions for a console application, so we can write this code with a lesser number of lines of code. To accomplish this goal, we need to extend the Scala class App. When we extend from App, we perform inheritance and we don't need to define the main function. We can just put all the code in the body of the class, which is very convenient and makes the code clean and simple to read.

Scala HelloWorld App in the Scala REPL

We will see how to create Scala HelloWorld App in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> object HelloWorld extends App { | println("Hello World") | } defined object HelloWorld scala> HelloWorld object HelloWorld scala> HelloWorld.main(null) Hello World scala>**

After coding the HelloWorld object in the Scala REPL we can ask the REPL what HelloWorld is, and as you might realize, the REPL will

answer that HelloWorld is an object. This is a very convenient Scala way to code console applications, because we can have a Hello World application with just three lines of code. Sadly, to have the same program in Java, it required way more code. Java is a great language for performance, but it is a verbose language compared with Scala, for instance.

Java HelloWorld application

We will see how to create Java HelloWorld application as follows: package scalabook.javacode.chap1; public class HelloWorld { public static void main(String args[]){ System.out.println("Hellow World"); } }

The Java app required six lines of code, while in Scala, we were able to do the same with 50% less code (three lines of code). This is a very simple application. When we are coding complex applications, this difference gets bigger, as a Scala application ends up with way less code than Java.

Remember, we use an object in Scala in order to have a Singleton (Design Pattern that makes sure you have just one instance of a class), and if we want the same in Java, the code would be something like the following: package scalabook.javacode.chap1; public class HelloWorldSingleton { private HelloWorldSingleton(){} private static class SingletonHelper{ private static final HelloWorldSingleton INSTANCE = new HelloWorldSingleton(); } public static HelloWorldSingleton getInstance(){ return SingletonHelper.INSTANCE; } public void sayHello(){ System.out.println("Hello World"); } public static void main(String[] args) { getInstance().sayHello(); } }

It's not just about the size of the code, but also about consistency and the language providing more abstractions for you. If you write less code, you will have fewer bugs in your software at the end of the day.

Scala language - the basics

Scala is a statically typed language with a very expressive type system which enforces abstractions in a safe yet coherent manner. All values in Scala are Java objects (primitives which are unboxed at runtime), because at the end of the day, Scala runs on the Java JVM. Scala enforces immutability as a core FP principle. This enforcement happens in multiple aspects of the Scala language, for instance, when you create a variable, you do it in an immutable way, when you use an collection, you will use a immutable collection. Scala also lets you use mutable variables and mutable structures, but by design, it favors immutable ones.

Scala variables - var and val

When you are coding in Scala, you create variables using the operator `var`, or you can use the operator `val`. The operator `var` allows you to create a mutable state, which is fine as long as you make it local, follow the CORE-FP principles and avoid a mutable shared state.

Scala REPL var usage

We will see how to use var in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> var x = 10 x: Int = 10 scala> x res0: Int = 10 scala> x = 11 x: Int = 11 scala> x res1: Int = 11 scala>**

However, Scala has a more interesting construct called `val`. Using the `val` operator makes your variables immutable, and this means you can't change the value once you've set it. If you try to change the value of the `val` variable in Scala, the compiler will give you an error. As a Scala developer, you should use the variable `val` as much as possible, because that's a good FP mindset, and it will make your programs better. In Scala, everything is an object; there are no primitives -- the `var` and `val` rules apply for everything it could but an `Int` or `String` or even a class.

Scala val usage at the Scala REPL

We will see how to use val in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x = 10 x: Int = 10 scala> x res0: Int = 10 scala> x = 11 <console>:12: error: reassignment to val x = 11 ^ scala> x res1: Int = 10 scala>**

Creating immutable variables

Right now, let's see how we define the most common types in Scala such as `Int`, `Double`, `Boolean`, and `String`. Remember, you can create these variables using `val` or `var` depending on your needs.

Scala variable type in the Scala REPL

We will see Scala variable type in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x = 10 x: Int = 10 scala> val y = 11.1 y: Double = 11.1 scala> val b = true b: Boolean = true scala> val f = false f: Boolean = false scala> val s = "A Simple String" s: String = A Simple String scala>**

For the variables in the preceding code, we did not define the type. Scala language figures it out for us. However, it is possible to specify the type if you want. In Scala, the type comes after the name of the variable.

Scala variables with explicit typing at the Scala REPL

We will see Scala variables with explicit typing at Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x:Int = 10 x: Int = 10 scala> val y:Double = 11.1 y: Double = 11.1 scala> val s:String = "My String " s: String = "My String " scala> val b:Boolean = true b: Boolean = true scala>**

Scala conditional and loops statements

Like any other language, Scala has support for conditional statements like `if` and `else`. While Java has a switch statement, Scala has a more powerful and functional structure called Pattern Matcher, which we will cover later in this chapter. Scala allows you to use `if` statements during variable assignments, which is very practical as well as useful.

If statements in Scala REPL

We will see how to use `if` statements in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x = 10 x: Int = 10 scala> if (x == 10) | println ("X is 10") X is 10 scala> val y = if (x == 10) 11 y: AnyVal = 11 scala> y res1: AnyVal = 11 scala>**

In the preceding code, you can see that we set the variable `y` based on an `if` condition. Scala `if` conditions are very powerful, and they also can be used in return statements.

If statements in return statements in Scala REPL

We will see how to use `if` statements in return statements in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x = 10 x: Int = 10 scala> def someFunction = if (x == 10) "X is 10" someFunction: Any scala> someFunction res0: Any = X is 10 scala>**

Scala supports `else` statements too, and you also can use them in variables and return statements as wellas follows: **~$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val x = 10 x: Int = 10 scala> if (x==10){ | println("X is 10") | } else { | println ("X is something else") | } X is 10 scala>**

Now you will learn how to use for loops in Scala. For loops are very powerful in Scala. We will start with the basics and later we will move on to functional loops used for comprehensions, also know as `List` comprehensions.

In Scala, for loops work with ranges, which is another Scala data structure that represents numbers from a starting point to an end point. The range is created using the left arrow operator(`<-`). Scala allows you to have multiple ranges in the same for loop as long as you use the semicolon(`;`).

You also can use `if` statements in order to filter data inside for loops, and work smoothly with `List` structures. Scala allows you to create variables inside a for loop as well. Right now, let's see some code which illustrates the various for loop usages in Scala language.

Basic for loop in Scala REPL

We will see how to use basic for loop in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> for (i <- 1 to 10) | println("i * " + i + " = " + i * 10) i * 1 = 10 i * 2 = 20 i * 3 = 30 i * 4 = 40 i * 5 = 50 i * 6 = 60 i * 7 = 70 i * 8 = 80 i * 9 = 90 i * 10 = 100 scala>**

Right now, we will create a for loop using a Scala data structure called `List`. This is very useful, because in the first line of code, you can define a `List` as well as set its values in the same line. Since we are using the `List` structure, you don't need to pass any other argument besides the `List` itself.

For with List in Scala REPL

We will see how to `use` for with `List` in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val listOfValues = List(1,2,3,4,5,6,7,8,9,10) listOfValues: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) scala> for (i<- listOfValues) println(i) 1 2 3 4 5 6 7 8 9 10 scala>**

Next, we can use for loops with `if` statements in order to apply some filtering. Later in this book, we will approach a more functional way to approach filtering using functions. For this code, let's say we want to get just the even numbers on the list and print them.

For with if statements for filtering - Scala REPL

We will see how to use `for` with `if` statements in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val listOfValues = List(1,2,3,4,5,6,7,8,9,10) listOfValues: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) scala> for (i<- listOfValues) if (i % 2== 0) println(i) 2 4 6 8 10 scala>**

Java code for filtering even numbers

In Scala language, we just need two lines of code to perform this filtering, whereas in Java it would have required at least eleven lines of code as you see in the following code: package scalabook.javacode.chap1; import java.util.Arrays; import java.util.List; public class ForLoopsEvenNumberFiltering { public static void main(String[] args) { List<Integer> listOfValues = Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9,10}); for(Integer i : listOfValues){ if (i%2==0) System.out.println(i); } } }

For comprehensions

Also known as list or sequence comprehensions, for comprehensions are one of the FP ways to perform loops. This is a language support to create `List` structure or collections based on other collections. This task is performed in a `SetBuilder` notation. Another way to accomplish the same goal would be by using the `Map` and `filter` functions, which we will cover later in this chapter. For comprehensions can be used in a generator form, which would introduce new variables and values, or in a reductionist way, which would filter values resulting into a new collection or sequence. The syntax is: `for (expt) yield e`, where the `yield` operator will add new values to a new collection/sequence that will be created from the original sequence.

For comprehension in Scala REPL

We will see how to use `for` comprehension in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val names = Set("Diego", "James", "John", "Sam", "Christophe") names: scala.collection.immutable.Set[String] = Set(John, Sam, Diego, James, Christophe) scala> scala> val brazilians = for { | name <- names | initial <- name.substring(0, 1) | } yield if (name.contains("Die")) name brazillians: scala.collection.immutable.Set[Any] = Set((), Diego) scala>**

In the preceding code, we create a set of names. As you can see, Scala, by default, prefers immutable data structures and uses `immutable.Set`. When we apply the `for` loop, we are simply filtering only the `names` which contain a specific substring, and then, using the `yield` operator, we are creating a new `Set` structure. The `yield` operator will keep the structure you are using. For instance, if we use `List` structure, it would create a `List` instead of a `Set structure`, the

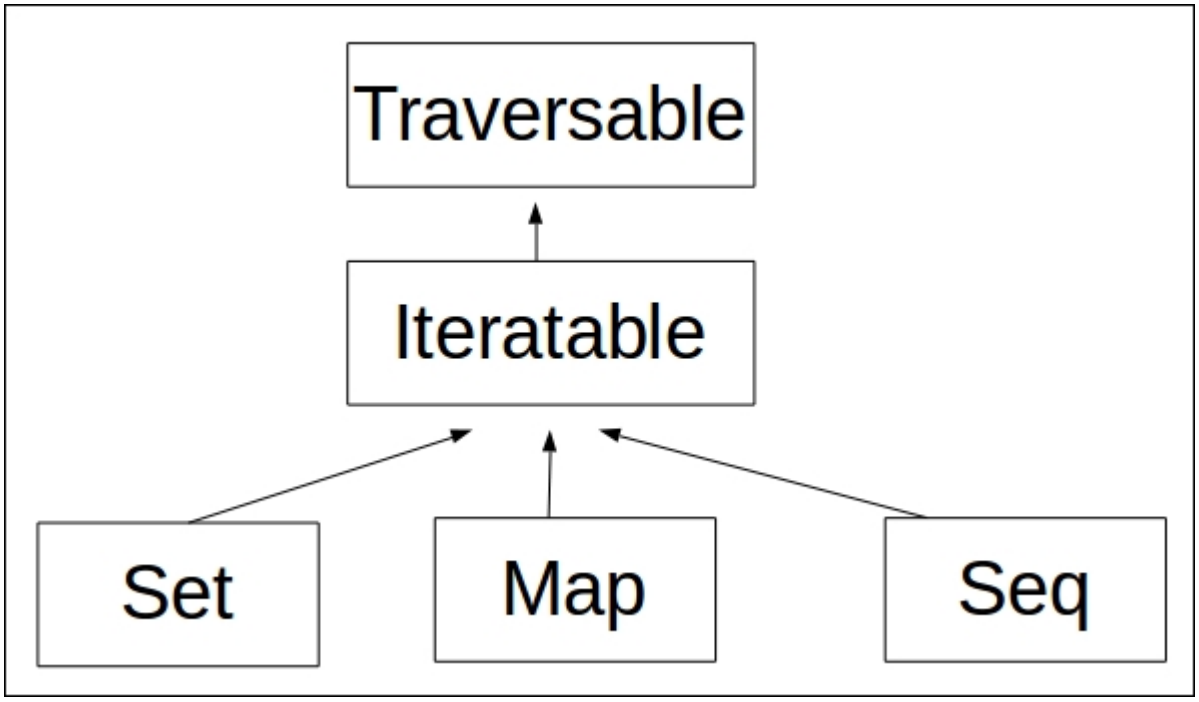
`yield` operator will always keep the same data collection you have on the variable. Another interesting aspect of the preceding code is the fact that we are holding the result of the `for` comprehension in a variable called Brazilians. Java does not have `for` comprehensions, but we could use similar code although it would require way more lines of code.

Java code for performing filtering with collections

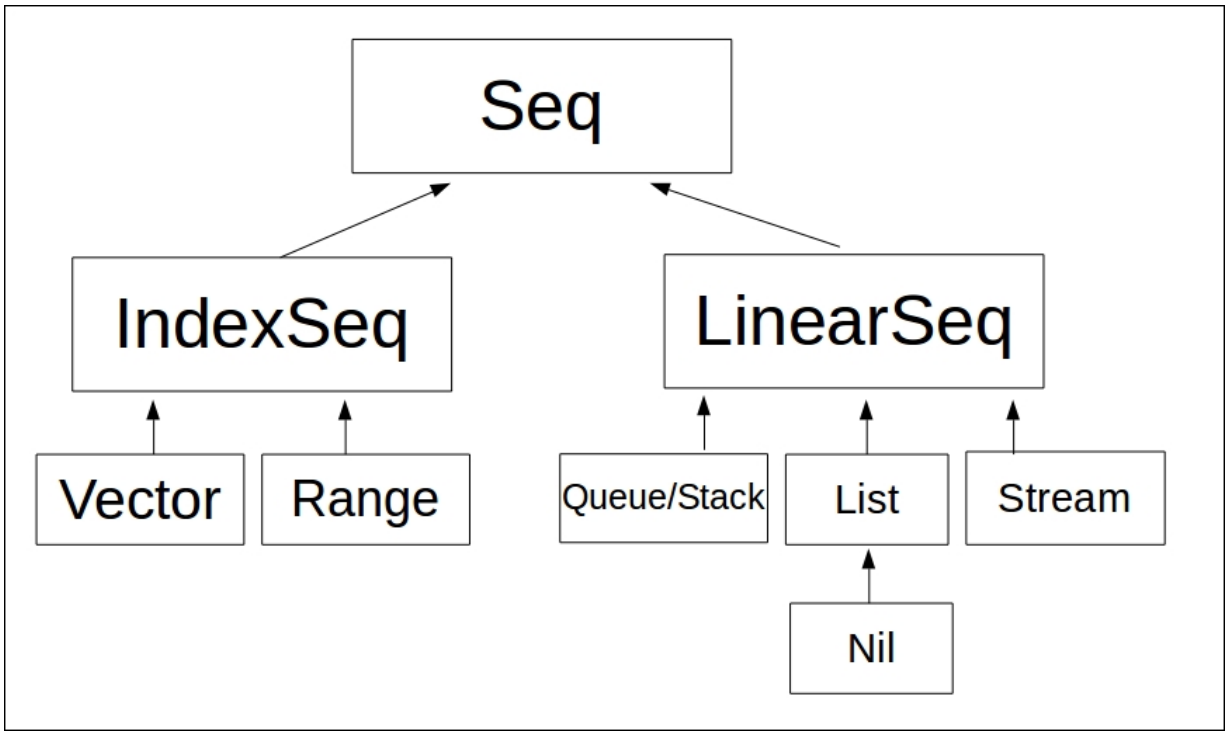
We will see how to use Java code for performing filtering with collections as follows: package scalabook.javacode.chap1; import java.util.LinkedHashSet; import java.util.Set; public class JavaNoForComprehension { public static void main(String[] args) { Set<String> names = new LinkedHashSet<>(); Set<String> brazillians = new LinkedHashSet<>(); names.add("Diego"); names.add("James"); names.add("John"); names.add("Sam"); names.add("Christophe"); for (String name: names){ if (name.contains("Die")) brazillians.add(name); } System.out.println(brazillians); } }

Scala collections

In the previous section, we saw how to create the `List` and `Set` structures in Scala in an immutable way. Now we will learn to work with the `List` and `Set` structures in a mutable way, and also with other collections such as sequences, tuples, and Maps. Let's take a look at the Scala collections hierarchy tree, as shown in the following diagram:



Now let's take a look at the Scala Seq class hierarchy. As you can see, Seq is traversable as well.



Scala collections extend from traversable, which is the main trait of all collection's descends. `List` structures, for instance, extend from Seq class hierarchy, which means sequence - `List` is a kind of sequence. All these trees are immutable or mutable depending on the Scala package you end up using.

Let's see how to perform basic mutable operations with `List` structures in Scala. In order to have filter and removal operations, we need use a `Buffer` sequence as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> var ms = scala.collection.mutable.ListBuffer(1,2,3) ms: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3) scala> ms += 4 res0: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4) scala> ms += 5 res1: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4, 5) scala> ms += 6 res2: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4, 5, 6) scala> ms(1) res3: Int = 2 scala> ms(5)**

res4: Int = 6 scala> ms -= 5 res5: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4, 6) scala> ms -= 6 res6: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2, 3, 4) scala>

Let's see the next set of code.

Creating, removing, and getting an item from a mutable list in Scala REPL

We will see how to create, remove, and get an item from a mutable list in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> var names = scala.collection.mutable.SortedSet[String]("Diego", "Poletto", "Jackson") names: scala.collection.mutable.SortedSet[String] = TreeSet(Diego, Jackson, Poletto) scala> names += "Sam" res2: scala.collection.mutable.SortedSet[String] = TreeSet(Diego, Jackson, Poletto, Sam) scala> names("Diego") res4: Boolean = true scala> names -= "Jackson" res5: scala.collection.mutable.SortedSet[String] = TreeSet(Diego, Poletto, Sam) scala>**

Have you ever wanted to return multiple values in a method? Well, in Java you have to create a class, but in Scala, there is a more convenient way to perform this task, and you won't need to create new classes each time. Tuples allow you to return or simply hold multiple values in methods without having to create a specific type.

Scala tuples

We will see Scala tuples as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
```

```
Type in expressions for evaluation. Or try :help.
scala> val config = ("localhost", 8080)
config: (String, Int) = (localhost,8080)
scala> config._1
res0: String = localhost
scala> config._2
res1: Int = 8080
scala>
```

Scala has special methods called `_1` and `_2` which you can use to retrieve a tuple's values. The only thing you have to keep in mind is the fact that values are kept in the order of insertion in the tuple.

Scala has a very practical and useful collection library. A Map, for instance, is a key/value pair that can be retrieved based on the key, which is unique. However, Map values do not need to be unique. Like other Scala collections, you have mutable and immutable Map collections. Keep in mind that Scala favors immutable collections over mutable ones.

Scala immutable Map in Scala REPL

We will see Scala immutable Map in Scala REPL as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val numbers = Map("one"   -> 1,
     |                   "two"   -> 2,
     |                   "three" -> 3,
     |                   "four"  -> 4,
     |                   "five"  -> 5,
     |                   "six"   -> 6,
     |                   "seven" -> 7,
     |                   "eight" -> 8,
     |                   "nine"  -> 9,
     |                   "ten"   -> 10)
numbers:
scala.collection.immutable.Map[String,Int] =
Map(four -> 4, three -> 3, two -> 2, six -> 6,
seven -> 7, ten -> 10, five -> 5, nine -> 9, one -
> 1, eight -> 8)
```

```
scala>
scala> numbers.keys
res0: Iterable[String] = Set(four, three, two,
six, seven, ten, five, nine, one, eight)
scala>
scala> numbers.values
res1: Iterable[Int] = MapLike(4, 3, 2, 6, 7, 10,
5, 9, 1, 8)
scala>
scala> numbers("one")
res2: Int = 1
scala>
```

As you can see, Scala uses `scala.collection.immutable.Map` when you create a Map using `Map()`. Both keys and values are iterable, and you can have access to all the keys with the `keys` method or to all the values using the `values` method.

Scala mutable Maps at Scala REPL

We will see Scala mutable Map in Scala REPL as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val map =
scala.collection.mutable.HashMap.empty[Int,String]
map: scala.collection.mutable.HashMap[Int,String]
= Map()
scala> map += (1 -> "one")
res0: map.type = Map(1 -> one)
scala> map += (2 -> "two")
res1: map.type = Map(2 -> two, 1 -> one)
scala> map += (3 -> "three")
res2: map.type = Map(2 -> two, 1 -> one, 3 ->
three)
scala> map += (4 -> "mutable")
res3: map.type = Map(2 -> two, 4 -> mutable, 1 ->
one, 3 -> three)
scala>
```

If you are dealing with mutable state, you have to be explicit and this is great in Scala, because it increases developers' awareness and avoids mutable shared state by default. So, in order to have a mutable Map, we need to explicitly create the Map with `scala.collection.mutable.HashMap`.

Monads

Monads are combinable parametrized container types which have support for higher-order functions. Remember higher-order functions are functions which receive functions as parameters and return functions as results. One of the most used functions in FP is Map. Map takes a function, applies it to each element in the container, and returns a new container.

Scala Map function in Scala REPL

We will see Map function in Scala REPL as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala>
scala> val numbers = List(1,2,3,4,5,6)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6)
scala> def doubleIt(i:Int):Double = i * 2
doubleIt: (i: Int)Double
scala> val doubled = numbers.map( doubleIt _ )
doubled: List[Double] = List(2.0, 4.0, 6.0, 8.0,
10.0, 12.0)
scala> val doubled = numbers.map( 2.0 * _ )
doubled: List[Int] = List(2.0, 4.0, 6.0, 8.0,
10.0, 12.0)
scala>
```

In the preceding code, we created a list of numbers containing 1,2,3,4,5, and 6. We also defined a Scala function called `doubleIt`, which receives an integer and multiplies it by `2.0` resulting in a double number. The `map` function calls `doubleIt` for each element in the `List (1,2,3,4,5,6)`, and the result is a new container, a brand new `List` instance containing the new values.

Scala has some syntactical sugar which helps us to be more productive. For instance, you may realize that in the previous code, we also did - `2.0 * _`. The underscore is a special operator for this specific case -- it means the current value is being iterated into the collection. Scala will create a function from this expression for us.

As you might have realized, `map` functions are pretty useful for lots of reasons: one reason is that you can do complex computations without using the `for` loop explicitly, and this makes your code functional. Secondly, we can use a `map` function to convert element types from one type to another. That's what we did in the previous code: we transformed a list of integers into a list of doubles. Look at the following: **scala> val one = Some(1) one: Some[Int] = Some(1) scala> val oneString = one.map(_.toString) oneString: Option[String] = Some(1)**

The `map` function operates over several data structures and not only collections, as you can see in the previous code. You can use the `map` function on pretty much everything in Scala language.

The `map` function is great, but you can end up with nested structures. That's why, when we are working with Monads, we use a slightly different version of the `map` function called `flatMap`, which works in a very similar way to the `map` function, but returns the values in a flat form instead of nested values.

In order to have a monad, you need to have a method called `flatMap`. Other function languages such as Haskell call `flatMap` as `bind`, and use the operator `>>=`. The syntax changes with the language, but the concept is the same.

Monads can be built in different ways. In Scala, we need a single argument constructor which will work as a monad factory. Basically, the constructor receives one type, `A`, and returns `Monad[A]` or just `M[A]`. For instance, `unit(A)` for a `List` will be `== List[A]` and `unit(A)`, where a is an Option `== Option[A]`. In Scala, you don't need to have unit; this is optional. To have a monad in Scala, you need to have map and `flatMap` implemented.

Working with Monads will make you write a little bit more code than before. However, you will get a way better API, which will be easier to reuse and your potential complexity will be managed, because you won't need to write a complex code full of `if` and for loops. The possibilities are expressed through the types, and the compiler can check it for you. Let us see a simple monad example in Scala language:

Option Monad in Scala

We will see option Monad in Scala as follows:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val a:Option[Int] = Some(1)
a: Option[Int] = Some(1)
scala> a.get
res0: Int = 1
scala> val b:Option[Int] = None
b: Option[Int] = None
scala> b.get
java.util.NoSuchElementException: None.get
  at scala.None$.get(Option.scala:347)
  at scala.None$.get(Option.scala:345)
  ... 32 elided
scala> b.getOrElse(0)
res2: Int = 0
scala> a == b
res3: Boolean = false
scala>
```

In Haskell, this also known as the Maybe monad. Option means optional value, because we are not 100% sure if the value will be present. In order to express a value, we use the `Some` type, and in order to express the lack of value, we use none. Option Monads are great, they make your code more explicit, because a method might receive or return an option, which means you are explicitly saying this could be null. However, this technique is not only more expressive but also safer, since you won't get a null pointer, because

you have a container around the value. Although, if you call the method `get` in `Option` and it is none, you will get a `NoSuchelementException`. In order to fix this, you can use the method `getOrElse`, and you can supply a fallback value which will be used in the case of none. Alright, but you might be wondering where the `flatMap` method is. Don't worry, Scala implements this method for us into the `Option` abstraction, so you can use it with no issues.

```
scala> val c = Some("one")
c: Some[String] = Some(one)
scala> c.flatMap( s => Some(s.toUpperCase) )
res6: Option[String] = Some(ONE)
```

The Scala REPL can perform autocomplete for you. If you type **C** + **Tab**, you will see all the available methods for the `Some` class. The `map` function is available for you to use, and as I said before, there is no unit function in Scala whatsoever. However, it is not wrong if you add in your APIs.

A list of all methods using the Scala REPL

Following are the list of all methods using the Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val c = Some("one")
c: Some[String] = Some(one)
scala> c.
++              count       foreach
iterator     productArity   seq         toBuffer
unzip
++:             drop        genericBuilder
last         productElement size
toIndexedSeq          unzip3
/:              dropRight   get
lastOption   productIterator slice      toIterable
view
:\              dropWhile   getOrElse
```

```
map           productPrefix  sliding     toIterator
withFilter
WithFilter      equals       groupBy
max
reduce          span         toLeft                x
addString       exists       grouped
maxBy
reduceLeft      splitAt      toList
zip
aggregate       filter       hasDefiniteSize
min           reduceLeftOption stringPrefix   toMap
zipAll
canEqual        filterNot    hashCode
minBy         reduceOption    sum          toRight
zipWithIndex
collect         find         head
mkString      reduceRight     tail         toSeq
collectFirst
flatMap         headOption   nonEmpty
reduceRightOption   tails           toSet
companion             flatten
init            orElse       repr
take
toStream
contains    fold                  inits
orNull          sameElements takeRight
toString
copy            foldLeft     isDefined
par
scan            takeWhile    toTraversable
copyToArray     foldRight       isEmpty
partition             scanLeft
to              toVector
copyToBuffer          forall
isTraversableAgain product scanRight
toArray
transpose
scala> c
```

Scala class, traits, and OO programming

As a hybrid post-functional language, Scala allows you to write OO code and create classes as well. Right now we will learn how to create classes and functions inside classes, and also how to work with traits, which are similar to Java interfaces in concept but way more powerful in practice.

A simple Scala class in Scala REPL

We will see a simple Scala class in Scala REPL as follows: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> class Calculator { | def add(a: Int, b: Int): Int = a + b | def multiply(n: Int, f: Int): Int = n * f | } defined class Calculator scala> scala> val c = new Calculator c: Calculator = Calculator@380fb434 scala> c.add(1,2) res0: Int = 3 scala> c.multiply(3,2) res1: Int = 6 scala>**

At first glance, the preceding code looks like Java. But let's add constructors, getters, and setters, and then you can see how much we can accomplish with just a few lines of code.

Scala plain old Java object in Scala REPL

Following is a Scala plain old Java object in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> class Person(| @scala.beans.BeanProperty var name:String = "", | @scala.beans.BeanProperty var age:Int = 0 |){ | name = name.toUpperCase | override def toString = "name: "**

+ name + " age: " + age | } defined class Person scala> scala> val p = new Person("Diego",31) p: Person = name: DIEGO age: 31 scala> val p1 = new Person(age = 31, name = "Diego") p1: Person = name: DIEGO age: 31 scala> p.getAge res0: Int = 31 scala> p1.getName res1: String = DIEGO scala>

Constructors in Scala are just lines of code. You might realize that we get the `name` variable, and apply a function to change the given name to upper case in the preceding example. If you want, you can put as many lines as you want, and you can perform as many computations as you wish.

On this same code, we perform method overriding as well, because we override the `toString` method. In Scala, in order to do an override, you need to use the `override` operator in front of the function definition.

We just wrote a **Plain Old Java Object** (**POJO**) with very few lines of code in Scala. Scala has a special annotation called `@scala.beans.BeanProperty`, which generates the getter and setter method for you. This is very useful, and saves lots of lines of code. However, the target needs to be public; you can't a apply `BeanProperty` annotation on top of a private `var` or `val` object.

Person class in Java

Following is a Person class in Java:

```
package scalabook.javacode.chap1;

    public class JavaPerson {

      private String name;
      private Integer age;

      public JavaPerson() {}

      public JavaPerson(String name, Integer age)
{
        super();
```

```
    this.name = name;
    this.age = age;
  }

  public JavaPerson(String name) {
    super();
    this.name = name;
  }

  public JavaPerson(Integer age) {
    super();
    this.age = age;
  }

  public Integer getAge() {
    return age;
  }

  public void setAge(Integer age) {
    this.age = age;
  }

  public String getName() {
    return name;
  }

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

}
```

Traits and inheritance

It's possible to do inheritance in Scala as well. For such a task, you use the operator `extend` after the class definition. Scala just allows you to extend one class, just like Java. Java does not allow multiple inheritance like C++. However, Scala allows it by using the Mixing technique with traits. Scala traits are like Java interface, but you can also add concrete code, and you are allowed to have as many traits as you want in your code.

Scala inheritance code in Scala REPL

Following is a Scala inheritance code in Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> class Person(
     |   @scala.beans.BeanProperty var name:String
= "",
     |   @scala.beans.BeanProperty var age:Int = 0
     | ){
     |    name = name.toUpperCase
     |    override def toString = "name: " + name
+ " age: " + age
     | }
defined class Person
scala>
scala> class LowerCasePerson(name:String,age:Int)
extends Person(name,age) {
     |    setName(name.toLowerCase)
     | }
defined class LowerCasePerson
scala>
scala> val p  = new LowerCasePerson("DIEGO
PACHECO",31)
p: LowerCasePerson = name: diego pacheco age: 31
scala> p.getName
res0: String = diego pacheco
scala>
```

Scala does not make constructors inheritance like Java. So you need to rewrite the constructors and pass the values through a super class. All code inside the class will be the secondary constructor. All code inside parentheses `()` in the class definition will be the primary constructor. It's possible to have multiple constructors using the `this` operator. For this particular implementation, we changed the default behavior and added new constructor code in order to make the given name lower case, instead of the default uppercase defined by the `Person` superclass.

Scala traits sample code in Scala REPL

Following is a Scala traits sample code in Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> trait Car
defined trait Car
scala>
scala> trait SportCar {
     |    val brand:String
     |    def run():String = "Rghhhhh Rghhhhh
Rghhhhh...."
     | }
defined trait SportCar
scala>
scala> trait Printable {
     |    def printIt:Unit
     | }
defined trait Printable
scala>
scala> class BMW extends Car with SportCar with
Printable{
     |    override val brand = "BMW"
     |    override def printIt:Unit =
println(brand + " does " + run() )
     | }
defined class BMW
scala>
scala> val x1 = new BMW
```

```
x1: BMW = BMW@22a71081
scala> x1.printIt
BMW does Rghhhhh Rghhhhh Rghhhhh....
scala>
```

In the preceding code, we created multiple traits. One is called Car, which is the mother trait. Traits support inheritance as well, and we have it with the `SportCar` trait which extends from the `Car` trait. The `SportCar` trait demands a variable called brand, and defines a concrete implementation of the function run. Finally, we have a class called `BMW` which extends from multiple traits -- this technique is called **mixing.**

Scala traits using variable mixing technique at Scala REPL

Following is a Scala traits using variable mixing technique at Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> trait SportCar { | def run():String = "Rghhhhh Rghhhhh Rghhhhh...." | } defined trait SportCar scala> scala> val bmw = new Object with SportCar bmw: SportCar = $anon$1@ed17bee scala> bmw.run res0: String = Rghhhhh Rghhhhh Rghhhhh.... scala>**

Scala is a very powerful language indeed. It's possible to add traits to a variable at runtime. When you define a variable, you can use the `with` operator after the assignment. This is a very useful feature, because it makes it easier to make function composition. You can have multiple specialized traits and just add them in your variables as you need them.

Scala allows you to create the `type` alias as well, this is a very simple technique which will increase the readability of your code. It's just a simple alias.

Scala type alias sample in Scala REPL

Following is a Scala type alias sample in Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> type Email = String
defined type alias Email
scala>
scala> val e = new Email("me@mail.com.br")
e: String = me@mail.com.br
scala>
```

When you are coding with Scala, it is highly recommended that you use the `type` alias and traits for everything, because that way you will get more advantages with your compiler, and you will avoid writing unnecessary code and unnecessary unit tests.

Case classes

We are not done yet in terms of the OO features in Scala; there is another very interesting way to work with classes in Scala: the so-called case classes. Case classes are great because you can have a class with way less number of lines of code and case classes can be part of a Pattern Matcher.

Scala case classes feature in Scala REPL

Following is a Scala case classes feature in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> case class Person(name: String, age: Int) defined class Person scala> val p = Person("Diego",31) p: Person = Person(Diego,31) scala> val p2 = Person("Diego",32) p2: Person = Person(Diego,32) scala> p.name res0: String = Diego scala> p.age res1: Int = 31 scala> p == p res2: Boolean = true scala> p.toString res3: String = Person(Diego,31) scala> p.hashCode res4: Int = 668670772 scala> p.equals(p2) res5: Boolean = false scala> p.equals(p) res6: Boolean = true scala>**

This is the Scala way to work with classes. Because this is so much easier and compact, you pretty much create a class with one line of code, and you can have the `equals` and `hashcode` methods for free.

Pattern Matcher

When you code in Java, you can use a Switch statement. However, in Scala, we have a more powerful feature called Pattern Matcher, which is a kind of switch but on steroids.

Simple Pattern Matcher in Scala

Following is a Simple Pattern Matcher in Scala:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> def resolve(choice:Int):String = choice
match {
     |     case 1 => "yes"
     |     case 0 => "no"
     |     case _ => throw new
IllegalArgumentException("Valid arguments are: 0
or 1. Your arg is:
     |           " + choice)
     | }
resolve: (choice: Int)String
scala> println(resolve(0))
no
scala> println(resolve(1))
yes
scala> try {
     |   println(resolve(33))
     | } catch{
     |   case e:Exception => println("Something
Went Worng. EX: " + e)
     | }
Something Went Worng. EX:
java.lang.IllegalArgumentException: Valid
arguments are: 0 or 1. Your arg is: 33
scala>
```

Scala uses Pattern Matcher for error handling. Java does not have Pattern Matcher like Scala. It's similar to a switch statement;

however, Pattern Matcher can be used in a method return statement as you can see in the preceding code. Scala developers can specify a special operator called `_` (Underscore), which allows you to specify anything in the Pattern Matcher scope. This behavior is similar to `else` in an `if` conditional. However, in Scala, you can use `_` in several places, and not only as the otherwise clause, like in Java switch.

Error handling in Scala is similar to error handling in Java. We use try...catch blocks. The main difference is that you have to use Pattern Matcher in Scala, which is great because it adds more flexibility to your code. Pattern Matcher in Scala can operate against many data structures like case classes, collections, integers, and strings.

The preceding code is pretty simple and straightforward. Next we will see a more complex and advanced code using the Scala Pattern Matcher feature.

Advanced pattern matcher in Scala REPL

Following is an Advanced Pattern Matcher using Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> def factorial(n:Int):Int = n match { | case 0 => 1 | case n => n * factorial(n - 1) | } factorial: (n: Int)Int scala> scala> println(factorial(3)) 6 scala> println(factorial(6)) 720 scala>**

Pattern Matcher can be used in a very functional way. For instance, in the preceding code, we use the Pattern Matcher for recursion. There is no need to create a variable to store the result, we can put the Pattern Matcher straight to the function return, which is very convenient and saves lots of lines of code.

Advanced complex pattern matcher in Scala REPL

Following is an Advanced complex Pattern Matcher using Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> trait Color defined trait Color scala> case class Red(saturation: Int) extends Color defined class Red scala> case class Green(saturation: Int) extends Color defined class Green scala> case class Blue(saturation: Int) extends Color defined class Blue scala> def matcher(arg:Any): String = arg match { | case "Scala" => "A Awesome Language" | case x: Int => "An Int with value " + x | case Red(100) => "Red sat 100" | case Red(_) => "Any kind of RED sat" | case Green(s) if s == 233 => "Green sat 233" | case Green(s) => "Green sat " + s | case c: Color => "Some Color: " + c | case w: Any => "Whatever: " + w | } matcher: (arg: Any)String scala> println(matcher("Scala")) A Awesome Language scala> println(matcher(1)) An Int with value 1 scala> println(matcher(Red(100))) Red sat 100 scala> println(matcher(Red(160))) Any kind of RED sat scala> println(matcher(Green(160))) Green sat 160 scala> println(matcher(Green(233))) Green sat 233 scala> println(matcher(Blue(111))) Some Color: Blue(111) scala> println(matcher(false)) Whatever: false scala> println(matcher(new Object)) Whatever: java.lang.Object@b56c222 scala>**

The Scala Pattern Matcher is really amazing. We just used an `if` statement in the middle of the Pattern Matcher, and also `_` to specify a match for any kind of red value. We also used case classes in the middle of the Pattern Matcher expressions.

Partial functions

Partial functions are great for function composition. They can operate with case statements as we just learned from Pattern Matcher. Partial functions are great in the sense of function composition. They allow us to define a function in steps. Scala frameworks and libraries use this feature a lot to create abstractions and callback mechanisms. It's also possible to check if a partial function is being supplied or not.

Partial functions are predictable, because the caller can check beforehand if the value will be applied to the partial function or not. Partial function can be coded with or without case-like statements.

Simple Partial function in Scala REPL

Following is a simple Partial function using Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val positiveNumber = new
PartialFunction[Int, Int] {
     |   def apply(n:Int) = n / n
     |   def isDefinedAt(n:Int) = n != 0
     | }
positiveNumber: PartialFunction[Int,Int] =
<function1>
scala>
scala> println( positiveNumber.isDefinedAt(6) )
true
scala> println( positiveNumber.isDefinedAt(0) )
false
scala>
scala> println( positiveNumber(6) )
1
scala> println( positiveNumber(0) )
  java.lang.ArithmeticException: / by zero
  at $anon$1.apply$mcII$sp(<console>:12)
```

```
  ... 32 elided
scala>
```

Partial functions are Scala classes. They have some methods you need to provide, for instance, `apply` and `isDefinedAt`. The function `isDefinedAt` is used by the caller to check if the `PartialFunction` will accept and operate with the value supplied. The `apply` function will do the work when the `PartialFunction` is executed by Scala.

Scala PartialFunction without OO using case statements in Scala REPL

Following is a Scala PartialFunction without OO using case statements in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> val positiveNumber:PartialFunction[Int, Int] = { | case n: Int if n != 0 => n / n | } positiveNumber: PartialFunction[Int,Int] = <function1> scala> scala> println(positiveNumber.isDefinedAt(6)) true scala> println(positiveNumber.isDefinedAt(0)) false scala> scala> println(positiveNumber(6)) 1 scala> println(positiveNumber(0)) scala.MatchError: 0 (of class java.lang.Integer) at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:253) at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:251) at $anonfun$1.applyOrElse(<console>:11) at $anonfun$1.applyOrElse(<console>:11) at scala.runtime.AbstractPartialFunction$mcII$sp.apply$mcII$sp (AbstractPartialFunction.scala:36) ... 32 elided scala>**

Scala was a more fluent way to work with `PartialFunction` using the `case` statements. When you use the `case` statements, you don't need to supply the `apply` and `isDefinedAt` functions, since the Pattern Matcher takes care of that.

PartialFunction composition in Scala REPL

Following is a PartialFunction composition in Scala REPL:

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> val even:PartialFunction[Int, String] = {
     |   case i if i%2 == 0 => "even"
     | }
even: PartialFunction[Int,String] = <function1>
scala>
scala> val odd:PartialFunction[Int, String] = {
case _ => "odd"}
odd: PartialFunction[Int,String] = <function1>
scala>
scala> val evenOrOdd:(Int => String) = even orElse
odd
evenOrOdd: Int => String = <function1>
scala>
scala> println( evenOrOdd(1) == "odd"  )
true
scala> println( evenOrOdd(2) == "even" )
true
scala>
```

Scala allows us to compose as many `PartialFunctions` as we want. `PartialFunction` composition happens with the `orElse` function. In the preceding code, we defined an immutable variable called `even`, which verifies even numbers. Secondly, we created a second immutable variable called `odd`, which checks for odd numbers. Then we did the composition, and created a third `PartialFunction` called `evenOrOdd` with compose even and odd using the `orElse` operator.

Package objects

Scala has packages like Java. However, Scala packages are also objects, and you can have code inside a package. Java does not have the same power as Scala in terms of packages. If you add code to a package, it will be available to all classes and functions within that package.

package.scala

Your `package.scala` file should contain the following code package com.packait.scala.book

package object commons {

val PI = 3.1415926

object constraintsHolder {

val ODD = "Odd"

val EVEN = "Even"

}

def isOdd(n:Int):String = if (n%2==0) constraintsHolder.ODD else null

def isEven(n:Int):String = if (n%2!=0) constraintsHolder.EVEN

```
else null


def show(s:String) = println(s)


}
```

This is the Scala package object. There is this special token called `package` object which you use to define common code to all classes, objects, and functions that are defined inside this package or sub-package. For this case, we define a value of PI as a constant and also one object holder containing the String values for `Odd` and `Even`. There are also three helper functions, which can and will be used by the classes inside this package.

MainApp.scala

Your `MainApp.scala` file should contain the following code package com.packait.scala.book.commons

```
object MainApp extends App {


show("PI is: " + PI)

show(constraintsHolder.getClass.toString())

show( isOdd(2) )

show( isOdd(6) )


show( isEven(3) )
```

```
show( isEven(7) )


}
```

As you can see in the preceding code, this new object is placed in the package: `com.packait.scala.book.commons`. Another interesting thing is the fact that we don't have any import statement here because of the `package` object feature. When you compile and run this program, you will see the following output: **PI is: 3.1415926 class com.packait.scala.book.commons.package$constraintsHolder$ Odd Odd Even Even**

Scala uses the `Package` object a great deal providing lots of shortcuts and convenience for all Scala developers. The following is the Scala `package` object definition: /* __

*\

** _____ ___ / / ___ Scala API **

** / __/ __// _ | / / / _ | (c) 2003-2013, LAMP/EPFL

**

** __\ \/ /__/ __ |/ /__/ __ | http://scala-lang.org/

**

** /____/___/_/ |_/____/_/ | |

**

** |/

**

*

*/

/**

* Core Scala types. They are always available without an explicit import.

* @contentDiagram hideNodes "scala.Serializable"

*/

package object scala {

type Throwable = java.lang.Throwable type Exception = java.lang.Exception type Error = java.lang.Error

type RuntimeException = java.lang.RuntimeException type NullPointerException =

java.lang.NullPointerException

type ClassCastException =

java.lang.ClassCastException

type IndexOutOfBoundsException =

java.lang.IndexOutOfBoundsException type ArrayIndexOutOfBoundsException =

java.lang.ArrayIndexOutOfBoundsException type StringIndexOutOfBoundsException =

```
java.lang.StringIndexOutOfBoundsException type
UnsupportedOperationException =

java.lang.UnsupportedOperationException type
IllegalArgumentException =

java.lang.IllegalArgumentException

type NoSuchElementException =

java.util.NoSuchElementException

type NumberFormatException =

java.lang.NumberFormatException

type AbstractMethodError =

java.lang.AbstractMethodError

type InterruptedException =

java.lang.InterruptedException


// A dummy used by the specialization annotation.

val AnyRef = new Specializable {

override def toString = "object AnyRef"

}


type TraversableOnce[+A] = scala.collection.TraversableOnce[A]
```

```
type Traversable[+A] = scala.collection.Traversable[A]
val Traversable = scala.collection.Traversable
type Iterable[+A] = scala.collection.Iterable[A]
val Iterable = scala.collection.Iterable
type Seq[+A] = scala.collection.Seq[A]
val Seq = scala.collection.Seq

type IndexedSeq[+A] = scala.collection.IndexedSeq[A]
val IndexedSeq = scala.collection.IndexedSeq
type Iterator[+A] = scala.collection.Iterator[A]
val Iterator = scala.collection.Iterator
type BufferedIterator[+A] = scala.collection.BufferedIterator[A]

type List[+A] = scala.collection.immutable.List[A]
val List = scala.collection.immutable.List
val Nil = scala.collection.immutable.Nil
type ::[A] = scala.collection.immutable.::[A]
val :: = scala.collection.immutable.::
val +: = scala.collection.+:
val :+ = scala.collection.:+
```

type Stream[+A] = scala.collection.immutable.Stream[A]

val Stream = scala.collection.immutable.Stream val #:: = scala.collection.immutable.Stream.#::

type Vector[+A] = scala.collection.immutable.Vector[A]

val Vector = scala.collection.immutable.Vector

type StringBuilder = scala.collection.mutable.StringBuilder val StringBuilder = scala.collection.mutable.StringBuilder

type Range = scala.collection.immutable.Range val Range = scala.collection.immutable.Range

// Numeric types which were moved into scala.math.*

type BigDecimal = scala.math.BigDecimal val BigDecimal = scala.math.BigDecimal

type BigInt = scala.math.BigInt

val BigInt = scala.math.BigInt

type Equiv[T] = scala.math.Equiv[T]

val Equiv = scala.math.Equiv

type Fractional[T] = scala.math.Fractional[T]

val Fractional = scala.math.Fractional

```
type Integral[T] = scala.math.Integral[T]
val Integral = scala.math.Integral

type Numeric[T] = scala.math.Numeric[T]
val Numeric = scala.math.Numeric

type Ordered[T] = scala.math.Ordered[T]
val Ordered = scala.math.Ordered

type Ordering[T] = scala.math.Ordering[T]
val Ordering = scala.math.Ordering

type PartialOrdering[T] = scala.math.PartialOrdering[T]
type PartiallyOrdered[T] = scala.math.PartiallyOrdered[T]

type Either[+A, +B] = scala.util.Either[A, B]
val Either = scala.util.Either

type Left[+A, +B] = scala.util.Left[A, B]
val Left = scala.util.Left
```

```
type Right[+A, +B] = scala.util.Right[A, B]

val Right = scala.util.Right


// Annotations which we might move to annotation.*

/*

type SerialVersionUID = annotation.SerialVersionUID

type deprecated = annotation.deprecated type deprecatedName =
annotation.deprecatedName type inline = annotation.inline

type native = annotation.native

type noinline = annotation.noinline type remote = annotation.remote

type specialized = annotation.specialized type transient =
annotation.transient type throws = annotation.throws

type unchecked = annotation.unchecked.unchecked type volatile =
annotation.volatile */

}
```

Functions

Like any great FP language, Scala has lots of built-in functions. These functions make our code more fluent and functional; now it's time to learn some of these functions: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> // Creates the numbers 1,2,3,4,5 and them multiply they by 2 and creates a new Vector scala> println ((1 to 5).map(_*2)) Vector(2, 4, 6, 8, 10) scala> scala> // Creates 1,2,3 and sum them all with each orher and return the total scala> println ((1 to 3).reduceLeft(_+_)) 6 scala> scala> // Creates 1,2,3 and multiply each number by it self and return a Vector scala> println ((1 to 3).map(x=> x*x)) Vector(1, 4, 9) scala> scala> // Creates numbers 1,2,3,4 ans 5 filter only Odd numbers them multiply them odds by 2 and return a Vector scala> println ((1 to 5) filter { _%2 == 0 } map { _*2 }) Vector(4, 8) scala> scala> // Creates a List with 1 to 5 and them print each element being multiplyed by 2 scala> List(1,2,3,4,5).foreach ((i:Int) => println(i * 2)) 2 4 6 8 10 scala> scala> // Creates a List with 1 to 5 and then print each element being multiplied by 2 scala> List(1,2,3,4,5).foreach (i => println(i * 2)) 2 4 6 8 10 scala> scala> // Drops 3 elements from the lists scala> println(List(2,3,4,5,6).drop(3)) List(5, 6) scala> println(List(2,3,4,5,6) drop 3) List(5, 6) scala> scala> // Zip 2 lists into a single one: It will take 1 element of each list and create a pair List scala> println(List(1,2,3,4).zip(List(6,7,8))) List((1,6), (2,7), (3,8)) scala> scala> // Take nested lists and create a single list with flat elements scala> println(List(List(1, 2), List(3, 4)).flatten) List(1, 2, 3, 4) scala> scala> // Finds a person in a List by Age scala> case class Person(age:Int,name:String) defined class Person scala> println(List(Person(31,"Diego"),Person(40,"Nilseu")).find((p:Person) => p.age <= 33)) Some(Person(31,Diego)) scala>**

Partial application

In Scala, the underscore(_) means different things in different contexts. The underscore can be used to partially apply a function. It means a value will be supplied later. This feature is useful for function composition and allows you to reuse functions. Let's see some code.

Partial function in Scala REPL

Following is an example using Partial function in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> def sum(a:Int,b:Int) = a+b sum: (a: Int, b: Int)Int scala> scala> val add6 = sum(6,_:Int) add6: Int => Int = <function1> scala> scala> println(add6(1)) 7 scala>**

In the preceding code, first, we define a function called `sum`, which takes two `Int` parameters and calculates a sum of these two parameters. Later, we define a function and hold it as a variable called `add6`. For the `add6` function definition, we just call the sum function passing `6` and `_`. Scala will get the parameter passed through `add6`, and pass it through the `sum` function.

Curried functions

This feature is very popular in function languages like Haskell. Curried functions are similar to partial applications, because they allow some arguments to pass now and others later. However, they are a little bit different.

Curried functions - Scala REPL

Following is an example using curried function in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> // Function Definition scala> def sum(x:Int) (y:Int):Int = x+y sum: (x: Int)(y: Int)Int scala> scala> // Function call - Calling a curried function scala> sum(2)(3) res0: Int = 5 scala> scala> // Doing partial with Curried functions scala> val add3 = sum(3) _ add3: Int => Int = <function1> scala> scala> // Supply the last argument now scala> add3(3) res1: Int = 6 scala>**

For the preceding code, we create a curried function in the function definition. Scala allows us to transform regular/normal functions into curried functions. The following code shows the usage of the `curried` function.

Curried transformation in Scala REPL

Following is an example using curried transformation in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> def normalSum(x:Int,y:Int):Int=x+y normalSum: (x: Int, y: Int)Int scala> scala> val curriedSum = (normalSum _).curried curriedSum: Int => (Int => Int) = <function1> scala> scala> val add3= curriedSum(3) add3: Int => Int = <function1> scala> scala> println(add3(3)) 6 scala>**

Operator overloading

Like C++, Scala permits operator overload. This feature is great for creating custom **Domain Specific Languages** (**DSL**), which can be useful to create better software abstractions or even internal or external APIs for developers, or for business people. You should use this feature with wisdom -- imagine if all frameworks decide to overload the same operators with implicits! You might run into trouble. Scala is a very flexible language compared to Java. However, you need to be careful, otherwise you could create code that's hard to maintain or even incompatible with other Scala applications, libraries, or functions.

Scala operator overloading in Scala REPL

Following is an example using Scala operator overloading in Scala REPL: **$ scala Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77). Type in expressions for evaluation. Or try :help. scala> case class MyNumber(value:Int){ | def +(that:MyNumber):MyNumber = new MyNumber(that.value + this.value) | def +(that:Int):MyNumber = new MyNumber(that + this.value) | } defined class MyNumber scala> val v = new MyNumber(5) v: MyNumber = MyNumber(5) scala> scala> println(v) MyNumber(5) scala> println(v + v) MyNumber(10) scala> println(v + new MyNumber(4)) MyNumber(9) scala> println(v + 8) MyNumber(13) scala>**

As you can see, we have two functions called `+`. One of this functions receives a `MyNumber` case class, and the other receives a `Int` value. You can use OO in Scala with regular classes and functions as well if you wish. We're also favoring immutability here because we always create a new instance of `MyNumber` when the operation `+` happens.

Implicits

Implicits allow you to do magic in Scala. With great power comes great responsibility. Implicits allow to you create very powerful DSL, but they also allow you to get crazy, so do it with wisdom. You are allowed to have implicit functions, classes, and objects. The Scala language and other core frameworks from the Scala ecosystem like Akka and PlayFramework use implicits many times.

Scala Implicits in SCALA REPL

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> implicit def transformStringtoInt(n:String)
= n.toInt
warning: there was one feature warning; re-run
with -feature for details
transformStringtoInt: (n: String)Int
scala>
scala> val s:String = "123456"
s: String = 123456
scala> println(s)
123456
scala>
scala> val i:Int = s
i: Int = 123456
scala> println(i)
123456
scala>
```

To use implicits, you need to use the keyword `implicit` before a function. Scala will implicitly call that function when it is appropriate. For this case, it will call to convert the `String` type to `Int` type as we can see.

Implicit Parameter at Scala REPL

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> implicit val yValue:Int = 6
yValue: Int = 6
scala> def sum(x:Int)(implicit yValue:Int) = x +
yValue
sum: (x: Int)(implicit yValue: Int)Int
scala> val result = sum(10)
result: Int = 16
scala> println(result)
16
scala>
```

For this other case, given in the last code, we use an implicit parameter in the function `sum`. We also used a curried function here. We defined the `implicit` function first, and then called the `sum` function. This technique is good for externalized functions configuration and values you would let it hard code. It also saves lines of code, because you don't need to pass a parameter to all functions all the time, so it's quite handy.

Futures

Futures enable an efficient way to write parallel operations in a nonblocking IO fashion. Futures are placeholder objects for values that might not exist yet. Futures are composable, and they work with callbacks instead of traditional blocking code.

Simple Future code in Scala REPL

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> import concurrent.Future
import concurrent.Future
scala> import
concurrent.ExecutionContext.Implicits.global
import
concurrent.ExecutionContext.Implicits.global
scala>
scala> val f: Future[String] = Future { "Hello
world!" }
f: scala.concurrent.Future[String] = Success(Hello
world!)
scala>
scala> println("Result: " + f.value.get.get)
Result: Hello world!
scala>
scala> println("Result: " + f)
Result: Success(Hello world!)
scala>
```

In order to work with futures in Scala, we have to import `concurrent.Future`. We also need an executor, which is a way to work with threads. Scala has a default set of execution services. You can tweak it if you like, however, for now we can just use the defaults; to do that, we just import `concurrent.ExecutionContext.Implicits.global`.

It's possible to retrieve the `Future` value. Scala has a very explicit API, which makes the developer's life easier, and also gives good samples for how we should code our own APIs. Future has a method called `value`, which returns `Option[scala.util.Try[A]]` where `A` is the generic type you are using for the future; for our case, it's a String `A`. `Try` is a different way to do a try...catch, and this is safer, because the caller knows beforehand that the code they are calling may fail. `Try[Optional]` means that Scala will try to run some code and the code may fail -- even if it does not fail, you might receive `None` or `Some`. This type of system makes everybody's lives better, because you can have `Some` or `None` as the Option return. Futures are a kind of callback. For our previous sample code, the result was obtained quite quickly, however, we often use futures to call external APIs, REST services, Microservices, SOAP Webservices, or any code that takes time to run and might not get completed. Futures also work with Pattern Matcher. Let's see another sample code.

A complete Future sample at Scala REPL

```
$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit
Server VM, Java 1.8.0_77).
Type in expressions for evaluation. Or try :help.
scala> import concurrent.Future
import concurrent.Future
scala> import
concurrent.ExecutionContext.Implicits.global
import
concurrent.ExecutionContext.Implicits.global
scala> import scala.util.{Success, Failure}
import scala.util.{Success, Failure}
scala> def createFuture():Future[Int] = {
     | Future {
     | val r = scala.util.Random
     | if (r.nextInt(100)%2==0) 0 else throw new
RuntimeException("ODD numbers are not good here :(
")
     | }
     | }
createFuture: ()scala.concurrent.Future[Int]
```

```
scala> def evaluateFuture(f:Future[_]) {
     | f.onComplete {
     | case Success(i) => println(s"A Success $i
")
     | case Failure(e) => println(s"Something went
wrong. Ex: ${e.getMessage}")
     | }
     | }
evaluateFuture: (f:
scala.concurrent.Future[_])Unit
scala> evaluateFuture(createFuture)
scala> Something went wrong. Ex: ODD numbers are
not good here :(
evaluateFuture(createFuture)
A Success 0
scala> evaluateFuture(createFuture)
Something went wrong. Ex: ODD numbers are not good
here :(
scala> evaluateFuture(createFuture)
Something went wrong. Ex: ODD numbers are not good
here :(
scala> evaluateFuture(createFuture)
A Success 0
scala> evaluateFuture(createFuture)
A Success 0
scala>
```

There is a function called `createFuture`, which creates `Future[Int]` each time you call it. In the preceding code, we use `scala.util.Random` to generate random numbers between 0 and 99. If the number is even, we return a `0`, which means success. However, if the number is odd, we return a `RuntimeException`, which will mean a failure.

There is a second function called `evaluateFuture`, which receives any Future. We allow a result of any kind of generic parameterized type of function, because we used the magic underscore `_`. Then we apply Pattern Matcher with two case classes: `Success` and `Failure`. In both the cases, we just print on `stdin`. We also use another interesting and handy Scala feature called String interpolation. We need to we start the String with `s` before `""`. This allows us to use expressions with `$` and `${}` to evaluate any variable in the context. This is a different approach for String concatenation from what we

have done so far. Later, we made `6` calls for the `evaluteFuture` function, passing a new Future each time, created by the function `createFuture`.

Reactive Programing and RxScala

Reactive programming is better, scalable, and a faster way to build applications. Reactive Programing can be done with OO languages, however, they make a lot of sense with FP languages. When FP is married to Reactive Programing, we get something called **Functional Reactive Programing** (**FRP**). Scala FRP can be used for many purposes like GUI, Robotics, and Music, because it gives you a better model to model time. Reactive programming is a new technique, which works with Streams(also known as Data Flows). Streams is a way to think and code applications in a way which can express data transformations and flow. The main idea is to propagate changes through a circuit or flow. Yes, we are talking about a new way to do async programming.

The main library for Reactive Programing is called **Reactive Extensions** (**Rx**) - http://reactivex.io/), originally built for .NET by Eric Meijer. It combines the best ideas from the Observer and Iterator Patterns, and FP. Rx has implementations for many languages like Scala, Java, Python, .NET, PHP, and others (https://github.com/ReactiveX). Coding with Rx is easy, and you can create Streams, combine with query-like operators, and also listen (subscribe) to any observable Streams to perform data transformations. Rx is used by many successful companies today like Netflix, GitHub, Microsoft, SoundCloud, Couchbase, Airbnb, Trello, and several others. In this book, we will use RxScala, which is the Scala implementation of the Reactive Streams.

The following table shows the main class/concepts you need to know in order to work with Rx.

Term / Class	Concept
Observable	Create async composable Streams from sources.
Observer	A callback function type.
Subscription	The bound between the Subscriber and the Observable. Receives notifications from Observables.

Reactive Streams is also the name of a common specification trying to consolidate and standardize the reactive stream processing, There are several implementations such as RxJava/RxScala, Reactor, Akka, Slick, and Vert.x. You can find more at https://github.com/reactive-streams/reactive-streams-jvm.

Back to the Observables -- we can perform all kinds of operations with observables. For instance, we can filter, select, aggregate, compose, perform time-based operations, and apply backpressure. There are two big wins with Observables instead of callbacks. First of all, Observables are not opinionated about how low-level I/O and threading happens, and secondly, when you are doing complex code, callbacks tend to be nested, and that is when things get ugly and hard to read. Observables have a simple way to do composition thanks to FP.

Observables push values to consumers whenever values are available, which is great because then the values can arrive in sync or async fashion. Rx provides a series of collection operators to do all sorts of data transformations you may need. Let's see some code now. We will use RxScala version 0.26.1, which is compatible with

RxJava version 1.1.1+. RxScala is just a wrapper for RxJava (Created by Netflix). Why not use RxJava straight? Because the syntax won't be pleasant; with RxScala, we can have a fluent Scala experience. RxJava is great, however, Java syntax for this is not pleasant - as Scala is, in fact, pretty ugly.

Simple Observables Scala with RxScala

```
package scalabook.rx.chap1

import rx.lang.scala.Observable
import scala.concurrent.duration._

object SimpleRX extends App {

  val o = Observable.
            interval(100 millis).
            take(5)

  o.subscribe( x => println(s"Got it: $x") )
  Thread.sleep(1000)

  Observable.
      just(1, 2, 3, 4).
      reduce(_+_).
      subscribe( r => println(s"Sum 1,2,3,4 is $r
in a Rx Way"))

}
```

If you run this preceding Scala program, you will see the following output:

Simple Observables Scala with RxScala - Execution in the console

```
Got it: 0
Got it: 1
```

```
Got it: 2
Got it: 3
Got it: 4
Sum 1,2,3,4 is 10 in a Rx Way
```

If you try to run this code in the Scala REPL, it will fail, because we need the RxScala and RxJava dependencies. For this, we will need SBT and dependency management. Do not worry, we will cover how to work with SBT in our Scala application in the next chapter.

Going back to the observables, we need to import the Scala Observable. Make sure you get it from the Scala package, because if you get the Java one, you will have issues: in the very first part of the code, we will get numbers starting from 0 each 100 milliseconds, and this code would run forever. To avoid this, we use the take function to put a limit into the collection, so we will get the first five values. Then, later, we subscribe to the observer, and when data is ready, our code will run. For the first sample, it's pretty easy, we are just printing the values we have got. There is a thread sleep in this program, otherwise, the program would terminate, and you would not see any value on the console.

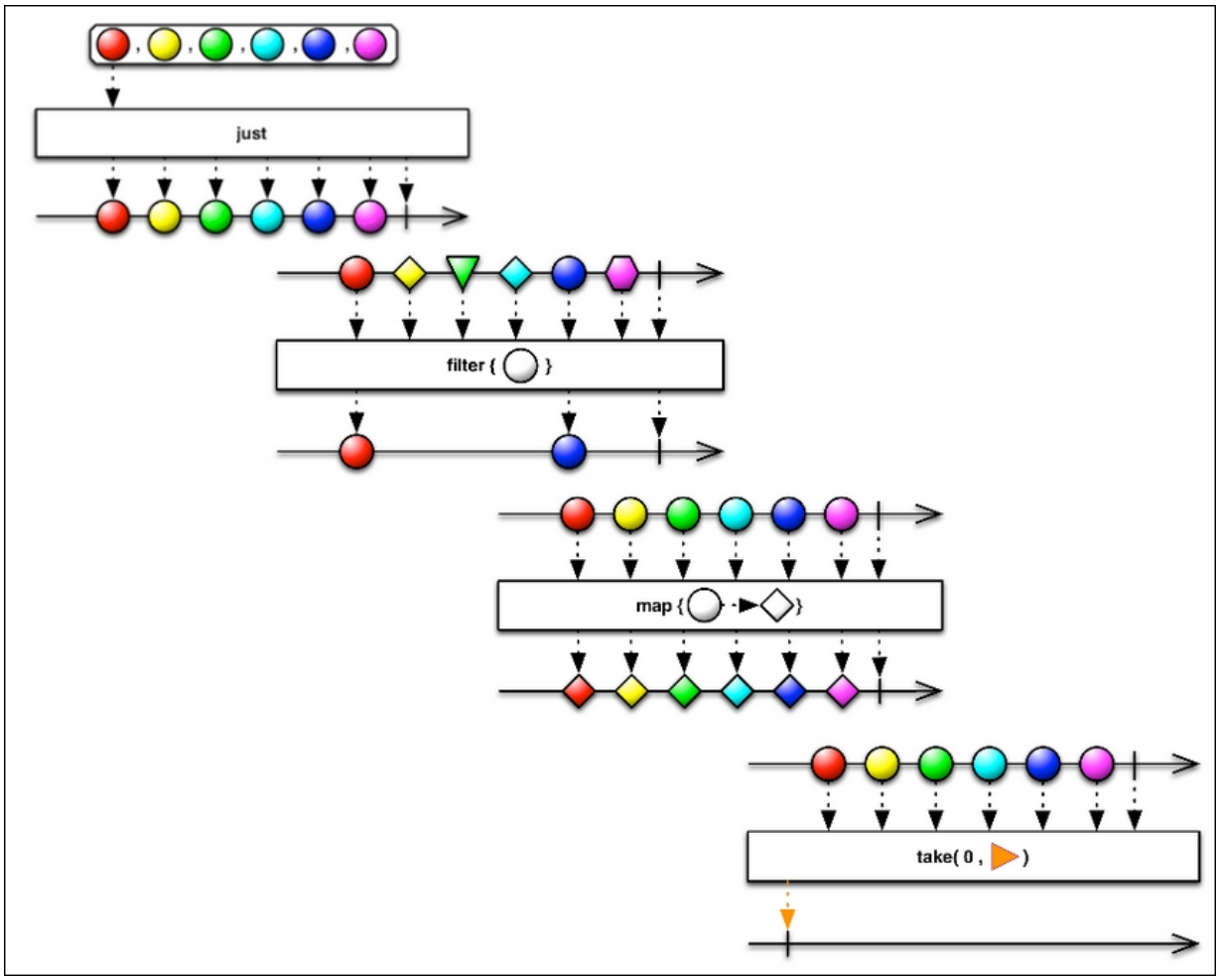
The second part of the code does something more interesting. First of all, it creates an Observable from a static list of values, which are 1,2,3, and 4. We apply a reduce function into the elements, which will sum all the elements with each other, and then we subscribe and print the result.

Complex Scala with RxScala Observables

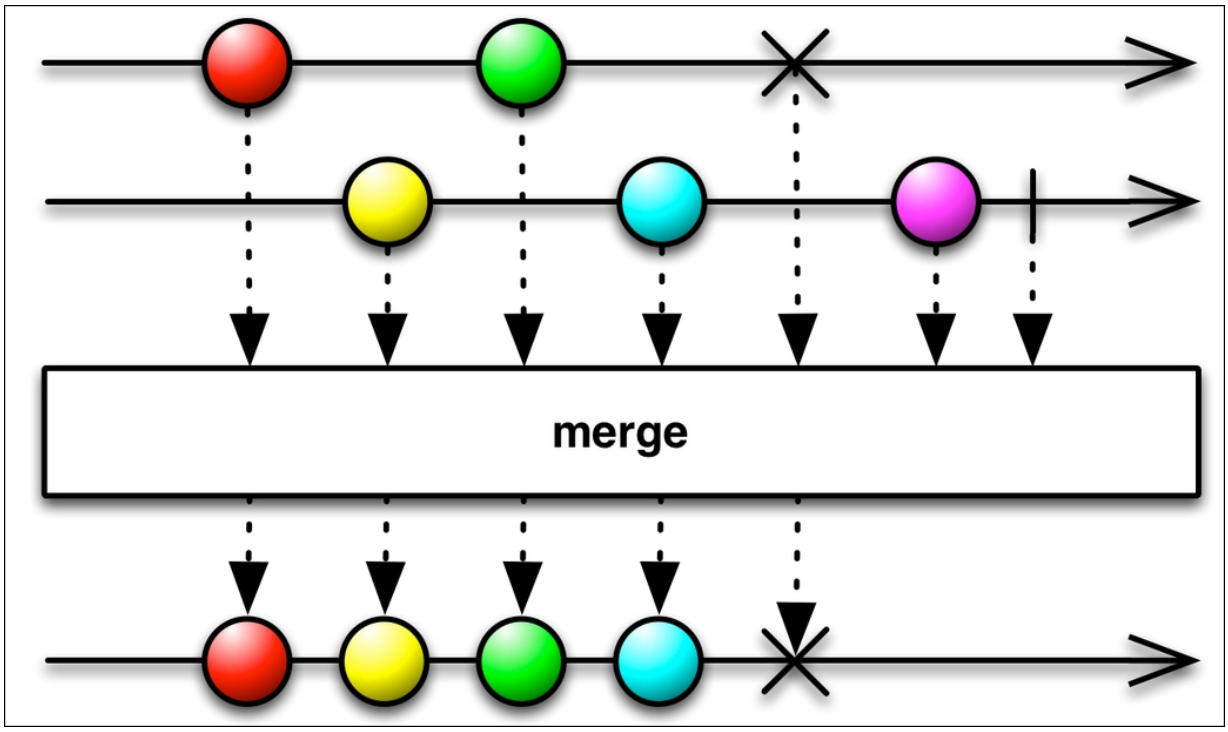
```
package scalabook.rx.chap1

import rx.lang.scala.Observable

object ComplexRxScala extends App {

  Observable.
      just(1,2,3,4,5,6,7,8,9,10).       //
```

```
1,2,3,4,5,6,7,8,9,10
      filter( x => x%2==0).              //
2,4,6,8,10
      take(2).                           // 2,4
      reduce(_+_).                       // 6
      subscribe( r => println(s"#1 $r"))

   val o1 = Observable.
            just(1,2,3,4,5,6,7,8,9,10).  //
1,2,3,4,5,6,7,8,9,10
            filter( x => x%2==0).        // 2, 4,
6, 8, 10
            take(3).                     // 2, 4
,6
            map( n => n * n)             // 4, 16,
36

   val o2 = Observable.
            just(1,2,3,4,5,6,7,8,9,10). //
1,2,3,4,5,6,7,8,9,10
            filter( x => x%2!=0).        // 1, 3,
5, 7, 9
            take(3).                     // 1, 3, 5
            map( n => n * n)             // 1, 9,
25

   val o3 = o1.
           merge(o2).                   // 2,16,
36, 1, 9, 25
            subscribe( r => println(s"#2 $r"))

}
```

The preceding first part of the code creates an Observable with numbers from 1 to 10, and then applies a `filter` function, which will get only the even numbers. It then reduces them, calculates their sum, and lastly, prints the solution. You can visualize it as depicted in the following image:



For the second part of the code, we create two different observables. The first one is with even numbers and the second one is with odd numbers. These two observables are decoupled from each other; you can control as many observables you want. Later on, the code uses a merge function to join these two observables into a third and new observable containing the content of the first and second observables.



Merging 2 Observables

There are many functions and options, and you can see the whole list at http://rxmarbles.com/ and https://github.com/ReactiveX/RxScala. For the sake of simplicity, for now, we are just working with numbers. Later, we will use this to do more advance compositions including database calls and external web services calls.

Summary

In this chapter, we learned the basic concepts of FP, Reactive Programing, and the Scala language. We learned about the basic constructs of the Scala language and Functional Programming, functions, collections, and OO in Scala, and concurrent programming with Futures.

Next, we will see how to use SBT to build Scala projects. We will learn how to compile and run Scala applications using SBT.

Chapter 2. Creating Your App Architecture and Bootstrapping with SBT

In the previous chapter, we learned about Functional Programing and Scala. This chapter will be focused on **Simple Build Tool** (**SBT**) and Activator in order to Bootstrap complex Scala and Play framework projects. Using SBT and Activator, we can perform several development tasks such as building, running tests, and deploying the application (which will be covered in detail in Chapter 10, *Scaling Up*). Let's get started.

In this chapter, we will see the following topics:

- SBT basics--installation, structure, and dependencies
- Activator basics--creating projects
- Overall architecture of our application

Introducing SBT

SBT is the ultimate Scala solution for building and packing Scala applications. SBT has lots of plugins, such as Eclipse and IntelliJ IDEA projects generation, which help a great deal when we are doing Scala development. SBT is built in Scala in order to help you build your Scala applications. However, SBT can still be used to build Java applications if you wish.

The core features of SBT are as follows:

- Scala-based build definition
- Incremental compilation
- Continuous compilation and testing

- Great support for testing libraries such as ScalaCheck, Specs, ScalaTest, and JUnit
- REPL integration
- Parallel Task execution

We will use SBT with the Typesafe Activator to Bootstrap our application later in this very chapter. Before doing so, we will play with SBT to learn the key concepts of setting up a build project for a Scala application. In this book, we will be using SBT version 0.13.11.

Installing SBT on Ubuntu Linux

Keep in mind that we need to have Java and Scala installed before installing SBT. If you don't have Java and Scala installed, go back to Chapter 1, *Introduction to FP, Reactive, and Scala* and follow the installation instructions. Open a terminal window, and run the following commands in order to download and install SBT:

```
$ cd /tmp
$ wget https://repo.typesafe.com/typesafe/ivy-
releases/org.scala-
sbt/sbt-launch/0.13.11/sbt-launch.jar?
_ga=1.44294116.1153786209.1462636319 -O sbt-
launch.jar
$ chmod +x sbt-launch.jar
$ mkdir ~/bin/ && mkdir ~/bin/sbt/
$ mv sbt-launch.jar ~/bin/sbt/
$ cd ~/bin/sbt/
$ touch sbt
```

Add the following content to the `~/bin/sbt/sbt` file:

```
#!/bin/bash
w
```

After saving the `~/bin/sbt/sbt` file, we need to give permission to execute the file with the following command:

```
$ chmod u+x ~/bin/sbt/sbt
```

Now we need to put SBT into the operational system path in order to be able to execute anywhere in the Linux terminal. We need to export SBT through the `PATH` command into the `~/.bashrc` file. Open the `~/.bashrc` file in your favorite editor, and add the following content:

```
export SBT_HOME=~/bin/sbt/
```

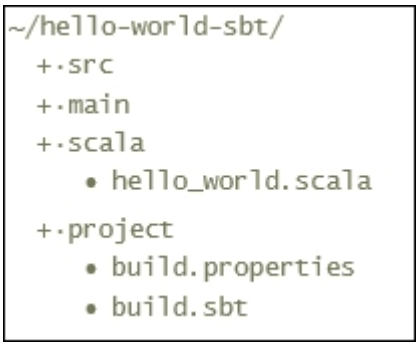
```
export PATH=$PATH:$SBT_HOME
```

We need to source the file using `$ source ~/.bashrc.`

Now we can run SBT and move on with the installation. When you now type `$ sbt` on your console, SBT will download all the dependencies required for use to run itself.

Getting started with SBT

Let's create folder named `hello-world-sbt`, and add the following project structure:



For `build.properties`, you need to have the following content:

build.properties

```
sbt.version=0.13.11
```

For `hello_world.scala`, we will use the following code:

hello_world.scala

```
object SbtScalaMainApp extends App {
  println("Hello world SBT / Scala App ")
}
```

For now we will use an SBT DSL. However, since SBT is written Scala, we can use the `build.scala` format if we wish. This is handy in some cases, because we can use any kind of Scala code in order to make the build more dynamic and to reuse code and tasks.

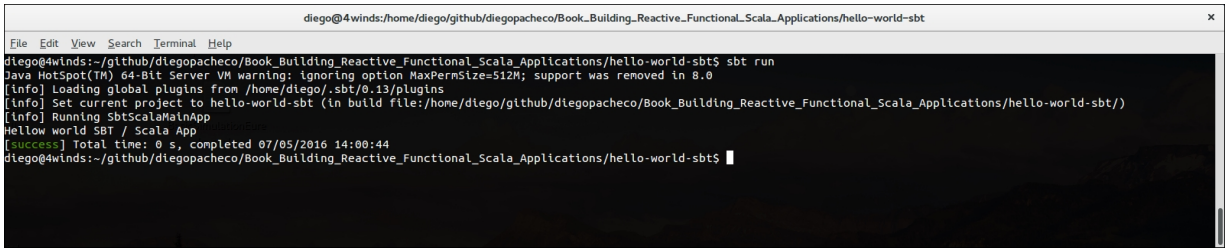
We will set some predefined variables, however, you can create your own variables, which can be used to avoid duplicate code. Finally, let's see the `build.sbt` file content as follows:

```
build.scala

    name := "hello-world-sbt"

    version := "1.0"

    scalaVersion := "2.11.8"
    scalaVersion in ThisBuild := "2.11.8"
```

In the preceding code, we have the name of the application, the version which will be used in the generated JAR file, and also the Scala version used on the application and the one used in the build process. We are ready to build this project, so open your terminal and type `$ sbt compile`.

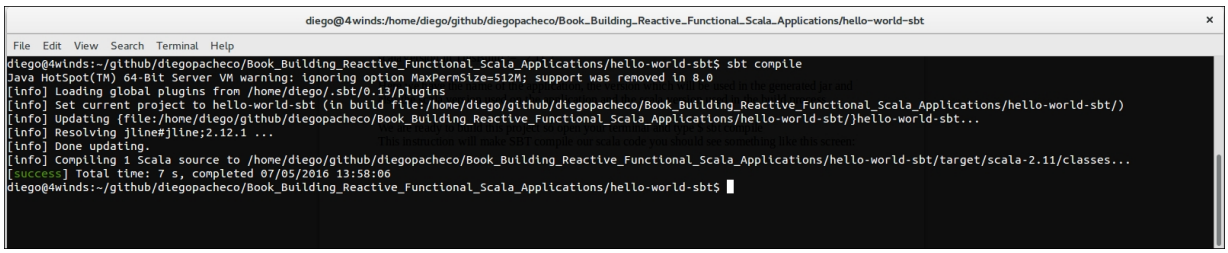
This instruction will make SBT compile our Scala code, and you should see something like this following screen:

```
$ sbt compile
```



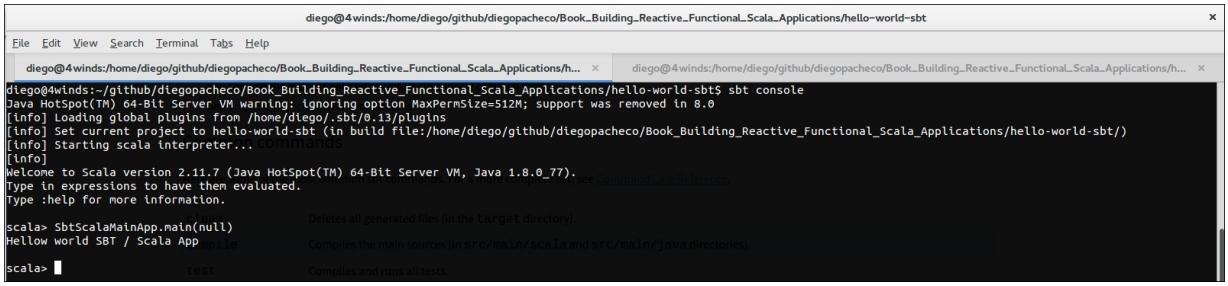
Congratulations! SBT just compiled our Scala application. Now we can run the application using SBT. In order to do this, we just need to type `$ sbt run` as follows:

```
$ sbt run
```



SBT makes it easier to test and play with your Scala application, because SBT has a REPL like the Scala REPL we were playing with in Chapter 1, *Introduction to FP, Reactive, and Scala*. The SBT REPL makes all the Scala code that you might have under the project available at the REPL.

Execute the command `$ sbt console.`



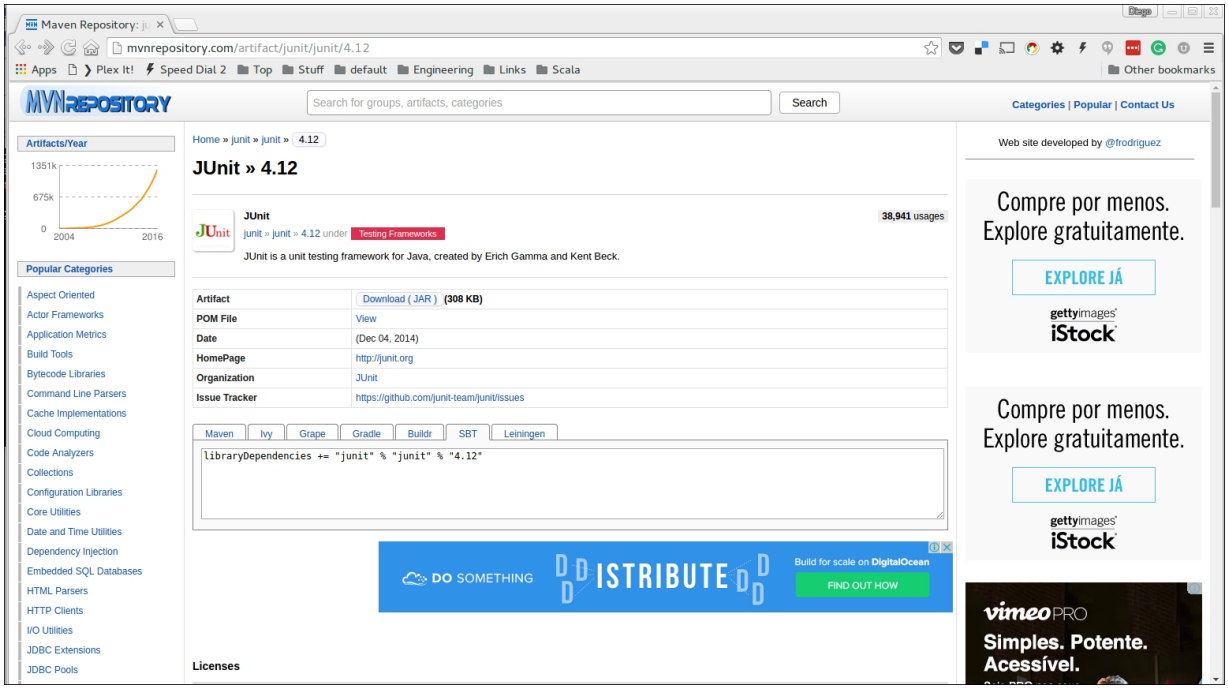
Once you are into the REPL, you can type any Scala code. As you must've realized, I just called the main Scala application directly via `$ SbtScalaMainApp.main(null)`.

Adding dependencies

Like any build tool, SBT allows you to resolve dependencies. SBT uses the Ivy / Maven2 patterns to resolve dependencies. So, if you are familiar with Maven2, Gradle, or Ant/Ivy, you will realize that setting SBT dependencies is the same, although with a different syntax. Dependencies are defined in the `build.sbt` file. There is no Scala development without unit tests. One of the most popular testing libraries is JUnit (http://junit.org/junit4/). JUnit works with Java and Scala projects. SBT will download and add JUnit to your Scala application `classpath` parameter. We need to edit the `build.sbt` file to add JUnit as a dependency as follows:

```
build.sbt
    name := "hello-world-sbt"

    version := "1.0"

    scalaVersion := "2.11.7"
    scalaVersion in ThisBuild := "2.11.7"

    libraryDependencies += "junit" % "junit" %
"4.12" % Test
    libraryDependencies += "com.novocode" %
"junit-interface" % "0.11"
    % "test"

    testOptions +=
Tests.Argument(TestFrameworks.JUnit, "-q", "-v")
```

As I mentioned before, SBT uses the same pattern as Maven2 / Ivy with: group ID + artifactid + version. If you don't know the pattern for the library you want to add, you can check out the Maven Repository website (they generate SBT configs as well) at the following link: http://mvnrepository.com/artifact/junit/junit/4.12.



SBT has scope for dependencies. We don't want to ship JUnit as part of the source code dependency. That's why we have the `% Test` after the dependency definition.

Once you have saved the file with the new content, you can run `$ sbt compile`. SBT will download JUnit for you and store the jar into the local Ivy repo files located at `/home/YOUR_USER/.ivy2/cache`. With the dependency in place, we can add more code and also use SBT to run our tests as follows:

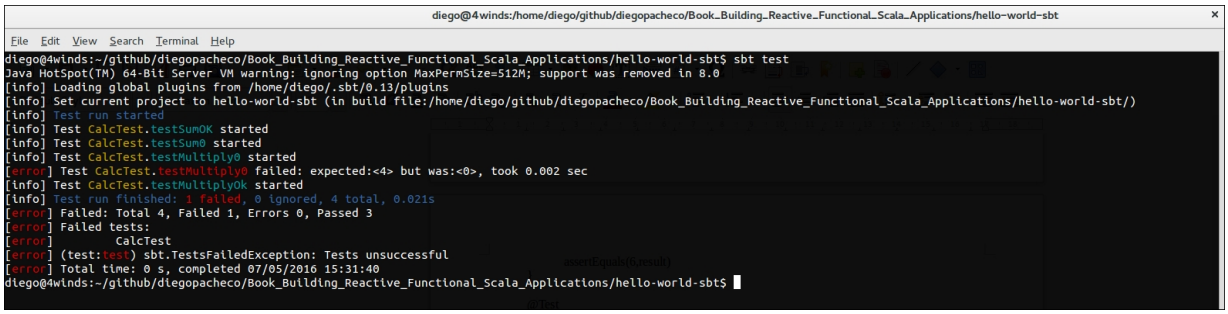
```
src/main/scala/calc.scala
    class Calculator{
      def sum(a:Int,b:Int):Int = {
        return a + b
      }
      def multiply(a:Int,b:Int):Int = {
        return a * b
      }
    }
```

In the preceding code, we just created a simple and straightforward calculator in Scala, which can add two integer numbers and also perform multiplication of two integers numbers. Now we can move on to the unit tests using JUnit. Tests need to be located in the `src/test/scala/` folder. Look at the following code:

`src/test/scala/calcTest.scala`

```
import org.junit.Test
import org.junit.Assert._

class CalcTest {
  @Test
  def testSumOK():Unit = {
    val c:Calculator = new Calculator()
    val result:Int = c.sum(1,5)
    assertNotNull(c)
    assertEquals(6,result)
  }

  @Test
  def testSum0():Unit = {
    val c:Calculator = new Calculator()
    val result:Int = c.sum(0,0)
    assertNotNull(c)
    assertEquals(0,result)
  }

  @Test
  def testMultiplyOk():Unit = {
    val c:Calculator = new Calculator()
    val result:Int = c.multiply(2,3)
    assertNotNull(c)
    assertEquals(6,result)
  }

  @Test
  def testMultiply0():Unit = {
    val c:Calculator = new Calculator()
    val result:Int = c.multiply(5,0)
    assertNotNull(c)
    assertEquals(4,result)
  }

}
```

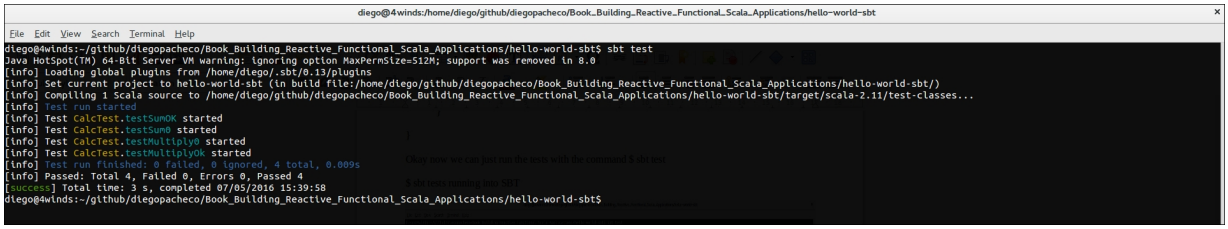
Okay, now we can just run the tests with the command `$ sbt test` as follows:



As you can see in the previous screenshot, all the tests are running. A test is created when we add the Java annotation `@Test`, and it needs to be a public function as well. There is one test, called `testMultiply0` , which fails, because it expects the result `4`, but `5` multiplied by `0` is zero, so the test is wrong. Let's fix this method by changing assertion to accept zero, like in the following code, and rerun the `$sbt test` as follows:

```
    @Test
        def testMultiply0():Unit = {
          val c:Calculator = new Calculator()
          val result:Int = c.multiply(5,0)
          assertNotNull(c)
          assertEquals(0,result)
        }
```

`$ sbt test` gives you the following result:



Hooray! All the tests passed. By default, SBT runs all your tests in parallel, which is great for speeding up build time - nobody likes to wait when doing builds, and Scala is not the fastest tech to build. However, you can disable parallel tests if you want by adding the following line into the `build.sbt`:

```
parallelExecution in Test := false
```

Generating Eclipse project files from SBT

SBT via plugins can generate Eclipse files. It's possible to add these plugins directly into your `build.sbt` file. However, there is a better solution. You can define global configurations, which are ideal, because you don't need to add in every simple `build.sbt` file you have. This also makes a lot of sense if you are working with multiple projects and/or you are working with open source projects because, as it is a matter of preference, people often do not versionate IDE files.

Go to the following directory if it exists, otherwise please create the following directory: `/home/YOUR_USER/.sbt/0.13/plugins`.

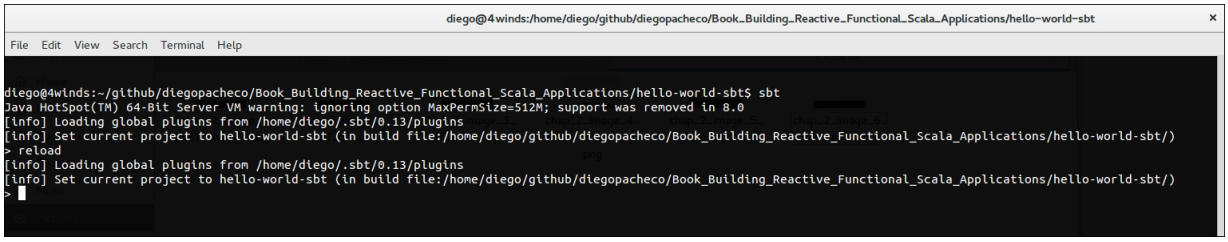
Now create the file `build.sbt` with the following content:

`/home/YOUR_USER/.sbt/0.13/plugins/build.sbt` Global config file

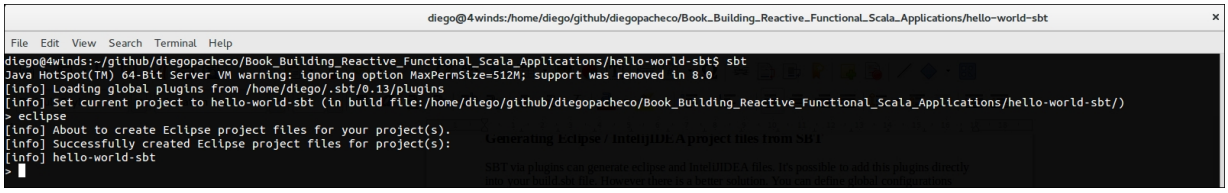
```
resolvers += Classpaths.typesafeResolver
    addSbtPlugin("com.typesafe.sbteclipse" %
"sbteclipse-plugin" %
    "4.0.0")
```

Once you save the file with this content, we can reload our SBT application by executing `$ sbt reload` , or quit the SBT console (***Ctrl*** + ***D***) and open sbt again using `$ sbt.`

```
$ sbt reload
```



Now we can generate Eclipse files by using the command `$ eclipse`.



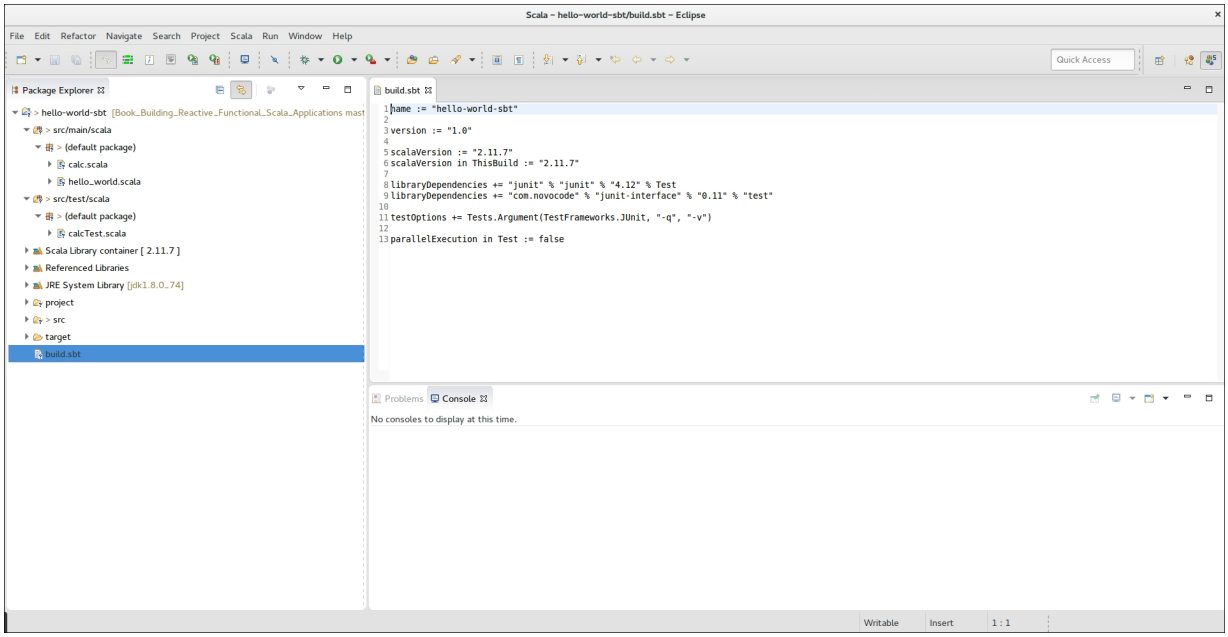
Once the generation is done, you can import the .project file generated into Eclipse.

By default, Eclipse does not attach source folders when generating the Eclipse project. If you want the source code (of third-party deps like Junit), you need to add an extra line into your `build.sbt` project. Adding source folders is often a good idea, otherwise, you can't do proper debugging without the source code.

build.sbt

```
EclipseKeys.withSource := true
```

The SBT Scala application imported into Eclipse is shown in the following screenshot:

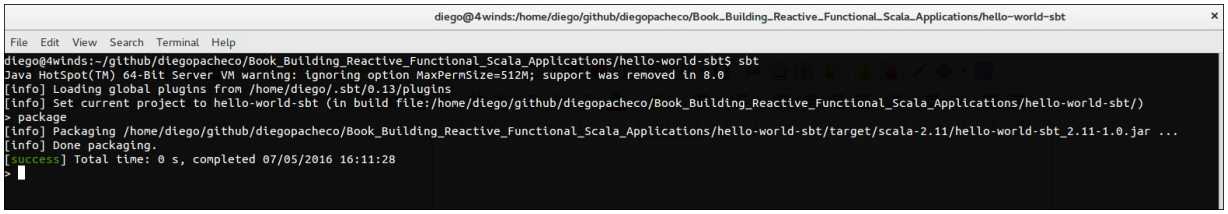

Scala – hello-world-sbt/build.sbt – Eclipse
File Edit Refactor Navigate Search Project Scala Run Window Help
Quick Access
Package Explorer
> hello-world-sbt [Book_Building_Reactive_Functional_Scala_Applications mast
> src/main/scala
> (default package)
calc.scala
hello_world.scala
> src/test/scala
> (default package)
calcTest.scala
Scala Library container [2.11.7]
Referenced Libraries
JRE System Library [jdk1.8.0_74]
project
> src
target
build.sbt
build.sbt
name := "hello-world-sbt"
version := "1.0"
scalaVersion := "2.11.7"
scalaVersion in ThisBuild := "2.11.7"
libraryDependencies += "junit" % "junit" % "4.12" % Test
libraryDependencies += "com.novocode" % "junit-interface" % "0.11" % "test"
testOptions += Tests.Argument(TestFrameworks.JUnit, "-q", "-v")
parallelExecution in Test := false
Problems
Console
No consoles to display at this time.
Writable
Insert
1 : 1

Application distribution

For this section, we will play with three different packaging solutions, which are as follows:

- The default SBT packagers
- SBT assembly plugin
- SBT native packager

SBT can generate jars by default. It is also possible to generate RPMs, DEBs, and even docker images via SBT plugins. First of all, let's generate an executable jar. This is done by the task package in SBT. Open your SBT console, and run the `$ sbt package`. However, we want to generate a `FAT` jar, which is a jar with all other dependencies (jars) of the application. In order to do that, we need to use another plugin called assembly.



The SBT package can generate a jar, but it does not ship the dependencies. In order to use the assembly plugin, create the file `project/assembly.sbt`, and add the content as follows:

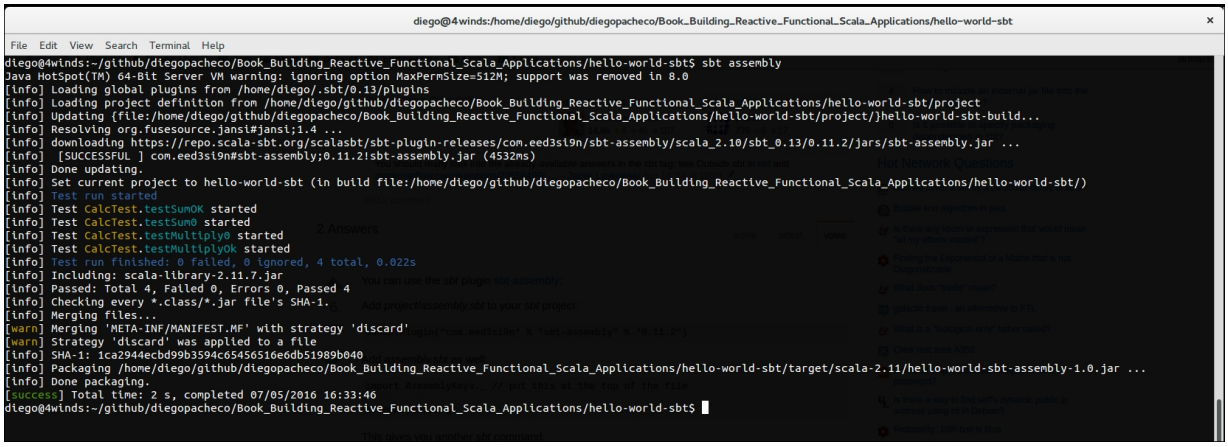
```
$ project/assembly.sbt
    addSbtPlugin("com.eed3si9n" % "sbt-assembly" %
"0.11.2")
```

In our `build.sbt`, we need to import the assembly plugin, like this:

`$ build.sbt` (put into the top of the file)

```
import AssemblyKeys._
    assemblySettings
```

Now we can run `$ sbt assembly` to generate our `FAT` jar.



There we go. Now we can run this as a normal Java application just using the command `java -jar` as follows:

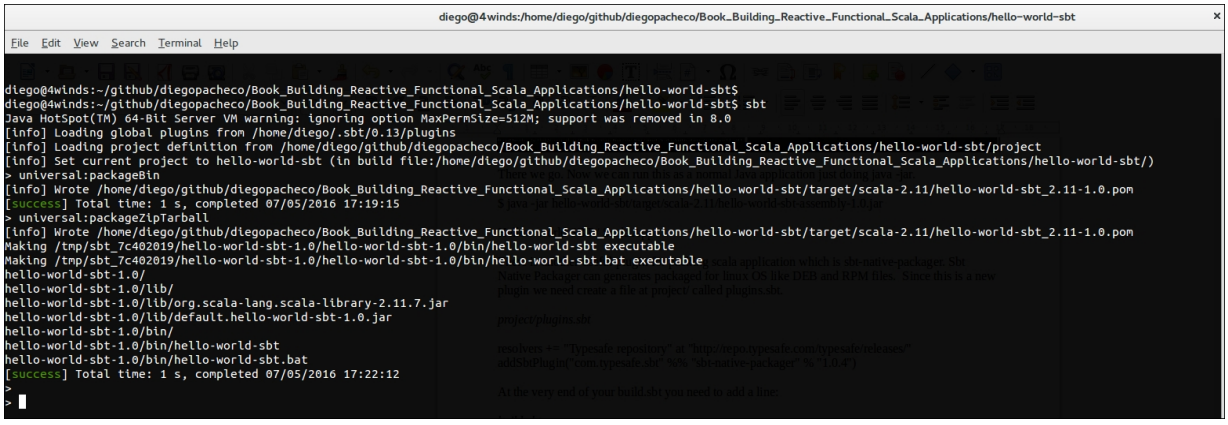
```
$ java -jar hello-world-sbt/target/scala-
2.11/hello-world-sbt-
    assembly-1.0.jar
```

Hello world SBT / Scala App

There is another useful plugin for packing the Scala application, which is `sbt-native-packager`. `sbt-native-packager` can generate packages for Linux OS like DEB and RPM files. Since this is a new plugin, we need to create a file called `plugins.sbt` in `project/` as follows: resolvers += "Typesafe repository" at "http://repo.typesafe.com/typesafe/releases/" addSbtPlugin("com.typesafe.sbt" %% "sbt-native-packager" % "1.0.4")

At the very end of your `build.sbt`, you need to add this line: enablePlugins(JavaAppPackaging)

Now we can generate packages with `sbt-native-packager` using `$ sbt universal:packageBin` or `$ sbt universal:packageZipTarball` .

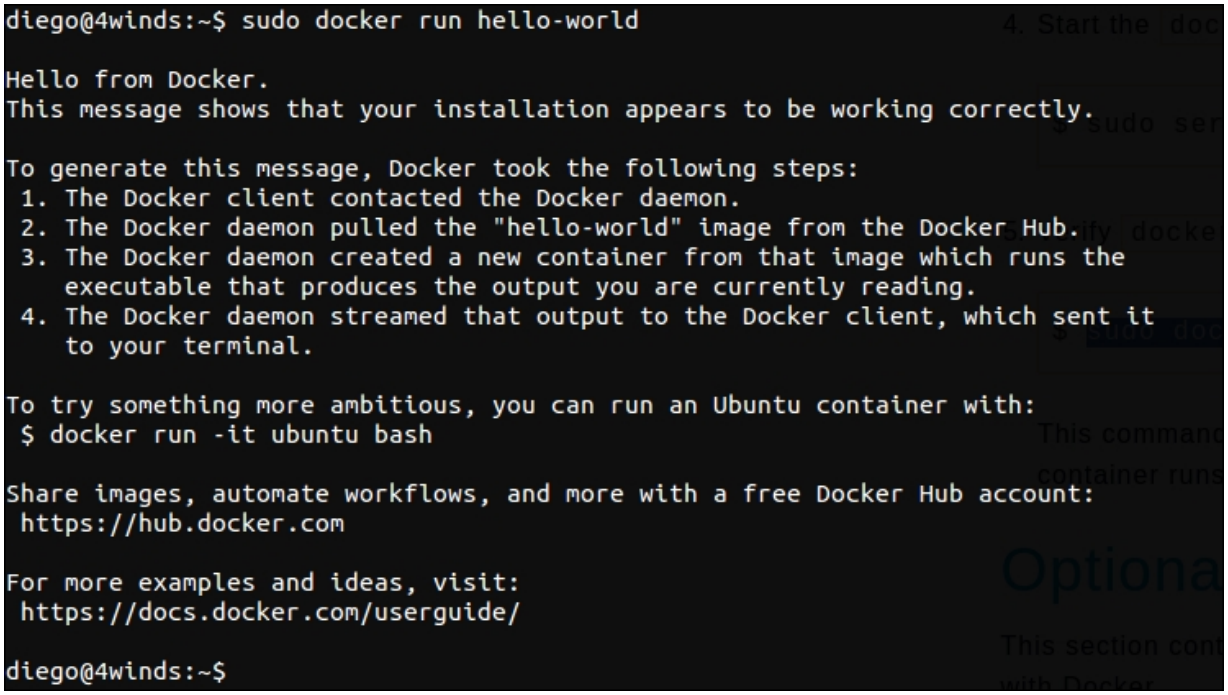


Now we have a ZIP and a TGZ file with your application in the folder `hello-world-sbt/target/universal/`. Inside this ZIP/TGZ file, we have our application in a jar format with all the dependencies; for now we just have Scala, but if we had more, they would be there as

well. There are SH and BAT scripts to run this application easily in Linux(SH) and Windows(BAT) respectively.

`sbt-native-packager` can also cook docker images. This is great, because that makes it easier to deploy applications into production environments. Our project is fully ready to bake docker images. We need to have docker installed on Linux; you can do so by running the following commands: **sudo apt-get update sudo apt-get install docker-engine sudo service docker start sudo docker run hello-world**

You should see something like the following screenshot if you've successfully installed Docker:



Now you can run `$ sbt`, and then generate your docker images by using the command `$ docker:publishLocal`. You will see an output similar to the following: **> docker:publishLocal [info] Wrote /home/diego/github/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/hello-world-sbt/target/scala-2.11/hello-world-sbt_2.11-1.0.pom [info] Sending build context**

to Docker daemon 5.769 MB [info] Step 1 : FROM java:latest [info] Pulling repository docker.io/library/java [info] 31ae46664586: Pulling image (latest) from docker.io/library/java [info] 31ae46664586: Pulling image (latest) from docker.io/library/java, endpoint: https://registry-1.docker.io/v1/ [info] 31ae46664586: Pulling dependent layers [info] e9fa146e2b2b: Pulling metadata [info] Status: Downloaded newer image for java:latest [info] docker.io/library/java: this image was pulled from a legacy registry. Important: This registry version will not be supported in future versions of docker. [info] ---> 31ae46664586 [info] Step 2 : WORKDIR /opt/docker [info] ---> Running in 74c3e354e9fd [info] ---> d67542bcaa1c [info] Removing intermediate container 74c3e354e9fd [info] Step 3 : ADD opt /opt [info] ---> f6cec2a2779f [info] Removing intermediate container 0180e167ae2d [info] Step 4 : RUN chown -R daemon:daemon . [info] ---> Running in 837ecff2ffcc [info] ---> 8a261bd9d88a [info] Removing intermediate container 837ecff2ffcc [info] Step 5 : USER daemon [info] ---> Running in 6101bd5b482b [info] ---> e03f5fa23bdf [info] Removing intermediate container 6101bd5b482b [info] Step 6 : ENTRYPOINT bin/hello-world-sbt [info] ---> Running in 43de9335129c [info] ---> eb3961f1e26b [info] Removing intermediate container 43de9335129c [info] Step 7 : CMD [info] ---> Running in 302e1fcd0a3d [info] ---> 04e7872e85fa [info] Removing intermediate container 302e1fcd0a3d [info] Successfully built 04e7872e85fa [info] Built image hello-world-sbt:1.0 [success] Total time: 447 s, completed 07/05/2016 17:41:47 >

You can confirm that there is a new docker image in your system just by running the command `$ docker ps:`

```
diego@4winds:~/github/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/hello-world-sbt$ docker images
REPOSITORY                TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
hello-world-sbt           1.0                 04e7872e85fa        53 seconds ago      654.4 MB
java                      latest              31ae46664586        2 days ago          642.9 MB
grafana/grafana           latest              4db697be86ae        4 weeks ago         213.8 MB
gliderlabs/alpine         latest              5b6d72625828        5 weeks ago         4.798 MB
tobegit3hub/seagull       latest              50efc40e5997        5 weeks ago         774.3 MB
hopsoft/graphite-statsd   latest              f4a323020cb8        5 weeks ago         806.6 MB
graylog2/allinone         latest              6e684c85c05e        7 weeks ago         870.3 MB
nginx                     latest              6f62f48c4e55        8 weeks ago         190.5 MB
redis                     latest              8d81cd6f6c5e        9 weeks ago         177.6 MB
```

The very first image is our docker image generated by the `sbt-native-packager` plugin with our Scala application. Congratulations! You have a docker container running with your Scala application. SBT Native Packager is really powerful, yet simple to use. You can get more details on the official documentation site (http://www.scala-sbt.org/sbt-native-packager/gettingstarted.html).

These are the basic things we need to know about SBT to build professional Scala applications. SBT has many other features and possibilities, which you can check it out at http://www.scala-sbt.org/0.13/docs/index.html. Next, we will learn about Typesafe Activator, which is a wrapper around SBT that makes it easy to use with Play framework applications.

Bootstrapping our Play framework app with Activator

Lightband (former Typesafe) has another tool called Activator (https://www.lightbend.com/community/core-tools/activator-and-sbt), which is a wrapper on top of SBT. Activator makes it easier to create Reactive applications using Scala, Akka, and the Play framework. Don't worry about the Play framework right now, because we will cover that in greater detail in Chapter 3, *Developing the UI with Play Framework*. Akka will be covered in detail in Chapter 8, *Developing a chat with Akka*.

Let's download and install Activator, and Bootstrap our architecture. Remember, we need to have Java 8 and Scala 2.11 already installed. If you don't have Java 8 or Scala 2.11, go back to Chapter 1, *Introduction to FP, Reactive, and Scala* and install them.

First of all, you need to download activator from here: https://www.lightbend.com/activator/download

I recommend that you download the minimal package, and let Activator download and install the rest of the other dependencies for you. You can download the minimal package here: https://downloads.typesafe.com/typesafe-activator/1.3.10/typesafe-activator-1.3.10-minimal.zip.

For this book, we will be using version 1.3.10. We need to put the `activator/bin` folder in the OS PATH.If you want, you can install Activator using the terminal, like this: If you want, you can install Activator using the terminal, like this: **$ cd /usr/local/ $ wget https://downloads.typesafe.com/typesafe-activator/1.3.10/typesafe-activator-1.3.10-minimal.zip $ tar -xzf typesafe-activator-1.3.10-minimal.zip $ rm -rf typesafe-activator-1.3.10-minimal.zip $ sudo echo 'export**

PATH=$PATH:/usr/local/typesafe-activator- 1.3.10-minimal/bin' >> ~/.bashrc $ source >> ~/.bashrc

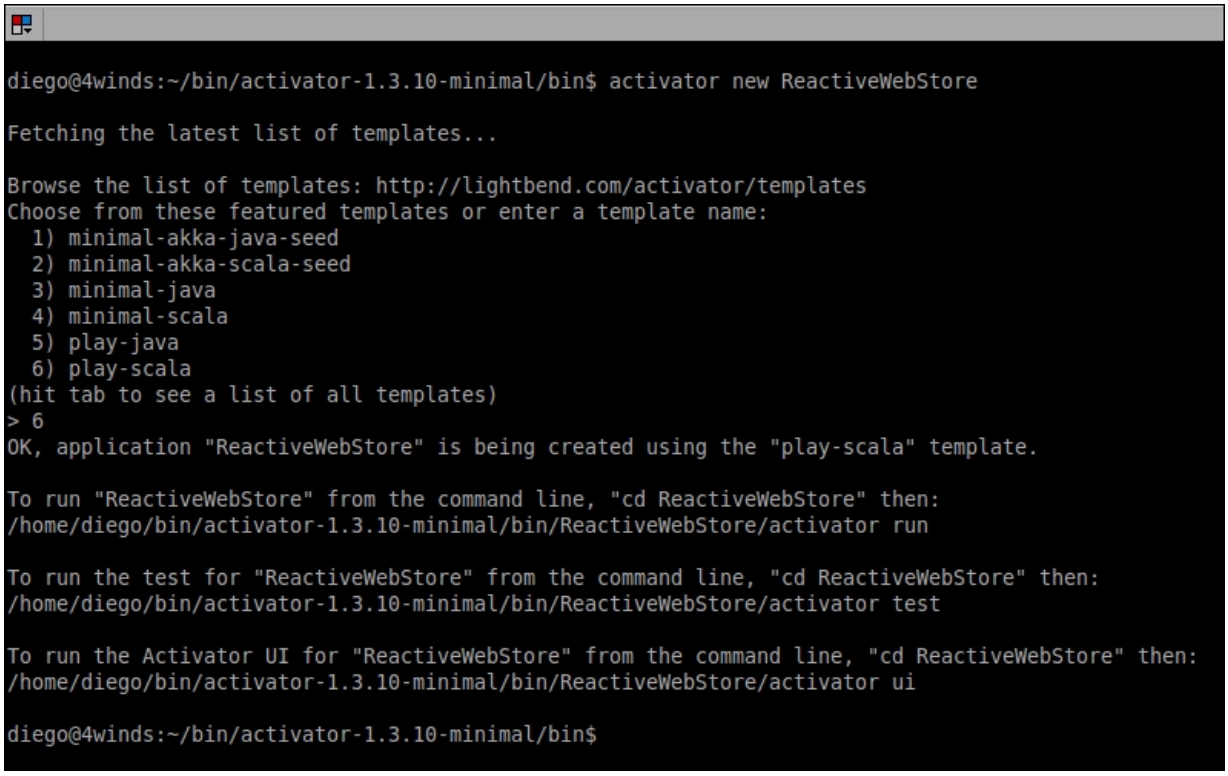
In order to test your installation, execute this command: **$ activator new ReactiveWebStore**

The preceding command will Bootstrap an architecture for you with Scala, Akka, Play framework, and SBT.

Activator will ask you a series of questions like such as what templates you might like to use. There are a couple of templates for Java applications, Scala applications, Akka applications, and Play applications. For now, we will pick option `6) play-scala`.

The first time you run Activator, it could take some time, because it will download all the dependencies from the web. When Activator finishes, you should see a folder called `ReactiveWebStore` in your file system.

The command `$ activator new ReactiveWebStore` shows the following result:



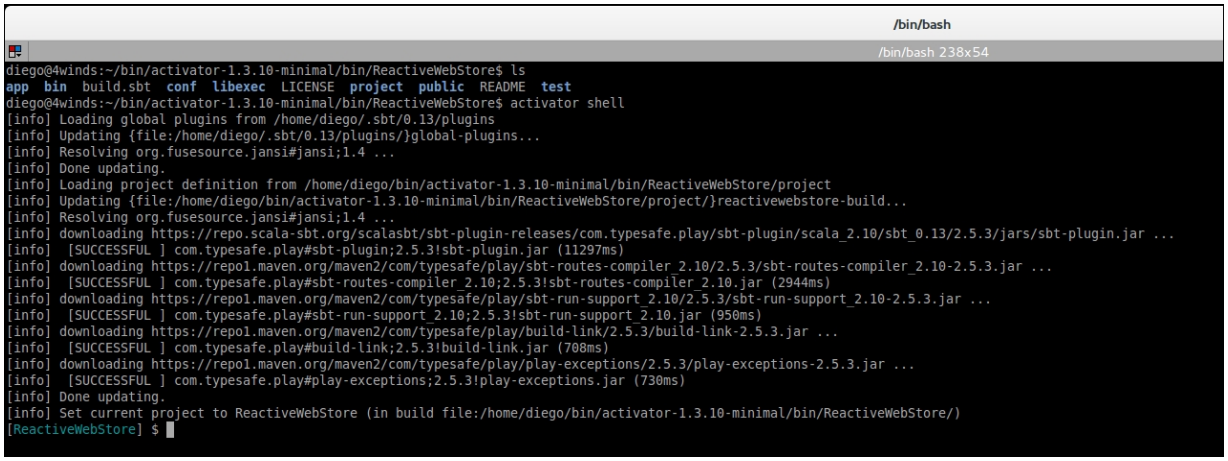
You should enter the `ReactiveWebStore` folder if you type `$ ll` into the console, and you should also see the following structure:
diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$ ll total 52 drwxrwxr-x 9 diego diego 4096 Mai 14 19:03 ./ drwxr-xr-x 3 diego diego 4096 Mai 14 19:03 ../ drwxrwxr-x 6 diego diego 4096 Mai 14 19:03 app/ drwxrwxr-x 2 diego diego 4096 Mai 14 19:03 bin/ -rw-rw-r-- 1 diego diego 346 Mai 14 19:03 build.sbt drwxrwxr-x 2 diego diego 4096 Mai 14 19:03 conf/ -rw-rw-r-- 1 diego diego 80 Mai 14 19:03 .gitignore drwxrwxr-x 2 diego diego 4096 Mai 14 19:03 libexec/ -rw-rw-r-- 1 diego diego 591 Mai 14 19:03 LICENSE drwxrwxr-x 2 diego diego 4096 Mai 14 19:03 project/ drwxrwxr-x 5 diego diego 4096 Mai 14 19:03 public/ -rw-rw-r-- 1 diego diego 1063 Mai 14 19:03 README drwxrwxr-x 2 diego diego 4096 Mai 14 19:03 test/

The content is explained as follows:

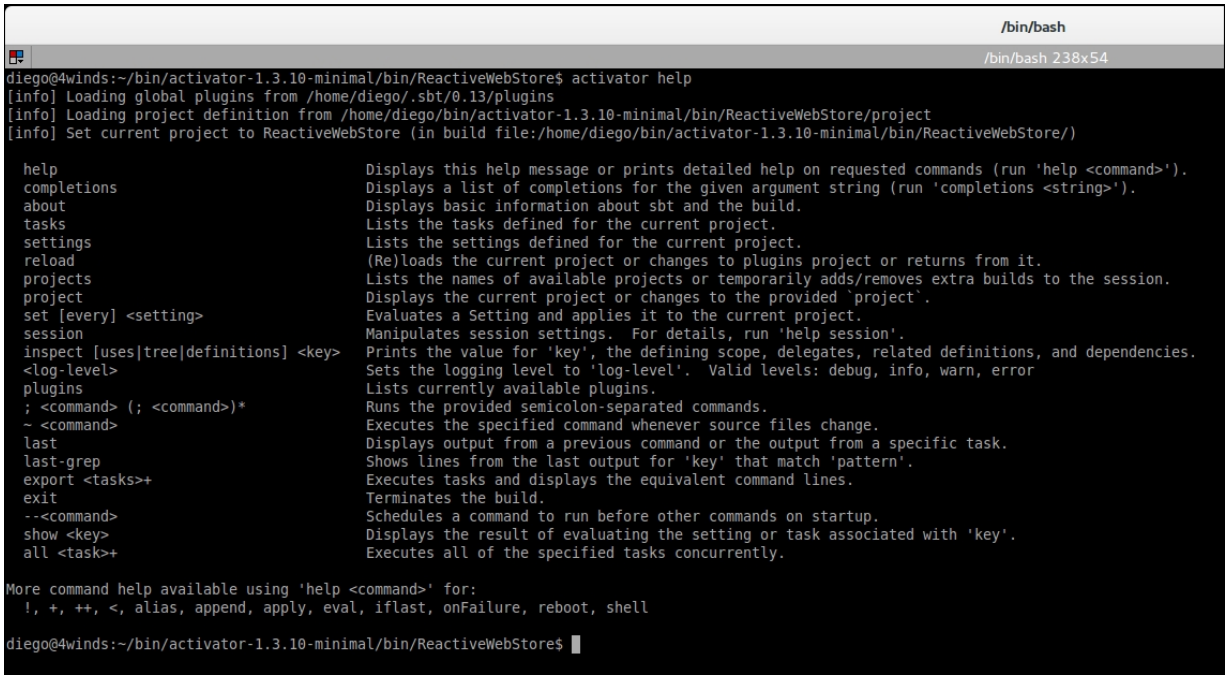
- **app**: This is the Play framework application folder where we will do the Scala web development.
- **build.sbt**: This is the build file; as you can see, Activator has generated the SBT build config for us.
- **conf**: This holds the application config files such as logging and Scala/Play app config.
- **project**: This is the SBT project folder where we define SBT plugins and SBT version.
- **test**: This holds the test source code for our application.
- **public**: This holds static HTML assets like Images, CSS, and JavaScript code.
- **bin**: This holds a copy of the activator script for Linux/Mac and Windows.
- **libexec**: This holds the Activator jar. This is pretty useful, because Activator has packed itself with our application. So, let's say you push this application for GitHub - when someone needs to access this app and download it from GitHub, the SBT file will be there, so they won't need to download it from the Internet. This is especially useful when you are provisioning and deploying applications in production, which this book will cover in detail in Chapter 10, *Scaling Up*.

Activator shell

Activator allows you to run REPL like we did in Scala and SBT. In order to get REPL access, you need to type the following on the console: $ activator shell



Activator has plenty of tasks that you can use. In order to know all the available commands, you can type `$ activator help` on the console.



Activator - compiling, testing, and running

Let's get to business now. We will compile, run tests, and run our web application using Activator and SBT. First of all, let's compile. Type `$ activator compile` on the console as follows: **$ activator compile diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$ activator compile [info] Loading global plugins from /home/diego/.sbt/0.13/plugins [info] Loading project definition from /home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/project [info] Set current project to ReactiveWebStore (in build file:/home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/) [info] Updating {file:/home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/}root... [info] Compiling 14 Scala sources and 1 Java source to /home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/target/scala-2.11/classes... [success] Total time: 154 s, completed 14/05/2016 19:28:03 diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$**

Let's run our tests now with the command `$ activator test`.

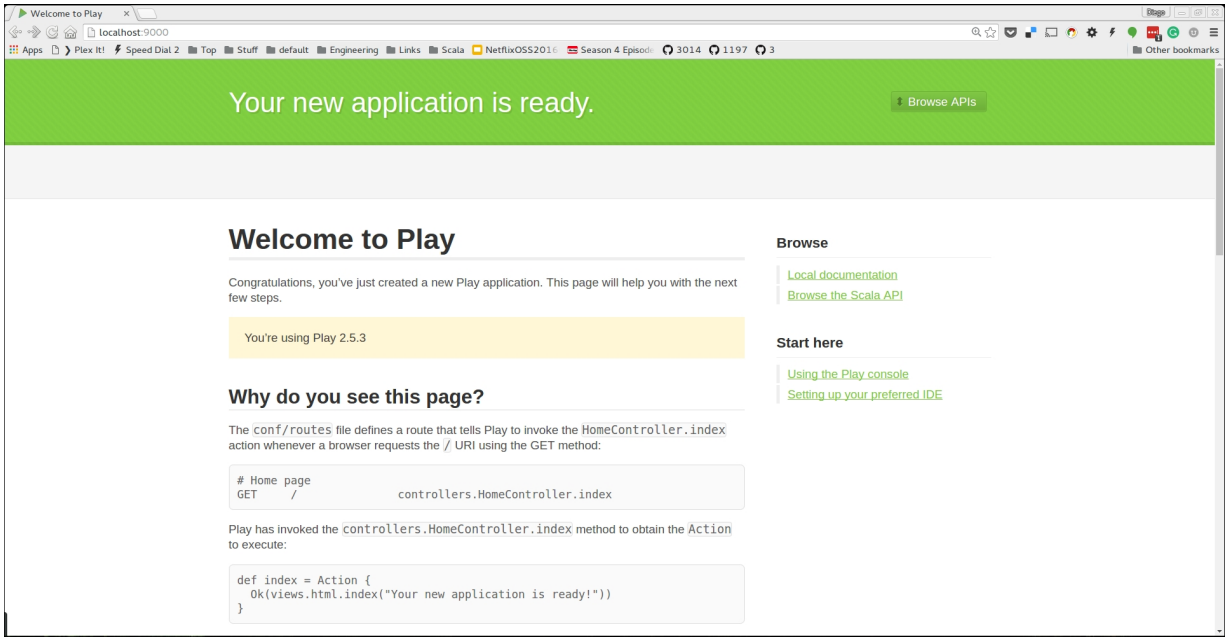
```
/bin/bash
/bin/bash 238x54
diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$ $ activator test
$: command not found
diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$ activator test
[info] Loading global plugins from /home/diego/.sbt/0.13/plugins
[info] Loading project definition from /home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/project
[info] Set current project to ReactiveWebStore (in build file:/home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/)
[info] Compiling 2 Scala sources to /home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/target/scala-2.11/test-classes...
[info] IntegrationSpec:
[info] Application
[info] application - ApplicationTimer demo: Starting application at 2016-05-14T22:30:11.464Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-05-14T22:30:13.643Z after 2s.
[info] - should work from within a browser
[info] ApplicationSpec:
[info] Routes
[info] application - ApplicationTimer demo: Starting application at 2016-05-14T22:30:13.786Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-05-14T22:30:13.851Z after 0s.
[info] - should send 404 on a bad request
[info] HomeController
[info] application - ApplicationTimer demo: Starting application at 2016-05-14T22:30:13.959Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-05-14T22:30:14.015Z after 1s.
[info] - should render the index page
[info] CountController
[info] application - ApplicationTimer demo: Starting application at 2016-05-14T22:30:14.118Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-05-14T22:30:14.200Z after 0s.
[info] - should return an increasing count
[info] ScalaTest
[info] Run completed in 7 seconds, 385 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 2, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 4, Failed 0, Errors 0, Passed 4
[success] Total time: 17 s, completed 14/05/2016 19:30:14
diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$
```

Finally, it's time to run your application. Type `$ activator run` on the console.

```
/bin/bash
/bin/bash 238x54
diego@4winds:~/bin/activator-1.3.10-minimal/bin/ReactiveWebStore$ activator run
[info] Loading global plugins from /home/diego/.sbt/0.13/plugins
[info] Loading project definition from /home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/project
[info] Set current project to ReactiveWebStore (in build file:/home/diego/bin/activator-1.3.10-minimal/bin/ReactiveWebStore/)

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

[info] p.c.s.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

Open your web browser, and go to the URL: `http://localhost:9000`.

You should see a screen like the following:


Welcome to Play
localhost:9000
Apps
Plex It!
Speed Dial 2
Top
Stuff
default
Engineering
Links
Scala
NetflixOSS2016
Season 4 Episode
3014
1197
3
Other bookmarks
Your new application is ready.
Browse APIs
Welcome to Play
Congratulations, you've just created a new Play application. This page will help you with the next few steps.
You're using Play 2.5.3
Why do you see this page?
The conf/routes file defines a route that tells Play to invoke the HomeController.index action whenever a browser requests the / URI using the GET method:
Home page
GET / controllers.HomeController.index
Play has invoked the controllers.HomeController.index method to obtain the Action to execute:
def index = Action {
Ok(views.html.index("Your new application is ready!"))
}
Browse
Local documentation
Browse the Scala API
Start here
Using the Play console
Setting up your preferred IDE

Summary

Congratulations! You've just Bootstrapped your first Scala / Play framework first. Activator makes our life easier. As you can see, with three commands, we were able to get a site up and running. You could do the same with just SBT, however, it would take more time, because we would need to get all the dependencies, configure all the source code structure, and add some sample HTML and Scala code. Thanks to Activator, we don't need to do any of that. However, we can still change all the SBT files and configs as per our wish. Activator is not tight with Scala or our application code, since it is more like a template-based code generator.

In the next chapter, we will be improving the application by adding validations, database persistence, Reactive Microservices calling using RxScala and Scala, and much more.

Chapter 3. Developing the UI with Play Framework

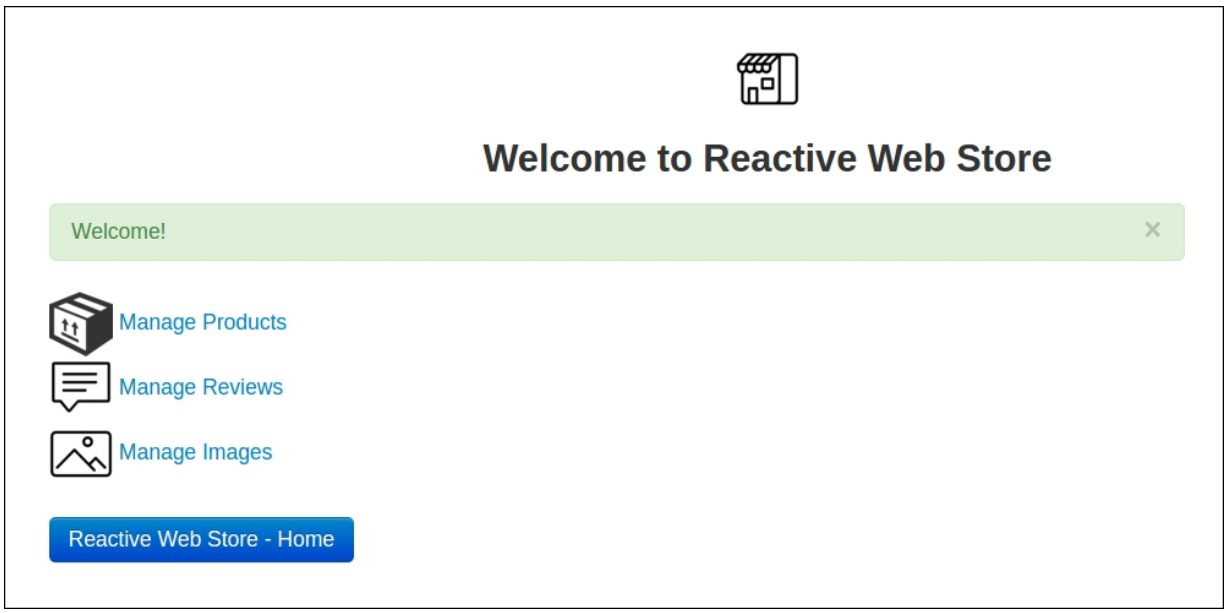
In the previous chapter, we performed bootstrapping on our application using Activator. In this chapter, we will continue developing our web application using Scala and Play framework. Play framework is great for web development because it is simple to use, and at the same time, very powerful. This is because it uses top-notch reactive solutions like spray, Akka, and Akka Stream under the hood. For this chapter, we will create the basic UI for some parts of our reactive web solution by adding validation and an in-memory store so you can feel the application working. We will use a little bit of CSS for styling, and JavaScript for some simple visualizations.

In this chapter, we will cover the following topics:

- Basics of web development with Scala and Play frameworks
- Creating your models
- Working with views and validations
- Working with session scopes

Getting started

Let's have a look at the preview of `Reactive Web Store`--the application that we will build.



For now, we will build three simple operations--**Create, Retrieve, Update,** and **Delete** (**CRUD**) in order to manage products, product reviews, and product images. We will create models, controllers, views, and routes for each CRUD.

Let's get started. First of all, we need to define our models. The models need to be located at `ReactiveWebStore/app/models`. Models are the CORE of the system and they represent the entity. We will use this entity later to store and retrieve data from a database later on in Chapter 6, *Persistence with Slick*. Our models should not have any UI logic, since we should use controllers for UI logic.

Creating our models

For our product model, we have a simple Scala case class in `Product.scala` as follows:

```
package models
    case class Product
    ( var id:Option[Long],
      var name:String,
      var details:String,
      var price:BigDecimal )
    {
      override def toString:String =
      {
        "Product { id: " + id.getOrElse(0) +
",name: " + name + ",
        details: "+ details + ", price: " + price
+ "}"
      }
    }
```

A product can have an optional ID, a name, details, and a price. We also override the `toString` method just for the sake of simplicity for logging. We also need to define models for image and review.

The following is the review model from `Review.scala`:

```
package models

    case class Review
    (var id:Option[Long],
      var productId:Option[Long],
      var author:String,
      var comment:String)
    {
      override def toString:String = {
        "Review { id: " + id + " ,productId: " +
        productId.getOrElse(0) + ",author: " +
author + ",comment:
        " + comment + " }"
```

```
    }
  }
```

For a review model, we have an optional ID, an optional `productId`, one author, and a comment. Validations will be done on the views. Now let's go for the image model.

The image model can be found in `Image.scala` as follows:

```
package models

    case class Image
    (var id:Option[Long],
      var productId:Option[Long],
      var url:String){
        override def toString:String = {
          "Image { productId: " +
productId.getOrElse(0) + ",url: "
          + url + "}"
        }
  }
```

For an image model, we have an optional ID, an optional `productId`, and the image URL.

The Play framework does the routing, and we need to define the routes at `ReactiveWebStore/conf/routes`. Keep in mind that the Play framework will validate all the routes, so you need to specify valid packages and classes. Play also creates something called reverse controller, which we will use later in the chapter. For now, let's define the routes. Reverse controller is generated by the Play framework with an action method which is the same as that of the original controller with the same signature, but it returns `play.api.mvc.Call` instead of `play.api.mvc.Action`.

Creating routes

The Play framework CRUD operations' routes for product, image, and review are as follows: # Routes

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

GET / controllers.HomeController.index

GET /assets/*file controllers.Assets.at(path="/public", file)

#

Complete CRUD for Product

#

GET /product controllers.ProductController.index GET /product/add controllers.ProductController.blank POST /product/ controllers.ProductController.insert POST /product/:id controllers.ProductController.update(id:Long) POST /product:id/remove controllers.ProductController.remove(id:Long) GET /product/details/:id controllers.ProductController.details(id:Long)

#

Complete GRUD for Review

#

GET /review controllers.ReviewController.index

GET /review/add controllers.ReviewController.blank POST /review/ controllers.ReviewController.insert POST /review/:id controllers.ReviewController.update(id:Long) POST

/review:id/remove controllers.ReviewController.remove(id:Long) GET /review/details/:id controllers.ReviewController.details(id:Long)

#

Complete CRUD for Image

#

GET /image controllers.ImageController.index

GET /image/add controllers.ImageController.blank POST /image/ controllers.ImageController.insert POST /image/:id controllers.ImageController.update(id:Long) POST /image:id/remove controllers.ImageController.remove(id:Long) GET /image/details/:id controllers.ImageController.details(id:Long)

The routes work like this--First you need to define the rest verb such as `GET`, `POST`, `PUT`, `DELETE`, and then you put in a `PATH` like `/image`. Finally, you specify which controller function will handle that route. Now we have the routes in place, we can move to the controllers. We will define the controllers for product, image and review.

All the routes follow the same logic. First we send the user to the web page where we list all the items (products, images, reviews)--this is represented by `GET /resource`, where the resource can be an image, product, or review, for instance. In order to get a specific resource, often by ID, we give the command `GET /resource (product, review or image)/ID. POST /resource` is used to perform the `UPDATE`.

In order to create a new item (product, review, or image), the pattern is `GET /resource/add` and `POST /resource/`. You may wonder why there are two routes to perform an insert. Well, that's because first of all we need to load the web page, and secondly, when the form is submitted, we need a new route to handle the values. There are two routes for an update as well for the same reason. If you want to `DELETE` a resource, the pattern is `POST /resource/ID/remove`. Lastly,

we have the details of the operation, which is used to show detailed information with regard to a specific item--the pattern is `GET /resource/details/ID`. With six routes, we can do a complete CRUD for a resource such as product, image, and review, or any other future resource that you may add to this application or your own applications.

Creating our controllers

Now let's move to the controllers used on the previous routes. The controllers need to be located at `ReactiveWebStore/app/controllers`. Controllers are bound between views (UI), models and service, which are responsible for business operations. It's always important to separate UI logic, which tends to be specific, from business logic, which tends to be more generic, and often, way more important.

Let's have a look at the product controller in `ProductController.scala` in the following code: @Singleton

class ProductController @Inject() (val messagesApi:MessagesApi,val service:IProductService) extends Controller with I18nSupport {

val productForm: Form[Product] = Form(

mapping(

"id" -> optional(longNumber), "name" -> nonEmptyText,

"details" -> text,

"price" -> bigDecimal

)(models.Product.apply)(models.Product.unapply))

def index = Action { implicit request => val products = service.findAll().getOrElse(Seq()) Logger.info("index called. Products: " + products) Ok(views.html.product_index(products)) }

```
def blank = Action { implicit request => Logger.info("blank called. ")
Ok(views.html.product_details(None, productForm)) }


def details(id: Long) = Action { implicit request => Logger.info("details
called. id: " + id) val product = service.findById(id).get
Ok(views.html.product_details(Some(id), productForm.fill(product)))

}


def insert()= Action { implicit request => Logger.info("insert called.")
productForm.bindFromRequest.fold(

form => {

BadRequest(views.html.product_details(None, form)) },

product => {

val id = service.insert(product)
Redirect(routes.ProductController.index).flashing("success"

-> Messages("success.insert", id)) })

}


def update(id: Long) = Action { implicit request =>
Logger.info("updated called. id: " + id)
productForm.bindFromRequest.fold(

form => {

Ok(views.html.product_details(Some(id), form)).flashing("error" ->
"Fix the errors!") },
```

```
product => {

service.update(id,product) Redirect(routes.ProductController.index).

flashing("success" -> Messages("success.update", product.name))

})

}



def remove(id: Long)= Action {

service.findById(id).map { product => service.remove(id)

Redirect(routes.ProductController.index).flashing("success" ->
Messages("success.delete", product.name)) }.getOrElse(NotFound)



}

}
```

The Play framework uses dependency injection and inversion of control using Google Guice. So, you can see at the top of the controller that we have the annotations `@Singleton` and `@Inject`. Singleton means that Guice will create a single instance of the class to handle all requests. Inject means we are injecting other dependencies into our controller, for instance, we inject `MessagesApi` in order to have the Play framework internalization support for string messages, and `IProductService`, that is, the product service that we will cover later in this chapter.

We also need to extend the Play class, `play.api.mvc.Controller`. Each function in a controller needs to return an action. This action

could be a view.The Play framework compiles all the views into Scala classes, so you can safely reference them into your controllers code.

All business operations are delegated to a trait called `IProductService`, which we will cover later in this chapter. We also log some information using the Logger class. Play Framework uses `Logback` as the default logging solution. Let's take a closer look at each controller function now.

The index function calls `IProductService`, and finds all the available products. If there are no products available, it returns an empty sequence, and then calls the product UI passing the collection of products.

The blank function renders a blank product form, so the user can have a blank product form on the UI in order to add data (insert operation). Play framework works with form binding. So, in each controller, you need to define how your form looks on the UI. That form mapping is done using `play.api.data.Form`. You can see the mapping on the immutable variable called `productForm`. The mapping is between the view(UI) and the model called product. Keep in mind that the name field is mapped as `NonEmptyText`, which means Play won't accept null or blank values. This is a great future, because we can do validations in a declarative way without having to write code. Price is defined as `BigDecimal`, so Play won't accept text, but only numbers.

The details function retrieves a product using `IProductService`, and redirects to the view. However, before doing the redirect, it binds the data with the form so the UI will load with all the data into the HTML inputs.

We also have the insert and update methods. They are all constructed with a `fold` method. The `fold` method has left and right, which means error or ok. The `fold` function is called from the mapped form and if there are no validation errors, it goes right, but if there are validations errors, it goes left. That's a very simple and

clean way to code the `update` and `insert` flows. With `fold`, we don't need code for an `if` statement. Once the validation is OK, we call `IProductService` to do an insert or update, and then we perform a redirect to the view. Messages are passed via scope. Play has options for scope--session or Flash. Session is for multiple requests, and the value will be stored in the client side. Flash is a request scope, and most of the times that is what you need to use. Here we are using the `Flash` scope, so it will only exit during that specific request. This feature is used to pass **Internationalization messages** (**i18n**), which are the result of the action. All the i18n messages need to be defined at `ReactiveWebStore/conf/messages` as follows: success.delete = OK '{0}' deleted!

success.insert = OK '{0}' created!

success.update = OK '{0}' updated!

error.notFound = Nothing Found with ID {0,number,0}

error.number = Not a valid number error.required = Missing value here

Lastly, we have the remove method. First of all, we need to make sure the product exists, so we do a `findById` using `IProductService`, and then we apply a map function. If the product doesn't exist, the Play framework has prebuilt HTTP error code messages like `NotFound`. If the product exists, we remove it using `IProductService`, and then we redirect to the UI with a flashing message. Now let's see the image and review controllers.

The review controller, `ReviewController.scala`, is as follows: @Singleton

class ReviewController @Inject()

(val messagesApi:MessagesApi,

val productService:IProductService, val service:IReviewService)

```
extends Controller with I18nSupport {

val reviewForm:Form[Review] = Form(

mapping(

"id" -> optional(longNumber), "productId" -> optional(longNumber),
"author" -> nonEmptyText, "comment" -> nonEmptyText )
(models.Review.apply)(models.Review.unapply))

def index = Action { implicit request => val reviews =
service.findAll().getOrElse(Seq()) Logger.info("index called. Reviews:
" + reviews) Ok(views.html.review_index(reviews)) }


def blank = Action { implicit request => Logger.info("blank called. ")
Ok(views.html.review_details(None,
reviewForm,productService.findAllProducts)) }


def details(id: Long) = Action { implicit request => Logger.info("details
called. id: " + id) val review = service.findById(id).get
Ok(views.html.review_details(Some(id),
reviewForm.fill(review),productService.findAllProducts)) }


def insert()= Action { implicit request => Logger.info("insert called.")
reviewForm.bindFromRequest.fold(

form => {

BadRequest(views.html.review_details(None,
form,productService.findAllProducts)) },

review => {
```

```
if (review.productId==null ||

review.productId.getOrElse(0)==0) {

Redirect(routes.ReviewController.blank).flashing("error" -> "Product
ID Cannot be Null!") }else {

Logger.info("Review: " + review) if (review.productId==null ||

review.productId.getOrElse(0)==0) throw new
IllegalArgumentException("Product Id Cannot Be Null") val id =
service.insert(review)
Redirect(routes.ReviewController.index).flashing("success" -

> Messages("success.insert", id)) }

})

}



def update(id: Long) = Action { implicit request =>
Logger.info("updated called. id: " + id)
reviewForm.bindFromRequest.fold(

form => {

Ok(views.html.review_details(Some(id),
form,productService.findAllProducts)).flashing("error" -> "Fix the
errors!")

},

review => {

service.update(id,review)

Redirect(routes.ReviewController.index).flashing("success" -
```

```
>Messages("success.update", review.productId)) })

}


def remove(id: Long)= Action {

service.findById(id).map { review => service.remove(id)

Redirect(routes.ReviewController.index).flashing("success" -

>Messages("success.delete", review.productId))
}.getOrElse(NotFound)

}



}
```

The review controller follows the same ideas and structure as the product controller. The only main difference is that here we need to Inject `IProductService`, because a review needs to belong to a product. Then we need to use `IProductService` in order to `findAllProduct`, because in the review view, we will have `SelectBox` with all the available products.

The image controller, `ImageController.scala`, is as follows:

```
@Singleton

class ImageController @Inject()

(val messagesApi:MessagesApi,

val productService:IProductService, val service:IImageService)

extends Controller with I18nSupport {
```

```
val imageForm:Form[Image] = Form(

mapping(

"id" -> optional(longNumber), "productId" -> optional(longNumber),
"url" -> text

)(models.Image.apply)(models.Image.unapply))

def index = Action { implicit request => val images =
service.findAll().getOrElse(Seq()) Logger.info("index called. Images:
" + images) Ok(views.html.image_index(images)) }



def blank = Action { implicit request => Logger.info("blank called. ")
Ok(views.html.image_details(None,
imageForm,productService.findAllProducts)) }



def details(id: Long) = Action { implicit request => Logger.info("details
called. id: " + id) val image = service.findById(id).get
Ok(views.html.image_details(Some(id),
imageForm.fill(image),productService.findAllProducts)) }



def insert()= Action { implicit request => Logger.info("insert called.")
imageForm.bindFromRequest.fold(

form => {

BadRequest(views.html.image_details(None, form,
productService.findAllProducts)) },

image => {
```

```
If (image.productId==null ||

image.productId.getOrElse(0)==0) {

Redirect(routes.ImageController.blank).

flashing("error" -> "Product ID Cannot be Null!") }else {

if (image.url==null || "".equals(image.url)) image.url =
"/assets/images/default_product.png"

val id = service.insert(image) Redirect(routes.ImageController.index).

flashing("success" -> Messages("success.insert", id)) }

})

}



def update(id: Long) = Action { implicit request =>
Logger.info("updated called. id: " + id)
imageForm.bindFromRequest.fold(

form => {

Ok(views.html.image_details(Some(id), form, null)).flashing("error" ->
"Fix the errors!") },

image => {

service.update(id,image) Redirect(routes.ImageController.index).

flashing("success" -> Messages("success.update", image.id))

})

}
```

```
def remove(id: Long)= Action {

service.findById(id).map { image => service.remove(id)

Redirect(routes.ImageController.index).flashing("success"

-> Messages("success.delete", image.id)) }.getOrElse(NotFound)

}

}
```

Image review works in a similar way to `ReviewController`. We need `IProductService` to get all the services.

Working with services

Services are where we put the business logic. We will look at reactive persistence in Chapter 6, *Persistence with Slick*. Right now, we don't have a database to persist information, so, for now, we will do an in-memory persistence.

First we will define the contract of our services. This is the Base API that we will use in the controllers. Let's take a look at the following trait in `BaseService.scala`: package services

import java.util.concurrent.atomic.AtomicLong import scala.collection.mutable.HashMap

trait BaseService[A] {

var inMemoryDB = new HashMap[Long,A]

var idCounter = new AtomicLong(0)

def insert(a:A):Long

def update(id:Long,a:A):Boolean def remove(id:Long):Boolean

def findById(id:Long):Option[A]

def findAll():Option[List[A]]

}

In the preceding code, we have an in-memory mutable `HashMap`, which is in our memory database where we will store products, images, and reviews. We also have an atomic counter with which we can generate IDs for our models. This is a trait using Generics--as you can see, here we have all the operations with `A`, which will be

specified later. Now we can move the service implementation for product, review, and image.

The `ProductService.scala` package is as follows: package services

import models.Product

import javax.inject._

trait IProductService extends BaseService[Product]{

def insert(product:Product):Long def update(id:Long,product:Product):Boolean def remove(id:Long):Boolean

def findById(id:Long):Option[Product]

def findAll():Option[List[Product]]

def findAllProducts():Seq[(String,String)]

}

@Singleton

class ProductService extends IProductService{

def insert(product:Product):Long = {

val id = idCounter.incrementAndGet() product.id = Some(id)

inMemoryDB.put(id, product) id

```
}

def update(id:Long,product:Product):Boolean = {
validateId(id)
product.id = Some(id)
inMemoryDB.put(id, product) true
}

def remove(id:Long):Boolean = {
validateId(id)
inMemoryDB.remove(id)
true
}

def findById(id:Long):Option[Product] = {
inMemoryDB.get(id)
}
```

```
def findAll():Option[List[Product]] = {

if (inMemoryDB.values == Nil ||

inMemoryDB.values.toList.size==0) return None
Some(inMemoryDB.values.toList) }



private def validateId(id:Long):Unit = {

val entry = inMemoryDB.get(id) if (entry==null) throw new
RuntimeException("Could not find Product: " + id)

}



def findAllProducts():Seq[(String,String)] = {

val products:Seq[(String,String)] = this .findAll()

.getOrElse(List(Product(Some(0),"","",0))) .toSeq

.map { product => (product.id.get.toString,product.name) }

return products

}

}
```

In the last code, we defined a trait called `IProductService`, which extends `BaseService` with a generic apply to product. The `ProductService` package implements `IProductService`. In Scala, we can have multiple classes in the same Scala file, so there is no need to create different files.

The code is very straightforward. There is a utility method here called `findAllProducts`, which is used by review and image controllers. Here we get all the elements on the in-memory hash map. If there are no elements, we return a list with empty product. Then we map the list to a `Seq` of tuple, which is required by the `SelectBox` checkbox that we will have in the view(UI). Now let's go for the image and review services as follows: package services

import javax.inject._

import models.Image

import scala.collection.mutable.HashMap import java.util.concurrent.atomic.AtomicLong

trait IImageService extends BaseService[Image]{

def insert(image:Image):Long def update(id:Long,image:Image):Boolean def remove(id:Long):Boolean

def findById(id:Long):Option[Image]

def findAll():Option[List[Image]]

}

@Singleton

class ImageService extends IImageService{

def insert(image:Image):Long = {

val id = idCounter.incrementAndGet(); image.id = Some(id)

inMemoryDB.put(id, image)

```
id
}

def update(id:Long,image:Image):Boolean = {
validateId(id)
image.id = Some(id)
inMemoryDB.put(id, image)
true
}

def remove(id:Long):Boolean = {
validateId(id)
inMemoryDB.remove(id)
true
}

def findById(id:Long):Option[Image] = {
inMemoryDB.get(id)
}
```

```
def findAll():Option[List[Image]] = {

if (inMemoryDB.values.toList == null ||

inMemoryDB.values.toList.size==0) return None
Some(inMemoryDB.values.toList) }



private def validateId(id:Long):Unit = {

val entry = inMemoryDB.get(id) If (entry==null) throw new
RuntimeException("Could not find Image: " + id)

}



}
```

In the preceding code, we have something pretty similar to that of `ProductService`. We have a trait called `IImageService` and the `ImageService` implementation. Now let's go for the review service implementation in `ReviewService.scala`as follows: package services

```
import javax.inject._

import models.Review

import scala.collection.mutable.HashMap import
java.util.concurrent.atomic.AtomicLong

trait IReviewService extends BaseService[Review]{

def insert(review:Review):Long def
update(id:Long,review:Review):Boolean def
remove(id:Long):Boolean

def findById(id:Long):Option[Review]
```

```
def findAll():Option[List[Review]]
}

@Singleton
class ReviewService extends IReviewService{

def insert(review:Review):Long = {
val id = idCounter.incrementAndGet(); review.id = Some(id)
inMemoryDB.put(id, review) id
}

def update(id:Long,review:Review):Boolean = {
validateId(id)
review.id = Some(id)
inMemoryDB.put(id, review) true
}

def remove(id:Long):Boolean = {
validateId(id)
inMemoryDB.remove(id)
```

```
true

}


def findById(id:Long):Option[Review] = {

inMemoryDB.get(id)

}


def findAll():Option[List[Review]] = {

if (inMemoryDB.values.toList == null ||

inMemoryDB.values.toList.size==0) return None
Some(inMemoryDB.values.toList) }


private def validateId(id:Long):Unit = {

val entry = inMemoryDB.get(id) If (entry==null) throw new
RuntimeException("Could not find Review: " + id)

}

}
```

In the preceding code, we have the `IReviewService` trait and the `ReviewService` implementation. We have validations on the service as well as a good practice.

Configuring the Guice module

We injected classes using `@Inject` in our controllers. The injection happens based on a trait; we need to define a concrete implementation for the traits we injected. The Play framework looks for Guice injections at the location `ReactiveWebStore/app/Module.scala`. Okay, so let's define our injections for the three controllers we just created.

The Guice module is found in `Module.scala`as follows: import com.google.inject.AbstractModule import java.time.Clock

import services.{ApplicationTimer}

import services.IProductService import services.ProductService

import services.ReviewService

import services.IReviewService

import services.ImageService

import services.IImageService

/**

* This class is a Guice module that tells Guice how to bind several * different types. This Guice module is created when the Play * application starts.

* Play will automatically use any class called `Module` that is in * the root package. You can create modules in other locations by * adding `play.modules.enabled` settings to the `application.conf`

```
* configuration file.

*/

class Module extends AbstractModule {



override def configure() = {

// Use the system clock as the default implementation of Clock
bind(classOf[Clock]).toInstance(Clock.systemDefaultZone) // Ask
Guice to create an instance of ApplicationTimer // when the
application starts.

bind(classOf[ApplicationTimer]).asEagerSingleton()
bind(classOf[IProductService]).to(classOf[ProductService]).

asEagerSingleton()

bind(classOf[IReviewService]).to(classOf[ReviewService]).

asEagerSingleton()

bind(classOf[IImageService]).to(classOf[ImageService]).

asEagerSingleton()

}

}
```

So we just need to add `bind` with our traits for the controllers, and then point to the controller implementation. They should also be created as singletons, as the Play framework starts our application. Here you also can define any other configuration or injection that our application may need. The last code defines three services: product service, `IReviewService`, and `IImageService`.

Working with views(UI)

The Play framework works with a Scala-based templating engine called Twirl. Twirl was inspired by ASP.NET Razor. Twirl is compact and expressive; you will see we can do more with less. Twirl template files are simple text files, however, the Play framework compiles the templates and turns them into Scala classes. You can mix HTML with Scala smoothly in Twirl.

The UI will be compiled into a Scala class, that can and will be referenced at our controllers, because we can route to a view. The nice thing about it is that this makes our coding way safer, since we have the compiler checking for us. The bad news is that you need to compile your UI, otherwise, your controllers won't find it.

Previously in this chapter, we defined controllers for products, images, and reviews, and we wrote the following code:

```
Ok(views.html.product_details(None, productForm))
```

With the preceding code, we redirect the user to a blank page for products so that the user can create a new product. We also can pass parameters to the UI. Since it is all Scala code, you are actually just calling a function as follows:

```
val product = service.findById(id).get
    Ok(views.html.product_details(Some(id),
productForm.fill(product)))
```

In the preceding code, we call the service to retrieve a product by ID, and then pass the object to the UI with the form being filled.

Let's continue building our application and create the UI for the products, reviews, and images. Since we are doing a CRUD, we will need more than one template file per CRUD. We will need the following structure:

- Index Template
    - List all items
    - Link to edit one item
    - Link to remove one item
    - Link to create a new Item
        - Detail Template
            - HTML Form to create a new Item
            - HTML form to edit an existing item(for update)

Having said that, we will have the following files:

- For Products:
    - product_index.scala.html
    - product_details.scala.html
        - For Image:
            - image_index.scala.html
            - image_details.scala.html
                - For Reviews:
                    - review_index.scala.html
                    - review_details.scala.html

For the sake of code reuse, we will create another file containing the basic structure of our UI, like CSS imports (CSS needs to be located at `ReactiveWebStore\public\stylesheets`), JavaScript imports, and page title so that we don't need to repeat that in all the templates for each CRUD. This page will be called: `main.scala.html`.

All the UI code should be located at `ReactiveWebStore/app/views`.

The main Scala with the UI index for all CRUD operations is in `main.scala.html`, as follows:

```
@(title: String)(content: Html)(implicit flash:
Flash)

    <!DOCTYPE html>
      <html lang="en">
        <head>
          <title>@title</title>
          <link rel="shortcut icon"
type="image/png"

href="@routes.Assets.at("images/favicon.png")">
          <link rel="stylesheet" media="screen"

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

href="@routes.Assets.at("stylesheets/bootstrap.min
.css")">
          <script
src="@routes.Assets.at("javascripts/jquery-
          1.9.0.min.js")" type="text/javascript">
</script>
          <script
src="@routes.Assets.at("javascripts/bootstrap.js")
"
          type="text/javascript"></script>
          <script
src="@routes.Assets.at("javascripts/image.js")"
          type="text/javascript"></script>
        </head>
        <body>
          <center><a href='/'><img height='42'
width='42'

src='@routes.Assets.at("images/rws.png")'></a>
          <h3>@title</h3></center>
          <div class="container">
            @alert(alertType: String) = {
              @flash.get(alertType).map { message
=>
                <div class="alert alert-
@alertType">
                  <button type="button"
class="close" data-
                  dismiss="alert">&times;</button>
                  @message
                </div>
```

```
                }
              }
            @alert("error")
            @alert("success")
            @content
            <a href="/"></a><BR>
            <button type="submit" class="btn btn-
primary"
            onclick="window.location.href='/'; " >
            Reactive Web Store - Home
            </button>
          </div>
        </body>
      </html>
```

In the preceding code, first of all, there is the following line at the very top:

```
@(title: String)(content: Html)(implicit flash:
Flash)
```

This means that we define the parameters this UI can receive. Here we expect a string title, which will be the page title, and there are some currying variables as well. You can get more details about currying in Chapter 1, *Introduction to FP, Reactive, and Scala*. So in currying, there are two things: First is HTML, which means you can pass HTML code to this function, and second, we have `Flash` which the Play framework will pass for us. `Flash` is used to get parameters between requests.

As you can see, later in the code we have `@title`, which means we retrieve the title of the parameters, and add the value to the HTML. We also print any error message or any validations issues, if present, with `@alert`. We import JQuery and Twitter Bootstrap, but we do not put hard-coded paths. Instead, we use the routers like `@routes.Assets.at`. The Javascripts still need to be located at `ReactiveWebStore\public\javascripts`.

Now other templates can work with `@main(..),` and they don't need any declaration of Javascript or CSS. They can add an extra HTML

code, which will be rendered on the previous code by `@content`. So, for the products, the content will be HTML product content, and so on for reviews, and images. Now we can move for the products UI.

Product index: UI index for products `product_index.scala.html`

```
@(products:Seq[Product])(implicit flash: Flash)

    @main("Products") {

      @if(!products.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>Name</th>
            <th>Details</th>
            <th>Price</th>
            <th></th>
          </tr>
          @for(product <- products) {
            <tr>
              <td><a
href="@routes.ProductController.

details(product.id.get)">@product.name</a></td>
              <td>@product.details</td>
              <td>@product.price</td>
              <td><form method="post" action=

"@routes.ProductController.remove(product.id.get)"
>
              <button class="btn btn-link"
type="submit">
              <i class="icon-trash">
</i>Delete</button>
              </form></td>
            </tr>
          }
        </table>
      }
      <p><a href="@routes.ProductController.blank"
class=
      "btn btn-success"><i class="icon-plus icon-
white">
      </i>Add Product</a></p>
    }
```

As you can see, there is HTML mixed with Scala code. Every time you need to run HTML, you just run it, and when you need run Scala code, you need use a special character, `@`. At the very top of the template, you can see the following code:

```
@(products:Seq[Product])(implicit flash: Flash)
```

Since this is Scala code in the end, and will be compiled, we need to define what parameters this UI template can receive. Here we expect a sequence of products. There is also a currying implicit variable called `Flash`, which will be provided by the Play framework, and we will use it for the message display. We also have the code-`@main("Products") { .. }`. This means that we call the main Scala template and add extra HTML--the product HTML. For this product UI, we list all the products based on the sequence of products. As you can see, we define an HTML table. We also validate if the sequence is not empty before listing all the products.

Now we can go for the details page for products in `product_details.scala.html` as follows:

```
@(id: Option[Long],product:Form[Product])(implicit
flash:Flash)

    @import play.api.i18n.Messages.Implicits._
    @import play.api.Play.current

    @main("Product: " +
product("name").value.getOrElse("")){

      @if(product.hasErrors) {
        <div class="alert alert-error">
          <button type="button" class="close"
data-
          dismiss="alert">&times;</button>
          Sorry! Some information does not look
right. Could you
          review it please and re-submit?
        </div>
      }
```

```
      @helper.form(action = if (id.isDefined)
        routes.ProductController.update(id.get)
else
        routes.ProductController.insert)
      {
        @helper.inputText(product("name"),
'_label -> "Product
        Name")
        @helper.inputText(product("details"),
'_label -> "Product
        Details")
        @helper.inputText(product("price"),
'_label -> "Price")
        <div class="form-actions">
          <button type="submit" class="btn btn-
primary">
            @if(id.isDefined) { Update Product }
else { New Product }
          </button>
        </div>
      }

    }
```

For this preceding UI, at the very top, we have the following line:

```
@(id: Option[Long],product:Form[Product])(implicit
flash:Flash)
```

This means that we expect an ID which is completely optional. This ID is used to know if we are dealing with an insert scenario or an update scenario, because we use the same UI for both insert and update. We also get the product form, which will be passed through `ProductController`, and we receive `Flash`, which will be provided by the Play framework.

We need to do some imports on the UI so that we can get access to the Play framework i18n support. This is done by the following:

```
@import play.api.i18n.Messages.Implicits._
    @import play.api.Play.current
```

Like the previous UI, we render this UI within the main Scala UI. So we don't need to specify JavaScript and CSS again. This is done by the following code:

```
@main("Product: " +
product("name").value.getOrElse("")){ .. }
```

Next we check if there are any validation errors. If there are, the users will need to fix the errors before moving on. This is done by the following code:

```
@if(product.hasErrors) {
      <div class="alert alert-error">
        <button type="button" class="close" data-
        dismiss="alert">&times;</button>
        Sorry! Some information does not look
right. Could you review
        it please and re-submit?
      </div>
    }
```

Now is the time to create the product form, which, in the end, will be mapped to HTML input boxes. We do this with the following code:

```
@helper.form(action = if (id.isDefined)
    routes.ProductController.update(id.get) else
    routes.ProductController.insert) {
      @helper.inputText(product("name"),
'_label -> "Product Name")
      @helper.inputText(product("details"),
'_label -> "Product
      Details")
      @helper.inputText(product("price"),
'_label -> "Price")
      <div class="form-actions">
        <button type="submit" class="btn btn-
primary">
          @if(id.isDefined) { Update Product }
else { New Product }
        </button>
      </div>
    }
```

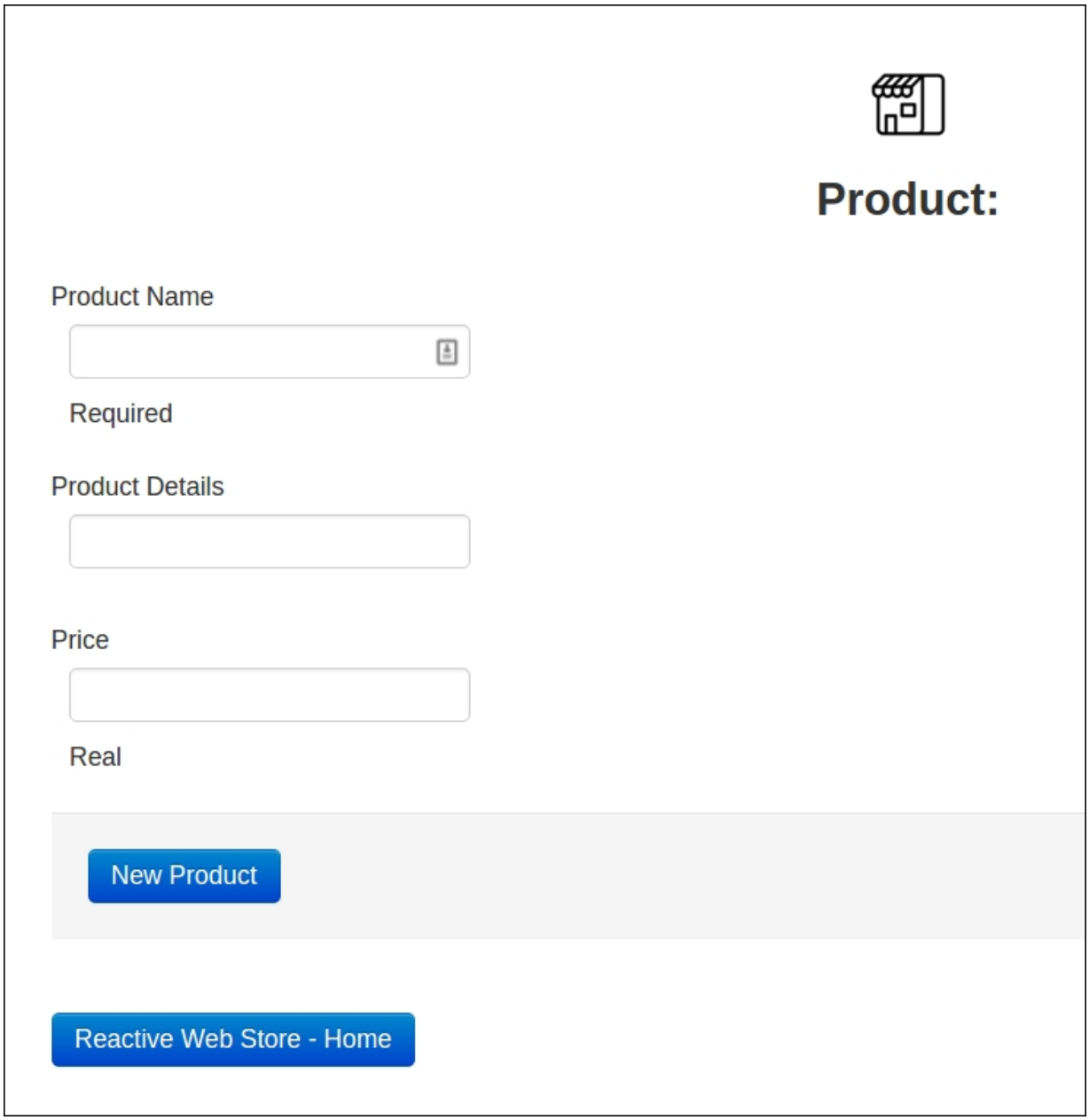
`@helper.form` is a special helper provided by the Play framework to create HTML forms easily. So, the action is the target where the form will be submitted. We need to do an `if` here, since we need to know if it is an update or an insert. Then we map all the fields we have for our product model with this code:

```
@helper.inputText(product("name"),    '_label ->
"Product Name")
    @helper.inputText(product("details"), '_label
-> "Product Details")
    @helper.inputText(product("price"),   '_label
-> "Price")
```

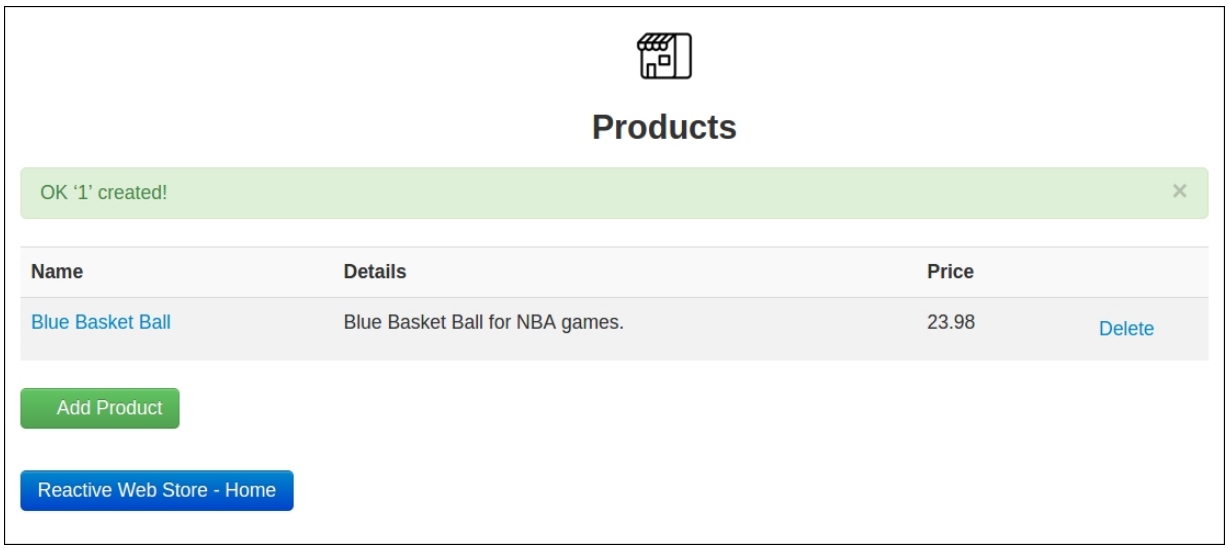
Remember, the product form comes from the product controller. For the helper, we just need to tell it which product field is for which HTML label, and that's it. This will produce the following UIs.

The following image shows the blank product index UI:

The insert UI form for product details looks as follows:



With products added, the product index UI appears as follows:



Now we can move to reviews. Let's go for the UIs.

The review index UI in `review_index.scala.html` is as follows:

```
@(reviews:Seq[Review])(implicit flash: Flash)

    @main("Reviews") {

      @if(!reviews.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>ProductId</th>
            <th>Author</th>
            <th>Comment</th>
            <th></th>
          </tr>
          @for(review <- reviews) {
            <tr>
              <td><a
href="@routes.ReviewController.details

(review.id.get)">@review.productId</a></td>
              <td>@review.author</td>
              <td>@review.comment</td>
              <td>
              <form method="post"
action="@routes.ReviewController.
              remove(review.id.get)">
```

```
                <button class="btn btn-link"
type="submit"><i class=
                "icon-trash"></i>Delete</button>
              </form></td>
            </tr>
          }
        </table>
      }
      <p><a href="@routes.ReviewController.blank"
class="btn btn-
      success"><i class="icon-plus icon-white">
</i>Add Review</a></p>
    }
```

So here we have the same things as we had for the products. Let's take a look at the details page for review now. You can find it in `review_details.scala.html`.

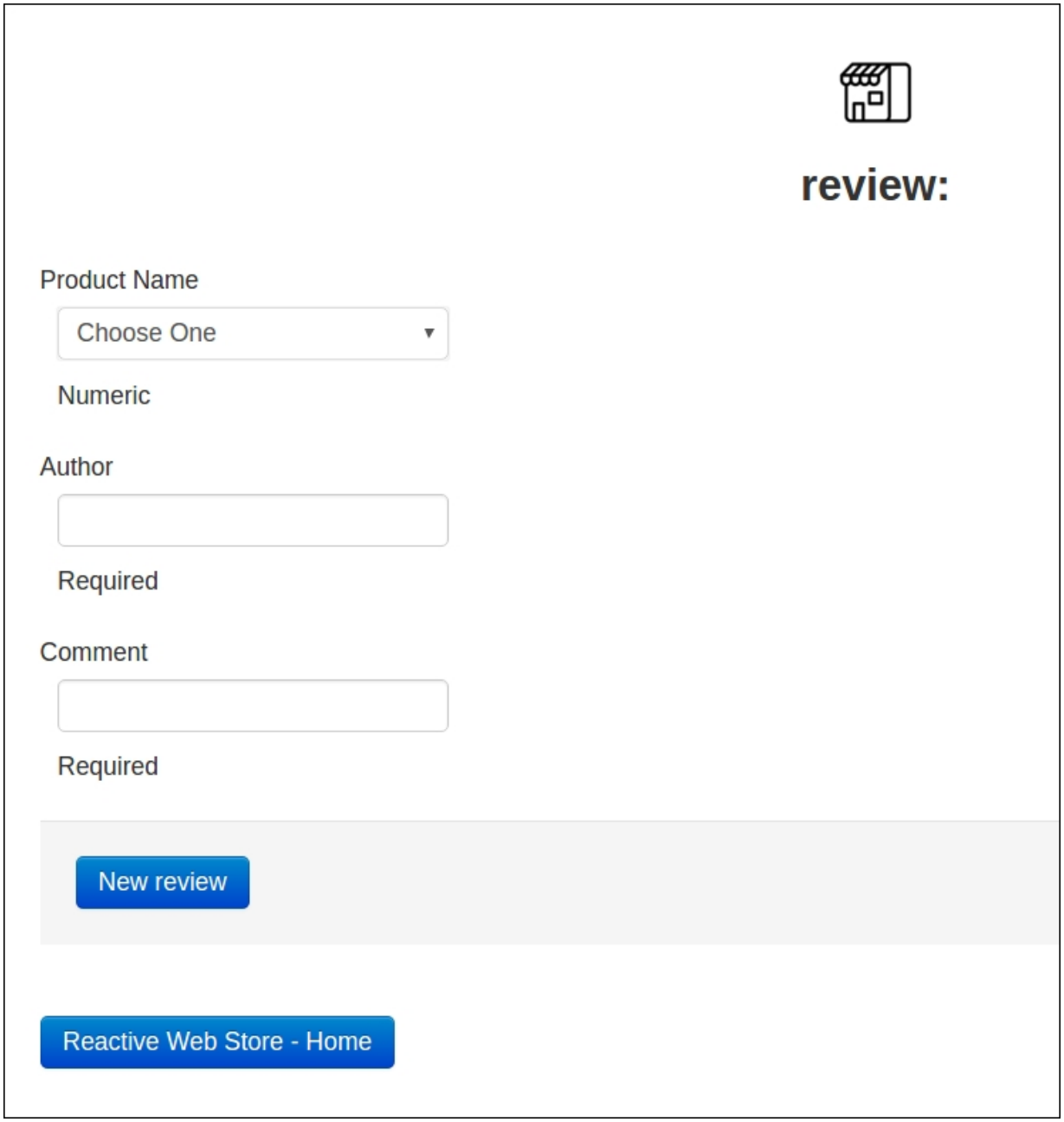
```
@(id: Option[Long],review:Form[Review],products:
    Seq[(String,String)])(implicit flash:Flash)

    @import play.api.i18n.Messages.Implicits._
    @import play.api.Play.current

    @main("review: " +
review("name").value.getOrElse("")){

      @if(review.hasErrors) {
        <div class="alert alert-error">
          <button type="button" class="close"
data-
          dismiss="alert">&times;</button>
          Sorry! Some information does not look
right. Could you
          review it please and re-submit?
        </div>
      }

      @helper.form(action = if (id.isDefined)
      routes.ReviewController.update(id.get) else
      routes.ReviewController.insert) {
        @helper.select(
          field = review("productId"),
          options = products,
          '_label -> "Product Name",
          '_default ->
```

```
review("productId").value.getOrElse("Choose
          One"))
          @helper.inputText(review("author"),
'_label -> "Author")
          @helper.inputText(review("comment"),
'_label ->
          "Comment")
          <div class="form-actions">
            <button type="submit" class="btn btn-
primary">
            @if(id.isDefined) { Update review }
else { New review }
          </button>
          </div>
      }

    }
```

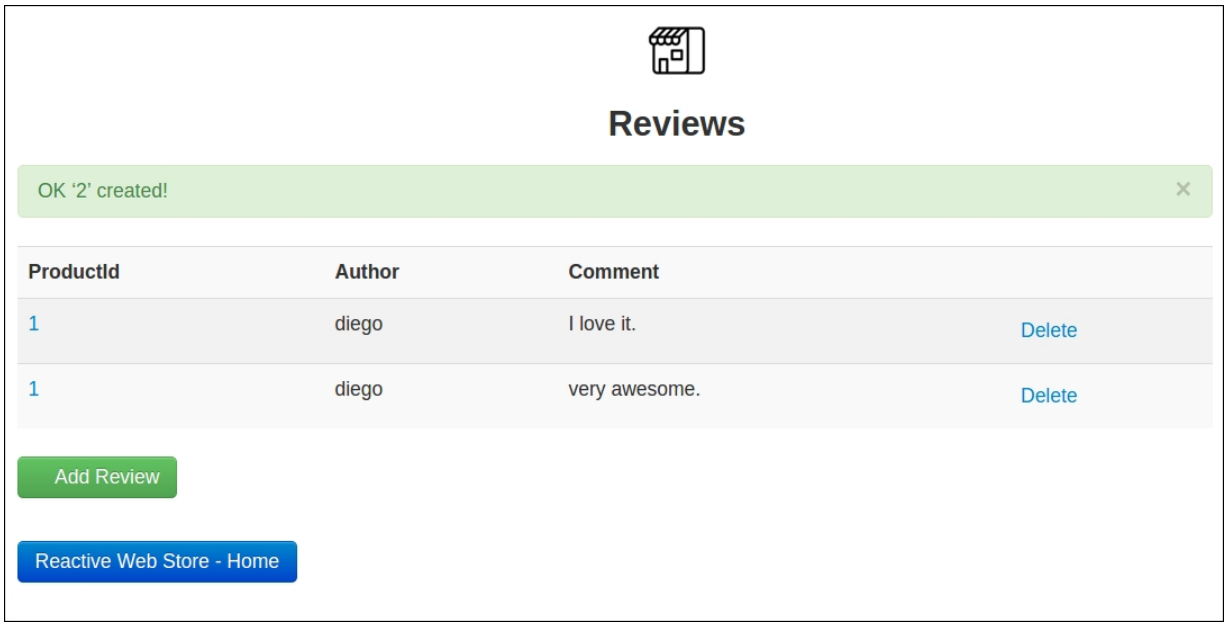
Here, in this last code, we have almost everything similar to what we had for products, however, there is one big difference. Review needs to be associated with a product ID. That's why, we need to have a select for the products, which is fulfilled by `products:Seq[(String,String)]`. This comes from the `ReviewController` code. This code produces the following UIs.

The blank review index UI is shown as follows:

The insert review details UI looks as follows:



The review index UI with reviews will look like the following image:



Now we can move to the last one: the image UI. The image UI is very similar to the review UI, because it depends on the product ID too. Let's go for it.

The image index UI has the following code in `image_index.scala.html`:

```
@(images:Seq[Image])(implicit flash:Flash)
    @main("Images") {
      @if(!images.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>ProductID</th>
            <th>URL</th>
            <th></th>
          </tr>
          @for(image <- images) {
            <tr>
              <td><a
href="@routes.ImageController.details
              (image.id.get)">@image.id</a></td>
              <td>@image.productId</td>
```

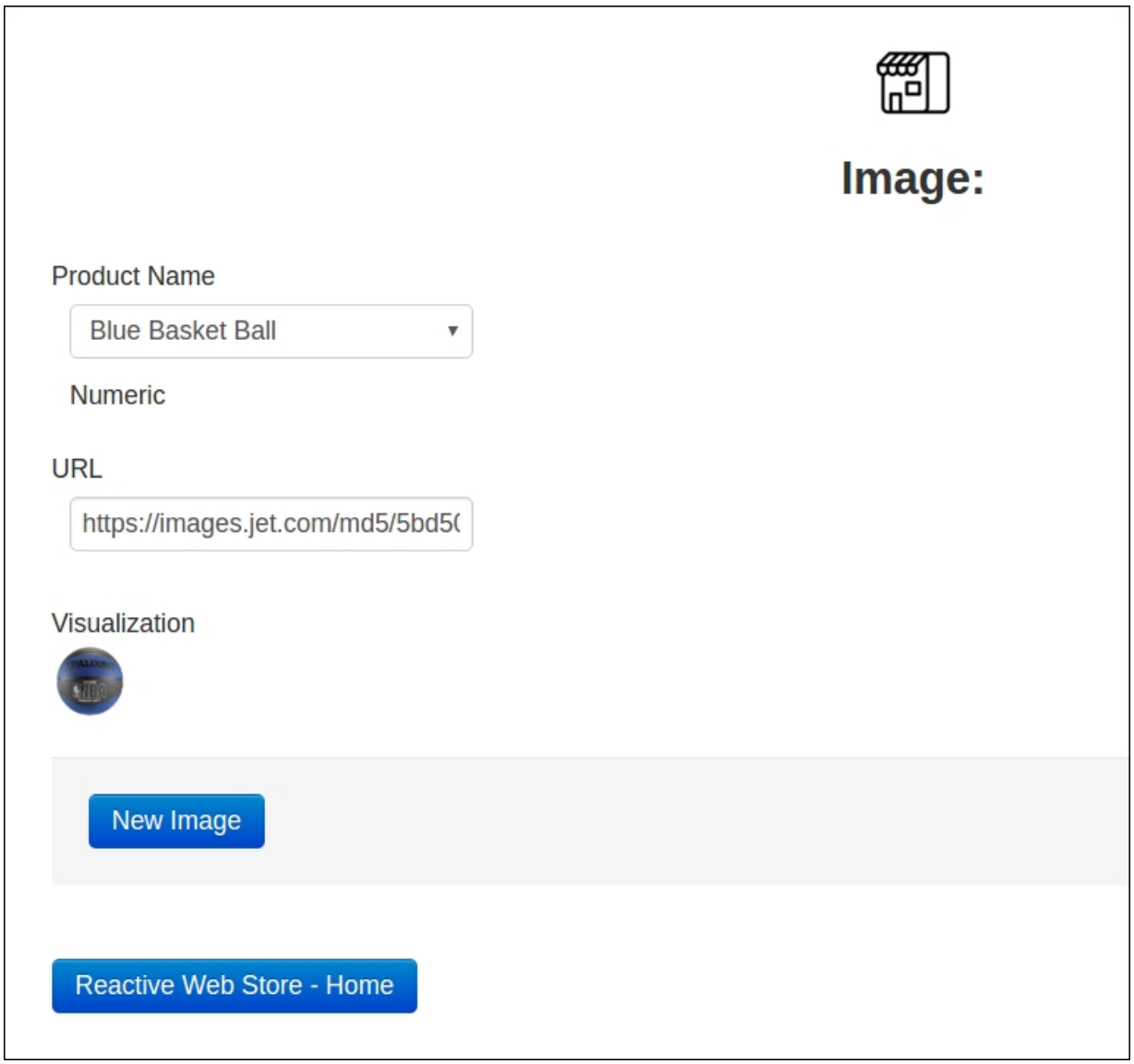
```
              <td>@image.url</td>
              <td><form method="post" action=
"@routes.ImageController.remove(image.id.get)">
                <button class="btn btn-link"
type="submit">
                <i class="icon-trash">
</i>Delete</button>
              </form></td>
            </tr>
          }
        </table>
      }
      <p><a href="@routes.ImageController.blank"
class=
      "btn btn-success"><i class="icon-plus icon-
white">
      </i>Add Image</a></p>
    }

    Image Details UI [image_details.scala.html]

    @(id:
Option[Long],image:Form[Image],products:Seq[(Strin
g,String)])
    (implicit flash:Flash)
    @import play.api.i18n.Messages.Implicits._
    @import play.api.Play.current
    @main("Image: " +
image("productId").value.getOrElse("")){
      @if(image.hasErrors) {
        <div class="alert alert-error">
          <button type="button" class="close"
data-
          dismiss="alert">&times;</button>
          Sorry! Some information does not look
right. Could you image
          it please and re-submit?
        </div>
      }

      @helper.form(action = if (id.isDefined)
      routes.ImageController.update(id.get) else
      routes.ImageController.insert) {
        @helper.select(field = image("productId"),
          options = products,
          '_label -> "Product Name",
          '_default ->
```

```
image("productId").value.getOrElse("Choose
          One")
        )
        @helper.inputText(
          image("url"),
          '_label       -> "URL",
          '_placeholder ->
"/assets/images/default_product.png",
          'onchange     ->
"javascript:loadImage();"
        )
        Visualization<br>
        <img id="imgProduct" height="42"
width="42"
        src="@image("url").value"></img>
        <div class="form-actions">
          <button type="submit" class="btn btn-
primary">
            @if(id.isDefined) { Update Image }
else { New Image }
          </button>
        </div>
      }

    }
```

This UI template will create the following HTML Pages:

The blank image index UI is shown in the following image:

The insert UI for image details looks as follows:



The following is the image index UI with items:

Now we have a complete working UI application. There are controllers, models, views, and simple services as well. We also have all validations in place.

Summary

In this chapter, you learned how to create controllers, models, services, views (using Twirl templating), Guice injections, and routing. We covered the principles of Scala Web Development using the Play framework. By the end of the chapter, we got the application with the Play framework up and running.

In the next chapter, we will learn more about services. As you may realize, we did some simple services in this chapter for products, reviews, and images, but now we will continue working with services.

Chapter 4. Developing Reactive Backing Services

In the previous chapter, you learned how to Bootstrap your application using Activator, and we developed our web application using Scala and the Play framework. Now we will enter into the reactive world of RxJava and RxScala.

In this chapter, we will cover the following topics:

- Reactive programming principles and the Reactive Manifesto
- Understanding the importance of non-blocking IO
- Observables, functions, and error handling with Rx
- Refactoring our controllers and models to call our services
- Adding RxScala to our services
- Adding logging

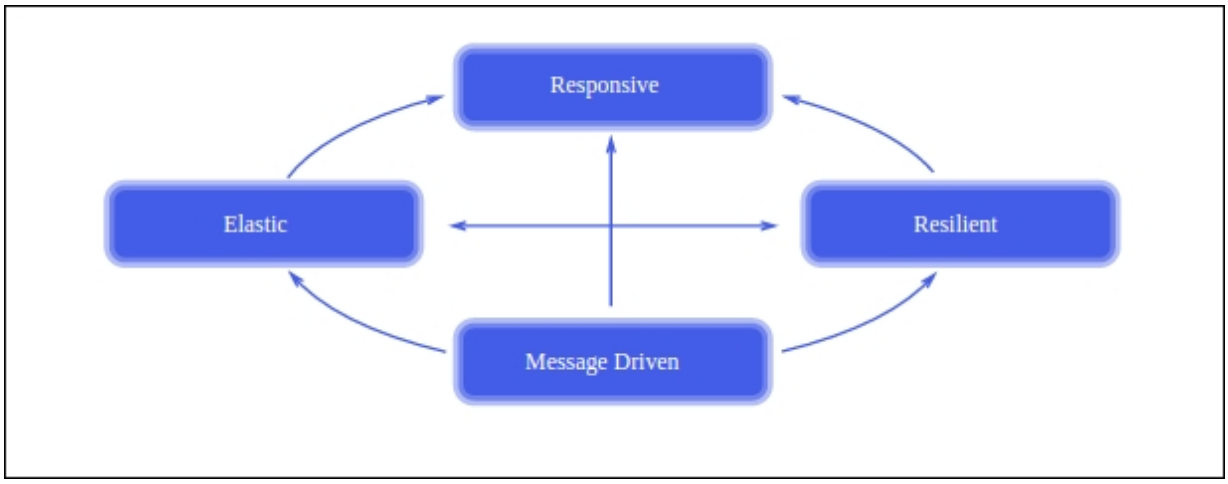
Getting started with reactive programming

Building applications today is harder than it was before. Everything now is more complex: we have to use more cores in processors, and we have cloud-native applications with hundreds of machines for a single service. Concurrent programming has always been hard, and it will always be so, because it is difficult to model time. In order to address this, we need to have a reactive style of architecture. In order to be able to handle more users and scale our applications, we need to leverage Async and non-blocking IO. To help us with this task, we can rely on RxJava and RxScala. Being reactive is not only about code but also about architectural principles.

The Reactive Manifesto captures these principles very well, and there are a couple of technologies that follow these principles in

order to be fully reactive.

The Reactive Manifesto can be shown as in the following diagram:



For more information, you can visit http://www.reactivemanifesto.org/.

The Reactive Manifesto describes what this reactive architecture/system looks like. Basically, there are the following four core principles underlining the reactive idea:

- **Responsive**: The system should respond in a timely manner. In other words, the system should detect problems quickly, and deal with them effectively, apart from providing rapid and consistent response time.
- **Resilient**: The system should stay responsive even after failure. This is done via replication containment, isolation, and delegation (https://en.wikipedia.org/wiki/Delegation_pattern). Containment and isolation are ideas that come from the naval industry, and are defined by the bulkhead pattern (https://en.wikipedia.org/wiki/Bulkhead_(partition)). Failures are contained at each component. Doing so makes sure that one

system's failure does not affect other systems. Recovery is delegated to another system, and not to the client.
- **Elastic**: The ability to increase and decrease resources for the system. This requires you design your system without **Single Point Of Failure** (**SPOF**), and design using shards and replication. Reactive systems are predictive and cost-effective.
- **Message-driven**: Reactive systems rely on asynchronous message passing to ensure loose coupling, isolation, and location transparency. By doing so, we can delegate failures as messages. This gives us elasticity, load management, and flow control. It's even possible to apply back-pressure (also known as throttling) when needed. All this should be done with non-blocking communication for better resource utilization.

Alright, let's use these principles practically in our application with RxScala. RxScala is just a Scala wrapper for RxJava, but it is better to use because it makes the code more functional, and you don't need to create objects such as `Action1`.

In our application, we have three major resources: products, reviews, and images. All products must have a price, so we will built a fully reactive price generator with the Play framework, RxScala, and Scala right now.

So first of all, we will play with RxScala in our Play application, then we will create a separate microservice, make reactive calls to that microservice, and retrieve our price suggestion for that service. All data flow transformations are using observables.

Let's create the routes for this controller at `ReactiveWebStore/conf/routes` , as follows:

```
#
    # Services
    #
    GET   /rx/prices
controllers.RxController.prices
    GET   /rx/aprices
controllers.RxController.pricesAsync
```

We have two routes here: one for a regular action, and another for an Async action that will return a Scala Future. Let's create a new controller called `Rx Controller.scala`. This controller needs to be located at `ReactiveWebStore/app/controller`.

Let's have a look at `RxController`, which is our reactive RxScala simple controller:

```
@Singleton
    class RxController @Inject()
(priceService:IPriceSerice) extends
    Controller {
      def prices = Action { implicit request =>
        Logger.info("RX called. ")
        import ExecutionContext.Implicits.global
        val sourceObservable =
priceService.generatePrices
        val rxResult = Observable.create {
sourceObservable.subscribe
      }
      .subscribeOn(IOScheduler())
      .take(1)
      .flatMap { x => println(x) ;
Observable.just(x) }
      .toBlocking
      .first
      Ok("RxScala Price suggested is = " +
rxResult)
    }

      def pricesAsync = Action.async { implicit
request =>
        Logger.info("RX Async called. ")
        import
play.api.libs.concurrent.Execution.Implicits.
        defaultContext
        val sourceObservable =
priceService.generatePrices
        val rxResult = Observable.create {
sourceObservable.subscribe
        }
        .subscribeOn(IOScheduler())
        .take(1)
        .flatMap { x => println(x) ;
Observable.just(x) }
```

```
            .toBlocking
            .first
            Future { Ok("RxScala Price sugested is = "
+ rxResult) }
        }
    }
```

So, in the very first method called `prices`, we return a regular Play framework Action. We receive `IPriceService` via dependency injection. This `IPriceService` is a reactive service, because it uses observables. So we call a method, `generatePrices`, which will return `Observable[Double]`. This will be our source observable, that is, the data source of our computation. Moving forward, we create a new observable subscribing into the source observable, and then we apply some transformation. For instance, we take just one element, and then we can perform transformation using `flatMap`. For this case, we do not really apply transformations. We use `flatMap` to simply print what we got, and then continue the chain. The next step is to call `toBlocking`, which will block the thread until the data is back. Once the data is back, we get the first element, which will be a double, and we return `Ok`.

Blocking sounds bad and we don't want that. Alternatively, we can use the `async` controller in the Play framework, which won't block the thread and return a Future. So that's the second method, called `pricesAsync`. Here we have similar observable code. However, in the end, we return a Future which is not blocking. However, we call `toBlocking` from the observable that will block the call, thus making it the same as the previous method. To be clear, Action is not bad. By default, everything is Async in Play, because even if you don't return an explicit Future, the Play framework creates a promise for `y` and makes you code Async. Using HTTP, you will block the thread at some point. If you want to be 100% non-blocking from end to end, you need to consider a different protocol such as web sockets.

Let's take a look at the service now. This service, and other services, need to be located at `ReactiveWebStore/apps/services`. First we will create `trait` to define the service behavior.

IPriceService - Scala trait

As you can see in the following code, we defined `IPriceService` with just one operation, that is, `generatePrices`, which returns `Observable[Double]`. The next step now is to define the service implementation. This code needs to be located in the same services folder as the previous trait:

```
trait IPriceSerice{
      def generatePrices:Observable[Double]
   }
```

PriceService - RxScala PriceService implementation

First we create `PublishSubject`, which is a way to generate data into observables. Scala has a nice way of generating infinite sequences using `Stream.continually`. So we pass a function which generates double random numbers from 0 to 1,000. This will happen forever and, because this is an expensive computation, we run it into a Future. The right thing to do will be to use a method after `Stream`, because this will finish the computation. For the sake of the exercise, we will keep it this way for now.

Each double random number is published into `PublishSubject` by the `onNext` method. Now let's move to the `generatePrices` method, which uses three observables to generate the number for us. To be clear, of course we could do a simpler solution here. However, we are doing it this way to illustrate the power of observables, and how you can use them in practice.

We have `Even` and `Odd` observables, both subscribing to `PublishSubject`, so they will receive infinite double numbers. There is a `flatMap` operation to add `10` to the number. Keep in mind that everything you do needs to be in an observable. So when you do

transformations with `flatMap`, you always need to return an observable.

Finally, we apply the filter function to get only even numbers on the `Even` observable and only odd numbers on the `Odd` observable. All this happens in parallel. `Even` and `Odd` observables do not wait for each other.

The next step is to merge both observables. We create a third observable that starts empty and then merges the infinite doubles from the `Even` observable with the infinite doubles from the `Odd` observable. Now is the time to limit the computation to only 10 numbers. We don't know how many odd or how many even numbers will be there because of Async. If you wish to control the number of odds and evens, you need to apply the `take` function on each observable.

Finally, we apply `foldLeft` to sum all the numbers and get a total. However, when we do that, we get only 90% of the numbers. This last observable is what is returned to the controller. Nothing is blocked here, and it's all Async and reactive.

You maybe wondering why `Stream.Continuously` generates different values all the time. That happens because we use a Call-by-Name function in Scala. We import the `nextDouble` function, and pass a function instead of the value of the function:

```
@Singleton
    class PriceService extends IPriceSerice{
      var doubleInfiniteStreamSubject =
PublishSubject.apply[Double]()
      Future {
        Stream.continually(nextDouble * 1000.0
).foreach {
          x => Thread.sleep(1000);
          doubleInfiniteStreamSubject.onNext(x)
        }
      }
      override def
generatePrices:Observable[Double] = {
```

```
        var observableEven = Observable.create {
        doubleInfiniteStreamSubject.subscribe }
        .subscribeOn(IOScheduler())
        .flatMap { x => Observable.from(
Iterable.fill(1)(x + 10) )
        }
        .filter { x => x.toInt % 2 == 0 }
        var observableOdd = Observable.create {
        doubleInfiniteStreamSubject.subscribe }
        .subscribeOn(IOScheduler())
        .flatMap { x => Observable.from(
Iterable.fill(1)(x + 10) )
        }
        .filter { x => x.toInt % 2 != 0 }
        var mergeObservable = Observable
        .empty
        .subscribeOn(IOScheduler())
        .merge(observableEven)
        .merge(observableOdd)
        .take(10)
        .foldLeft(0.0)(_+_)
        .flatMap { x => Observable.just( x - (x *
0.9) ) }
         return mergeObservable
      }
    }
```

We need to register this service in Guice in `Module.scala` located in the default package at `ReactiveWebStore/app`.

Guice Injection - Module.scala

Your `Module.scala` file should look something like this:

```
class Module extends AbstractModule {
      override def configure() = {
        // Use the system clock as the default
implementation of Clock

bind(classOf[Clock]).toInstance(Clock.systemDefaul
tZone)
        // Ask Guice to create an instance of
ApplicationTimer when
        //the
        // application starts.
```

```
        bind(classOf[ApplicationTimer]).asEagerSingleton()

        bind(classOf[IProductService]).to(classOf[ProductS
        ervice]).
                asEagerSingleton()

        bind(classOf[IReviewService]).to(classOf[ReviewSer
        vice]).
                asEagerSingleton()

        bind(classOf[IImageService]).to(classOf[ImageServi
        ce]).
                asEagerSingleton()

        bind(classOf[IPriceSerice]).to(classOf[PriceServic
        e]).
                asEagerSingleton()
              }
            }
```

In order to compile and run the preceding code, we need to add an extra SBT dependency. Open `build.sbt`, and add RxScala. We will also add another dependency, which is ws, a play library to make web service calls. We will use it later in this chapter.

Your `build.sbt` should look something like this:

```
        name := """ReactiveWebStore"""
            version := "1.0-SNAPSHOT"
            lazy val root = (project in
        file(".")).enablePlugins(PlayScala)
            scalaVersion := "2.11.7"

            libraryDependencies ++= Seq(
              jdbc,
              cache,
              ws,
              "org.scalatestplus.play" %% "scalatestplus-
        play" % "1.5.1" %
              Test,
              "com.netflix.rxjava" % "rxjava-scala" %
        "0.20.7"
            )
```

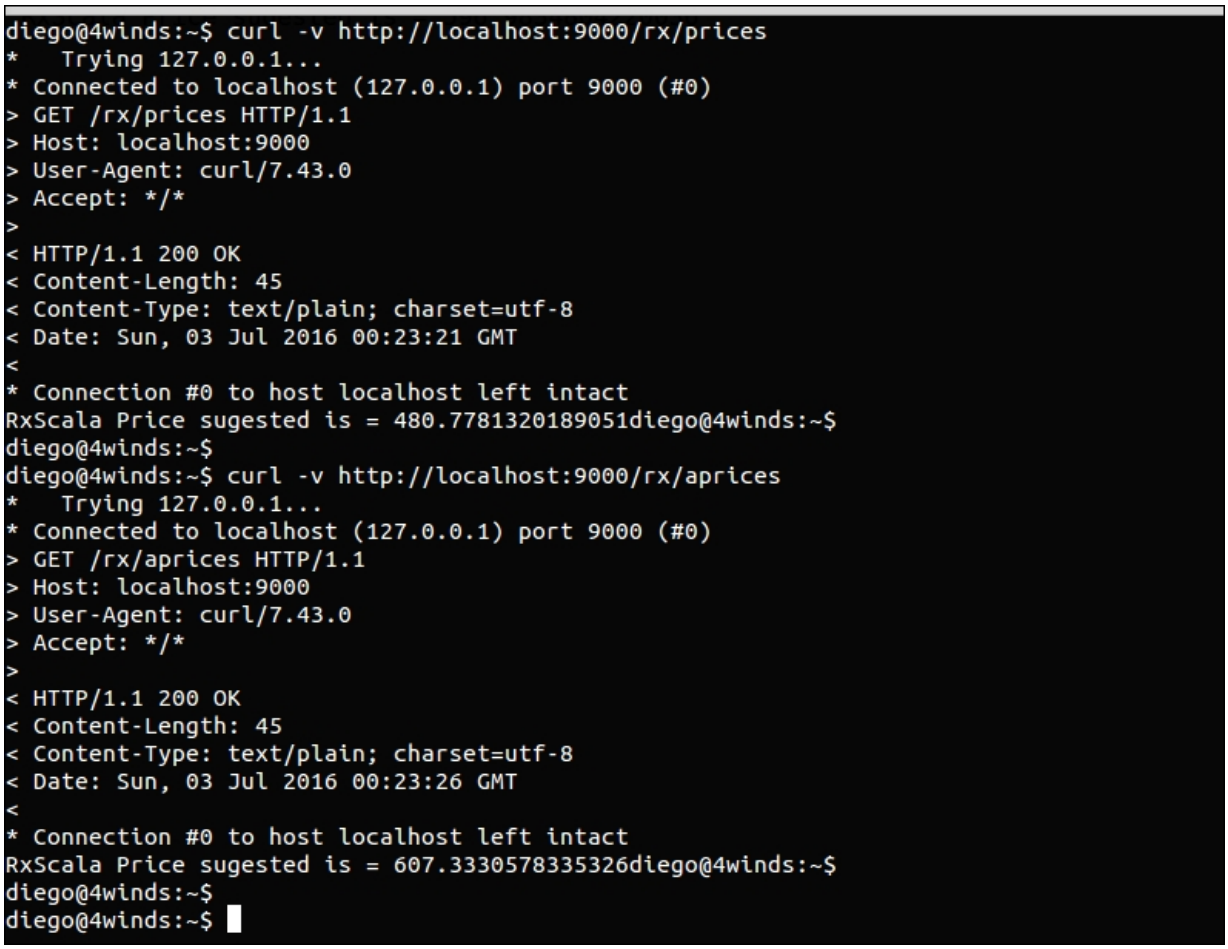
```
resolvers += "scalaz-bintray" at
"http://dl.bintray.com/scalaz/releases"
resolvers += DefaultMavenRepository
```

Now we can compile and run this code using `activator run`.

We can call this new route now using a CURL call. If you prefer, you can just open your browser and do it there.

```
curl -v http://localhost:9000/rx/prices
curl -v http://localhost:9000/rx/aprices
```

We will see the following result:



Great! We have RxScala working with the Play framework. Now we will refactor our code to make it even more interesting. So we will create a microservice using the Play framework, and we will externalize this random number generator to the microservice.

We need to create a new Play framework application. We will pick the option number `6) play-scala` application template. Open your console, and then type the following command:

```
$ activator new ng-microservice
```

You will see this result:

```
diego@4winds:/tmp$ activator new ng-microservice

Fetching the latest list of templates...

Browse the list of templates: http://lightbend.com/activator/templates
Choose from these featured templates or enter a template name:
  1) minimal-akka-java-seed
  2) minimal-akka-scala-seed
  3) minimal-java
  4) minimal-scala
  5) play-java
  6) play-scala
(hit tab to see a list of all templates)
> 6
OK, application "ng-microservice" is being created using the "play-scala" template.

To run "ng-microservice" from the command line, "cd ng-microservice" then:
/tmp/ng-microservice/activator run

To run the test for "ng-microservice" from the command line, "cd ng-microservice" then:
/tmp/ng-microservice/activator test

To run the Activator UI for "ng-microservice" from the command line, "cd ng-microservice" then:
/tmp/ng-microservice/activator ui

diego@4winds:/tmp$ cd ng-microservice/
diego@4winds:/tmp/ng-microservice$ ll
total 88
drwxrwxr-x  9 diego diego  4096 Jul  2 17:35 ./
drwxrwxrwt 26 root  root  36864 Jul  2 17:35 ../
drwxrwxr-x  6 diego diego  4096 Jul  2 17:35 app/
drwxrwxr-x  2 diego diego  4096 Jul  2 17:35 bin/
-rw-rw-r--  1 diego diego   345 Jul  2 17:35 build.sbt
drwxrwxr-x  2 diego diego  4096 Jul  2 17:35 conf/
-rw-rw-r--  1 diego diego    80 Jul  2 17:35 .gitignore
drwxrwxr-x  2 diego diego  4096 Jul  2 17:35 libexec/
-rw-rw-r--  1 diego diego   591 Jul  2 17:35 LICENSE
drwxrwxr-x  2 diego diego  4096 Jul  2 17:35 project/
drwxrwxr-x  5 diego diego  4096 Jul  2 17:35 public/
-rw-rw-r--  1 diego diego  1063 Jul  2 17:35 README
drwxrwxr-x  2 diego diego  4096 Jul  2 17:35 test/
diego@4winds:/tmp/ng-microservice$
```

Let's create the routes on `ng-microservice`. We won't have any UI here, since this will be a microservice. We need to add a route at `ng-microservice/conf/routes`:

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

    GET /double
controllers.NGServiceEndpoint.double
```

```
    GET /doubles/:n
controllers.NGServiceEndpoint.doubles(n:Int)
```

Now let's define the controller. This is not a regular controller, because this one won't serve UI views. Instead, it will serve JSON to the microservice consumers. Here we will have just one consumer, which will be `ReactiveWebStore`. However, it is possible to have as many consumers as you wish such as other microservices or even mobile applications.

NGServiceEndpoint

For this controller, we just have two routes. The routes are `double` and `doubles`. The first route returns a double from the service, and the second one returns a list of doubles generated in a batch. For the second method, we get a list of doubles and transform that list in JSON using the Play framework utility library called `Json`:

```
class NGServiceEndpoint @Inject()
(service:NGContract) extends
    Controller {
      def double = Action {
        Ok(service.generateDouble.toString())
      }
      def doubles(n:Int) = Action {
        val json =
Json.toJson(service.generateDoubleBatch(n))
        Ok(json)
      }
    }
```

The next step is create `trait` for the microservice. This trait is also the service contract in **Service Oriented Architecture** (**SOA**) terms, that is, the capabilities that the microservice offers.

The `NGContract.scala` file should look something like this:

```
trait NGContract {
       def generateDouble:Double
```

```
    def generateDoubleBatch(n:Int):List[Double]
}
```

Let's look at the Service implementation of this microservice:

```
package services
    import scala.util.Random
    import scala.util.Random.nextDouble
    class NGServiceImpl extends NGContract{
      override def generateDouble:Double = {
        Stream.continually(nextDouble * 1000.0 )
        .take(1)
      }
      override def
generateDoubleBatch(n:Int):List[Double] = {
        require(n >= 1, "Number must be bigger
than 0")
        val nTimes:Option[Int] = Option(n)
        nTimes match {
          case Some(number:Int) =>
          Stream.continually(nextDouble * 1000.0 )
          .take(n)
          .toList
          case None =>
          throw new IllegalArgumentException("You
need provide a valid
          number of doubles you want.")
        }
      }
    }
```

This service implementation does not have any RxScala code. However, it is very functional. We have two methods implemented here. The methods are `generateDouble` and `generateDoubleBatch`, which receive, through parameters, the number of doubles you want it to generate for you. For the first operation (`generateDouble`), we use `Stream.continually` to generate infinite random doubles, then we multiply these numbers by 1,000, and then take just 1 and return it.

The second operation is very similar. However, we have to add some validations to make sure that the numbers of double are present.

There are a couple of ways to do it. One way use the assert method in Scala. The second way is the pattern matcher, which is nice because we don't need to write an `if` statement.

This technique is very common in the Scala community. So we create an option that takes the number, and then we pattern-match it. If there is a number present, the case `Some` method will be triggered, otherwise, the case `None` will be called.

After these validations, we can use `Stream` to generate as many numbers as requested. Before we run the code, we need to define the Guice injections. This file is located in the default package at `ng-microservice/app/`:

```
class Module extends AbstractModule {
      override def configure() = {
        // Use the system clock as the default
implementation of Clock

bind(classOf[Clock]).toInstance(Clock.systemDefaul
tZone)

bind(classOf[NGContract]).to(classOf[NGServiceImpl
]).
        asEagerSingleton()
      }
    }
```

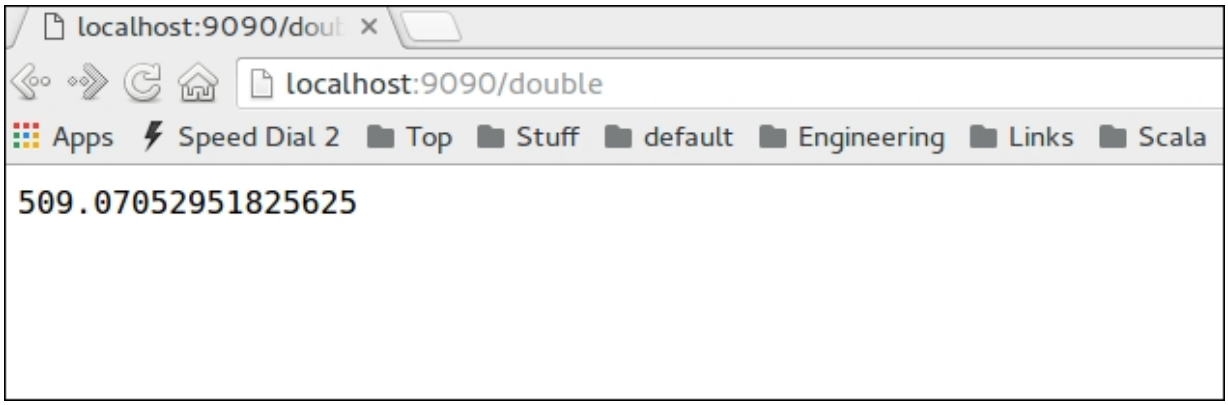
Now its time to compile and run our microservice. Since we already have a play application called `ReactiveWebStore` running on `port 9000`, you will have trouble if you simply run the microservice. To fix this, we need to run it on a different port. Let's use `9090` for the microservice. Open the console, and execute the command `$ activator` followed by `$ run 9090`:

```
ng-microservice$ activator -
Dsbt.task.forcegc=false
[info] Loading global plugins from
/home/diego/.sbt/0.13/plugins
[info] Loading project definition from
/home/diego/github/diegopacheco/Book_Building_Reac
```

```
tive_Functional_Scala_Applications/Chap4/ng-
microservice/project
[info] Set current project to ng-microservice (in
build
file:/home/diego/github/diegopacheco/Book_Building
_Reactive_Functional_Scala_Applications/Chap4/ng-
microservice/)
[ng-microservice] $ run 9090

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

[info] p.c.s.NettyServer - Listening for HTTP on
/0:0:0:0:0:0:0:0:9090

(Server started, use Ctrl+D to stop and go back to
the console...)
```

We can test our microservice by calling the two operations that we have. So let's open the web browser and do it.

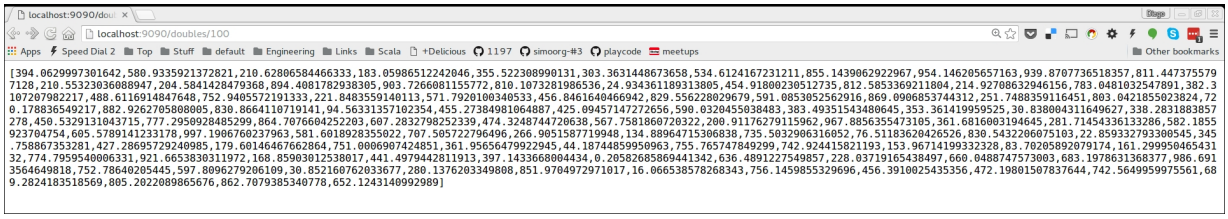
The double microservice call at `http://localhost:9090/double` looks as follows:



Every time you call this operation, you'll see a different random double number. Now we can try out the next operation: the one you

pass for the number of doubles you want. This operation will return a list of doubles in the JSON format.

The call for double in batches microservice at `http://localhost:9090/doubles/100` looks like the following screenshot:



It works! We have 100 doubles here. Now that we have a microservice working, we can go back to our `ReactiveWebStore` and change our RxScala code. We will create new controllers. We will also update the existing code for products to call our new code, and suggest a price for the user on the UI, all in a reactive manner. Keep in mind that you need to have `ng-microservice` running; otherwise, `ReactiveWebStore` won't be able to retrieve doubles.

Play framework and high CPU usage

If you notice your CPU usage going higher than it should, do not worry; there is a fix for it. Actually, the issue is related to SBT. Just make sure that, when you run Activator, you pass the following parameter:

```
$ activator -Dsbt.task.forcegc=false.
```

Going back to `ReactiveWebStore`, let's create new routes. Open `ReactiveWebStore/conf/routes`:

```
GET   /rnd/double

controllers.RndDoubleGeneratorController.rndDouble
    GET   /rnd/call
controllers.RndDoubleGeneratorController.rndCall
    GET   /rnd/rx
controllers.RndDoubleGeneratorController.rxCall
    GET   /rnd/rxbat

controllers.RndDoubleGeneratorController.rxScalaCa
llBatch
```

Once we have the new routes, we need to create the new controller. This controller needs to be located with the other controllers at `ReactiveWebStore/app/controllers`.

RndDoubleGeneratorController

The `RndDoubleGeneratorController` class file should look like this:

```
@Singleton
    class RndDoubleGeneratorController @Inject()
(service:IRndService)
    extends Controller {
```

```
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
      def rndDouble = Action { implicit request =>
        Ok( service.next().toString() )
      }
      def rndCall = Action.async { implicit
request =>
        service.call().map { res => Ok(res) }
      }
      def rxCall = Action { implicit request =>

Ok(service.rxScalaCall().toBlocking.first.toString
())
      }
      def rxScalaCallBatch = Action { implicit
request =>

Ok(service.rxScalaCallBatch().toBlocking.first.toS
tring())
      }
    }
```

All methods in the preceding controller call the service `IRndService`. All operations in `RndService` call `ng-microservice`. Here we have some flavors of operations, which will be covered in great detail next when we explore the service implementation.

There are some interesting things here: for instance, for the second operation called `rndCall`, we see `Action.async` in use, which means our controller will return a Future, and this Future comes from the service. We also execute a `Map` to transform the result into an `Ok`.

The last operation called `rxScalaCallBatch` is the most interesting, and the one we will be using for our UI. However, you can use the other one if you wish, since they all return doubles, and that's good.

IRndService.scala - Scala trait

Let's look at the service definition. First of all, we need to define a trait for the service that will define the operations we need:

```
trait IRndService {
      def next():Double
      def call():Future[String]
      def rxScalaCall():Observable[Double]
      def rxScalaCallBatch():Observable[Double]
    }
```

RndService.scala - RndService implementation

Now we can move to the real service implementation. This needs to be located at `ReactiveWebStore/app/services`:

```
@Singleton
    class RndService @Inject() (ws: WSClient)
extends IRndService {
      import
play.api.libs.concurrent.Execution.Implicits.
      defaultContext
      override def next():Double = {
        val future =
ws.url("http://localhost:9090/double").get().map
        { res => res.body.toDouble }
        Await.result(future, 5.seconds)
      }
      override def call():Future[String] = {

ws.url("http://localhost:9090/double").get().map
        { res => res.body }
      }
      override def
rxScalaCall():Observable[Double] = {
        val doubleFuture:Future[Double] =

ws.url("http://localhost:9090/double").get().map {
x =>
          x.body.toDouble }
        Observable.from(doubleFuture)
      }
     // Continue ...
```

In order to call our microservice (`ng-microservice`), we need to inject a special Play framework library called `ws`, a utility library to call web

services. We inject it by adding the code (`ws:WSClient`) into the class definition.

When you call something with `ws`, it returns a Future. We need to have the Future executors in place. That's why the import `defaultContext` is very important, and you cannot skip it.

For the method, as you can see, we next call our microservice at `http://localhost:9090/double` to get a single double. We map this result, and get the body of the result, which will be the double itself.

For this method, we use `Await.result`, which will block and wait for the result. If the result is not back in five seconds, this code will fail.

The second method called `call` does the same, but the main difference is that we are not blocking the service; instead, we are returning a Future to the controller.

Finally, the last method called `rxScalaCall` does the same: it calls our microservice using the `ws` library. However, we return an observable. Observables are great because they can used as a Future.

Now it is time to go check out the final operation and the most interesting one. For this same class, we need to add another method such as this one:

The method `rxScalaCallBatch` in `RndService.scala` is as follows:

```
override def rxScalaCallBatch():Observable[Double]
= {
      val doubleInfiniteStreamSubject =
PublishSubject.apply[Double]()
      val future =
ws.url("http://localhost:9090/doubles/10")
      .get()
      .map { x =>
Json.parse(x.body).as[List[Double]] }
      future.onComplete {
        case Success(l:List[Double]) => l.foreach
```

```
{ e =>
        doubleInfiniteStreamSubject.onNext(e) }
        case Failure(e:Exception) =>
        doubleInfiniteStreamSubject.onError(e)
      }
      var observableEven = Observable.create {
        doubleInfiniteStreamSubject.subscribe }
      .onErrorReturn { x => 2.0 }
      .flatMap { x => Observable.from(
Iterable.fill(1)(x + 10) ) }
      .filter { x => x.toInt % 2 == 0 }
      .flatMap { x => println("ODD: " + x) ;
Observable.just(x) }
      var observableOdd = Observable.create {
        doubleInfiniteStreamSubject.subscribe }
        .onErrorReturn { x => 1.0 }
        .flatMap { x => Observable.from(
Iterable.fill(1)(x + 10) ) }
        .filter { x => x.toInt % 2 != 0 }
        .flatMap { x => println("EVEN: " + x) ;
Observable.just(x) }
        var mergeObservable = Observable
        .empty
        .merge(observableEven)
        .merge(observableOdd)
        .take(10)
        .foldLeft(0.0)(_+_)
        .flatMap { x => Observable.just( x - (x *
0.9) ) }
        mergeObservable
      }
```

So, first we create `PublishSubject` in order to be able to produce data for the observables. Then we make the `ws` call to our microservice. The main difference now is that we call the batches operation and order 10 doubles. This code happens in a future, so it is non-blocking.

We then use the `Map` function to transform the result. The `ng-microservice` function will return JSON, so we need to deserialize this JSON into Scala objects. Finally, we run a pattern matcher in the Future result. If the result is a success, it means everything is good. So, for each double, we publish into the observables using

`PublishSubject`. If the service is down or we have a problem, we publish an error to the observables downstream.

Next we create three observables: one for odd numbers, one for even numbers, and a third one which will merge the other two and do extra computation. The way we did the conversion between Future and Observable is ideal, because it is non-blocking.

Here we have code very similar to what we had before for the Rx controller. The main difference is that we have error handling, because `ng-microservice` might never return, as it may be down or just not working. So we need to start working with fallbacks. Good fallbacks are key to error handling for Reactive applications. Fallbacks should be sort of static; in other words, they should not fail at all.

We provided two fallback methods: one for the `Odd` Observable and the other for the `Even` Observable. These fallbacks are done by setting the method `OnErrorReturn`. So for the even one, the fallback is static and the value is 2, and for the odd one the value is 1. This is great, because even with failure our application continues to work.

You might realize we are not using the take function this time. So will this code run forever? No, because `ng-microservice` just returns 10 doubles. Finally, we merge the observables into a single observable, add all the numbers, get 90% of the value, and return an observable.

Module.scala - Guice Injections

Now hook this new service in Guice. Let's change the Guice `Module.scala` located at `ReactiveWebStore/apps/Module.scala`:

```
class Module extends AbstractModule {
      override def configure() = {

bind(classOf[Clock]).toInstance(Clock.systemDefaul
tZone)

bind(classOf[ApplicationTimer]).asEagerSingleton()
```

```
bind(classOf[IProductService]).to(classOf[ProductS
ervice]).
        asEagerSingleton()

bind(classOf[IReviewService]).to(classOf[ReviewSer
vice]).
        asEagerSingleton()

bind(classOf[IImageService]).to(classOf[ImageServi
ce]).
        asEagerSingleton()

bind(classOf[IPriceSerice]).to(classOf[PriceServic
e]).
        asEagerSingleton()

bind(classOf[IRndService]).to(classOf[RndService])
.
        asEagerSingleton()
      }}
```

Next we need to create a JQuery function in JavaScript to call our new controller. This function needs to be located at `ReactiveWebStore/public/javascripts`.

The following is `price.js`, the JQuery function that calls our controller:

```
/**
     * This functions loads the price in the HTML
component.
    */
    function loadPrice(doc){
      jQuery.get(
"http://localhost:9000/rnd/rxbat", function(
      response ) {
        doc.getElementById("price").value =
parseFloat(response)
      }).fail(function(e) {
        alert('Wops! We was not able to call
        http://localhost:9000/rnd/rxba. Error: ' +
e.statusText);
```

```
      });
    }
```

We just have a single function called `loadPrice`, which receives a document. We use the `JQuery.get` method to call our controller, and parse the response, adding to the HTML text box called `price`. If something goes wrong, we alert the user that it was not possible to load a price.

main.scala.html

We need to change our `main.scala` code located at `ReactiveWebStore/app/views/main.scala.html` in order to import a new JavaScript function:

```
@(title: String)(content: Html)(implicit flash:
Flash)
    <!DOCTYPE html>
    <html lang="en">
      <head>
        <title>@title</title>
        <link rel="shortcut icon" type="image/png"

href="@routes.Assets.at("images/favicon.png")">
        <link rel="stylesheet" media="screen"

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

href="@routes.Assets.at("stylesheets/bootstrap.min
.css")">
        <script
src="@routes.Assets.at("javascripts/jquery-
        1.9.0.min.js")" type="text/javascript">
</script>
        <script
src="@routes.Assets.at("javascripts/bootstrap.js")
"
        type="text/javascript"></script>
        <script
src="@routes.Assets.at("javascripts/image.js")"
        type="text/javascript"></script>
        <script
src="@routes.Assets.at("javascripts/price.js")"
```

```
        type="text/javascript"></script>
      </head>
      <body>
        <center><a href='/'><img height='42'
width='42'
        src='@routes.Assets.at("images/rws.png")'>
</a>
        <h3>@title</h3></center>
        <div class="container">
          @alert(alertType: String) = {
            @flash.get(alertType).map { message =>
              <div class="alert alert-@alertType">
                <button type="button"
class="close" data-
                  dismiss="alert">&times;</button>
                  @message
              </div>
            }
          }
          @alert("error")
          @alert("success")
          @content
          <a href="/"></a><BR>
          <button type="submit" class="btn btn-
primary"
            onclick="window.location.href='/'; " >
           Reactive Web Store - Home
          </button>
        </div>
      </body>
    </html>
```

product_details.scala.html

Finally, we need to change our product view in order to add a button to load the price from the controller. Let's change the `product_details` view at `ReactiveWebStore/app/views/product_details.scala.html`:

```
@(id: Option[Long],product:Form[Product])(implicit
flash:Flash)
    @import play.api.i18n.Messages.Implicits._
    @import play.api.Play.current
    @main("Product: " +
product("name").value.getOrElse("")){
```

```
      @if(product.hasErrors) {
        <div class="alert alert-error">
          <button type="button" class="close"
data-
          dismiss="alert">&times;</button>
          Sorry! Some information does not look
right. Could you
          review it please and re-submit?
        </div>
      }

      @helper.form(action = if (id.isDefined)
      routes.ProductController.update(id.get) else
      routes.ProductController.insert) {
        @helper.inputText(product("name"), '_label
-> "Product Name")
        @helper.inputText(product("details"),
'_label -> "Product
        Details")
        @helper.inputText(product("price"),
'_label -> "Price")
        <div class="form-actions">
        <button type="button" class="btn btn-
primary"
        onclick="javascript:loadPrice(document);"
>Load Rnd
        Price</button>
        <button type="submit" class="btn btn-
primary">
          @if(id.isDefined) { Update Product }
else { New Product }
        </button>
        </div>
      }
    }
```

Great! Now we have a button that loads the data from the controller using JQuery. You can realize that the button `Load Rnd Price` has an `onClick` property, which calls our JavaScript function.

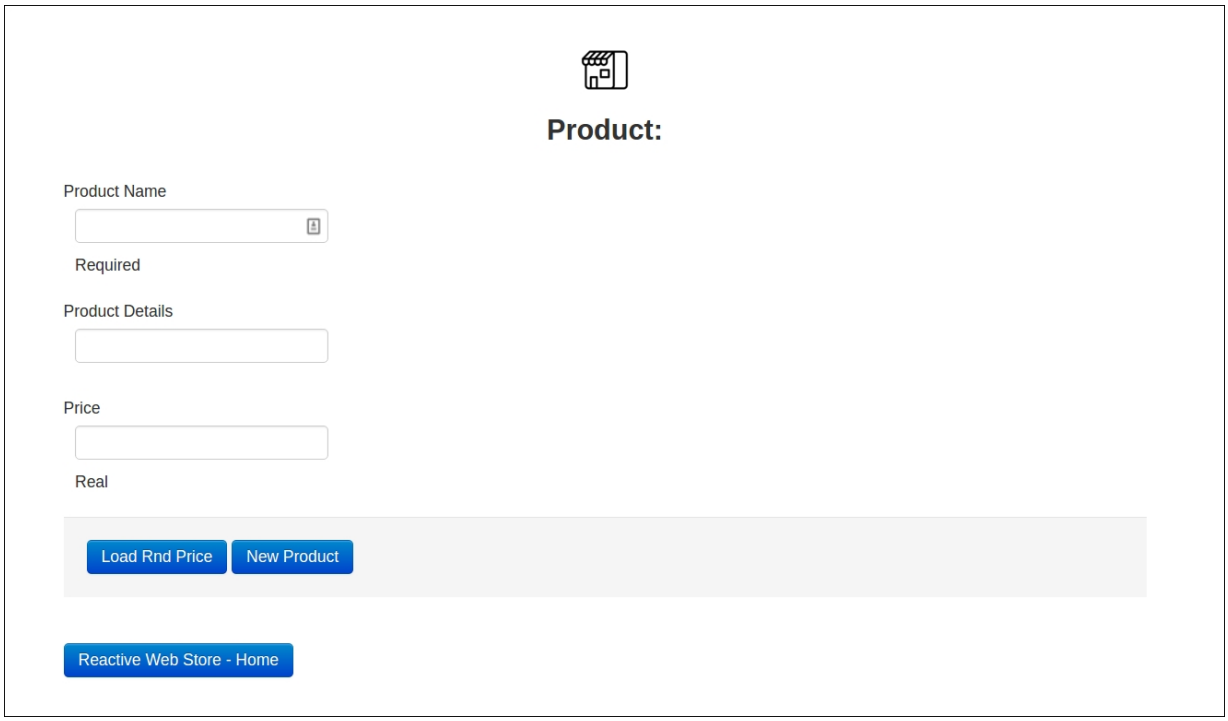
Now you need to open your console and type `$ activator run` to compile and run the changes as we did to `ReactiveWebStore`.

This command will give the following result:

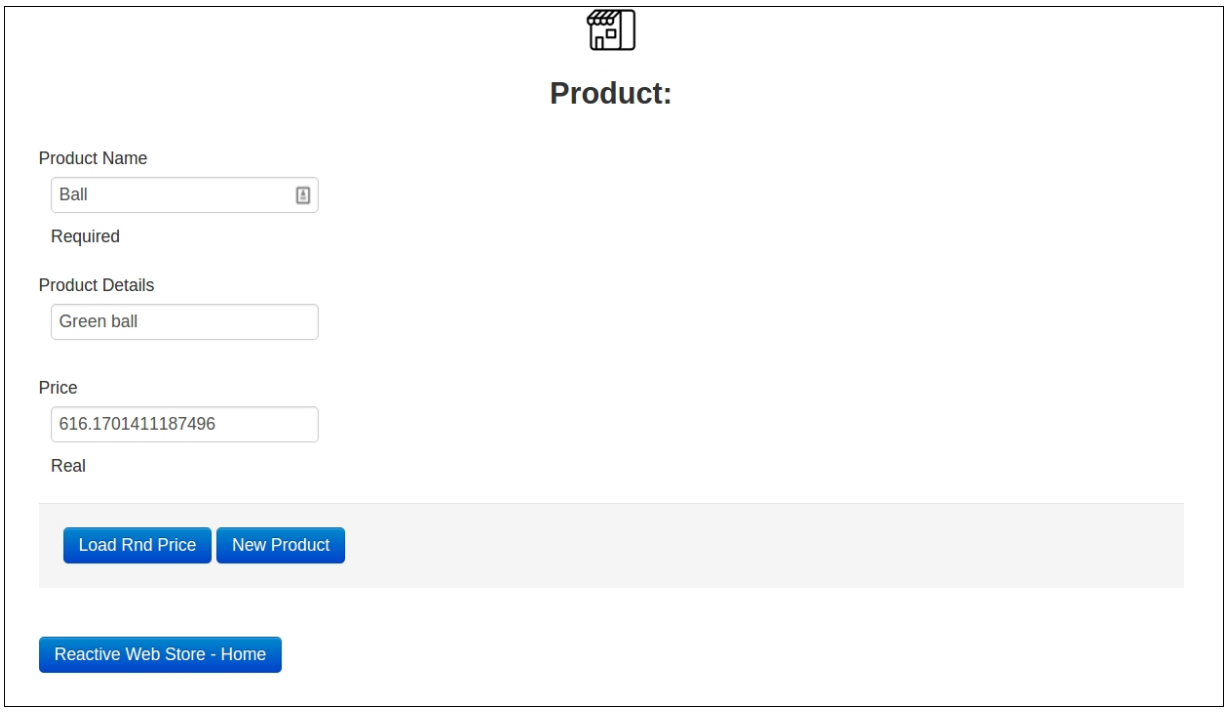
```
ReactiveWebStore$ activator -
Dsbt.task.forcegc=false
[info] Loading global plugins from
/home/diego/.sbt/0.13/plugins
[info] Loading project definition from
/home/diego/github/diegopacheco/Book_Building_Reac
tive_Functional_Scala_Applications/Chap4/ReactiveW
ebStore/project
[info] Set current project to ReactiveWebStore (in
build
file:/home/diego/github/diegopacheco/Book_Building
_Reactive_Functional_Scala_Applications/Chap4/Reac
tiveWebStore/)
[ReactiveWebStore] $ run
--- (Running the application, auto-reloading is
enabled) ---
[info] p.c.s.NettyServer - Listening for HTTP on
/0:0:0:0:0:0:0:0:9000
(Server started, use Ctrl+D to stop and go back to
the console...)
[info] application - ApplicationTimer demo:
Starting application at 2016-07-03T02:35:54.479Z.
[info] play.api.Play - Application started (Dev)
```

So open you browser at `http://localhost:9000`, and go to the product page to see our need feature integrated and working like a charm. Keep in mind that you need to have `ng-microservice` working in another console window; otherwise, our application will use static fallbacks.

The new product feature will be shown at `http://localhost:9000/product/add` as seen in the following screenshot:



So if you click on the **Load Rnd Price** button, you will see something like this:



If you take a look at the application log in the activator console, you will see something similar to this:

```
[info] application - ApplicationTimer demo:
Starting application at 2016-07-03T02:35:54.479Z.
[info] play.api.Play - Application started (Dev)
[info] application - index called. Products:
List()
[info] application - blank called.
ODD: 722.8017048639501
EVEN: 863.8229024202085
ODD: 380.5549208988492
EVEN: 947.6312814830953
ODD: 362.2984794191124
ODD: 676.978825910585
ODD: 752.7412673916701
EVEN: 505.3293481709368
EVEN: 849.9768444508936
EVEN: 99.56583617819769
```

Alright, that's it. We have everything working!

Summary

In this chapter, you learned the core principles of reactive application guided by the Reactive Manifesto. You also learned how to create reactive applications using RxScala. We then explained how to call other internal and external web services using the ws library. Then you learned to serialize and deserialize JSON using the `Json` Library. Moving on, you learned how to create simple microservices using the Play framework support.

In the next chapter, we will continue building our application, and learn how to test our application with JUnit and ScalaTest.

Chapter 5. Testing Your Application

In the chapters so far, we bootstrapped our application using Activator, developed our web application using Scala and the Play framework, and added a Reactive microservice call using RxScala for data flow computations. Now we will go ahead and enter the unit test and controller testing using the BDD and Play framework.

In this chapter, we will cover the following topics:

- Unit testing principles
- Testing Scala applications with JUnit
- Behavior-driven development principles
- Testing with ScalaTest
- Testing Play framework applications with ScalaTest
- Running tests in Activator / SBT

The need for `testingTest` is a fundamental and very important part of software development. Without tests, we cannot be sure that our code works. We should perform tests on almost all the code we produce. There are things that don't make sense for testing, for instance, case classes and classes that just represent structural objects, or, in other words, classes without functions. If you have code that applies computations, transformations, and validations, you definitely want to test this code with a good code coverage, which refers to features or any important code that is not just structural.

Test coverage is important, because it allows us to do refactoring (improve application code quality by reducing code, creating generic code, or even deleting code) with trust. This is because, if you have tests and if you do something wrong by accident, your tests will let you know. This is all about having short cycles of feedback. The earlier the better; you want to know if you have introduced bugs as

soon, and as close to development, as possible. Nobody likes to discover bugs that could be caught by a simple test in production.

Unit testing principles

Unit testing is the smallest unit of testing that you could possibly apply. You need to apply it at the class level. So a unit test will cover your class with all the functions you have there. But hold on a minute, a class often has dependencies, and these dependencies might have other dependencies, so how do you test that? We need to have mocks, simple dumb objects that simulate other classes' behavior. This is an important technique to isolate code and allow unit testing.

Making code testable

Unit testing is simple: basically, we call a function by passing arguments to it, and then we check the output to see if it matches our expectations. This is also called asserts or assertions. So, unit testing is about asserts. Sometimes, your code might not be testable. For instance, let's say you have a function that returns a unit and has no parameters. This is very tough to test, because it implies that the function is full of side-effects. If you remember what we discussed in Chapter 1, *Introduction to FP, Reactive, and Scala*, this is against FP principles. So, if we have this case, we need to refactor the code to make the function return something, and then we can test it.

Isolation and self-contained tests

Unit tests should be self-contained, which means that a unit test's classes should not depend on any particular order of execution. Let's say you have a unit test class with two test functions. So, each test should test just one function at a time, and both functions should be

able to run in any order whatsoever. Otherwise, the tests will be fragile and hard to maintain in the long run.

Effective naming

Effective naming is essential. The test function needs to say exactly what the test does. This is important because, when the test fails, it is easier to figure out what went wrong and why. Following the same idea, when you do assertions, you should assert just one thing at a time. Imagine that you need to test whether a web service returns a valid JSON. This particular JSON could have two fields: first name and last name. So, you will make one assert for the name and an other for the last name. This way, it will be easier to understand what the test does, and to troubleshoot when the test fails.

Levels of testing

When we run tests, we often do it in layers. Unit testing is the basic level; however, there are other levels such as controller tests, integration tests, UI tests, End-to-End tests, stress tests, and so many others. For this book, we will cover unit tests, controller tests, and UI tests using Junit and `ScalaTest`,Play's framework support.

Testing with Junit

If you come from a Java background, it is highly possible that you have already worked with Junit. It's possible to test with Junit using the Scala and Play framework. However, this is not the best practice when we are creating applications with the Play framework, since it favors **Behavior Driven Development** (**BDD**) testing with Scala Spec. For this chapter, we will cover how to perform all sorts of test using BDD and Play. Right now, let's take a look at how we can do unit testing with Junit before we move to BDD.

```
@RunWith(classOf[Suite])

@Suite.SuiteClasses(Array(classOf[JunitSimpleTest]
))
    class JunitSimpleSuiteTest
    class JunitSimpleTest extends PlaySpec with
AssertionsForJUnit {
      @Test def testBaseService() {
        val s = new ProductService
        val result = s.findAll()
        assertEquals(None, result)
        assertTrue( result != null)
        println("All good junit works fine with
ScalaTest and Play")
      }
    }
```

So what we have in the preceding code is a class that extends `PlaySpec`, and adds a trait called `AssertionForJunit`. Why don't we have the classical Junit class here? Because the Play framework is set up to run Scala tests, so this bridge allows us to run Junit by `ScalaTest` Play framework constructs.

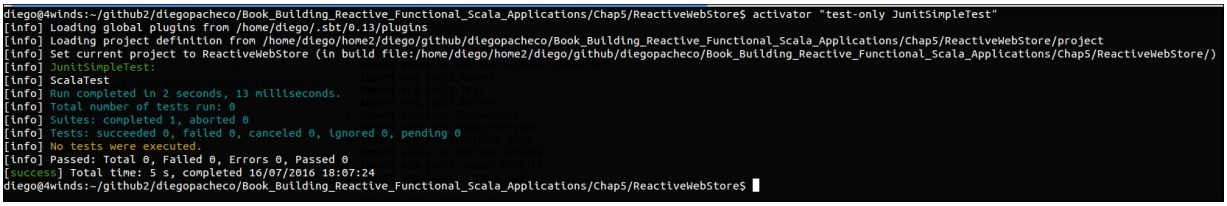
Then we have a test function called `testBaseServer` , which uses the `@Test` annotation from JUnit. Inside the test method, we create an instance of `ProductService`, and then we call the function `findAll`.

Finally, we have assertions that will check if the result is what we are expecting. So we don't have products, because we did not insert them earlier. Hence, we expect to have `None` as the response, and the result should also not be `null`.

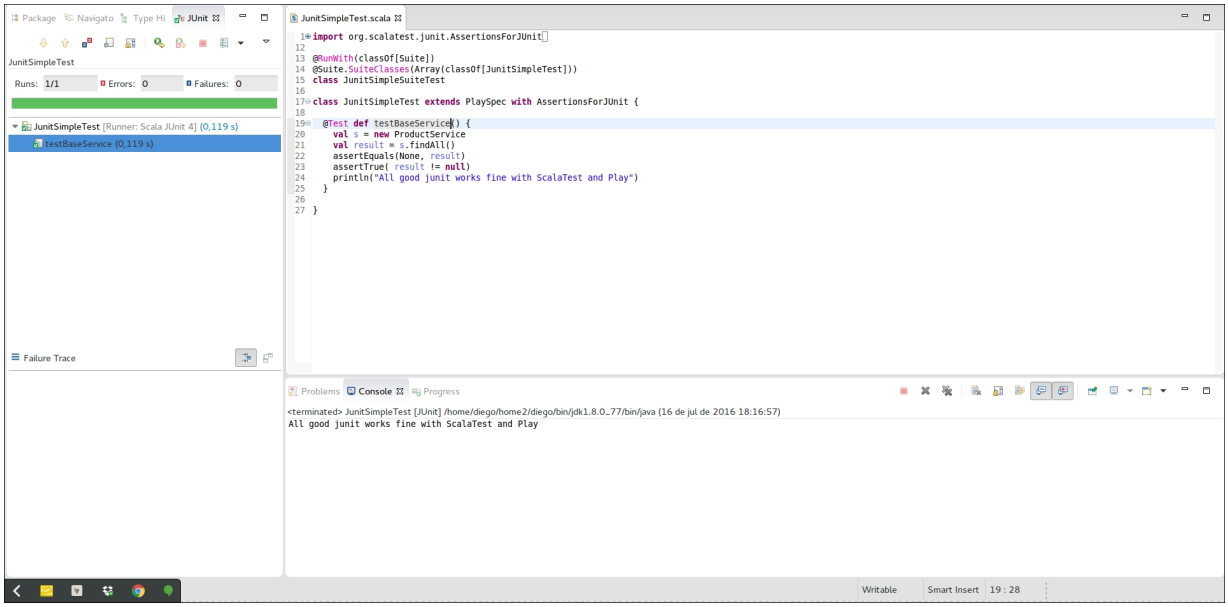
You can run this in your console using the following command:

```
$ activator "test-only JunitSimpleTest"
```

You will see the result shown in the next screenshot:



As you can see, our test was executed without any issues. It's also possible to run this test and a normal test in Junit using the Eclipse IDE. You just right-click on the file and select **Run As: Scala Junit Test;** refer to the following screenshot:


Package
Navigato
Type Hi
JUnit
JunitSimpleTest
Runs: 1/1
Errors: 0
Failures: 0
JunitSimpleTest [Runner: Scala JUnit 4] (0,119 s)
testBaseService (0,119 s)
Failure Trace
JunitSimpleTest.scala
1 import org.scalatest.junit.AssertionsForJUnit
12
13 @RunWith(classOf[Suite])
14 @Suite.SuiteClasses(Array(classOf[JunitSimpleTest]))
15 class JunitSimpleSuiteTest
16
17 class JunitSimpleTest extends PlaySpec with AssertionsForJUnit {
18
19 @Test def testBaseService() {
20 val s = new ProductService
21 val result = s.findAll()
22 assertEquals(None, result)
23 assertTrue(result != null)
24 println("All good junit works fine with ScalaTest and Play")
25 }
26
27 }
Problems
Console
Progress
<terminated> JunitSimpleTest [JUnit] /home/diego/home2/diego/bin/jdk1.8.0_77/bin/java (16 de jul de 2016 18:16:57)
All good junit works fine with ScalaTest and Play
Writable
Smart Insert
19 : 28

Behavior-Driven Development - BDD

Behavior-Driven Development (**BDD**) is an agile development technique, that focuses on the engagement between developers and non-technical people such as product owners from the business. The idea is pretty simple: use the same language as the business uses in order to extract the reason why the code you are building exists in the first place. BDD ends up minimizing the translation between tech language and business language, creating more synergy and less noise between information technology and business.

BDD tests describe what the application needs to do, and how it behaves. It's very common to write down these tests using pair programing between business people and developers. `ScalaTest` is a BDD framework. Play framework has a great integration with `ScalaTest`. Let's get started with `ScalaTest` and Play right now.

MyFirstPlaySpec.scala - First BDD with ScalaTest and the Play framework

The `MyFirstPlaySpec.scala` class should contain the following code:

```
class MyFirstPlaySpec extends PlaySpec {
      "x + 1 " must {
        "be 2 if x=1 " in {
          val sum = 1 + 1
          sum mustBe 2
        }
        "be 0 if x=-1 " in {
          val sum = -1 + 1
          sum mustBe 0
        }
      }
    }
```

So, you create a class called `MyFirstPlaySpec`, and we extend it from `PlaySpec` in order to get Play framework `ScalaTest` support. Then we create two functions to test the sum of two numbers. In the first test, `1 +1` should be `2`, and in the second, `-1 + 1` should be `0`. When we execute `mustBe`, it is the same thing as doing an assert in Junit. The main difference here is that the test has behavior explicitly on the Spec. Now we can run the test by typing the following:

```
$ activator "test-only MyFirstPlaySpec"
```

You will see the following result:

```
diego@4winds:~/github2/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore$ activator "test-only MyFirstPlaySpec"
[info] Loading global plugins from /home/diego/.sbt/0.13/plugins
[info] Loading project definition from /home/diego/home2/diego/github/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore/project
[info] Set current project to ReactiveWebStore (in build file:/home/diego/home2/diego/github/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore/)
[info] Compiling 1 Scala source to /home/diego/home2/diego/github/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore/target/scala-2.11/test-classes...
[info] MyFirstPlaySpec:
[info] x + 1
[info] - must be 2 if x=1
[info] - must be 0 if x=-1
[info] ScalaTest
[info] Run completed in 1 second, 837 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
[success] Total time: 12 s, completed 16/07/2016 18:44:44
diego@4winds:~/github2/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore$
```

Testing with Play framework support

Now we will continue building our application. Let's add BDD tests in our application. We will start doing tests for your services. We have to test `ProductService`, `ImageService`, and `ReviewService`.

Your `ProductServiceTestSpec.scala file` should contain the following code:

```
class ProductServiceTestSpec extends PlaySpec {
      "ProductService" must {
        val service:IProductService = new
ProductService
        "insert a product properly" in {
          val product = new
models.Product(Some(1),
          "Ball","Awesome Basketball",19.75)
          service.insert(product)
        }
        "update a product" in {
          val product = new
models.Product(Some(1),
          "Blue Ball","Awesome Blue
Basketball",19.99)
          service.update(1, product)
        }
        "not update because does not exit" in {
          intercept[RuntimeException]{
            service.update(333,null)
          }
        }
        "find the product 1" in {
           val product = service.findById(1)
           product.get.id mustBe Some(1)
           product.get.name mustBe "Blue Ball"
           product.get.details mustBe "Awesome
Blue Basketball"
           product.get.price mustBe 19.99
        }
        "find all" in {
          val products = service.findAll()
```

```
            products.get.length mustBe 1
            products.get(0).id mustBe Some(1)
            products.get(0).name mustBe "Blue Ball"
            products.get(0).details mustBe "Awesome
Blue Basketball"
            products.get(0).price mustBe 19.99
          }
          "find all products" in {
            val products = service.findAllProducts()
            products.length mustBe 1
            products(0)._1 mustBe "1"
            products(0)._2 mustBe "Blue Ball"
          }
          "remove 1 product" in {
            val product = service.remove(1)
            product mustBe true
            val oldProduct = service.findById(1)
            oldProduct mustBe None
          }
          "not remove because does not exist" in {
            intercept[RuntimeException]{
              service.remove(-1)
            }
          }
        }
      }
```

For this test, we are testing all public functions available in `ProductService`. The tests are pretty straightforward: we call a specific service operation, such as `findById`, and then we check the result to make sure that all the data that is supposed to be there is present.

There are scenarios where the service should return an exception, for instance, if you try to remove something that does not exist. If you take a look at the last test function called `"not remove because does not exist"`, we should get an exception. However, there is a bug in the service code. Run the tests, and then you will see it.

ProductService.scala - FIX the code issue

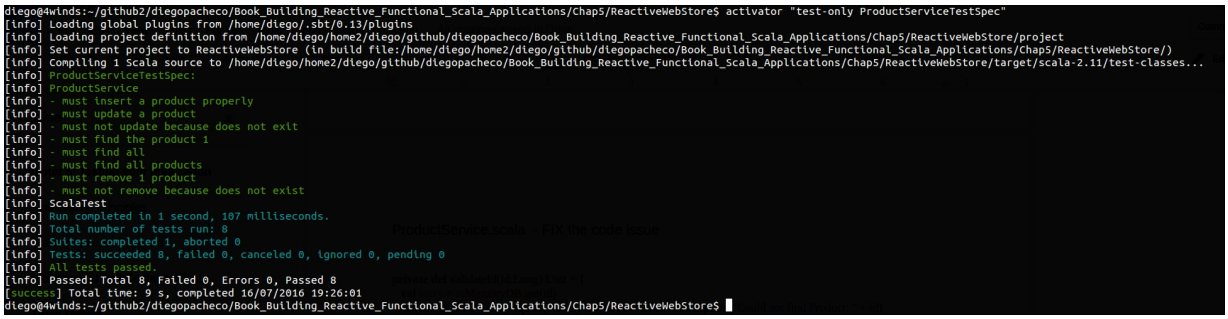
That's the great thing about tests: they show issues in our code so that we can fix them before the code goes to production and affects the user experience. To fix the last test, we need to go to the `ProductService` class and fix a method. This is how we fix it:

```
private def validateId(id:Long):Unit = {
      val entry = inMemoryDB.get(id)
      if (entry==null || entry.equals(None)) throw
new
      RuntimeException("Could not find Product: "
+ id)
    }
```

All set now, everything is okay. The Play framework supports testing for expected exceptions using the intercept function to pass the expected exception, Let's run the test in the console using the `activator` command.

```
$ activator "test-only ProductServiceTestSpec"
```

After executing this command, we get the following:



ImageServiceTestSpec.scala - ImageService Test

Alright, Now we can add tests for `ImageService` as follows:

```
class ImageServiceTestSpec extends PlaySpec {
      "ImageService" must {
        val service:IImageService = new
ImageService
        "insert a image properly" in {
          val image = new
models.Image(Some(1),Some(1),
          "http://www.google.com.br/myimage")
          service.insert(image)
        }
        "update a image" in {
          val image = new
models.Image(Some(2),Some(1),
          "http://www.google.com.br/myimage")
          service.update(1, image)
        }
        "not update because does not exist" in {
          intercept[RuntimeException]{
            service.update(333,null)
          }
        }
        "find the image1" in {
          val image = service.findById(1)
          image.get.id mustBe Some(1)
          image.get.productId mustBe Some(1)
          image.get.url mustBe
"http://www.google.com.br/myimage"
        }
        "find all" in {
          val reviews = service.findAll()
          reviews.get.length mustBe 1
          reviews.get(0).id mustBe Some(1)
          reviews.get(0).productId mustBe Some(1)
          reviews.get(0).url mustBe
"http://www.google.com.br/myimage"
        }
        "remove 1 image" in {
          val image = service.remove(1)
          image mustBe true
          val oldImage = service.findById(1)
          oldImage mustBe None
        }
        "not remove because does not exist" in {
          intercept[RuntimeException]{
            service.remove(-1)
          }
        }
```

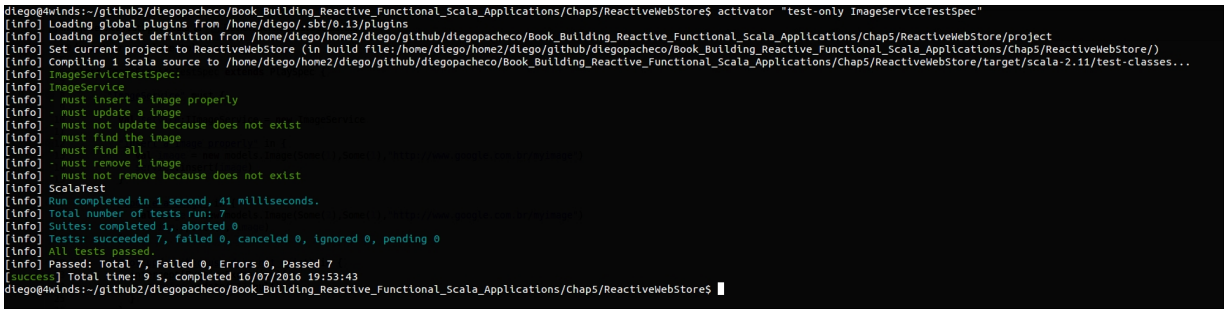
```
    }
  }
```

So these are the BDD tests for `ImageService`. We have covered all the available functions on the service. Like in the `ProductService` class, we also have tests for unfortunate scenarios where we expect exceptions to happen.

Sometimes, we need to call more than one function to test a specific function or a specific test case. For example, in `"remove 1 image"`, we first delete an image. Our test case checks for an image that does not exist. Let's run the tests on the Activator console.

```
$ activator "test-only ImageServiceTestSpec"
```

The following result will be obtained:



ReviewServiceTestSpec.scala - ReviewService test

Next, we need add tests for the review service. Here we go.

```
class ReviewServiceTestSpec extends PlaySpec {
      "ReviewService" must {
        val service:IReviewService = new
ReviewService
```

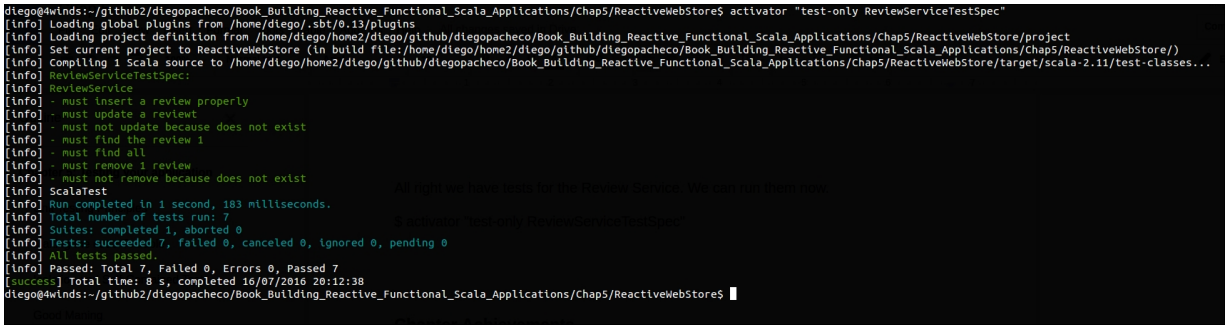
```
        "insert a review properly" in {
          val review = new
models.Review(Some(1),Some(1),"diegopacheco",
          "Testing is Cool")
          service.insert(review)
        }
        "update a review" in {
          val review = new
models.Review(Some(1),Some(1),
          "diegopacheco","Testing so so Cool")
          service.update(1, review)
        }
        "not update because does not exist" in {
          intercept[RuntimeException]{
            service.update(333,null)
          }
        }
        "find the review 1" in {
          val review = service.findById(1)
          review.get.id mustBe Some(1)
          review.get.author mustBe "diegopacheco"
          review.get.comment mustBe "Testing so so
Cool"
          review.get.productId mustBe Some(1)
        }
        "find all" in {
          val reviews = service.findAll()
          reviews.get.length mustBe 1
          reviews.get(0).id mustBe Some(1)
          reviews.get(0).author mustBe
"diegopacheco"
          reviews.get(0).comment mustBe "Testing
so so Cool"
          reviews.get(0).productId mustBe Some(1)
        }
        "remove 1 review" in {
          val review = service.remove(1)
          review mustBe true
          val oldReview= service.findById(1)
          oldReview mustBe None
        }
        "not remove because does not exist" in {
          intercept[RuntimeException]{
            service.remove(-1)
          }
        }
```

```
    }
  }
```

Alright, we have tests for the review service. We can run them now.

```
$ activator "test-only ReviewServiceTestSpec"
```

Here is the output:



Testing routes

The Play framework allows us to test routes as well. This is good, because as our application grows and we refactor the code, we can be 100% sure that our routes are functioning. Route testing could be easily confused with controller testing. The main difference is that, with routing testing, we should test if we are able to reach the routes and that's it. After route testing, we will cover controller testing in detail.

RoutesTestingSpec.scala - Play framework route testing

Your `RoutesTestingSpec.scala` file should contain the following code:
class RoutesTestingSpec extends PlaySpec with OneAppPerTest {

"Root Controller" should {

"route to index page" in {

val result = route(app, FakeRequest(GET, "/")).get status(result) mustBe OK

contentType(result) mustBe Some("text/html") contentAsString(result) must include ("Welcome to Reactive Web Store")

}

}

"Product Controller" should {

"route to index page" in {

```
val result = route(app, FakeRequest(GET, "/product")).get
status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("Product") }

"route to new product page" in {

val result = route(app, FakeRequest(GET, "/product/add")).get

status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("Product") }

"route to product 1 details page page" in {

try{

route(app, FakeRequest(GET, "/product/details/1")).get }catch{

case e:Exception => Unit }

}

}

"Review Controller" should {

"route to index page" in {

val result = route(app, FakeRequest(GET, "/review")).get
status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("Review") }

"route to new review page" in {
```

```
val result = route(app, FakeRequest(GET, "/review/add")).get

status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("review") }

"route to review 1 details page page" in {

try{

route(app, FakeRequest(GET, "/review/details/1")).get }catch{

case e:Exception => Unit }

}

}

"Image Controller" should {

"route to index page" in {

val result = route(app, FakeRequest(GET, "/image")).get
status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("Image") }

"route to new image page" in {

val result = route(app, FakeRequest (GET, "/image/add")).get
status(result) mustBe OK

contentType(result) mustBe Some("text/html")
contentAsString(result) must include ("image") }

"route to image 1 details page page" in {
```

```
try{

route(app, FakeRequest(GET, "/image/details/1")).get }catch{

case e:Exception => Unit }

}

}

}
```

So here we have tests for all our main controllers, which are root, product, review, and image. `RootController` is the controller of the main page when you visit `http://localhost:9000`.

There is a special helper function called `route` in the Play framework, which helps us to test routes. Then we use `FakeRequest` pass the path to the route. It's possible to test the status code and content type of the page to which the router routed our request.

For product, image, and review controllers, you can see we are trying to call an item that does not exist. That's why we have the try...catch, because we expect to have an exception happening there.

```
$  activator "test-only RoutesTestingSpec"
```

Executing this preceding command produces the following result:

```
[info] RoutesTestingSpec:
[info] Root Controller
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:30.346Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:32.122Z after 2s.
[info] - should route to index page
[info] Product Controller
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:32.566Z.
[info] application - index called. Products: List()
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:32.680Z after 0s.
[info] - should route to index page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:35.782Z.
[info] application - blank called.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:35.986Z after 0s.
[info] - should route to new product page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:36.367Z.
[info] application - details called. id: 1
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:36.458Z after 0s.
[info] - should route to product 1 details page page
[info] Review Controller
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:37.972Z.
[info] application - index called. Reviews: List()
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:38.105Z after 1s.
[DEBUG] [07/16/2016 20:34:38.181] [application-akka.actor.default-dispatcher-4] [EventStream] shutting down: StandardOutLogger started
[info] - should route to index page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:38.707Z.
[info] application - blank called.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:38.975Z after 0s.
[info] - should route to new review page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:39.791Z.
[info] application - details called. id: 1
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:39.923Z after 0s.
[DEBUG] [07/16/2016 20:34:39.972] [application-akka.actor.default-dispatcher-6] [EventStream] shutting down: StandardOutLogger started
[info] - should route to review 1 details page page
[info] Image Controller
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:41.403Z.
[info] application - index called. Images: List()
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:41.464Z after 0s.
[info] - should route to index page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:42.175Z.
[info] application - blank called.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:42.277Z after 0s.
[info] - should route to new image page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T03:34:43.075Z.
[info] application - details called. id: 1
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T03:34:43.145Z after 0s.
[info] - should route to image 1 details page page
[info] ScalaTest
[info] Run completed in 21 seconds, 865 milliseconds.
[info] Total number of tests run: 10
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 10, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Controller testing

We did unit tests, we did route tests, and now is the time to add controller tests. Controller tests are similar to routes tests, but they are not the same. For instance, our controller always respond to UI pages, so we expected to create specific HTML pages based on each method. The Play framework has integration with Selenium, which is a testing framework for UIs, and a controllers that allows you to simulate web browsers, and you can do pretty much the same as you would by clicking on the pages like a real user.

So let's get started. First, we will start with `RndDoubleGeneratorControllerTestSpec`.

RndDoubleGeneratorControllerTestSpec.scala - RndDoubleGeneratorController tests

The `RndDoubleGeneratorControllerTestSpec.scala` file should contain the following code:

```
class RndDoubleGeneratorControllerTestSpec
    extends PlaySpec
    with OneServerPerSuite with OneBrowserPerSuite
with HtmlUnitFactory
    {
      val injector = new GuiceApplicationBuilder()
      .injector
      val ws:WSClient =
injector.instanceOf(classOf[WSClient])
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
      "Assuming ng-microservice is down rx number
should be" must {
        "work" in {
          val future =
ws.url(s"http://localhost:${port}/rnd/rxbat")
```

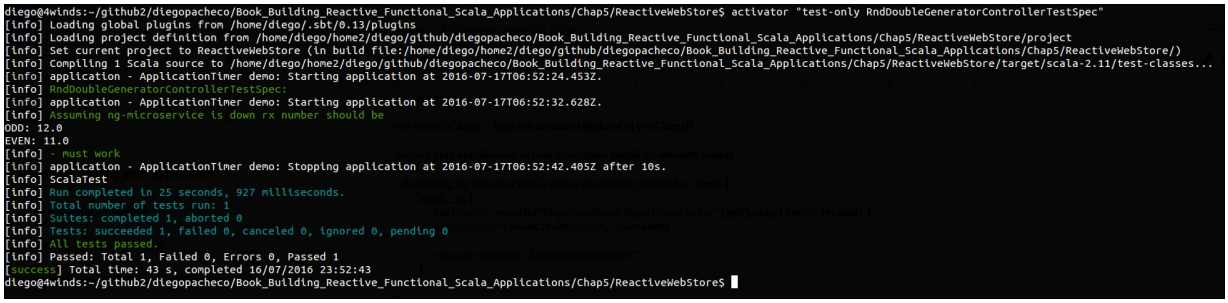
```
            .get().map { res => res.body }
            val response = Await.result(future,
15.seconds)
            response mustBe "2.3000000000000007"
          }
        }
    }
```

This class has some interesting things. For instance, we inject `WSClient` using Google Guice with `GuiceApplicationBuilder`. Secondly, we assume that the `ng-microservice` we created in the previous chapter is down, so we can predict the response coming from the fallback.

We call the controller using `WSClient`, and then we map the response to return the body content as a string. So this will be an Async Future, and in order to get the result, we use `Await` to wait five seconds for the response to come back. Once the response is back, we make sure the result is 2.3. If the result does not come back in 15s, the test will fail. Run the following command:

```
$ activator "test-only
RndDoubleGeneratorControllerTestSpec"
```

Now you will see the following result:



All right, now we have a controller test fully working using Guice injections and the `WSClient` Play framework library. Let's now make controller tests for the product, image, and review controllers.

IntegrationSpec.scala

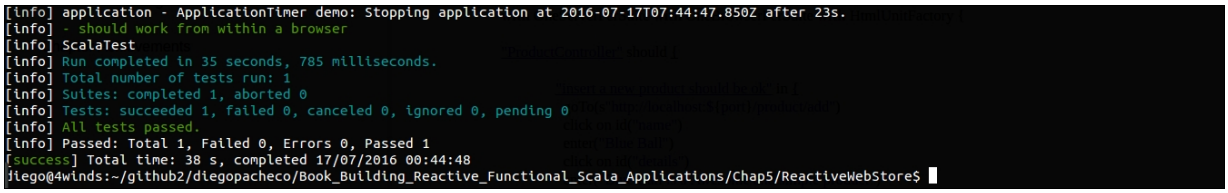
We can test our main page to check if it is okay. This is a very simple test, and gets you ready for the next tests. So, here we go.

```
class IntegrationSpec extends PlaySpec with
OneServerPerTest with
    OneBrowserPerTest with HtmlUnitFactory {

      "Application" should {
        "work from within a browser" in {
           go to ("http://localhost:" + port)
           pageSource must include ("Welcome to
Reactive Web Store")
        }
      }
    }
```

This test is very easy. We just call the main page, and we check if it contains the text `Welcome to Reactive WebStore`. Let's run it.

```
$ activator "test-only IntegrationSpec"
```

The result after running this test is shown in the following image:



ProductControllerTestSpec.scala

Now we will look at the product controller test spec:

```
class ProductControllerTestSpec extends PlaySpec
with
    OneServerPerSuite with OneBrowserPerSuite with
HtmlUnitFactory {
      "ProductController" should {
        "insert a new product should be ok" in {

goTo(s"http://localhost:${port}/product/add")
          click on id("name")
          enter("Blue Ball")
          click on id("details")
          enter("Blue Ball is a Awesome and simple
product")
          click on id("price")
          enter("17.55")
          submit()
        }
        "details from the product 1 should be ok"
in {

goTo(s"http://localhost:${port}/product/details/1"
)
          textField("name").value mustBe "Blue
Ball"
          textField("details").value mustBe "Blue
Ball is a Awesome
          and simple product"
          textField("price").value mustBe "17.55"
        }
        "update product 1 should be ok" in {

goTo(s"http://localhost:${port}/product/details/1"
)
          textField("name").value = "Blue Ball 2"
          textField("details").value = "Blue Ball
is a Awesome and
          simple product 2 "
          textField("price").value = "17.66"
          submit()

goTo(s"http://localhost:${port}/product/details/1"
)
          textField("name").value mustBe "Blue
Ball 2"
          textField("details").value mustBe "Blue
```

```
Ball is a Awesome
          and simple product 2 "
          textField("price").value mustBe "17.66"
        }
        "delete a product should be ok" in {

goTo(s"http://localhost:${port}/product/add")
          click on id("name")
          enter("Blue Ball")
          click on id("details")
          enter("Blue Ball is a Awesome and simple
product")
          click on id("price")
          enter("17.55")
          submit()

goTo(s"http://localhost:${port}/product")
          click on id("btnDelete")
        }
      }
    }
```

So, for the product controller, we simulate a web browser using Selenium Play's framework support. We test basic controller functionality such as inserting a new product, details for a specific product, and updating and removing a product.

For insert, we go to the new product form using `goTo`. We use `$port` as a variable. We do this because the Play framework will boot up the application for us, but we don't know in which port. So we need to use this variable in order to get to the product controller.

Then we click on each text field using the `click` function, and we enter values using the `enter` function. After filling in the whole form, we submit it using the `submit` function.

For details, we just go to the product details page, and we check if the text fields have the values that we are expecting. We do that using the `textField.value` function.

In order to check the product update function, we need to first update the product definition, and then go to details to see if the values we

changed are there.

Finally, we test the `delete` function. For this function, we need to click on a button. We need to set the ID of the button in order to have this working. We need to do a small refactoring in our UI to have the ID there.

product_index.scala.html

Your `product_index.scala.html` file should contain the following code:

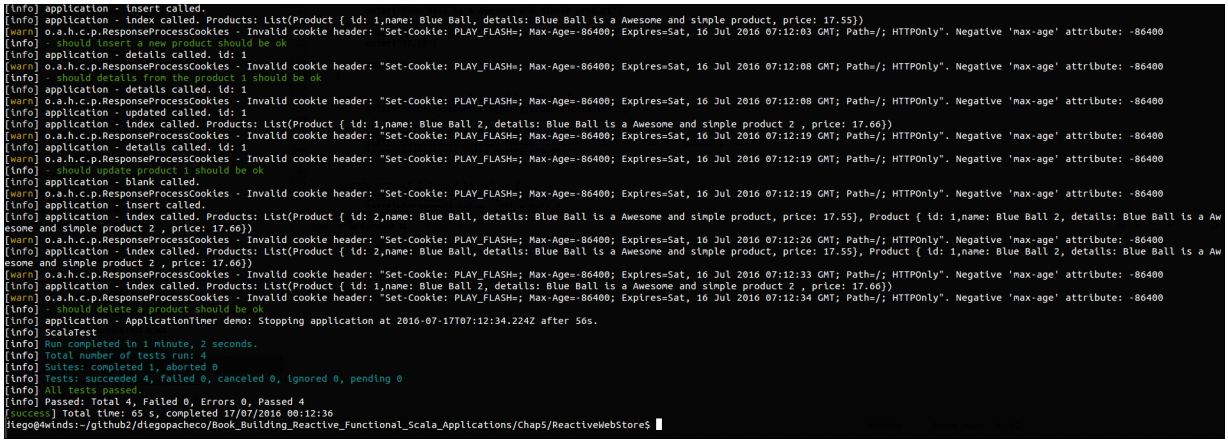
```
@(products:Seq[Product])(implicit flash: Flash)
    @main("Products") {
      @if(!products.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>Name</th>
            <th>Details</th>
            <th>Price</th>
            <th></th>
          </tr>
          @for(product <- products) {
            <tr>
              <td><a
href="@routes.ProductController.details
              (product.id.get)">@product.name</a>
</td>
              <td>@product.details</td>
              <td>@product.price</td>
              <td><form method="post" action=

"@routes.ProductController.remove(product.id.get)"
>
                <button id="btnDelete"
name="btnDelete" class="btn
                btn-link" type="submit"><i
class="icon-
                trash"></i>Delete</button>
              </form></td>
            </tr>
          }
        </table>
      }
```

```
        <p><a href="@routes.ProductController.blank"
        class="btn btn-success"><i class="icon-plus
icon-white">
        </i>Add Product</a></p>
    }
```

All set. Now we can run our tests on Activators using the console.

```
$ activator "test-only ProductControllerTestSpec"
```

This preceding command shows the result in the following screenshot:



Since this test runs the application for real and calls the controller simulating the web browser, this test could take some time. Now it's time to move to the `ImageController` tests.

ImageControllerTestSpec.scala

Your `product_index.scala.html` should contain the following code:

```
class ImageControllerTestSpec extends PlaySpec
with OneServerPerSuite with OneBrowserPerSuite
with
    HtmlUnitFactory {
```

```
      "ImageController" should {
        "insert a new image should be ok" in {

goTo(s"http://localhost:${port}/product/add")
          click on id("name")
          enter("Blue Ball")
          click on id("details")
          enter("Blue Ball is a Awesome and simple
product")
          click on id("price")
          enter("17.55")
          submit()

goTo(s"http://localhost:${port}/image/add")
          singleSel("productId").value = "1"
          click on id("url")
          enter("http://myimage.com/img.jpg")
          submit()
        }
        "details from the image 1 should be ok" in
{

goTo(s"http://localhost:${port}/image/details/1")
          textField("url").value mustBe
"http://myimage.com/img.jpg"
        }
        "update image 1 should be ok" in {

goTo(s"http://localhost:${port}/image/details/1")
          textField("url").value =
"http://myimage.com/img2.jpg"
          submit()

goTo(s"http://localhost:${port}/image/details/1")
          textField("url").value mustBe
"http://myimage.com/img2.jpg"
        }
        "delete a image should be ok" in {

goTo(s"http://localhost:${port}/image/add")
          singleSel("productId").value = "1"
          click on id("url")
          enter("http://myimage.com/img.jpg")
          submit()
          goTo(s"http://localhost:${port}/image")
          click on id("btnDelete")
        }
```

```
    }
  }
```

First of all, we need to go to the product controller to insert a product; otherwise, we cannot do image operations, since they all need a product ID.

image_index.scala.html

Your `image_index.scala.html` file should contain the following code:

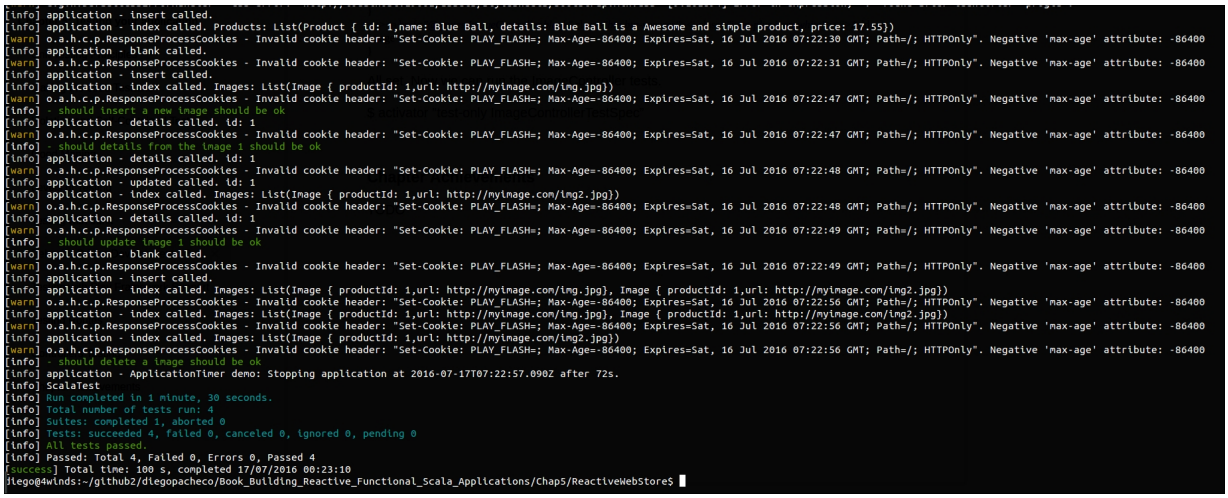
```
@(images:Seq[Image])(implicit flash:Flash)
    @main("Images") {
      @if(!images.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>ProductID</th>
            <th>URL</th>
            <th></th>
          </tr>
          @for(image <- images) {
            <tr>
              <td><a
href="@routes.ImageController.details
              (image.id.get)">@image.id</a></td>
              <td>@image.productId</td>
              <td>@image.url</td>
              <td><form method="post" action=

"@routes.ImageController.remove(image.id.get)">
                <button id="btnDelete"
name="btnDelete" class="btn
                btn-link" type="submit"><i
class="icon-
                trash"></i>Delete</button></form>
              </td>
            </tr>
          }
        </table>
      }
      <p><a href="@routes.ImageController.blank"
      class="btn btn-success"><i class="icon-plus
icon-white">
```

```
      </i>Add Image</a></p>
    }
```

All set. Now we can run the `ImageController` tests.

```
$ activator "test-only ImageControllerTestSpec"
```

The result is shown in the following screenshot:



`ImageController` has passed all its tests. Now we will move to the `ReviewController` tests.

ReviewControllerTestSpec.scala

Your `ReviewControllerTestSpec.scala` file should contain the following code:

```
class ReviewControllerTestSpec extends PlaySpec
with OneServerPerSuite with OneBrowserPerSuite
    with HtmlUnitFactory {
      "ReviewController" should {
        "insert a new review should be ok" in {
```

```
goTo(s"http://localhost:${port}/product/add")
          click on id("name")
          enter("Blue Ball")
          click on id("details")
          enter("Blue Ball is a Awesome and simple
product")
          click on id("price")
          enter("17.55")
          submit()

goTo(s"http://localhost:${port}/review/add")
          singleSel("productId").value = "1"
          click on id("author")
          enter("diegopacheco")
          click on id("comment")
          enter("Tests are amazing!")
          submit()
        }
        "details from the review 1 should be ok"
in {

goTo(s"http://localhost:${port}/review/details/1")
          textField("author").value mustBe
"diegopacheco"
          textField("comment").value mustBe "Tests
are amazing!"
        }
        "update review 1 should be ok" in {

goTo(s"http://localhost:${port}/review/details/1")
          textField("author").value =
"diegopacheco2"
          textField("comment").value = "Tests are
amazing 2!"
          submit()

goTo(s"http://localhost:${port}/review/details/1")
          textField("author").value mustBe
"diegopacheco2"
          textField("comment").value mustBe "Tests
are amazing 2!"
        }
        "delete a review should be ok" in {

goTo(s"http://localhost:${port}/review/add")
          singleSel("productId").value = "1"
          click on id("author")
          enter("diegopacheco")
```

```
            click on id("comment")
            enter("Tests are amazing!")
            submit()
            goTo(s"http://localhost:${port}/review")
            click on id("btnDelete")
            }
    }
```

First of all, we need to go to the product controller to insert a product; otherwise, we cannot do image operations, since they all need a product ID.

For insert, we go to the new product form using goto. We use `$port` as a variable. We do this because the Play framework will boot up the application for us, but we don't know on which port, so we need to use this variable in order to get to the product controller.

Then we click on each text field using the `click` function, and we enter values using the `enter` function. After filling in the whole form, we submit it using the `submit` function.

For details, we just go to the product details page and check if the text fields have the values that we expect. We do that using the `textField.value` function.

In order to check the product update function, we need to first update the product definition, and then go to the details to see if the values we changed are there.

Finally, we test the `delete` function. For this function, we need to click on a button. We need to set the ID of the button in order to have this working. Then we do a small refactoring in our UI to have the ID there.

review_index.scala.html

Your `review_index.scala.html` file should contain the following code:

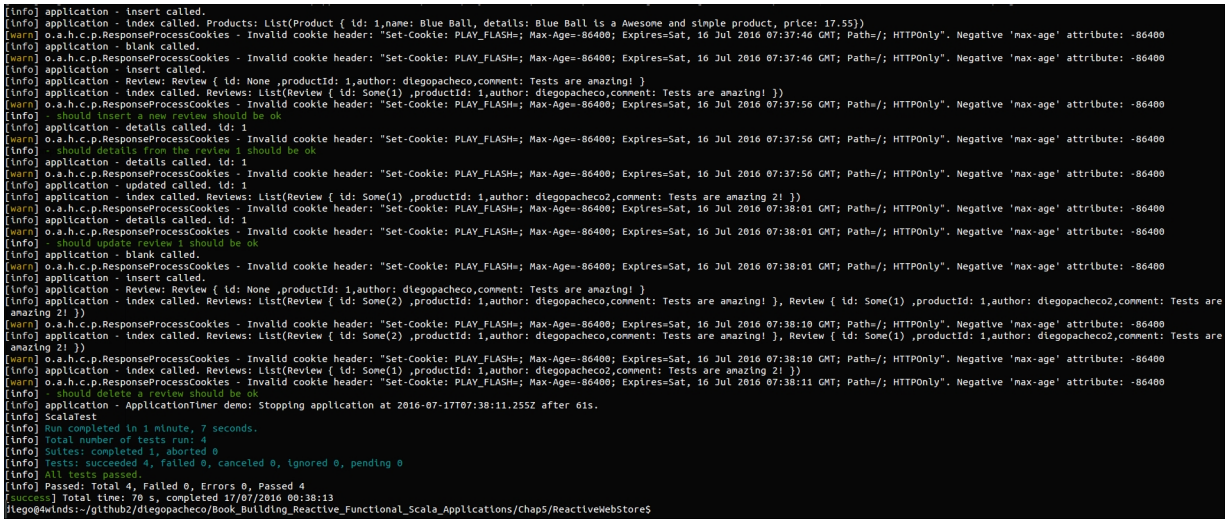
```
@(reviews:Seq[Review])(implicit flash: Flash)
    @main("Reviews") {
```

```
      @if(!reviews.isEmpty) {
        <table class="table table-striped">
          <tr>
            <th>ProductId</th>
            <th>Author</th>
            <th>Comment</th>
            <th></th>
          </tr>
          @for(review <- reviews) {
            <tr>
              <td><a
href="@routes.ReviewController.

details(review.id.get)">@review.productId</a></td>
              <td>@review.author</td>
              <td>@review.comment</td>
              <td>
                <form method="post"
action="@routes.

ReviewController.remove(review.id.get)">
                  <button id="btnDelete"
name="btnDelete"
                  class="btn btn-link"
type="submit"><i class="icon-
                  trash"></i>Delete</button>
                </form>
              </td>
            </tr>
          }
        </table>
      }
      <p><a href="@routes.ReviewController.blank"
class="btn btn-
      success"><i class="icon-plus icon-white">
</i>Add Review</a></p>
    }
```

Finally, we can run the tests on the console.

```
$ activator "test-only ReviewControllerTestSpec"
```

The test will show the following output:



Alright, `ReviewController` has passed all our tests.

It's a very good practice to have the tests separated by type. However, if you want, you could mix all the tests such as unit testing, controller testing, and route testing in one single file.

ApplicationSpec.scala

Your `ApplicationSpec.scala` should have the following code:

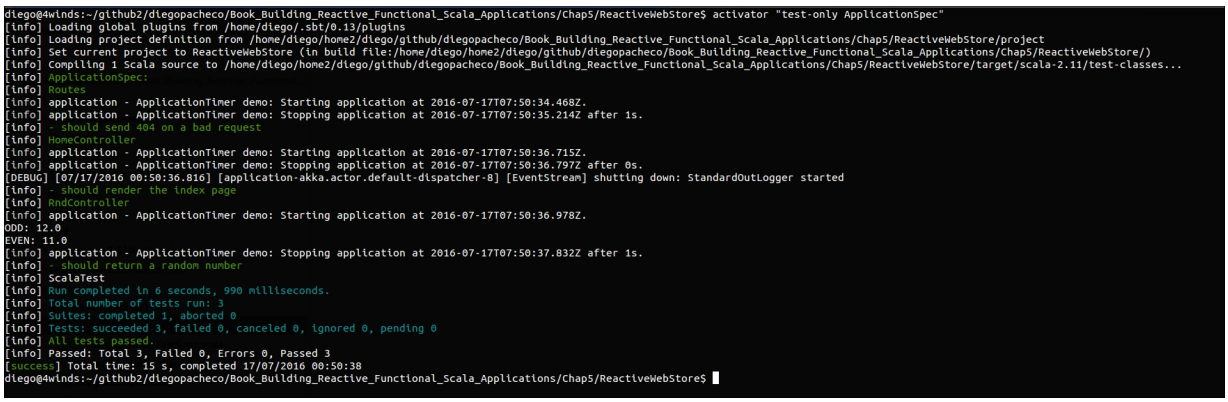
```
class ApplicationSpec extends PlaySpec with
OneAppPerTest {
      "Routes" should {
        "send 404 on a bad request" in {
          route(app, FakeRequest(GET,
"/boum")).map(status(_)) mustBe
          Some(NOT_FOUND)
        }
      }
      "HomeController" should {
        "render the index page" in {
          val home = route(app, FakeRequest(GET,
"/")).get
          status(home) mustBe OK
          contentType(home) mustBe
Some("text/html")
          contentAsString(home) must include
```

```
("Welcome to Reactive Web
          Store")
        }
      }
      "RndController" should {
        "return a random number" in {
          // Assuming ng-microservice is down
otherwise will fail.
          contentAsString(route(app,
FakeRequest(GET,
          "/rnd/rxbat")).get) mustBe
"2.30000000000000007"
        }
      }
    }
```

We can run these mixed tests, and they will all pass.

```
$ activator "test-only ApplicationSpec"
```

You will see this result:



OK, we are almost done. We just need to add some tests for the microservice called `ng-microservice` , which we created in Chapter 4, *Developing Reactive Backing Services*.

NGServiceImplTestSpec.scala

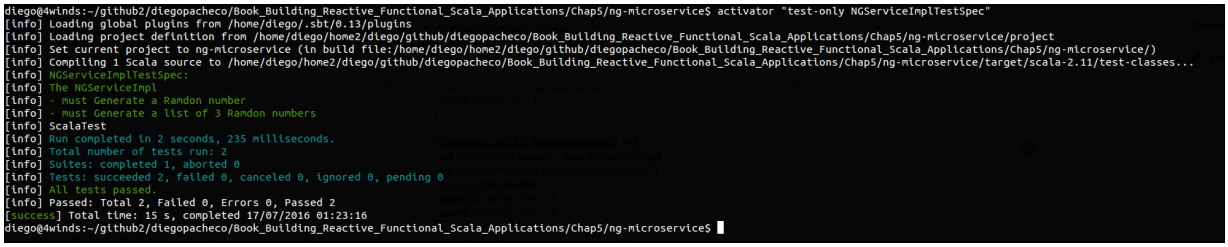
Your NGServiceImplTestSpec.scala file should have the following code:

```
class NGServiceImplTestSpec extends PlaySpec {
      "The NGServiceImpl" must {
        "Generate a Ramdon number" in {
          val service:NGContract = new
NGServiceImpl
          val double = service.generateDouble
          assert( double >= 1 )
        }
        "Generate a list of 3 Ramdon numbers" in {
          val service:NGContract = new
NGServiceImpl
          val doubles =
service.generateDoubleBatch(3)
          doubles.size mustBe 3
          assert( doubles(0) >= 1 )
          assert( doubles(1) >= 1 )
          assert( doubles(2) >= 1 )
        }
      }
    }
```

So here, in the preceding code, we have two methods to test the two operations that we have in our microservice. First we generate one double, and then we ask for a list of three doubles. As you can see, we just check if we get a positive double back from the service, and that's it. Since the result is not predictable, this is a good way to test it. Sometimes, even when the result is predictable, you want tests like this. Why? Because it makes the tests more reliable, and often, when we use too many hardcore values. The values could be changing and breaking our tests, and that's not cool. Let's run it on the console.

```
$  activator "test-only NGServiceImplTestSpec"
```

Here is the result that we get:



Now let's move on to the controller, and do some controller testing.

NGServiceEndpointControllerTest.scala

Your `NGServiceEndpointControllerTest.scala` file should contain the following code:

```
class NGServiceEndpointControllerTest extends
PlaySpec with OneServerPerSuite with
    OneBrowserPerSuite with HtmlUnitFactory {
      val injector = new GuiceApplicationBuilder()
      .injector
      val ws:WSClient =
injector.instanceOf(classOf[WSClient])
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
      "NGServiceEndpointController" must {
        "return a single double" in {
          val future =
ws.url(s"http://localhost:${port}/double")
          .get().map { res => res.body }
          val response = Await.result(future,
15.seconds)
          response must not be empty
          assert( new java.lang.Double(response)
>= 1 )
        }
        "return a list of 3 doubles" in {
          val future =
ws.url(s"http://localhost:${port}/doubles/3")
          .get().map { res => res.body }
          val response = Await.result(future,
15.seconds)
```

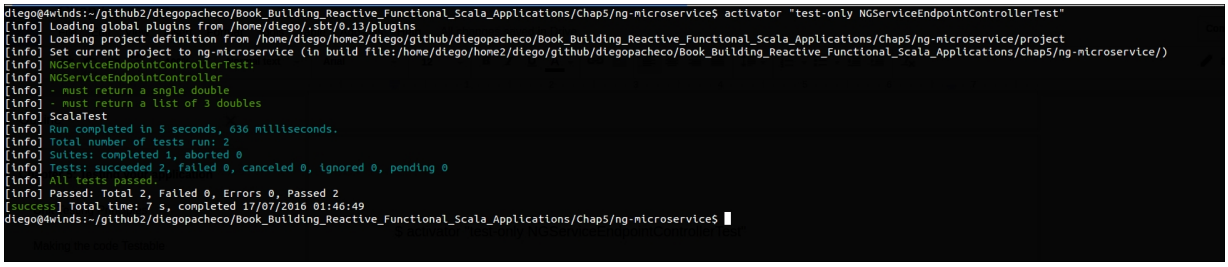
```
            response must (not be empty and include
("[") and
            include ("]"))
          }
        }
      }
```

Here we have to inject the `WSClient` library so we can call the controller. This controller has two methods like the service we tested before. The second method returns a JSON structure. Then we check for `"["and "]"` to make sure that the array is present, since this is a list of three numbers.

We use the assert function to check the response from the controller, and to be 100% sure that everything is okay. Let's run it.

```
$ activator "test-only
NGServiceEndpointControllerTest"
```

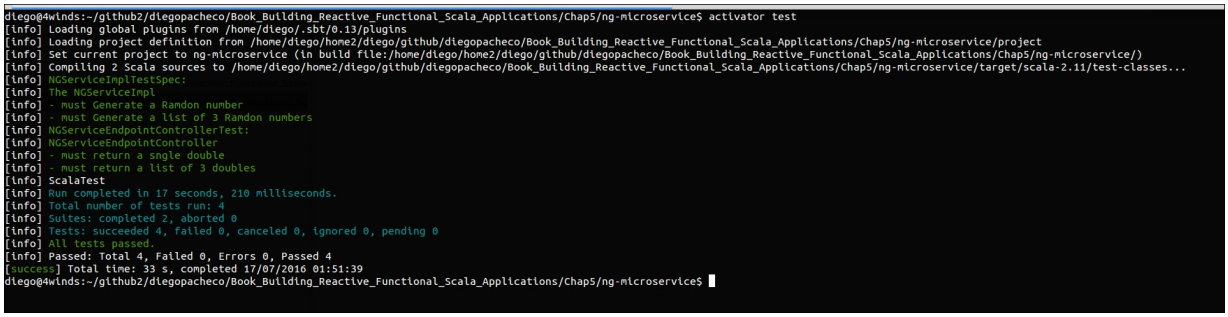
Refer to the following screenshot to see the test result:



Great! We have covered pretty much everything .

In this chapter, we ran all kinds of tests. We always used the command `$ activator "test-only xxx"`; the reason for this is to save time. However, it is very common to run all tests. You can do that in both projects; we have to just type `$ activator test`.

When running all the tests in the `ng-microservice` project, we get the result shown in the following screenshot:



On the other hand, running all the tests in the `ReactiveWebStore` project gives the result shown in the next screenshot:

```
[info] application - details called. id: 1
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:55:56.304Z after 0s.
[info] - should route to review 1 details page page
[info] Image Controller
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:55:57.022Z.
[info] application - index called. Images: List()
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:55:57.079Z after 0s.
[info] - should route to index page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:55:58.601Z.
[info] application - blank called.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:55:58.658Z after 0s.
[info] - should route to new image page
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:55:59.294Z.
[info] application - details called. id: 1
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:55:59.372Z after 0s.
[info] - should route to image 1 details page page
[info] ApplicationSpec:
[info] Routes
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:56:00.066Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:56:00.233Z after 0s.
[info] - should send 404 on a bad request
[info] HomeController
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:56:01.116Z.
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:56:01.204Z after 0s.
[DEBUG] [07/17/2016 01:56:01.364] [application-akka.actor.default-dispatcher-6] [EventStream] shutting down: StandardOutLogger started
[info] - should render the index page
[info] RndController
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:56:01.991Z.
ODD: 12.0
EVEN: 11.0
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:56:02.593Z after 1s.
[DEBUG] [07/17/2016 01:56:02.740] [application-akka.actor.default-dispatcher-2] [EventStream] shutting down: StandardOutLogger started
[info] - should return a random number
[info] JunitSimpleTest:
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:56:05.572Z.
[info] RndDoubleGeneratorControllerTestSpec:
[info] application - ApplicationTimer demo: Starting application at 2016-07-17T08:56:06.148Z.
[info] Assuming ng-microservice is down rx number should be
ODD: 12.0
EVEN: 11.0
[info] - must work
[info] application - ApplicationTimer demo: Stopping application at 2016-07-17T08:56:06.906Z after 0s.
[DEBUG] [07/17/2016 01:56:06.939] [application-akka.actor.default-dispatcher-3] [EventStream] shutting down: StandardOutLogger started
All good junit works fine with ScalaTest and Play
[info] ScalaTest
[info] Run completed in 3 minutes, 3 seconds.
[info] Total number of tests run: 51
[info] Suites: completed 12, aborted 0
[info] Tests: succeeded 51, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 52, Failed 0, Errors 0, Passed 52
[success] Total time: 189 s, completed 17/07/2016 01:56:09
diego@4winds:~/github2/diegopacheco/Book_Building_Reactive_Functional_Scala_Applications/Chap5/ReactiveWebStore$
```

Summary

In this chapter, you learned how to do tests. We added several tests for your Scala and Play framework projects. You also learned about unit testing principles, testing Scala applications with JUnit, BDD principles, testing with `ScalaTest`, testing Play framework applications with `ScalaTest`, and running tests in Activator / SBT.

In the next chapter, you will learn more about persistence using Slick, which is reactive. We will also change our tests a little bit in order to work with a database.

Chapter 6. Persistence with Slick

In the previous chapters, you learned to bootstrap your application using Activator, we developed our web application using Scala and Play framework, and we added Reactive microservices calls using RxScala for data flow computations. We performed unit tests and controller testing using the BDD and Play frameworks.

In this chapter, you will learn how to achieve relational database persistence. So far, we have our application up and running. However, we are using in-memory persistence with HashMap. Now we will upgrade our application to use proper persistence. In order to achieve this, we will use a reactive database persistence framework called Slick.

In this chapter, we will cover the following topics:

- Principles of database persistence with Slick
- Working with Functional Relational Mapping in our application
- Creating Queries with SQL support
- Improving code with Async database operations

We will do some refactoring in our application. Step-by-step, we will create all the tables and persistence classes that we need to have Slick working with our Play framework application. Tests will be refactored as well in order to test the application logic, and skip the database persistence part.

Introducing the Slick framework

Scala Language Integrated Connection Kit (**Slick**) is a Scala modern framework, which allows working with data using abstractions that are very similar to Scala collections. You can write

database queries in both SQL and Scala code. Writing Scala code instead of SQL is better, because we leverage the compiler, and hence, this approach is less error-prone. Also, it becomes easier to maintain the code, since the compiler will help you by pointing out where the code breaks, if it happens.

Slick is a **Functional Relational Mapping** (**FRM**) library. Those of you who come from a Java background, and are familiar with **Object Relational Mapping** (**ORM**) frameworks such as Hibernate, will see that Slick has similar concepts. Basically, you create a Scala class, which will explicitly map to a relational table. Slick FRM ideas are inspired by Microsoft's LINQ framework.

Slick is reactive by design, and works in an asynchronous non-blocking IO model. Using Slick you have the following advantages:

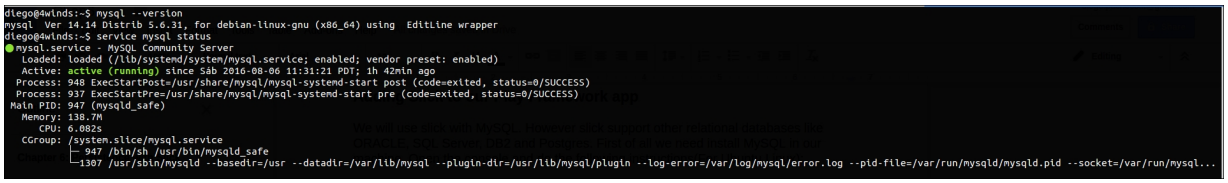
- **Resilience**: A common issue is that a heavy load on the DB and application creates more threads, and makes the situation worse. Slick can fix this problem, because it queues database operations in the DB.
- **Efficient Resources utilization**: Slick can be tuned for parallelism in terms of the number of active jobs and suspended database sessions. Slick also has a clean demarcation between I/O and CPU-intensive code.

MySQL setup

We will use Slick with MySQL 5.6. However, Slick supports other relational databases like Oracle, SQL Server, DB2, and Postgres. First of all, we need to install MySQL in our machine. Open the console, and perform the following steps (for Ubuntu Linux, other OS (Windows/Mac), and distros, check out http://dev.mysql.com/downloads/mysql/):

```
$ sudo apt-get install mysql-server -y
$ mysql --version
$ service mysql status
```

After installation with `apt-get`, when you run the other two commands, you should see an output like this:



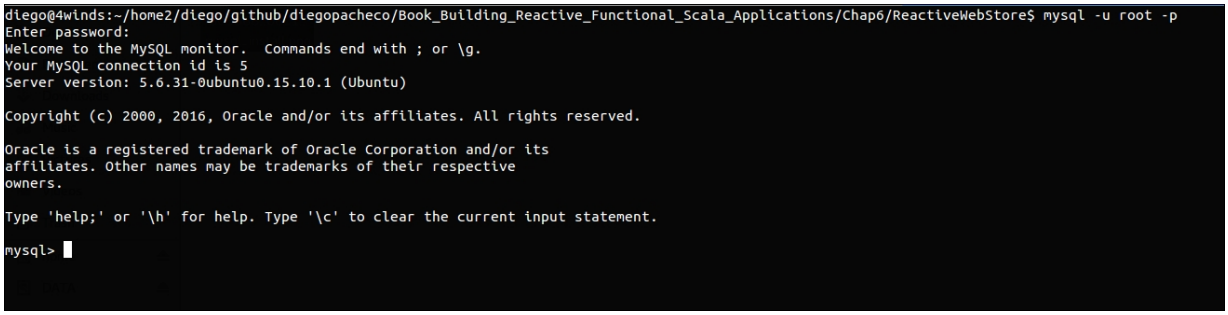
MySQL Installation

Once the installation is done and the MySQL server is up and running, we can move on and create the database. In order to get this, we will need to open the MySQL console. For development reasons, I did not put a password for root. However, for production, it is strongly recommended that you do use a strong password.

Execute the following command:

```
$ mysql -u root -p
```

This will give output as follows:

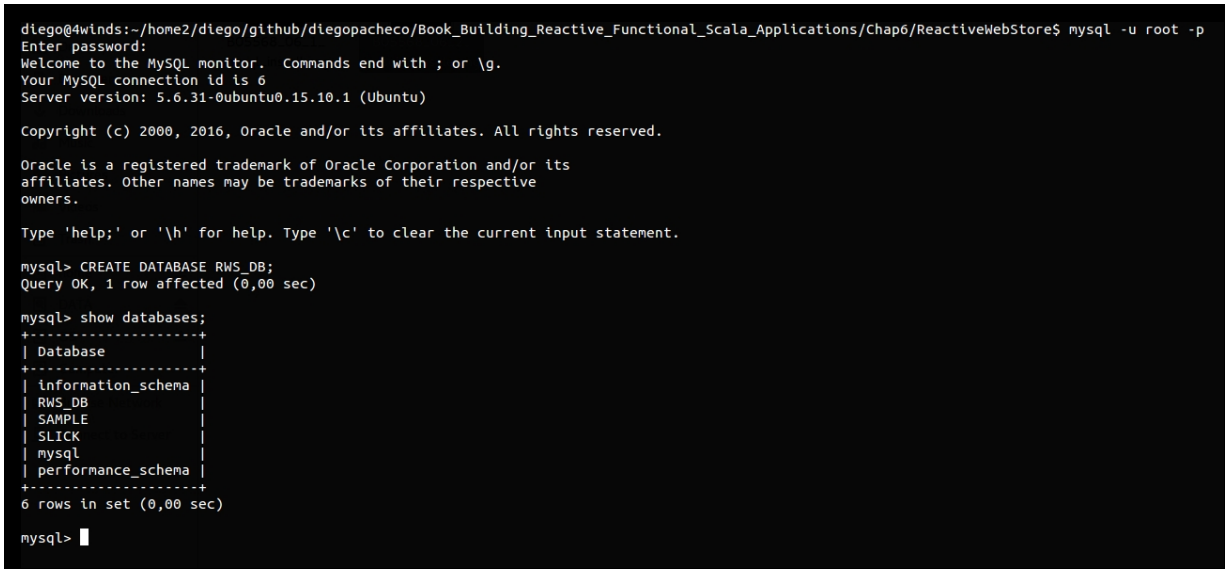


MySQL Console

Once you enter the MySQL console, you can create the database. We will create a database named `RWS_DB` using the following command:

```
$ CREATE DATABASE RWS_DB;
```

You will see the following result:



You can type `$ SHOW DATABASES;` in order to get a list of all the available databases in MySQL. All set, we have our database up and running.

Configuring Slick in our Play framework app

First, we need to add dependencies to the `build.sbt` file. We will need to remove or comment a library called `JDBC` and add the `play-slick` libraries and MySQL java driver.

This is done as follows:

```
name := """ReactiveWebStore"""
    version := "1.0-SNAPSHOT"
    lazy val root = (project in
file(".")).enablePlugins(PlayScala)
    scalaVersion := "2.11.7"
    libraryDependencies ++= Seq(   //jdbc,
      cache,
      ws,
      "org.scalatestplus.play" %% "scalatestplus-
play" % "1.5.1" %
      Test,
      "com.netflix.rxjava" % "rxjava-scala" %
"0.20.7",
      "com.typesafe.play" %% "play-slick" %
"2.0.0",
      "com.typesafe.play" %% "play-slick-
evolutions" % "2.0.0",
      "mysql" % "mysql-connector-java" % "6.0.3"
    )
    resolvers += "scalaz-bintray" at
"http://dl.bintray.com/scalaz/releases"
    resolvers += DefaultMavenRepository
```

As you can see in the preceding code, we comment out the `JDBC` library, and add three new dependencies:

```
"com.typesafe.play" %% "play-slick" % "2.0.0",
    "com.typesafe.play" %% "play-slick-evolutions"
% "2.0.0",
    "mysql" % "mysql-connector-java" % "6.0.3"
```

You can go to the console and run the commands `$ activator`, `$ reload`, and `$ compile` in order to get SBT to download all the new dependencies.

Configure the database connection

Slick needs to be configured to access the MySQL database that we created. Under the folder `ReactiveWebStore/conf`, we need to edit the `application.conf` file and add the database connection URL and settings as follows:

```
# Default database configuration

slick.dbs.default.driver="slick.driver.MySQLDriver
$"

slick.dbs.default.db.driver="com.mysql.cj.jdbc.Dri
ver"

slick.dbs.default.db.url="jdbc:mysql://127.0.0.1:3
306/RWS_DB?
    useUnicode=true&useJDBCCompliantTimezoneShift=

true&useLegacyDatetimeCode=false&serverTimezone=UT
C"
    slick.dbs.default.db.user="root"
    slick.dbs.default.db.password=""
```

FPM Mapping

The next step is to create FPM mapping between our Scala code and MySQL tables under `ReactiveWebStore/app`, we will create a new package called `dao`. DAO stands for **Database Access Object** (**DAO**), and is a well-known OO pattern. So we will create some DAO classes here. First we will define a base trait, which will define the behavior and code capability for each of our `dao` packages.

We will start with `BaseDao.scala`:

```
package dao
    import slick.lifted.TableQuery
    import scala.concurrent.Future
    /**
    * Defines base dao structure every dao should
have.
    */
    trait BaseDao[T] {
      def toTable():TableQuery[_]
      def findAll():Future[Seq[T]]
      def remove(id:Long):Future[Int]
      def insert(o:T):Future[Unit]
      def update(o:T):Future[Unit]
      def findById(id:Long):Future[Option[T]]
    }
```

We will have three `dao` packages: `ProductDao`, `ImageDao`, and `ReviewDao`. Each `dao` will be able to perform an operation, but over a different MySQL table. According to the trait we just defined, we will be able to do the following:

- **findAll**: Find all data for a specific table
- **remove**: Delete an item in a table by ID
- **insert**: Add new data to a table
- **update**: Update data in a table
- **findbyId**: Get a specific record in a table, filtered by ID
- **toTable**: Return the table FRM mapping for that `dao`.

ProductDao

We will start with having a look at `ProductDao.scala`:

```
package dao
    trait IProductDao extends BaseDao[Product]{
      def findAll(): Future[Seq[Product]]
      def findById(id:Long):
Future[Option[Product]]
      def remove(id:Long): Future[Int]
      def insert(p:Product): Future[Unit]
      def update(p2:Product): Future[Unit]
    }
    class ProductDao @Inject() (protected val
dbConfigProvider: DatabaseConfigProvider)
    extends
    HasDatabaseConfigProvider[JdbcProfile] with
IProductDao {
      import driver.api._
      class ProductTable(tag: Tag) extends
Table[Product](tag,
      models.ProductDef.toTable) {
        def id = column[Option[Long]]("ID",
O.PrimaryKey)
        def name = column[String]("NAME")
        def details = column[String]("DETAILS")
        def price = column[BigDecimal]("PRICE")
        def * = (id, name, details, price) <>
(Product.tupled,
        Product.unapply _)
      }
      override def toTable =
TableQuery[ProductTable]
      private val Products = toTable()
      override def findAll(): Future[Seq[Product]]
=
      db.run(Products.result)
      override def findById(id:Long):
Future[Option[Product]] =
      db.run(Products.filter( _.id ===
id).result.headOption)
      override def remove(id:Long): Future[Int] =
      db.run(Products.filter( _.id === id).delete)
      override def insert(p:Product): Future[Unit]
= db.run(Products
```

```
      += p).map { _ => () }
      override def update(p2:Product) =
Future[Unit] {
        db.run(
          Products.filter(_.id === p2.id)
          .map(p => (p.name,p.details, p.price))
          .update((p2.name,p2.details,p2.price))
        )
      }
    }
```

This is the `dao` implementation for `Product`. We have lots of new concepts here, so let's take a look at each step, one at a time. As you can see, a trait called `IProductDao` which extends from `BaseDao` using generics to specify the model `Product`.

This trait is important for dependency injection using Guice. We will have two `dao` implementations for each model: one implementation using Slick and MySQL, and the other using our previous `inMemory` database for testing purpose.

There is a class there called `ProductDao` which is the `dao` implementation using Slick. We need Guice to inject a class here, called `DatabaseConfigProvider`, which will be used to perform the database operations. `ProductDao` also needs to extend `HasDatabaseConfigProvider[JdbcProfile]` to work with the database.

We also need to import the `driver` api via the following command:

```
Import driver.api._
```

The next step is to create FRM mapping with a class called `ProductTable`, which extends table passing the model, which in our case is a product. You also need to announce the name of the MySQL table. In order to get the table name, we use a companion object, which we need to create around our models. We do it this way in order to avoid duplicating the MySQL table name everywhere.

In the `ProductTable` table, you can see some functions such as `id`, `name`, `price`, and `details`. These are the exact name of the fields of `model.Product`. However, we have to add the mapping to the MySQL table on the right side. We do it using a function called `column` where we pass the type and the exact MySQL field name.

Finally, we need to run a special projection function called `*` to pass all the fields on the model, which are being mapped to the relational database.

Now we can move to the `dao` operations. As you can see, all the functions use `db.run` to perform data access. This is great because, as you can realize, they return a Future so the `dao` won't be blocking, and you can do something else, for instance, more database operations, pre-optimizations, and validations.

Once we have a `ProductTable` table, we can create a Slick `TableQuery` with it to perform database operations as if they are Scala functions. In order to list all the available products, we just use this command:

```
db.run(Products.result)
```

It is as simple as that. This code will return a `Future[Seq[Products]]`. We can also filter by ID using this:

```
db.run(Products.filter( _.id === id).result.headOption)
```

So, `first _.id` is the `id` on the database, and `id` is the one that comes by parameter. After getting the result, you can see that we called another function called `headOption`, which makes sure that we get the result as an option. This is a great pattern to rely on, since the data might not be there on the table, and we avoid getting `NoSuchElementException`.

Removing a product is fairly trivial as well. We just use the following:

```
db.run(Products.filter( _.id === id).delete)
```

This preceding code returns `Future[Int]`, counting the number of items that were deleted. If the record ID is not found in the database, the result will be `0`. We expect it to be always `1`, since we are going to delete by ID. However, the API is generic, and if, let's say, you delete by name or another field, you might have multiple deletes. That's why it is an `Int` and not a Boolean.

Inserting data is easy too; we just give the following command:

```
db.run(Products += p).map { _ => () }
```

As you can see, it is a very simple `map` function as if we were adding an element to a list. This code returns unit, which means nothing. However, we still have a Future, so this code is not blocking.

To perform an update, there is a little bit more code, but it is still simple at the end of the day.

```
db.run(
      Products.filter(_.id === p2.id)
      .map(p => (p.name,p.details, p.price))
      .update((p2.name,p2.details,p2.price))
    )
```

First we need to apply a filter to select the records that we will update. We pass the ID, because we just want to update a single record. Then we need to apply a `map` function to pick the fields that we want to update; finally, we perform the update, passing the new values to the `update` function.

Let's take a look at the companion object for the product model.

Here is the code for `Models.Product.scala`:

```
object ProductDef{
      def toTable:String = "Product"
```

```
}
```

As you can see, this is a simple helper companion object to hold the MySQL table name.

ReviewDAO

We are done with `ProductDao`, and now we need to move to the review model and create `dao` for reviews. We will perform steps similar to the ones we did for the product.

`ReviewDao.scala` is as follows:

```
package dao
    trait IReviewDao extends BaseDao[Review]{
      def findAll(): Future[Seq[Review]]
      def findById(id:Long):
Future[Option[Review]]
      def remove(id:Long): Future[Int]
      def insert(p:Review): Future[Unit]
      def update(p2:Review): Future[Unit]
    }
    class ReviewDao @Inject() (protected val
dbConfigProvider: DatabaseConfigProvider)
    extends HasDatabaseConfigProvider[JdbcProfile]
with IReviewDao {
      import driver.api._
      class ReviewTable(tag: Tag) extends
Table[Review](tag,
      models.ReviewDef.toTable) {
        def id = column[Option[Long]]("ID",
O.PrimaryKey)
        def productId = column[Option[Long]]
("PRODUCT_ID")
        def author = column[String]("AUTHOR")
        def comment = column[String]("COMMENT")
        def * = (id, productId, author,comment) <>
(Review.tupled,
        Review.unapply _)
      }
      override def toTable =
TableQuery[ReviewTable]
      private val Reviews = toTable()
      override def findAll(): Future[Seq[Review]]
```

```
=
      db.run(Reviews.result)
      override def findById(id:Long):
Future[Option[Review]] =
      db.run(Reviews.filter( _.id ===
id).result.headOption)
      override def remove(id:Long): Future[Int] =
      db.run(Reviews.filter( _.id === id).delete)
      override def insert(r:Review): Future[Unit]
=
      db.run(Reviews += r).map { _ => () }
      override def update(r2:Review) =
Future[Unit] {
        db.run(
          Reviews.filter(_.id === r2.id)
          .map(i => (i.productId,i.author,
i.comment))

.update((r2.productId,r2.author,r2.comment))
          )
      }
    }
```

In the preceding code, we have the elements that we saw in `ProductDao`. There is an interface for `dao` called `IReviewDao`, which extends `BaseDao` using the review model. We have the `ReviewDao` implementation with the `ReviewTable` FRM mapping. We also have a companion object for the review model.

`Review.scala` is as follows:

```
object ReviewDef{
      def toTable:String = "Review"
    }
```

ImageDao

Now we need to move to our last `dao`, `ImageDao`. Like `ProductDao` and `ReviewDao`, we will go through the same ideas and concepts as implementation too.

We will now look at `ImageDao.scala`:

```
package dao
    trait IImageDao extends BaseDao[Image]{
      def findAll(): Future[Seq[Image]]
      def findById(id:Long): Future[Option[Image]]
      def remove(id:Long): Future[Int]
      def insert(p:Image): Future[Unit]
      def update(p2:Image): Future[Unit]
    }
    class ImageDao @Inject() (protected val
dbConfigProvider: DatabaseConfigProvider)
    extends
    HasDatabaseConfigProvider[JdbcProfile]
    with IImageDao {
      import driver.api._
      class ImageTable(tag: Tag) extends
Table[Image](tag,
      models.ImageDef.toTable) {
        def id = column[Option[Long]]("ID",
O.PrimaryKey)
        def productId = column[Option[Long]]
("PRODUCT_ID")
        def url = column[String]("URL")
        def * = (id, productId, url) <>
(Image.tupled, Image.unapply
        _)
      }
      override def toTable =
TableQuery[ImageTable]
      private val Images = toTable()
      override def findAll(): Future[Seq[Image]] =
      db.run(Images.result)
      override def findById(id:Long):
Future[Option[Image]] =
      db.run(Images.filter( _.id ===
id).result.headOption)
      override def remove(id:Long): Future[Int] =
      db.run(Images.filter( _.id === id).delete)
      override def insert(i:Image): Future[Unit] =
      db.run(Images += i).map { _ => () }
      override def update(i2:Image) = Future[Unit]
      {db.run(
        Images.filter(_.id === i2.id)
        .map(i => (i.productId,i.url))
        .update((i2.productId,i2.url))
      )}
    }
```

We also need to have a companion object helper for the image.

`Image.scala` is as follows:

```
object ImageDef{
      def toTable:String = "Image"
    }
```

Slick evolutions

Slick won't create the table for us, unless we create an evolution. Slick keeps track of the database state and creates and applies SQL commands for us. Evolutions need be located at `ReactiveWebStore/conf/evolutions/default`, where `default` is the name of the database we configured in `application.conf`. Evolutions need to be named in a sequential way so that we can preserve order and Slick can keep track of the changes. Right now, we will create an evolution for `ProductDao`, because we need a product table.

The code will be as follows with the name `1.sql`: **# --- !Ups CREATE TABLE Product (ID INT NOT NULL AUTO_INCREMENT,NAME VARCHAR(100) NOT NULL,DETAILS VARCHAR(250),PRICE DOUBLE NOT NULL,PRIMARY KEY (ID)); # --- !Downs # drop table "Product";**

We need evolutions for th review and image as well. So we need to create `2.sql` for the image and `3.sql` for the review.

The code will be as follows for `2.sql`: **# --- !Ups CREATE TABLE Image (ID INT NOT NULL AUTO_INCREMENT,PRODUCT_ID INT NOT NULL,URL VARCHAR(250),PRIMARY KEY (ID)); # --- !Downs # drop table "Product";**

The code will be as follows with the name `3.sql`: **# --- !Ups CREATE TABLE Review (ID INT NOT NULL AUTO_INCREMENT,PRODUCT_ID INT NOT NULL,AUTHOR VARCHAR(250),COMMENT VARCHAR(250),PRIMARY KEY (ID)); # --- !Downs # drop table "Review";**

Refactoring services

We need to change the default base trait for our `dao` packages to return Futures now.

Let's start with `BaseServices.scala`: package services import scala.concurrent.Future trait BaseService[A] {

def insert(a:A):Future[Unit]

def update(id:Long,a:A):Future[Unit]

def remove(id:Long):Future[Int]

def findById(id:Long):Future[Option[A]]

def findAll():Future[Option[Seq[A]]]

}

This last implementation reflects what's happening in the `dao` packages. Now we can move to the services implementation, and proceed with our refactoring.

Next we see `ProductService.scala`: package services trait IProductService extends BaseService[Product]{

def insert(product:Product):Future[Unit]

def update(id:Long,product:Product):Future[Unit]

def remove(id:Long):Future[Int]

def findById(id:Long):Future[Option[Product]]

def findAll():Future[Option[Seq[Product]]]

def findAllProducts():Seq[(String,String)]

```
}
@Singleton
class ProductService
@Inject() (dao:IProductDao)
extends IProductService{
import play.api.libs.concurrent.Execution.Implicits.
defaultContext
def insert(product:Product):Future[Unit] = {
dao.insert(product);
}
def update(id:Long,product:Product):Future[Unit] = {
product.id = Option(id.toInt) dao.update(product)
}
def remove(id:Long):Future[Int] = {
dao.remove(id)
}
def findById(id:Long):Future[Option[Product]] = {
dao.findById(id)
}
def findAll():Future[Option[Seq[Product]]] = {
```

dao.findAll().map { x => Option(x) }

}

private def validateId(id:Long):Unit = {

val future = findById(id) val entry = Awaits.get(5, future) if (entry==null || entry.equals(None)) throw new RuntimeException("Could not find Product: " + id) }

def findAllProducts():Seq[(String,String)] = {

val future = this.findAll() val result = Awaits.get(5, future) val products:Seq[(String,String)] = result .getOrElse(Seq(Product(Some(0),"","",0))) .toSeq

.map { product => (product.id.get.toString,product.name) }

return products

}

}

There are a couple of changes here. First, we inject an `IProductDao`, and let Guice figure out the right injection that we need to be able to test with our old `in-memory` HashMap implementation, which will be covered later in this chapter.

The changes involve new function signatures, using `Awaits`, and using `Seq` instead of `List`.

Let's move to on `ReviewService.scala` now.

```
package services trait IReviewService extends
BaseService[Review]{


    def insert(review:Review):Future[Unit]
```

```
      def
update(id:Long,review:Review):Future[Unit]

      def remove(id:Long):Future[Int]

      def findById(id:Long):Future[Option[Review]]

      def findAll():Future[Option[Seq[Review]]]

    }

    @Singleton

    class ReviewService @Inject() (dao:IReviewDao)
extends IReviewService{

      import play.api.libs.concurrent.Execution.

      Implicits.defaultContext

      def insert(review:Review):Future[Unit] = {

      dao.insert(review);}

      def
update(id:Long,review:Review):Future[Unit] = {
```

```
        review.id = Option(id.toInt)
dao.update(review)

      }

      def remove(id:Long):Future[Int] = {

        dao.remove(id)

      }

      def findById(id:Long):Future[Option[Review]]
= {

        dao.findById(id)

      }

      def findAll():Future[Option[Seq[Review]]] =
{

        dao.findAll().map { x => Option(x) }

      }

      private def validateId(id:Long):Unit = {
```

```
        val future = findById(id) val entry =
Awaits.get(5, future) if (entry==null ||
entry.equals(None)) throw new
RuntimeException("Could not find Review: " + id) }


    }
```

In the preceding code, we have the same kind of changes that we made for the product. Let's move to `ImageService.scala`, which is our last service.

```
package services trait IImageService extends
BaseService[Image]{


      def insert(image:Image):Future[Unit]


      def update(id:Long,image:Image):Future[Unit]


      def remove(id:Long):Future[Int]


      def findById(id:Long):Future[Option[Image]]


      def findAll():Future[Option[Seq[Image]]]


    }


    @Singleton
```

```
    class ImageService @Inject() (dao:IImageDao)
extends IImageService {

      import play.api.libs.concurrent.Execution.

      Implicits.defaultContext

      def insert(image:Image):Future[Unit] = {

        dao.insert(image)

      }

      def update(id:Long,image:Image):Future[Unit]
= {

        image.id = Option(id.toInt)
dao.update(image)

      }

      def remove(id:Long):Future[Int] = {

        dao.remove(id)

      }
```

```
      def findById(id:Long):Future[Option[Image]]
= {

        dao.findById(id)

      }

      def findAll():Future[Option[Seq[Image]]] = {

        dao.findAll().map { x => Option(x) }

      }

      private def validateId(id:Long):Unit = {

        val future = findById(id) val entry =
Awaits.get(5, future) if (entry==null
||entry.equals(None) ) throw new
RuntimeException("Could not find Image: " + id) }

    }
```

We have refactored all services to use the new `dao` packages implementation. Now the next step is the move to the controllers.

Refactoring controllers

Now we have all the `dao` packages implemented with the respective database evolutions. However, our services expected a different contract, since we were using an in-memory database before. Let's refactor the product controller: package controllers @Singleton

class ProductController @Inject() (val messagesApi:MessagesApi,val service:IProductService) extends Controller with I18nSupport {

val productForm: Form[Product] = Form(

mapping(

"id" -> optional(longNumber), "name" -> nonEmptyText, "details" -> text,

"price" -> bigDecimal

)(models.Product.apply)(models.Product.unapply))

def index = Action { implicit request => val products = Awaits.get(5,service.findAll()) .getOrElse(Seq())

Logger.info("index called. Products: " + products) Ok(views.html.product_index(products)) }

def blank = Action { implicit request => Logger.info("blank called. ") Ok(views.html.product_details(None, productForm)) }

def details(id: Long) = Action { implicit request => Logger.info("details called. id: " + id) val product = Awaits.get(5,service.findById(id)).get Ok(views.html.product_details(Some(id), productForm.fill(product))) }

def insert()= Action { implicit request => Logger.info("insert called.") productForm.bindFromRequest.fold(

```
form => {

BadRequest(views.html.product_details(None, form)) },

product => {

service.insert(product) Redirect(routes.ProductController.index).

flashing("success" -> Messages("success.insert", "new product
created")) }

)

}

def update(id: Long) = Action { implicit request =>
Logger.info("updated called. id: " + id)
productForm.bindFromRequest.fold(

form => {

Ok(views.html.product_details(Some(id), form)).flashing("error" ->
"Fix the errors!") },

product => {

service.update(id,product)
Redirect(routes.ProductController.index).flashing ("success" ->
Messages("success.update", product.name)) }

)

}

def remove(id: Long)= Action {

import play.api.libs.concurrent.Execution.

Implicits.defaultContext
```

val result = Awaits.get(5,service.findById(id)) result.map { product =>

service.remove(id)

Redirect(routes.ProductController.index).flashing("success"

-> Messages("success.delete", product.name))
}.getOrElse(NotFound)

}

}

There are two big changes in the preceding code despite the new function signatures. First, we use a utility function called `get` from a class called `Awaits`. This is needed so that we wait for the result to come back from the database. Second, when we flash the result, we no longer show the `id`, we just display a text message. Let's take a look at the `Awaits` implementation in `Utils.Awaits.scala`, which is as follows: package utils object Awaits {

def get[T](sec:Int,f:Future[T]):T = {

Await.result[T](f, sec seconds) }

}

`Awaits` is just a simple utility class that waits for a period of time to get a Future result. We need to add some tweaks in `ReviewController` and `ImageController` as well.

We will first explore `ReviewController.scala`: package controllers @Singleton

class ReviewController @Inject() (val messagesApi:MessagesApi,val productService:IProductService, val service:IReviewService)

extends Controller with I18nSupport {

```
val reviewForm:Form[Review] = Form(

mapping(

"id" -> optional(longNumber), "productId" -> optional(longNumber),
"author" -> nonEmptyText, "comment" -> nonEmptyText )
(models.Review.apply)(models.Review.unapply) )

def index = Action { implicit request => val reviews =
Awaits.get(5,service.findAll()).getOrElse(Seq()) Logger.info("index
called. Reviews: " + reviews) Ok(views.html.review_index(reviews)) }

def blank = Action { implicit request => Logger.info("blank called. ")
Ok(views.html.review_details(None,
reviewForm,productService.findAllProducts)) }

def details(id: Long) = Action { implicit request => Logger.info("details
called. id: " + id) val review = Awaits.get(5,service.findById(id)).get
Ok(views.html.review_details(Some(id),
reviewForm.fill(review),productService.findAllProducts)) }

def insert()= Action { implicit request => Logger.info("insert called.")
reviewForm.bindFromRequest.fold(

form => {

BadRequest(views.html.review_details(None,
form,productService.findAllProducts)) },

review => {

if (review.productId==null || review.productId.isEmpty) {

Redirect(routes.ReviewController.blank).flashing("error"

-> "Product ID Cannot be Null!") }else {

Logger.info("Review: " + review) if (review.productId==null ||
```

```
review.productId.getOrElse(0)==0) throw new
IllegalArgumentException("Product Id Cannot Be Null")
service.insert(review) Redirect(routes.ReviewController.index).

flashing("success" -> Messages("success.insert", "new Review"))

}

}

)

}

def update(id: Long) = Action { implicit request =>
Logger.info("updated called. id: " + id)
reviewForm.bindFromRequest.fold(

form => {

Ok(views.html.review_details(Some(id),
form,productService.findAllProducts)).

flashing("error" -> "Fix the errors!") },

review => {

service.update(id,review) Redirect(routes.ReviewController.index).

flashing("success" -> Messages("success.update",
review.productId))

}

)

}

def remove(id: Long)= Action {
```

import play.api.libs.concurrent.Execution.

Implicits.defaultContext

val result = Awaits.get(5,service.findById(id)) result.map { review =>

service.remove(id)

Redirect(routes.ReviewController.index).flashing("success" -

> Messages("success.delete", review.productId))
}.getOrElse(NotFound)

}

}

For `ReviewController`, we have made the same changes that we did for the product, that is, the use of `Awaits` and labels on flash returns.

Let's move on to the final controller: `ImageController.scala.`

```
package controllers @Singleton


    class ImageController @Inject() (val
messagesApi:MessagesApi,


    val productService:IProductService, val
service:IImageService)


    extends Controller with I18nSupport {


      val imageForm:Form[Image] = Form(
```

```
        mapping(

          "id" -> optional(longNumber),
"productId" -> optional(longNumber), "url" -> text

        )(models.Image.apply)
(models.Image.unapply) )

      def index = Action { implicit request => val
images =
Awaits.get(5,service.findAll()).getOrElse(Seq())
Logger.info("index called. Images: " + images)
Ok(views.html.image_index(images)) }

      def blank = Action { implicit request =>
Logger.info("blank called. ")
Ok(views.html.image_details(None,
imageForm,productService.findAllProducts)) }

      def details(id: Long) = Action { implicit
request => Logger.info("details called. id: " +
id) val image =
Awaits.get(5,service.findById(id)).get
Ok(views.html.image_details(Some(id),
imageForm.fill(image),productService.findAllProduc
ts)) }

      def insert()= Action { implicit request =>
Logger.info("insert called.")
imageForm.bindFromRequest.fold(

          form => {
```

```
BadRequest(views.html.image_details(None, form,
productService.findAllProducts)) },

          image => {

            if (image.productId==null ||

            image.productId.getOrElse(0)==0) {

Redirect(routes.ImageController.blank).flashing
("error" -> "Product ID Cannot be Null!") }else {

              if (image.url==null ||
"".equals(image.url)) image.url =
"/assets/images/default_product.png"

              service.insert(image)
Redirect(routes.ImageController.index).

              flashing("success" ->
Messages("success.insert", "new image"))

            }

          }

        )
```

```
      }

      def update(id: Long) = Action { implicit
request => Logger.info("updated called. id: " +
id) imageForm.bindFromRequest.fold(

          form => {

            Ok(views.html.image_details(Some(id),
form, null)).flashing("error" -> "Fix the
errors!") },

          image => {

            service.update(id,image)
Redirect(routes.ImageController.index).flashing
("success" -> Messages("success.update",
image.id)) }

        )

      }

      def remove(id: Long)= Action {

        import play.api.libs.concurrent.Execution.

        Implicits.defaultContext
```

```
        val result =
Awaits.get(5,service.findById(id)) result.map {
image =>


          service.remove(id)



Redirect(routes.ImageController.index).flashing("s
uccess"


          -> Messages("success.delete", image.id))
}.getOrElse(NotFound)


      }


    }
```

Configuring DAO packages in Guice

We need to configure the injections for the three new `dao` packages that we created. So we to need to add three lines in the file `Module.scala`. Please open the file in your IDE, and add the following content:

```
bind(classOf[IProductDao]).to(classOf[ProductDao])
.asEagerSingleton()

bind(classOf[IImageDao]).to(classOf[ImageDao]).asE
agerSingleton()

bind(classOf[IReviewDao]).to(classOf[ReviewDao]).a
sEagerSingleton
```

The whole file, `Module.scala`, should look like this:

```
/**
    * This class is a Guice module that tells
Guice how to bind several
    * different types. This Guice module is
created when the Play
    * application starts.
    * Play will automatically use any class called
`Module` that is in
    * the root package. You can create modules in
other locations by
    * adding `play.modules.enabled` settings to
the `application.conf`
    * configuration file.
    */
    class Module extends AbstractModule {
      override def configure() = {
        // Use the system clock as the default
implementation of Clock

bind(classOf[Clock]).toInstance(Clock.systemDefaul
tZone)
        // Ask Guice to create an instance of
```

```
ApplicationTimer when
        //the application starts.

bind(classOf[ApplicationTimer]).asEagerSingleton()
        bind(classOf[IProductService]).

to(classOf[ProductService]).asEagerSingleton()
        bind(classOf[IReviewService]).

to(classOf[ReviewService]).asEagerSingleton()
        bind(classOf[IImageService]).

to(classOf[ImageService]).asEagerSingleton()
        bind(classOf[IPriceSerice]).

to(classOf[PriceService]).asEagerSingleton()
        bind(classOf[IRndService]).
        to(classOf[RndService]).asEagerSingleton()
        bind(classOf[IProductDao]).
        to(classOf[ProductDao]).asEagerSingleton()
        bind(classOf[IImageDao]).
        to(classOf[ImageDao]).asEagerSingleton()
        bind(classOf[IReviewDao]).
        to(classOf[ReviewDao]).asEagerSingleton()
      }
    }
```

Refactoring tests

As you might expect, most tests are no longer working. We will need to perform some refactoring here as well. We will refactor our former `dao` to make it generic, and it will be used in integration tests (end-to-end tests).

Since we will create a generic dao system in memory for end-to-end testing purposes, we need to change our models a little bit. First, we need to create a base trait for all the models. This is needed so we can treat our models as equals.

Let's have a look at `models.BaseModel.scala`:

```
package models
    trait BaseModel {
      def getId:Option[Long]
      def setId(id:Option[Long]):Unit
    }
```

We also need to make all our models implement this new trait. So we will need to change the Scala code for the product, image, and review. This is very trivial: we just add a getter and a setter for the `id` field. You can also use `scala.bean.BeanProperty` instead of writing one by yourself.

Your `models.Product.scala` file should look something like this:

```
package models
    case class Product
    ( var id:Option[Long],
      var name:String,
      var details:String,
    var price:BigDecimal )
    extends BaseModel{
      override def toString:String = {
        "Product { id: " + id.getOrElse(0) +
",name: " + name + ",
        details: "+ details + ", price: " + price
```

```
+ "}"
      }
      override def getId:Option[Long] = id
      override def setId(id:Option[Long]):Unit =
this.id = id
    }
    object ProductDef{
      def toTable:String = "Product"
    }
```

As you can see in the preceding code, we extend the `BaseModel` method, and implement `getId` and `setId`. We need to do the same for the review and image models.

Your `models.Review.scala` file should look like this:

```
package models
    case class Review
    (var id:Option[Long],
      var productId:Option[Long],
      var author:String,
    var comment:String)
    extends BaseModel{
      override def toString:String = {
        "Review { id: " + id + " ,productId: " +
          productId.getOrElse(0) + ",author: " +
author + ",comment: "
         + comment + " }"
      }
      override def getId:Option[Long] = id
      override def setId(id:Option[Long]):Unit =
this.id = id
    }
    object ReviewDef{
      def toTable:String = "Review"
    }
```

Now we move on to the last model. We need to implement it in `Image.scala`.

Your `models.Image.scala` file should look like this:

```
package models
case class Image
(var id:Option[Long],
  var productId:Option[Long],
var url:String)
extends BaseModel {
  override def toString:String = {
    "Image { productId: " + productId.getOrElse(0)
+ ",url: " +
    url + "}"
  }
  override def getId:Option[Long] = id
  override def setId(id:Option[Long]):Unit =
this.id = id
}
object ImageDef{
  def toTable:String = "Image"
}
```

Generic mocks

Now we have all that we need to create a generic mock implementation and mock all the `dao` packages. Under the location `ReactiveWebStore/test/`, we will create a package called mocks, and create a call, `GenericMockedDao`.

Your `GenericMockedDao.scala` file should look like this:

```
package mocks
import models.BaseModel
class GenericMockedDao[T <: BaseModel] {
  import java.util.concurrent.atomic.AtomicLong
  import scala.collection.mutable.HashMap
  import scala.concurrent._
  import play.api.libs.concurrent.Execution.
  Implicits.defaultContext
  var inMemoryDB = new HashMap[Long,T]
  var idCounter = new AtomicLong(0)
  def findAll(): Future[Seq[T]] = {
    Future {
      if (inMemoryDB.isEmpty) Seq()
      inMemoryDB.values.toSeq
    }
  }
  def findById(id:Long): Future[Option[T]] = {
    Future {
      inMemoryDB.get(id)
    }
  }
  def remove(id:Long): Future[Int] = {
    Future {
      validateId(id)
      inMemoryDB.remove(id)
      1
    }
  }
  def insert(t:T): Future[Unit] = {
    Future {
      val id = idCounter.incrementAndGet();
      t.setId(Some(id))
      inMemoryDB.put(id, t)
      Unit
```

```
    }
  }
  def update(t:T): Future[Unit] = {
    Future {
      validateId(t.getId.get)
      inMemoryDB.put(t.getId.get, t)
      Unit
    }
  }
  private def validateId(id:Long):Unit = {
    val entry = inMemoryDB.get(id)
    if (entry==null || entry.equals(None)) throw
new
    RuntimeException("Could not find Product: " +
id)
  }
}
```

So the `GenericMockedDao` call expects the `Generic` parameter, which could be any class extending from `BaseModel`. Then we use an in-memory HashMap implementation and a counter to simulate database operations. We run all the operations inside Futures, so we don't break the new signature the code is expecting. Now we can create three `MockedDaos` for each model we need: product, review, and image.

Your `mocks.ProductMockedDao.scala` file should look like this:

```
package mocks
class ProductMockedDao extends IProductDao {
  val dao:GenericMockedDao[Product] = new
  GenericMockedDao[Product]()
  override def findAll(): Future[Seq[Product]] = {
    dao.findAll()
  }
  override def findById(id:Long):
Future[Option[Product]] = {
    dao.findById(id)
  }
  override def remove(id:Long): Future[Int] = {
    dao.remove(id)
  }
  override def insert(p:Product): Future[Unit] = {
    dao.insert(p)
```

```
  }
  override def update(p2:Product): Future[Unit] =
{
    dao.update(p2)
  }
  override def toTable:TableQuery[_] = {
    null
  }
}
```

As you can see here, we implement the `IProdutDao` trait, and we delegate all operations to `genericMockedDao`. Since everything is in-memory, we don't need to implement the `toTable` function. We need to do the same for the review and image.

Your `mocks.ReviewMockedDao.scala` file should look like this:

```
package mocks
class ReviewMockedDao extends IReviewDao {
  val dao:GenericMockedDao[Review] = new
  GenericMockedDao[Review]()
  override def findAll(): Future[Seq[Review]] = {
    dao.findAll()
  }
  override def findById(id:Long):
Future[Option[Review]] = {
    dao.findById(id)
  }
  override def remove(id:Long): Future[Int] = {
    dao.remove(id)
  }
  override def insert(p:Review): Future[Unit] = {
    dao.insert(p)
  }
  override def update(p2:Review): Future[Unit] = {
    dao.update(p2)
  }
  override def toTable:TableQuery[_] = {
    null
  }
}
```

Exactly like product, we delegate all operations to `GenericMockedDao`. Now let's move to the last one, the image, and then we can fix the

tests.

Your `mocks.ImageMockedDao.scala` file should look like this:

```
package mocks
class ImageMockedDao extends IImageDao {
  val dao:GenericMockedDao[Image] = new
GenericMockedDao[Image]()
  override def findAll(): Future[Seq[Image]] = {
    dao.findAll()
  }
  override def findById(id:Long):
Future[Option[Image]] = {
    dao.findById(id)
  }
  override def remove(id:Long): Future[Int] = {
    dao.remove(id)
  }
  override def insert(p:Image): Future[Unit] = {
    dao.insert(p)
  }
  override def update(p2:Image): Future[Unit] = {
    dao.update(p2)
  }
  override def toTable:TableQuery[_] = {
    null
  }
}
```

Okay, we have all the mocks that we need for now. We can move on to fix the test specs. We need to fix services tests and controller test. Services tests will use mocks. Controllers tests, however, will use the real database implementation. We need to use other utility classes for controller tests. Located in the test source folder, we need to create a package called `utils`.

Your `utils.DBCleaner.scala` file should look like this:

```
package utils
    object DBCleaner {
      val pool = Executors.newCachedThreadPool()
      implicit val ec =
ExecutionContext.fromExecutorService(pool)
```

```
      def cleanUp():Unit = {
        Class.forName("com.mysql.cj.jdbc.Driver")
        val db =   Database.forURL
        ("jdbc:mysql://127.0.0.1:3306/RWS_DB?
useUnicode=

true&useJDBCCompliantTimezoneShift=true&useLegacyD
atetimeCode=
        false&serverTimezone=UTC",
        "root", "")
        try{
          Await.result(
            db.run(
              DBIO.seq(
                sqlu""" DELETE FROM Product; """,
                sqlu""" DELETE FROM Image; """,
                sqlu""" DELETE FROM Review; """,
                sqlu""" ALTER TABLE Product
AUTO_INCREMENT = 1 """,
                sqlu""" ALTER TABLE Image
AUTO_INCREMENT = 1 """,
                sqlu""" ALTER TABLE Review
AUTO_INCREMENT = 1 """
              )
            )
          ,20 seconds)
        }catch{
          case e:Exception => Unit
        }
      }
    }
```

`DBCleaner` will connect to the real database and perform delete statements to clean up all table data. After deleting all data in the tables, we also reset the sequence in the database; otherwise, our tests will not have the predictability we need to do assertions.

As you can see in `db.run`, we can use `DBIO.seq`, which allows us to execute multiple instructions on the database. Here we are not using Scala code. We are using pure SQL statements, since we need to use very specific MySQL functions to reset the sequences.

If you need, you could use all these functions in your application. This is useful if you need to use a specific database function, if you

have a very complex query, or sometimes, because there is a performance issue.

Most fixes we do now center around using `Awaits` to wait for the Future result, and also using our new mocks. For the controller test, we need to call the `DBCleaner` function as well.

Service tests

Now we will create tests for services to test them. So let's get started.

Your `ProductServiceTestSpec.scala` file should look like this: class ProductServiceTestSpec extends PlaySpec {

"ProductService" must {

val service:IProductService = new ProductService(new ProductMockedDao)

"insert a product properly" in {

val product = new models.Product(Some(1),"Ball","Awesome Basketball",19.75)

service.insert(product)

}

"update a product" in {

val product = new models.Product(Some(1),"Blue Ball","Awesome Blue Basketball",19.99) service.update(1, product)

}

"not update because does not exit" in {

intercept[RuntimeException]{

service.update(333,null)

}

}

```
"find the product 1" in {

val product = Awaits.get(5, service.findById(1)) product.get.id
mustBe Some(1) product.get.name mustBe "Blue Ball"

product.get.details mustBe "Awesome Blue Basketball"

product.get.price mustBe 19.99

}

"find all" in {

val products = Awaits.get(5, service.findAll()) products.get.length
mustBe 1

products.get(0).id mustBe Some(1) products.get(0).name mustBe
"Blue Ball"

products.get(0).details mustBe "Awesome Blue Basketball"

products.get(0).price mustBe 19.99

}

"find all products" in {

val products = service.findAllProducts() products.length mustBe 1

products(0)._1 mustBe "1"

products(0)._2 mustBe "Blue Ball"

}

"remove 1 product" in {

val product = Awaits.get(5, service.remove(1)) product mustBe 1
```

val oldProduct = Awaits.get(5,service.findById(1)) oldProduct mustBe None

}

"not remove because does not exist" in {

intercept[RuntimeException]{

Awaits.get(5,service.remove(-1)) }

}

}

}

As you can see in the preceding code, most fixes center around the new signatures and the fact we are using Futures and need to use the `Awaits` utility and mocks. We test the service without the database call via this code: val service:IProductService = new ProductService(new ProductMockedDao)

We can move on to the next service, which will be the review.

Your `ReviewServiceTestSpec.scala` file should look like this: class ReviewServiceTestSpec extends PlaySpec {

"ReviewService" must {

val service:IReviewService = new ReviewService(new ReviewMockedDao)

"insert a review properly" in {

val review = new models.Review (Some(1),Some(1),"diegopacheco","Testing is Cool") service.insert(review)

```
}

"update a reviewt" in {

val review = new models.Review
(Some(1),Some(1),"diegopacheco","Testing so so Cool")
service.update(1, review)

}

"not update because does not exist" in {

intercept[RuntimeException]{

Awaits.get(5, service.update(333,null)) }

}

"find the review 1" in {

val review = Awaits.get(5,service.findById(1)) review.get.id mustBe
Some(1) review.get.author mustBe "diegopacheco"

review.get.comment mustBe "Testing so so Cool"

review.get.productId mustBe Some(1) }

"find all" in {

val reviews = Awaits.get(5,service.findAll()) reviews.get.length
mustBe 1

reviews.get(0).id mustBe Some(1) reviews.get(0).author mustBe
"diegopacheco"

reviews.get(0).comment mustBe "Testing so so Cool"

reviews.get(0).productId mustBe Some(1) }
```

"remove 1 review" in {

val review = Awaits.get(5, service.remove(1)) review mustBe 1

val oldReview= Awaits.get(5, service.findById(1)) oldReview mustBe None

}

"not remove because does not exist" in {

intercept[RuntimeException]{

Awaits.get(5, service.remove(-1)) }

}

}

}

That was the review spec service test code. We apply the same changes as we did for product. Now we need to move on to the last service test, which will be the image.

Your `ImageServiceTestSpec.scala` file should look like this: class ImageServiceTestSpec extends PlaySpec {

"ImageService" must {

val service:IImageService = new ImageService(new ImageMockedDao)

"insert a image properly" in {

val image = new models.Image (Some(1),Some(1),"http://www.google.com.br/myimage") service.insert(image)

```
}

"update a image" in {

val image = new models.Image
(Some(2),Some(1),"http://www.google.com.br/myimage")
service.update(1, image)

}

"not update because does not exist" in {

intercept[RuntimeException]{

Awaits.get(5, service.update(333,null)) }

}

"find the image" in {

val image = Awaits.get(5,service.findById(1)) image.get.id mustBe
Some(1)

image.get.productId mustBe Some(1) image.get.url mustBe
"http://www.google.com.br/myimage"

}

"find all" in {

val reviews = Awaits.get(5,service.findAll()) reviews.get.length
mustBe 1

reviews.get(0).id mustBe Some(1) reviews.get(0).productId mustBe
Some(1) reviews.get(0).url mustBe
"http://www.google.com.br/myimage"

}
```

```
"remove 1 image" in {

val image = Awaits.get(5,service.remove(1)) image mustBe 1

val oldImage = Awaits.get(5,service.findById(1)) oldImage mustBe
None

}

"not remove because does not exist" in {

intercept[RuntimeException]{

Awaits.get(5,service.remove(-1)) }

}

}

}
```

We have fixed all the services tests. Now we need to fix the controller tests.

Controller tests

Now let's fix the controller tests. The first one will be the product controller.

Your `ProductControllerTestSpec.scala` file should look like this: class ProductControllerTestSpec extends

PlaySpec

with OneServerPerSuite with OneBrowserPerSuite with HtmlUnitFactory {

"ProductController" should {

DBCleaner.cleanUp()

"insert a new product should be ok" in {

goTo(s"http://localhost:${port}/product/add") click on id("name")

enter("Blue Ball")

click on id("details")

enter("Blue Ball is a Awesome and simple product") click on id("price")

enter("17.55")

submit()

}

"details from the product 1 should be ok" in {

goTo(s"http://localhost:${port}/product/details/1") textField("name").value mustBe "Blue Ball"

```
textField("details").value mustBe "Blue Ball is a Awesome and simple product"

textField("price").value mustBe "17.55"

}

"update product 1 should be ok" in {

goTo(s"http://localhost:${port}/product/details/1")
textField("name").value = "Blue Ball 2"

textField("details").value = "Blue Ball is a Awesome and simple product 2 "

textField("price").value = "17.66"

submit()

goTo(s"http://localhost:${port}/product/details/1")
textField("name").value mustBe "Blue Ball 2"

textField("details").value mustBe "Blue Ball is a Awesome and simple product 2 "

textField("price").value mustBe "17.66"

}

"delete a product should be ok" in {

goTo(s"http://localhost:${port}/product/add") click on id("name")

enter("Blue Ball")

click on id("details")

enter("Blue Ball is a Awesome and simple product") click on id("price")
```

enter("17.55")

submit()

goTo(s"http://localhost:${port}/product") click on id("btnDelete")

}

"Cleanup db in the end" in {

DBCleaner.cleanUp()

}

}

}

The Controller product test needs to call the `DBCleaner` function at the beginning of the test to make sure that the database is in a well-known state; additionally, and after running all the tests, we need to clean up the database just to be safe.

We will now apply the same changes for the review and image controller tests.

Your `ReviewControllerTestSpec` file should look like this: class ReviewControllerTestSpec extends PlaySpec

with OneServerPerSuite with OneBrowserPerSuite with HtmlUnitFactory {

DBCleaner.cleanUp()

"ReviewController" should {

"insert a new review should be ok" in {

goTo(s"http://localhost:${port}/product/add") click on id("name")

```
enter("Blue Ball")

click on id("details")

enter("Blue Ball is a Awesome and simple product") click on
id("price")

enter("17.55")

submit()

goTo(s"http://localhost:${port}/review/add")
singleSel("productId").value = "1"

click on id("author")

enter("diegopacheco")

click on id("comment")

enter("Tests are amazing!")

submit()

}

"details from the review 1 should be ok" in {

goTo(s"http://localhost:${port}/review/details/1")
textField("author").value mustBe "diegopacheco"

textField("comment").value mustBe "Tests are amazing!"

}

"update review 1 should be ok" in {

goTo(s"http://localhost:${port}/review/details/1")
textField("author").value = "diegopacheco2"
```

```
textField("comment").value = "Tests are amazing 2!"

submit()

goTo(s"http://localhost:${port}/review/details/1")
textField("author").value mustBe "diegopacheco2"

textField("comment").value mustBe "Tests are amazing 2!"

}

"delete a review should be ok" in {

goTo(s"http://localhost:${port}/review/add")
singleSel("productId").value = "1"

click on id("author")

enter("diegopacheco")

click on id("comment")

enter("Tests are amazing!")

submit()

goTo(s"http://localhost:${port}/review") click on id("btnDelete")}

"Cleanup db in the end" in {

DBCleaner.cleanUp()

}

}

}
```

Alright, we have the tests for the review controller fixed. Now we can move to the last controller test for the image.

Your `ImageControllerTestSpec.scala` file should look like this: class ImageControllerTestSpec extends PlaySpec

with OneServerPerSuite with OneBrowserPerSuite with HtmlUnitFactory {

DBCleaner.cleanUp()

"ImageController" should {

"insert a new image should be ok" in {

goTo(s"http://localhost:${port}/product/add") click on id("name")

enter("Blue Ball")

click on id("details")

enter("Blue Ball is a Awesome and simple product") click on id("price")

enter("17.55")

submit()

goTo(s"http://localhost:${port}/image/add") singleSel("productId").value = "1"

click on id("url")

enter("https://thegoalisthering.files.wordpress.com/2012/01/

bluetennisball_display_image.jpg") submit()

}

"details from the image 1 should be ok" in {

goTo(s"http://localhost:${port}/image/details/1") textField("url").value mustBe

"https://thegoalisthering.files.wordpress.com/2012/01/

bluetennisball_display_image.jpg"

}

"update image 1 should be ok" in {

goTo(s"http://localhost:${port}/image/details/1") textField("url").value =

"https://thegoalisthering.files.wordpress.com/2012/01/

bluetennisball_display_image2.jpg"

submit()

goTo(s"http://localhost:${port}/image/details/1") textField("url").value mustBe

"https://thegoalisthering.files.wordpress.com/2012/01/

bluetennisball_display_image2.jpg"

}

"delete a image should be ok" in {

goTo(s"http://localhost:${port}/image/add")
singleSel("productId").value = "1"

click on id("url")

enter("https://thegoalisthering.files.wordpress.com/2012/01/

bluetennisball_display_image.jpg") submit()

goTo(s"http://localhost:${port}/image") click on id("btnDelete")

}

"Cleanup db in the end" in {

DBCleaner.cleanUp()

}

}

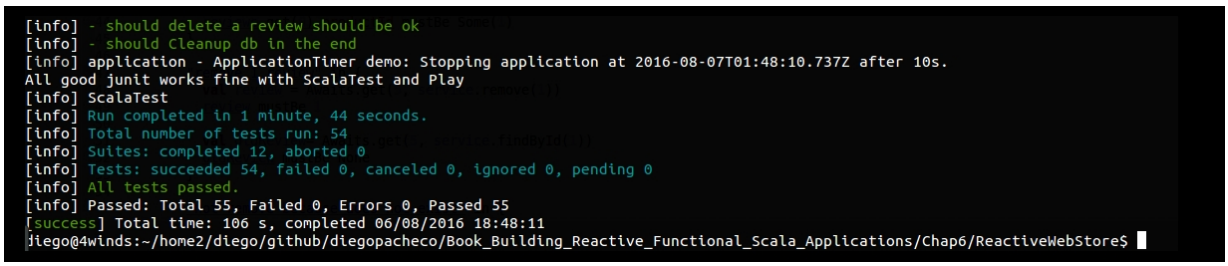
}

All right, all the controller tests are fixed now. We can run all the tests to double check whether everything is OK.

Run the following command:

```
$ activator test
```

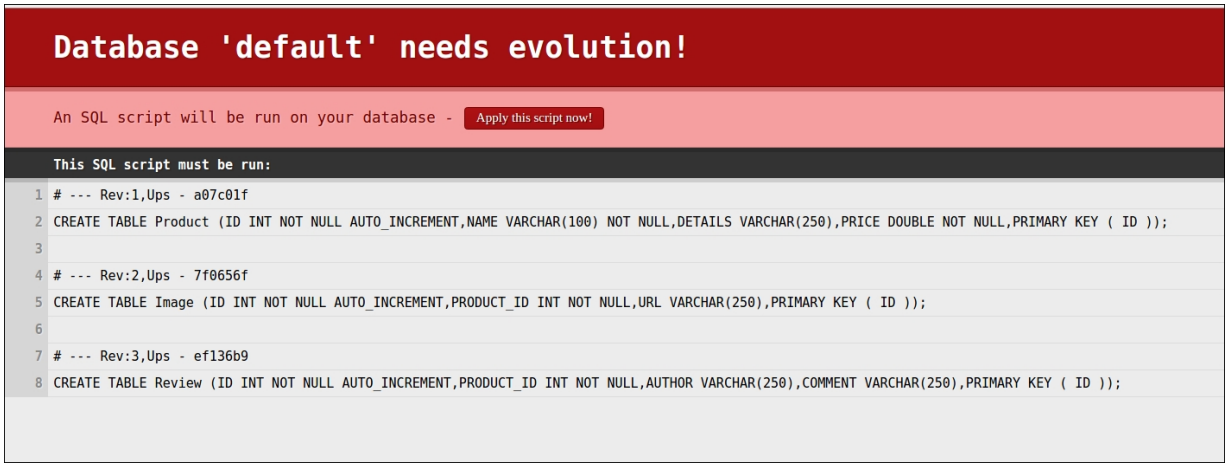
You get output as shown in the following screenshot:



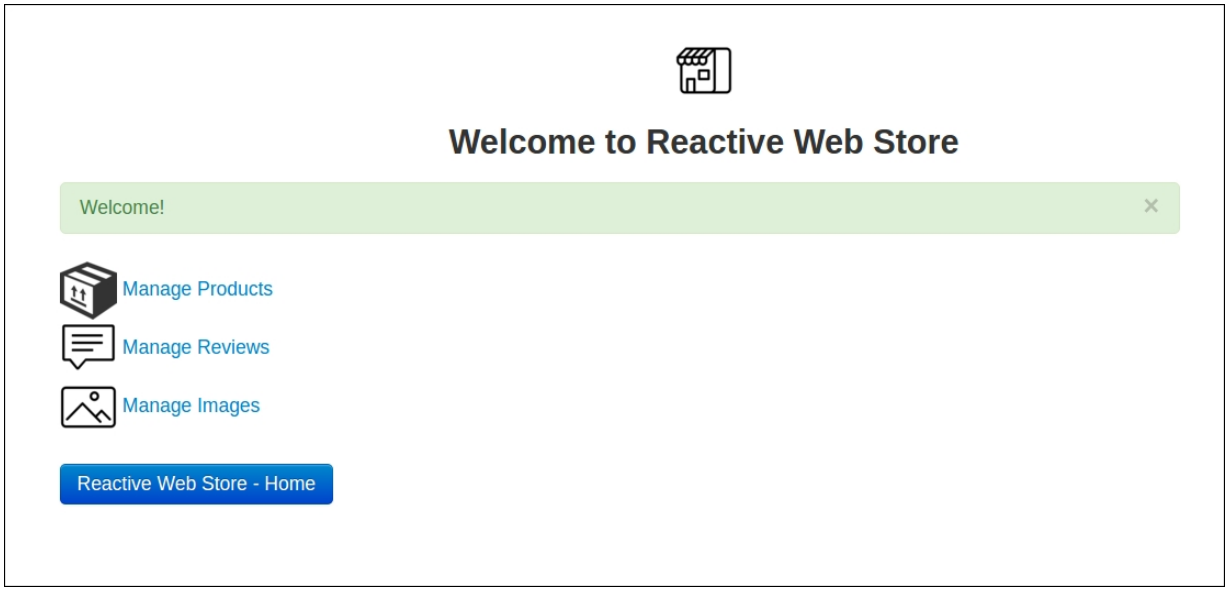
If you have problems running the application (covered in the next section), apply the evolution, and then you can run the tests again. Tests might take some time, depending on your hardware.

Running the application

Now it is time to run the application using `$ activator run`. Open your web browser, and go to `http://localhost:9000/`. Once you do that, Play will detect that the application needs evolutions, and will apply the three evolutions we have (`1.sql`, `2.sql`, and `3.sql`). However, you will need to click on the button to apply the evolution.



After you click on the red button, **Apply this script now!**, Slick will create the tables, and redirect you to the application.



Summary

With this, we reach the end of the chapter. You learned how to perform database persistence using Slick. You also learned how to do FRM mapping, and we refactored our application and tests so they work with reactive persistence and the Play framework. We then explained how to access the database using Scala code, and perform operations using SQL.

In the following chapter, we will see more about reports, and we will use our database to generate reports based on our Play framework application.

Chapter 7. Creating Reports

Up until now, we learned how to bootstrap our application using Activator, develop our web application using the Scala and Play framework, and a add reactive microservices call using RxScala for data flow computations. We also performed unit test and controller testing using the BDD and Play framework. Then, we persisted data into MySQL using Slick. Now we will move on with our application.

In this chapter, you will learn how to write reports with JasperReports. JasperReports is a very solid reporting solution for Java, and it can be used in Scala very easily. We will create database reports using Jasper, and change our application to have such functionality.

In this chapter, we will cover the following topics:

- Understanding JasperReports
- Adding database reports to our application

Introducing JasperReports

JasperReports (http://community.jaspersoft.com/project/jasperreports-library) is a very popular and solid reports solution that can generate reports in several formats, such as:

- HTML
- Excel
- Word
- Open Office format
- PDF

In order to get your reports, you have a visual tool called Jaspersoft Studio, in which you can drag and drop elements such as labels,

images, data fields, and much more. Jasper will store this metadata (the report definition) in an XML file, also known as **JRXML**. If you want, you can edit and work with this XML without any editor; however, it is way better to use the Jaspersoft Studio tool to gain productivity.

Jasper can work with several data sources, such as databases, XML, or even objects in memory. For this book, we will use the database datasource to access our MySQL database.

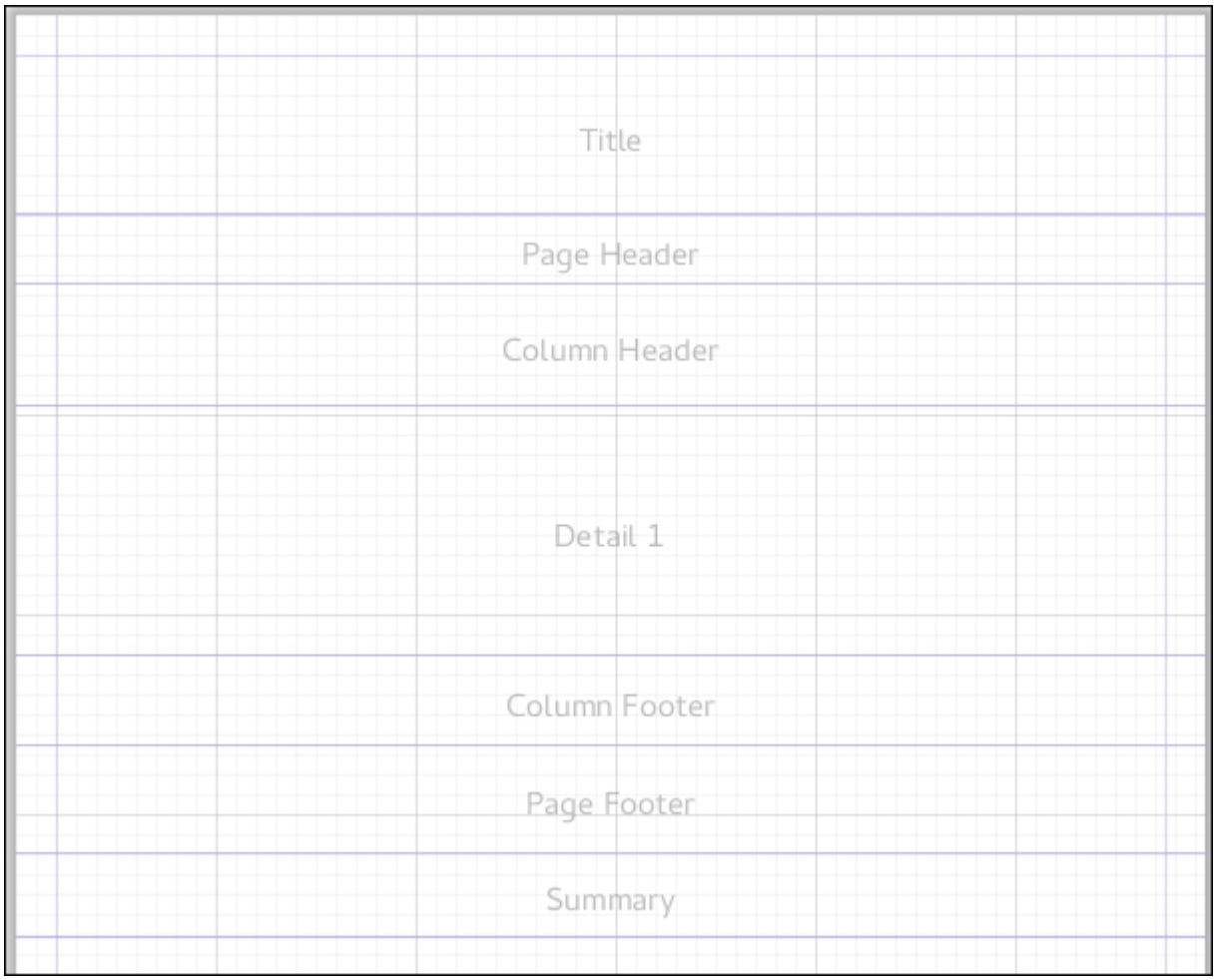
JasperReports workflow

JasperReports has the same execution stages, including compiling your reports and rendering in a specific format, for example, HTML. The first stage is the report design. If you are not using the Jaspersoft Studio visual tool, we assume that you have your JRXML file. The next step is to compile JRXML into a Jasper file. This compilation phase doesn't need to happen every time; it's needed only if you change the JRXML. Otherwise, you can use the same Jasper file. There are some strategies to cache the Jasper file, so basically you can do it on the build time or you can cache on demand in the application. For our application, we will be using the second approach--caching on demand in the application.

The next phase is to render or export. You can export the report to the many formats Jasper supports, such as HTML, EXCEL, or PDF, for instance. It's possible to use the same report layout and export to as many formats as you like. For our application, we will be using the PDF format.

Jasper sessions

A JRXML has sections that are evaluated in different ways and at different times. The following diagram shows all the available sessions:



The different sections are as follows:

- **Title**: This is printed just one time
- **Page Header**: This is printed at the beginning of all printed pages

- **Column Header**: This is printed at the beginning of each detail column
- **Detail**: This is where every record read from the data source is printed
- **Column Footer**: This is printed at the end of each detail column
- **Page Footer**: This is printed at the end of all printed pages
- **Summary**: This is printed at the end of the report, and is often used to show calculations, totals, and summarizations in general

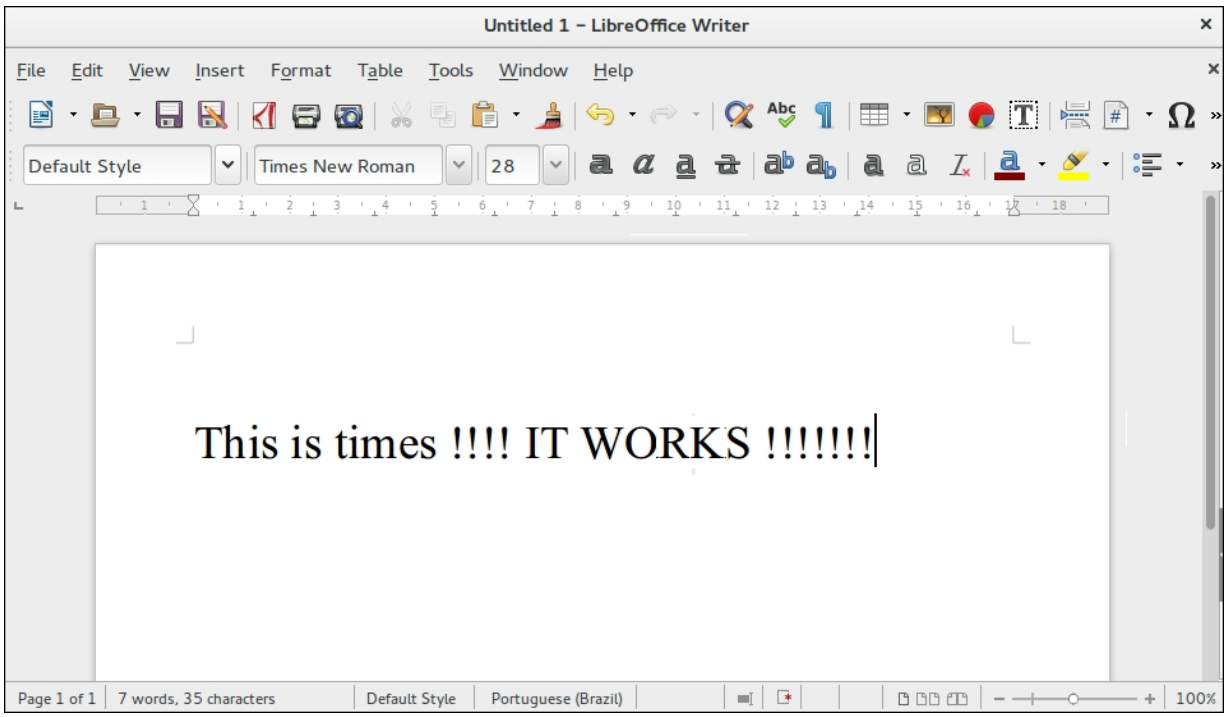
Jasper is a very flexible report solution that also allows us to run groovy scripts inside a Jasper report to do dynamic calculations as well as dynamic layouts. This is useful if you do not want to print a page given some condition, or based on what you have in the database, or you do not want to show some data.

Next, we will install Jaspersoft Studio and start creating reports for our application.

Installing Jaspersoft Studio 6

For this, you will need to have Java 8 installed. If you don't have it, go back to Chapter 1, *Introduction to FP, Reactive, and Scala*, and follow the setup instructions. Jasper is really great because it works on multiple platforms; however, it works better on Windows. We are using Linux, so we will need to deal with fonts. JasperReports uses lots of Microsoft's core fonts, such as Arial and Times New Roman. There are some options to have the sources on Linux. You can look for a mscorefonts installer on Linux or just copy the fonts from Windows.

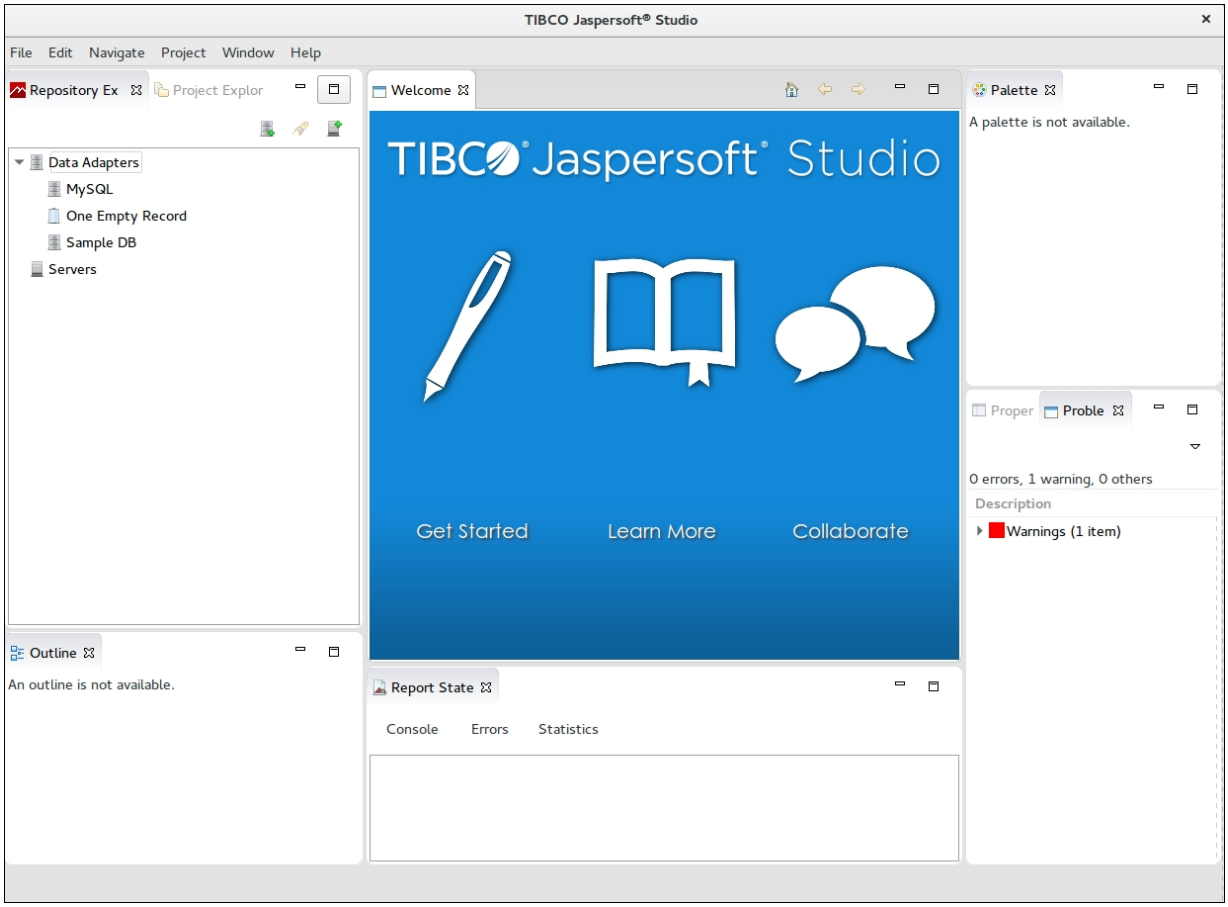
If you have a dual boot Linux/Windows installation, you can go to your Windows drive at the location `WindowsDrive/Windows/Fonts`. You will need to copy all font files to `/usr/share/fonts` on Linux and run `$ sudo fc-cache -fv`. This might take some time--for my Windows installation, it was about ~300 MB of fonts. You can test whether you have Windows core fonts on Linux. Open the writer and check for the fonts. You should see something similar to this:



Why is this so important? Because Jasper won't work if you don't have the right font in place. It will just throw you random exceptions that will not make sense, but it is very likely to be related to missing fonts.

Once we have the fonts, we can go ahead and download Jaspersoft Studio 6. For this book, we will be using the 6.2.2 version. You can download it from http://community.jaspersoft.com/project/jasperreports-library/releases. If you are on Linux, it's highly recommended to use the `DEB` package; otherwise, you will need to install several other dependencies.

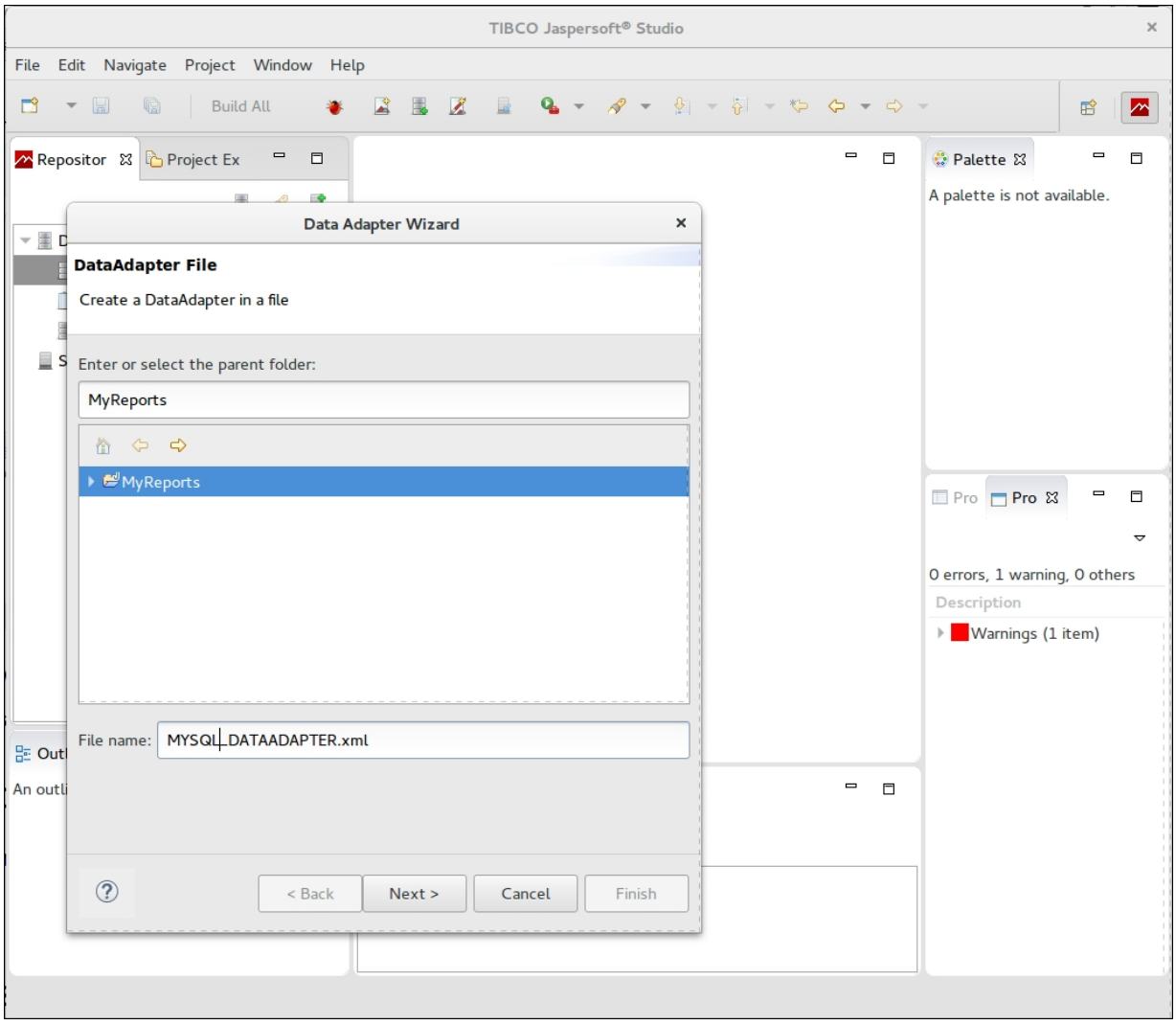
Once you download and install Jaspersoft Studio and open the program, you will see a UI similar to this one here:



We have successfully installed Jaspersoft Studio. Now, we will need to configure our MySQL data source in order to start creating reports for our application.

Configuring MySQL Data Adapter in Jaspersoft Studio

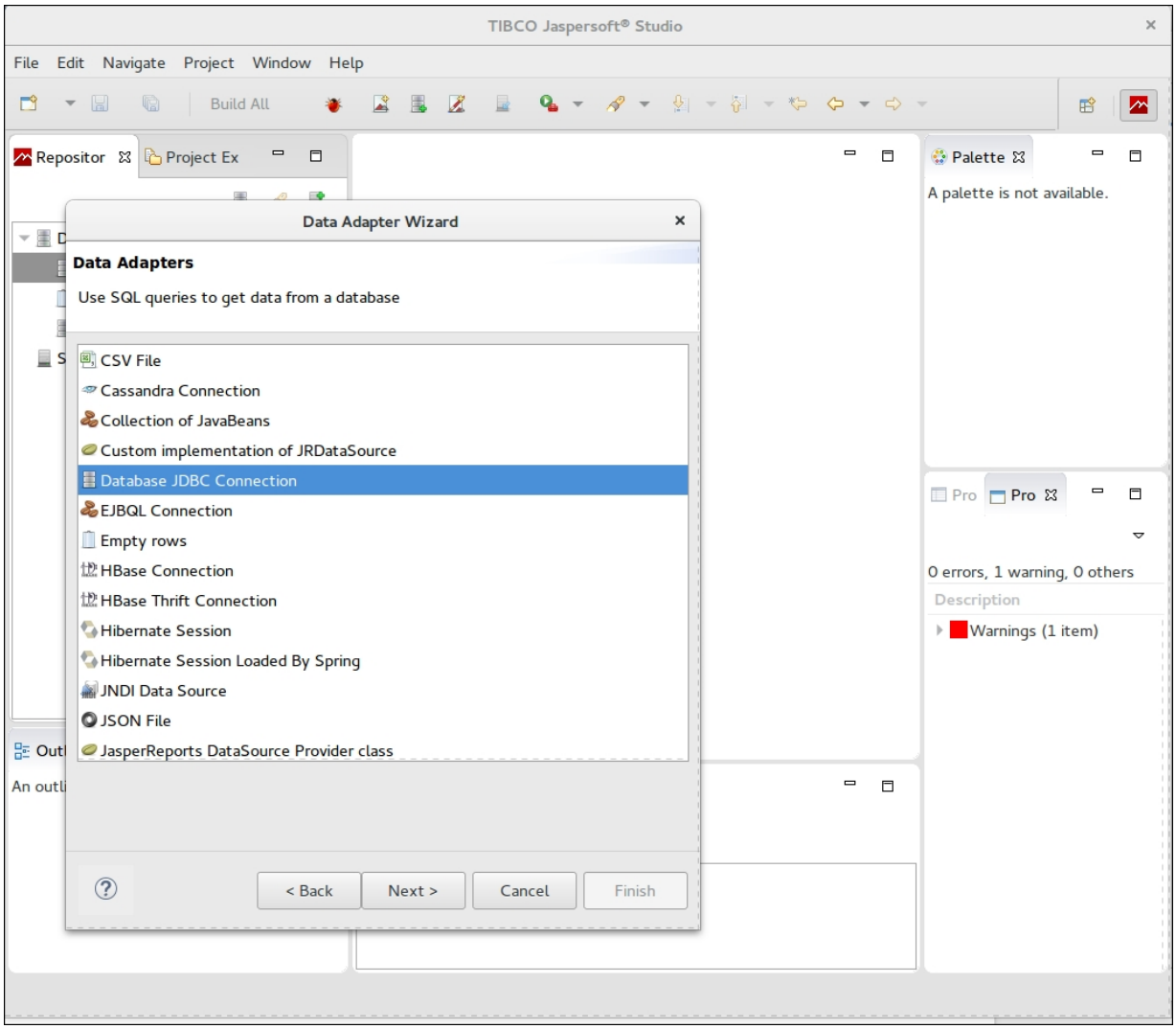
Open Jaspersoft Studio 6 and click on **File** | **New** | **Data Adapter Wizard**. You will see the following screen:



File name should be `MYSQL_DATAADAPTER.xml`, and then you can click **Next >**.

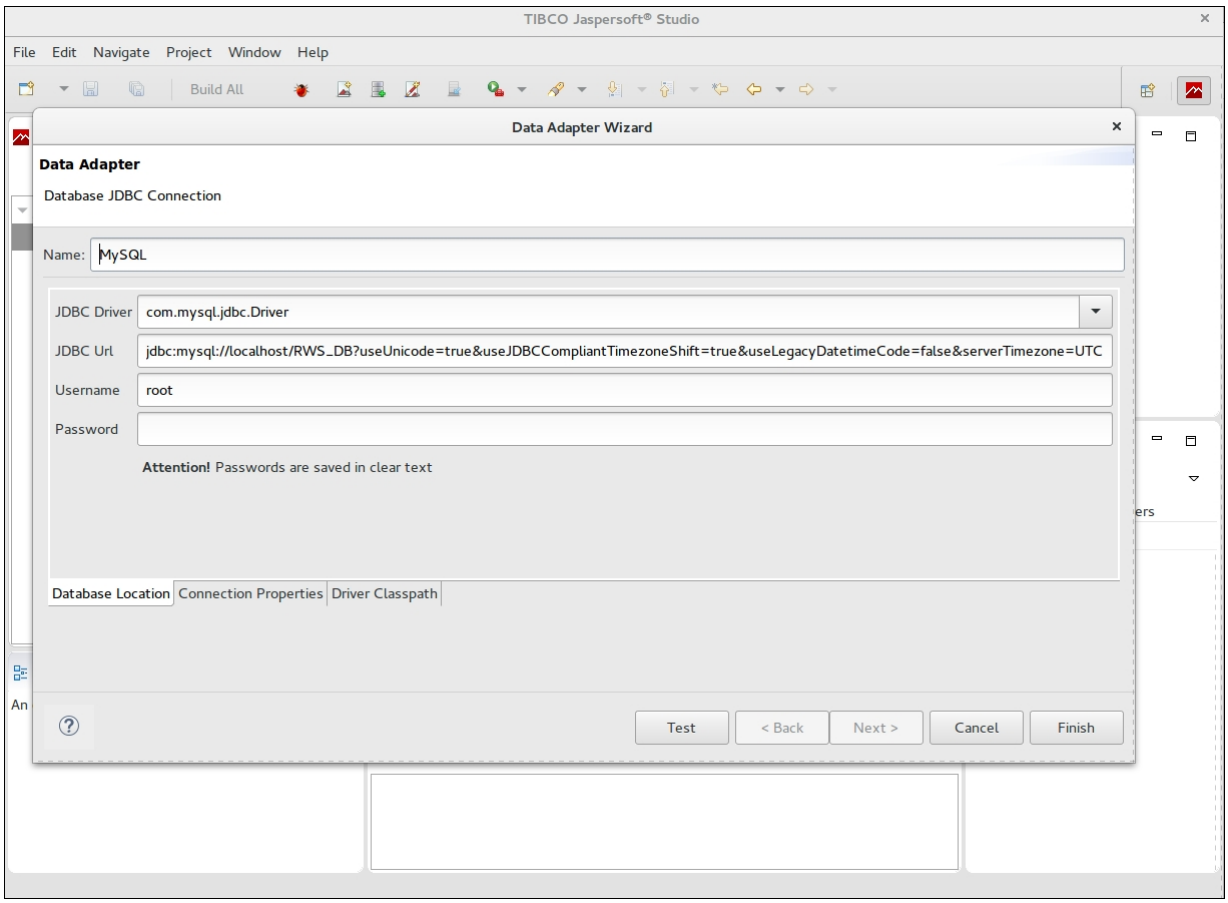
Next, we will need to choose the type of database adapter. There are several options, such as Cassandra, MongoDB, HBase, JSON file,

and so on.



We need to pick **Database JDBC Connection** and click **Next >**.

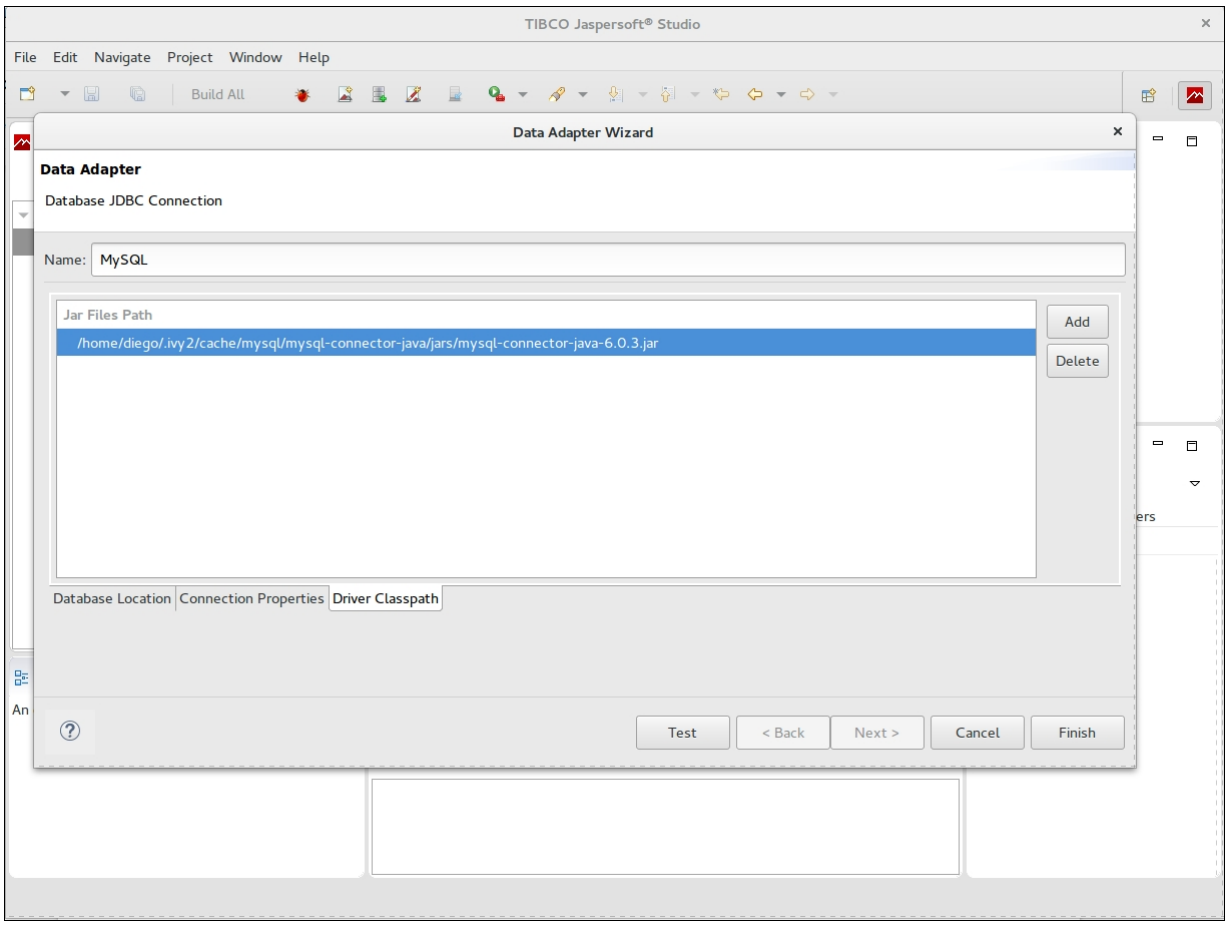
Now, we will need to configure the connection details.



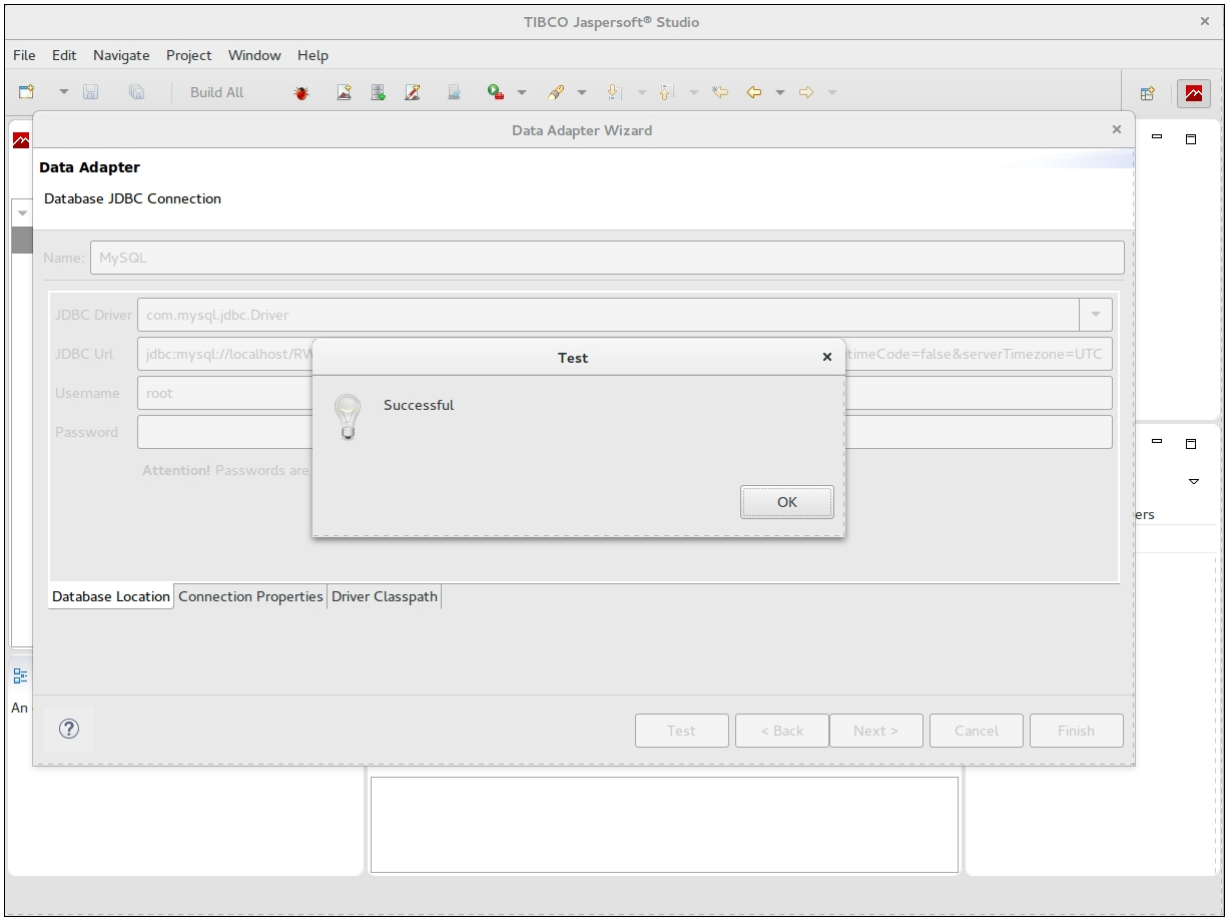
The fields should be completed as follows:

- **Name:** `MySQL`
- **JDBC Driver:** `com.mysql.jdbc.Driver`
- **JDBC Url:** `jdbc:mysql://localhost/RWS_DB?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC`
- **Username:** `root`
- **Password:** `This needs be blank, or put the password if you are using one.`

We will also need to configure the driver into the Jaspersoft Studio classpath. As we are running the application in the same box, we already have the MySQL driver download with SBT on the `~/.ivy2/cache/mysql/mysql-connector-java/jars/mysql-connector-java-6.0.3.jar` folder. We will just need to point it out on the third tab, called `Driver Classpath`.



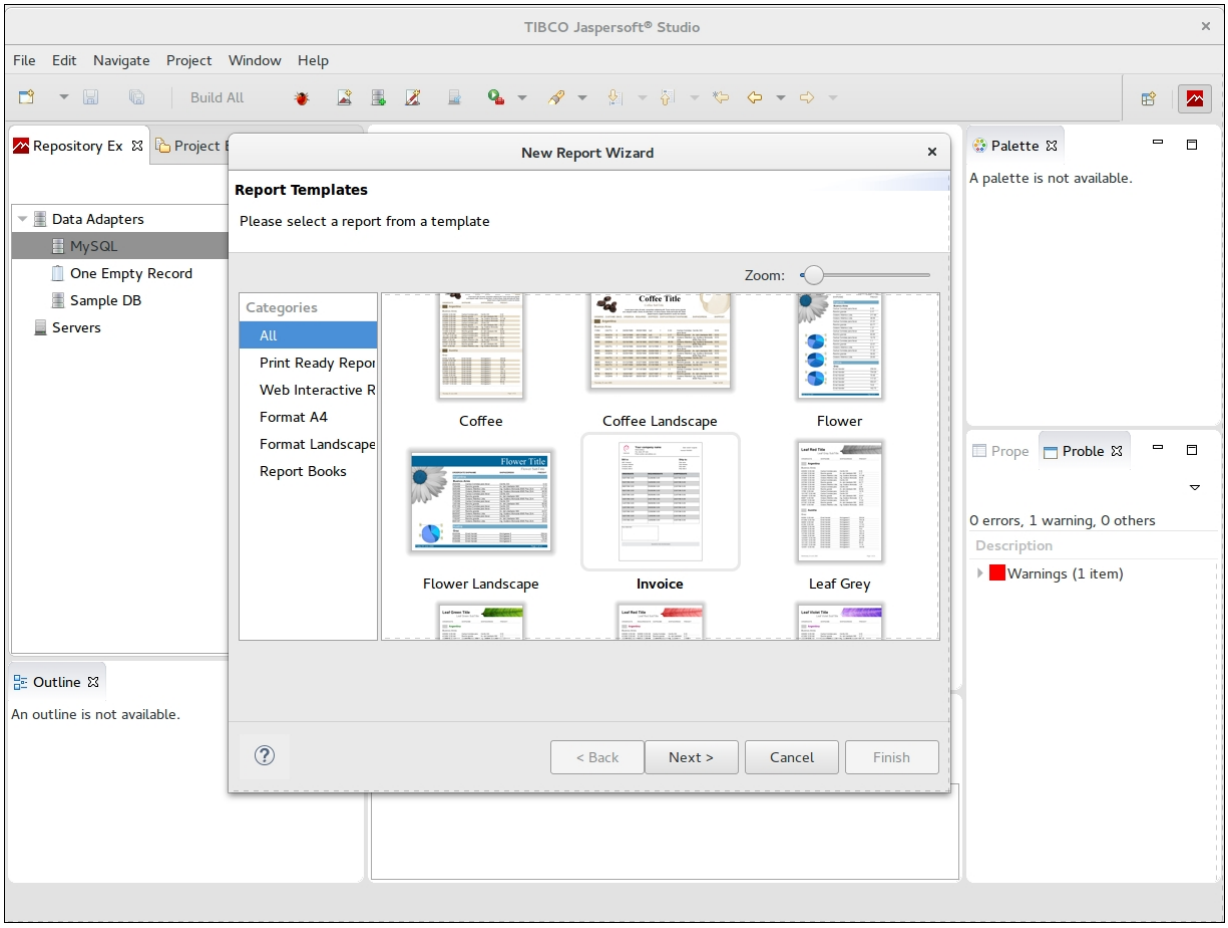
Now we can test the connection to see if it's all good.



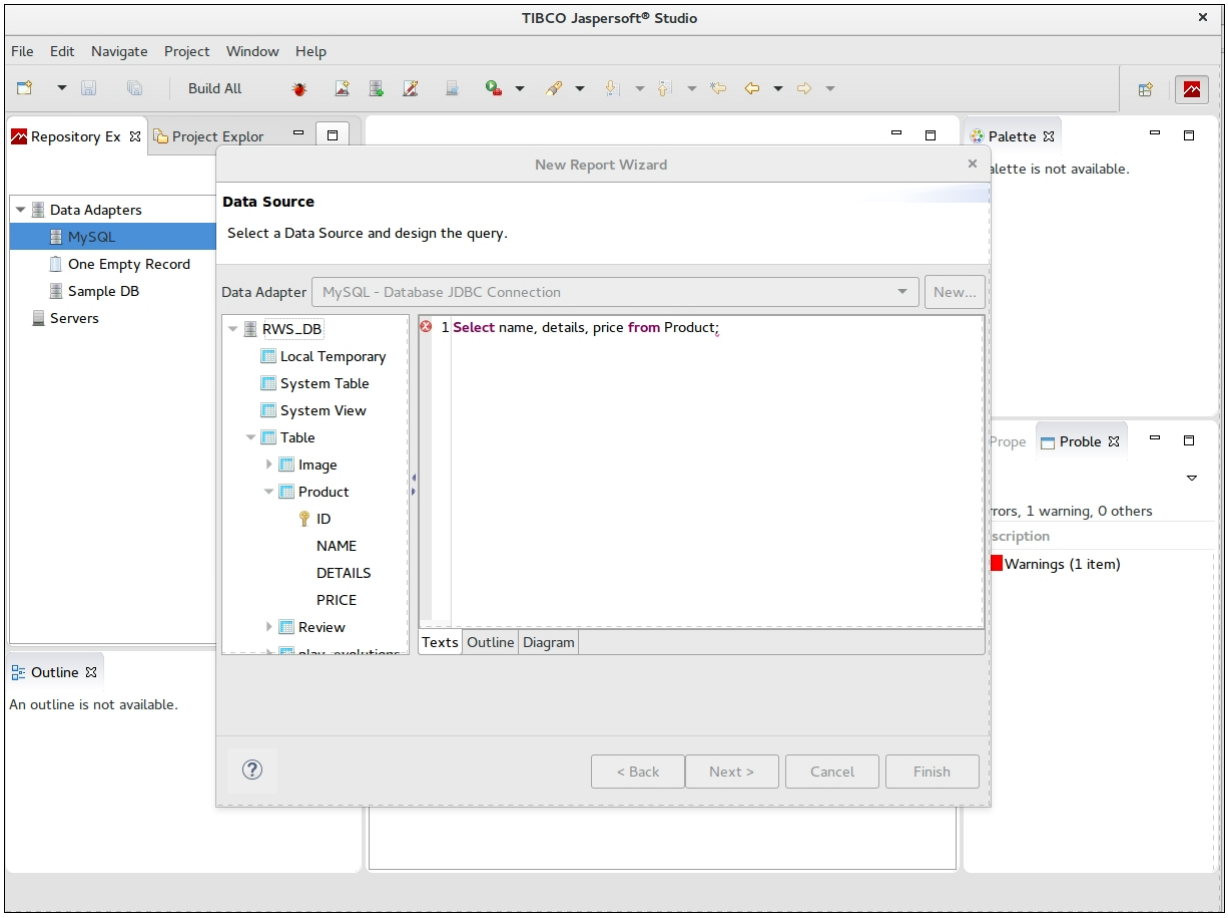
Great! Now we have our MySQL Database Adapter configured, and we are ready to start creating reports for your application.

Creating a product report

To create a product report, click on **File** | **New** | **Jasper Report**. Then select the **Invoice** template.

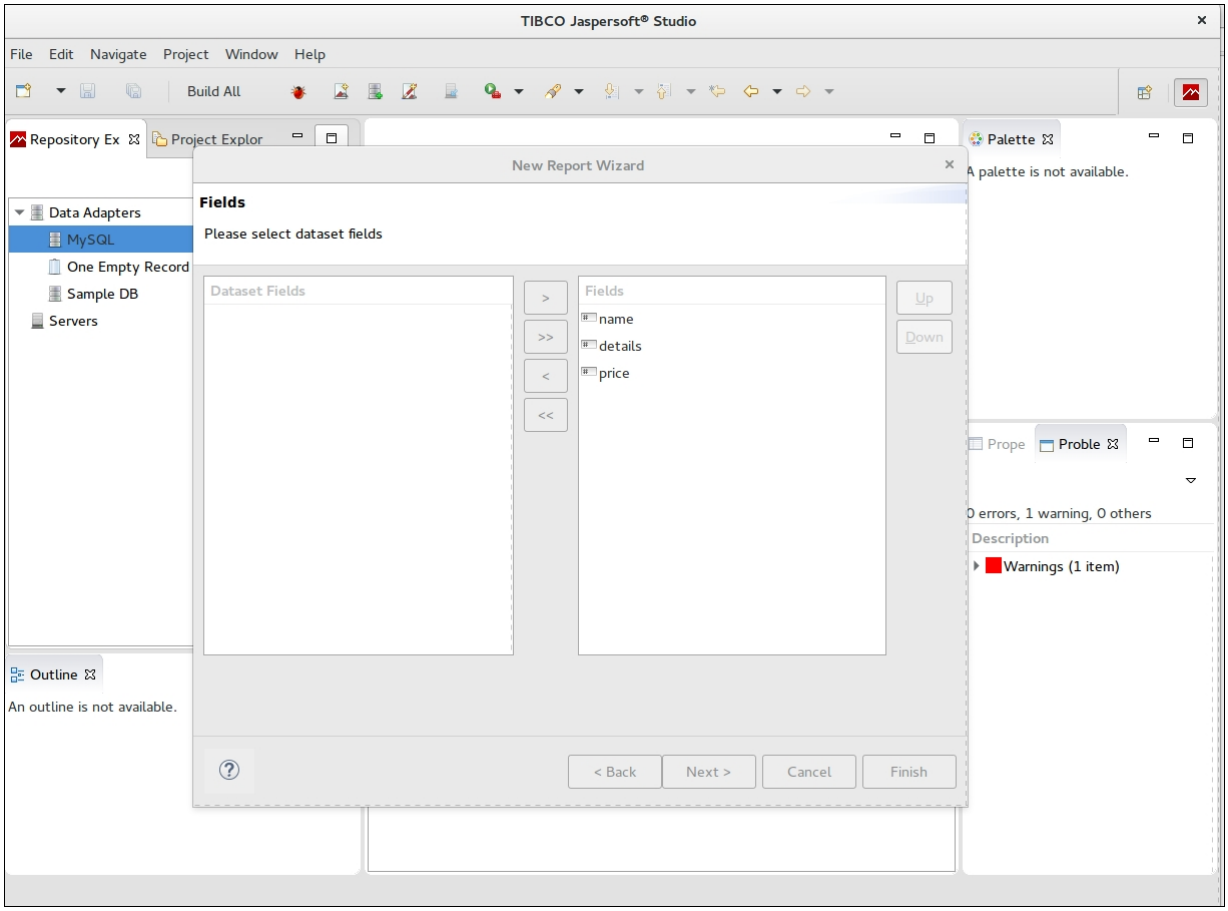


Now you can click on **Next >** and we will set up the name of the report. The file name will be `Products.jrxml`. Click on **Next >**. Then, we will need to select the **Data Source:** `MySQL`.



Now, you will need to run the `Select name, details, price from Product;` query.

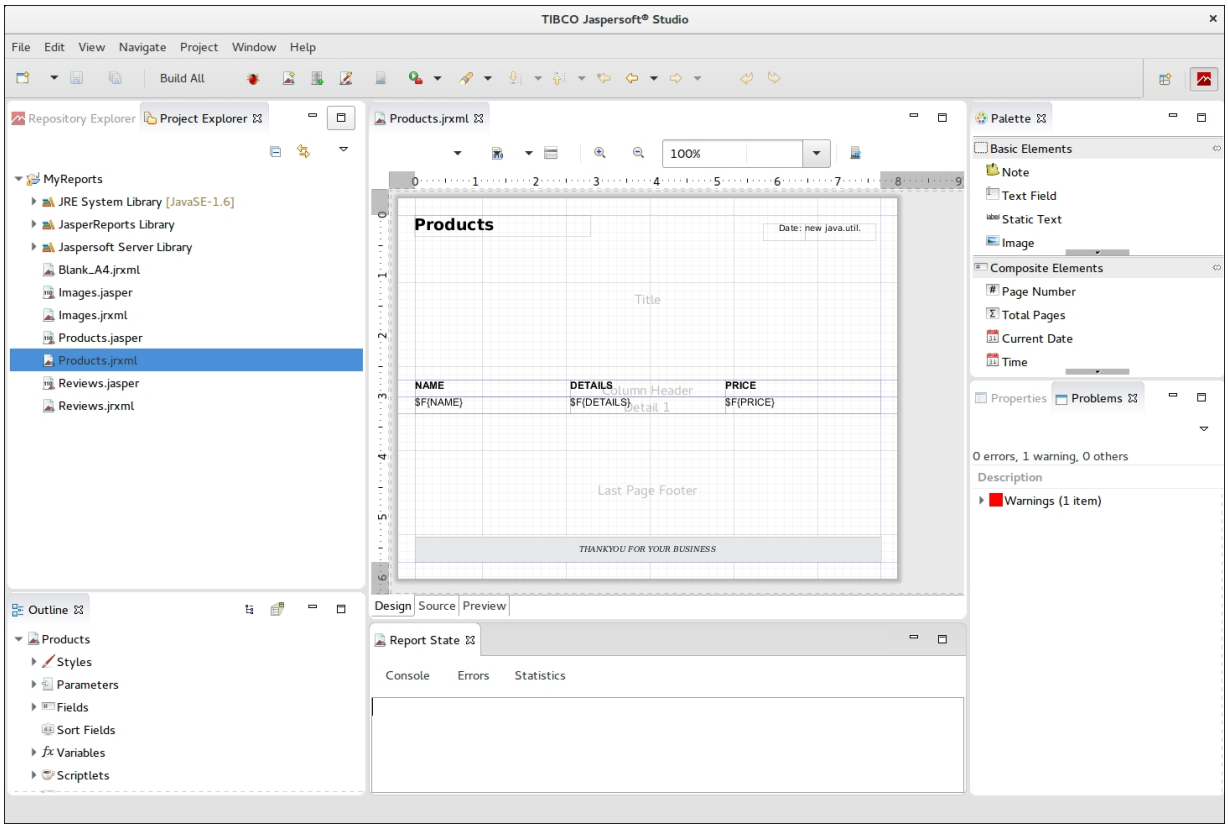
After setting the SQL query, you can click on **Next >**.



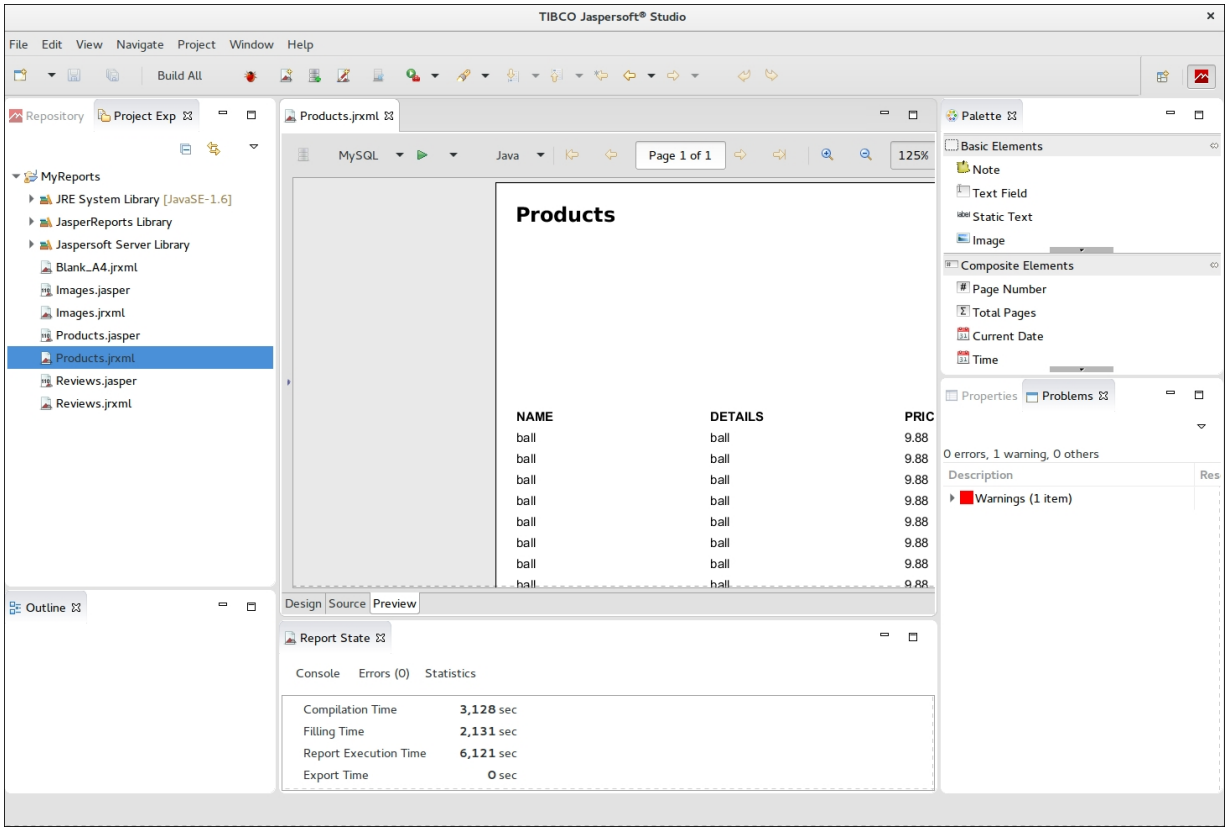
Next, you will need to pick the fields that will be used in the report. Select all the fields from the left list and move them to the right list. Then click on **Next >**. We don't need group ordering for this report, so just skip the group and click on **Next >** again.

Congratulations! We finished the setup. We can take a look (have a report preview) of the report using Jaspersoft Studio. Just click on the new report called **Products.jxml**. We will remove all the fields that we don't need, as well as the logo. Then the report will look like

this:



We will change the title to `Products` and drop all other information but the **NAME**, **DETAILS**, and **PRICE** headers, which will be retained. We will also keep the **$F{NAME}**, **$F{DETAILS}**, and **$F{PRICE}** fields, which will come from the MySQL database.



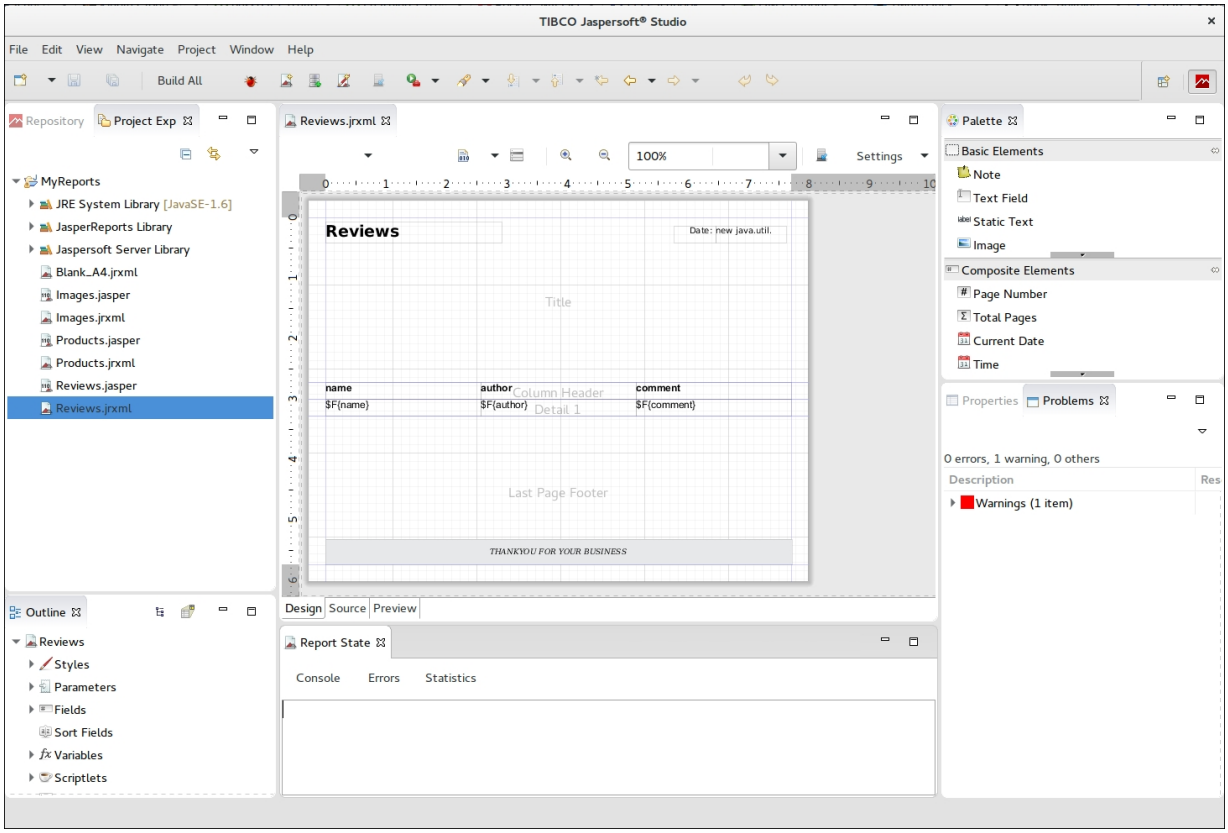
Now, we can see the report preview. We will need to click on the bottom tab named **Preview**. There are several preview options. We will have to pick **MySQL** as a data source from the top of the screen and the exported format; here, we are using **Java** to see the UI. You can also pick other formats, such as **PDF,** for instance.

Next, we will need to create reports for reviews and images.

Creating a review report

Now, we will create the review report. We will use a very similar process to the one that we used for the product report. Let's get started on creating a review report:

1. Click on **File** | **New** | **Jasper Report**. Select the **Invoice** template and click on **Next >**.
2. The file name will be `Reviews.jrxml`. Then click on **Next >**.
3. Choose **MySQL** from **Data Adapter** and click on **Next >**.
4. **Query(Text)** should contain the following code snippet: Select p.name, r.author, r.comment from Product p, Review r where p.id = r.product_id;
5. Then click **Next >**.
6. Select all the fields: **name**, **author**, and **comment** then click **Next**.>
7. Let's skip the **group by** section and click on **Next** and then **Finish**.
8. We will remove all template labels and fields and just keep the database fields, so we should have something like this:



That's it! We have the review report. If you like, you can click on the **Preview** tab at the bottom of the screen and select **MySQL** and **Java** to see the report. Keep in mind that you will need to have data; otherwise, it will be empty.

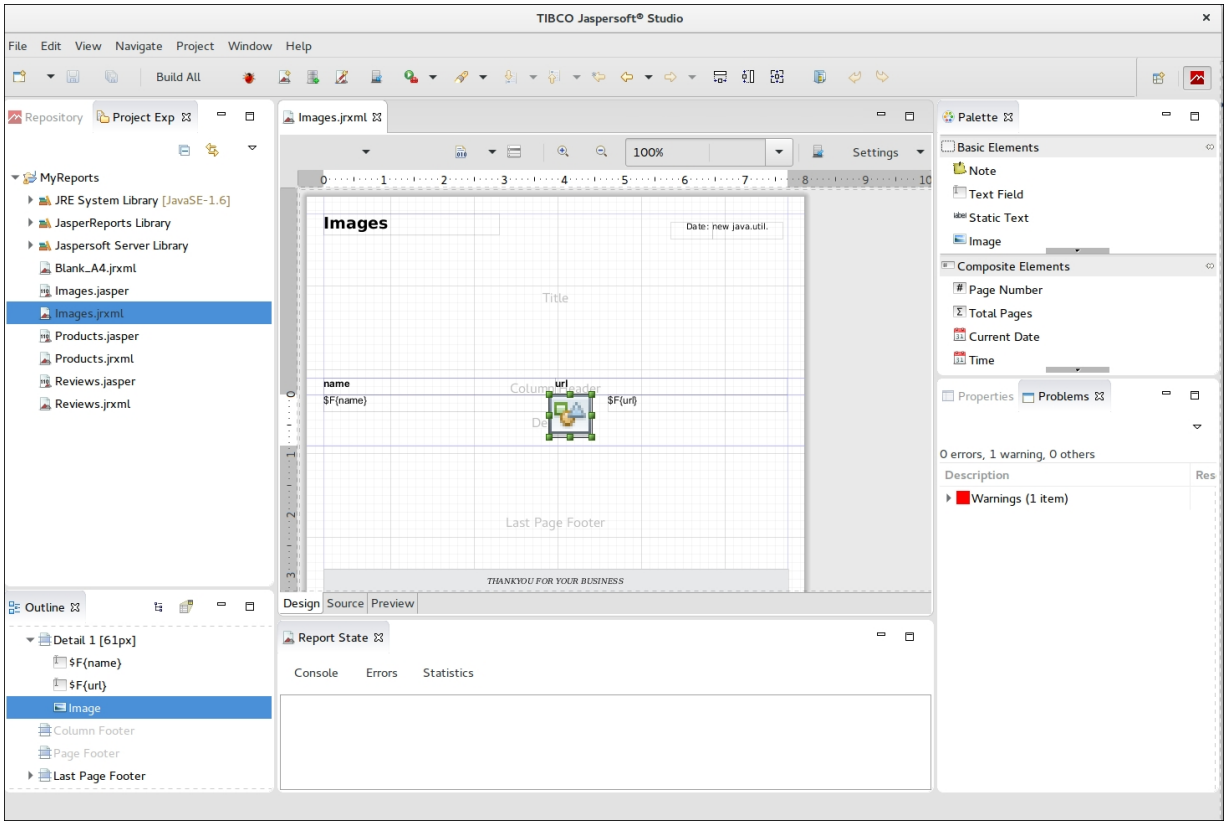
Creating an image report

Now we will create the image report. We will follow a very similar process to the one we used for the product and review report. As we have an image URL, we will also display the image, so we will need to use a different component. Let's get started on creating an image report:

1. Click on **File** | **New** | **Jasper Report**.
2. Select the **Invoice** template and click on **Next**.
3. The file name will be `Images.jrxml`. Then click on **Next**.
4. Choose **MySQL** from **Data Adapter** and click on **Next >**.
5. **Query(Text)** should contain the following code snippet: Select p.name, i.url from Image i, Product p where p.id = i.product_id;
6. Then click **Next >**.
7. Select all the fields: **name**, **url** , and then click **Next >**.
8. Let's skip the group by section and click on **Next >** and then **Finish**.

Now we will need to remove all labels and fields, as we did for the other reports, and just keep the labels and fields from the data adapter.

We need to add an image component called **Image**. You can find it in the palette at the right-hand side called **Basic Elements**. Just drag and drop it into the detail band, as shown in the following screenshot:



Select a custom expression and then type `$F{url}`.

That's it! Now that we have the image report with images, it's time to change the Play framework application in order to render this report in PDF format over there.

Integrating JasperReports with Play framework

We will need to create a new folder under `ReactiveWebStore/app` called `reports`. Then, we will copy all three new `.jrxml` files from the Jaspersoft Studio to this folder and set up the build dependencies.

build.sbt

First of all, we will need to add new dependencies to the `build.sbt` file.

Your `build.sbt` file should look like this after adding the Jasper dependencies:

```
libraryDependencies ++= Seq(
      // .... Other dependencies ....
      "net.sf.jasperreports" % "jasperreports" %
"6.2.2"  withSources()
      ,"net.sf.jasperreports" % "jasperreports-
functions" % "6.2.2",
      "net.sf.jasperreports" % "jasperreports-
chart-themes" % "6.2.2"
    )
    resolvers += "Jasper" at

"https://jaspersoft.artifactoryonline.com/jasperso
ft/repo/"
    resolvers += "JasperSoft" at

"https://jaspersoft.artifactoryonline.com/jasperso
ft/jaspersoft-
    repo/"
    resolvers += "Jasper3rd" at

"https://jaspersoft.artifactoryonline.com/jasperso
ft/
    jaspersoft-3rd-party/"
    resolvers += "mondrian-repo-cache" at
```

```
"https://jaspersoft.artifactoryonline.com/jasperso
ft/
    mondrian-repo-cache/"
    resolvers += "spring-mil" at
"http://repo.spring.io/libs-milestone"
    resolvers += "spring-rel" at
"http://repo.spring.io/libs-release"
    resolvers += "oss" at

"https://oss.sonatype.org/content/groups/public/"
```

So, basically, we added all JasperReports dependencies and resolvers, which are a bunch of remote repositories where SBT can look for the `jar` files. You can run the `$ activator compile` command on the console in order to reload the new dependencies. After running `compile`, it is important to generate eclipse files again, so you will need to run `$ activator eclipse`.

Generic report builder

Now is the time to code in Scala. We will create a generic report builder in Scala. Under `ReactiveWebStore/app/reports`, we will create a new Scala class called `ReportBuilder.scala`.

Your `ReportBuilder.scala` file should have the following code:

```
package reports
    object ReportBuilder {
      private var
reportCache:scala.collection.Map[String,Boolean] =
      new
scala.collection.mutable.HashMap[String,Boolean].e
mpty
      def generateCompileFileName(jrxml:String):
String =
      "/tmp/report_" + jrxml + "_.jasper"
      def compile(jrxml:String){
        if(reportCache.get(jrxml).getOrElse(true))
{

JasperCompileManager.compileReportToFile( new
          File(".").getCanonicalFile +
"/app/reports/" + jrxml ,
```

```
          generateCompileFileName(jrxml))
          reportCache += (jrxml -> false)
        }
      }
      def toPdf(jrxml:String):ByteArrayInputStream
= {
        try {
          val os:OutputStream = new
ByteArrayOutputStream()
          val
reportParams:java.util.Map[String,Object] = new
          java.util.HashMap()
          val con:Connection =
DriverManager.getConnection
          ("jdbc:mysql://localhost/RWS_DB?
user=root&password

=&useUnicode=true&useJDBCCompliantTimezoneShift

=true&useLegacyDatetimeCode=false&serverTimezone=U
TC")
          compile(jrxml)
          val jrprint:JasperPrint =
JasperFillManager.fillReport
          (generateCompileFileName(jrxml),
reportParams, con)
            val exporter:JRPdfExporter = new
JRPdfExporter()
          exporter.setExporterInput(new
SimpleExporterInput(jrprint))
          exporter.setExporterOutput
          (new
SimpleOutputStreamExporterOutput(os));
          exporter.exportReport()
          new ByteArrayInputStream

((os.asInstanceOf[ByteArrayOutputStream]).toByteAr
ray())
        }catch {
          case e:Exception => throw new
RuntimeException(e)
        }
      }
    }
```

First of all, we are setting a temporary directory to store the Jasper compiled files in the `generateCompileFileName` function. As you can see, we are storing the compiled reports at `/tmp/`. If you don't use Linux, you will need to change this path.

Next, we have the compile function, which receives a JRXML report in parameter. There is a report cache `Map` object to perform an on-demand cache for the Jasper files. This map has the JRXML report, is the key as a Boolean file. This solution allows you to compile reports on demand.

Finally, we have the `toPdf` function that will receive the `jrxml` function and compile the report that is needed. This function uses `DriverManager` to get the SQL connection in order to send the connection to the Jasper engine. Finally, there is the fill, process managed by `JasperFillManager`, which will receive the Jasper file and reports parameters (for us, an empty map) and the SQL connection.

After filling the report with data from database, we can export the report in PDF using the `JRPdfExporter` command. As this is a generic function, we will return a `ByteArrayInputStream`, which is an in-memory stream structure.

Now, the next step is to change our controllers in order to be able to generate reports for products, reviews, and images.

Adding the report to the product controller

We will need to change the product controller in order to expose the new report function.

Your `ProductController.scala` file, after adding the report function, should look something like this:

```
@Singleton
    class ProductController @Inject() (val
```

```
messagesApi:MessagesApi,val
    service:IProductService) extends Controller
with I18nSupport {
      //... rest of the controller code...
      def report() = Action {
        import play.api.libs.concurrent.
        Execution.Implicits.defaultContext
        Ok.chunked( Enumerator.fromStream(
        ReportBuilder.toPdf("Products.jrxml") ) )
        .withHeaders(CONTENT_TYPE ->
"application/octet-stream")
        .withHeaders(CONTENT_DISPOSITION ->
"attachment;
          filename=report-products.pdf"
        )
      }
    }
```

Right here, we have a new function called `report`. We will need to use our `ReportBuilder` method passing the `Products.jrxml` as parameter. We are using the `Ok.chunked` function in order to be able to stream the report to the browser. We are also setting some response headers, such as the content type and the name of the file, which will be reported to `products.pdf`.

Now, we will apply the same code to the review and image controllers.

Adding the report to the review controller

Now is the time to create the report function for the review controller. Here we go.

Your `ReviewController.scala` file, after adding a report function, should look something like this:

```
@Singleton
    class ReviewController @Inject()
    (val messagesApi:MessagesApi,
      val productService:IProductService,
```

```
    val service:IReviewService)
    extends Controller with I18nSupport {
      //... rest of the controller code...
      def report() = Action {
        import play.api.libs.concurrent.Execution.
        Implicits.defaultContext
        Ok.chunked( Enumerator.fromStream(
        ReportBuilder.toPdf("Reviews.jrxml") ) )
        .withHeaders(CONTENT_TYPE ->
"application/octet-stream")
        .withHeaders(CONTENT_DISPOSITION ->
"attachment;
        filename=report-reviews.pdf")
      }
    }
```

We have the same logic here as we have for the product controller. The main difference is the `jrxml` file and the filename response header. Now, we can move to the last controller--the image controller.

Adding the report to the image controller

Finally, we will apply the same logic here as we did for the product and review controller, but now it is time to change the image controller.

Your `ImageController.scala` file, after adding report function, should look something like this:

```
@Singleton
    class ImageController @Inject()
    (val messagesApi:MessagesApi,
      val productService:IProductService,
    val service:IImageService)
    extends Controller with I18nSupport {
      // ... rest of the controller code ...
      def report() = Action {
        import play.api.libs.concurrent.Execution.
        Implicits.defaultContext
        Ok.chunked( Enumerator.fromStream(
```

```
        ReportBuilder.toPdf("Images.jrxml") ) )
        .withHeaders(CONTENT_TYPE ->
"application/octet-stream")
        .withHeaders(CONTENT_DISPOSITION ->
"attachment;
        filename=report-images.pdf")
      }
    }
```

Alright, we have finished all the controllers. However, we will need to configure routes, otherwise, we won't be able to call the controllers--this is the next step.

Routes - adding new report routes

Now, we will need to add the new routes for the reports. For that, we will edit the `conf/routes` file, as follows:

```
GET     /reports
controllers.HomeController.reports
    #
    # Reports
    #
    GET  /product/report
controllers.ProductController.report
    GET  /review/report
controllers.ReviewController.report
    GET  /image/report
controllers.ImageController.report
```

We are done with routes now, and we need to change the UI in order to expose the new report functionality. We will create a new view containing all reports, and, for the sake of ease, we will also add a button for each resource UI (product, review, and image).

New centralized reports UI

We will need to create a new view at `ReactiveWebStore/views/reports_index.scala.html`.

Your `reports_index.scala.html` file should look something like this: @()(implicit flash: Flash) @main("Reports") { <img height="42" width="42" src="@routes.Assets.at("images/product.png")"> Products Report
 <img height="42" width="42" src="@routes.Assets.at("images/review.png")"> Reviews Report
 <img height="42" width="42" src="@routes.Assets.at("images/image.png")"> Images Report
 }

So here, we will basically list all resources--product, review, and images and link the relative controllers, and when the user clicks on the respective link a PDF report will be downloaded. Now we need to edit each resource (product, image, and review) view in order to add a link for the reports there as well.

Adding the report button for each view

Let's edit the product view first.

Your `product_index.scala.html` file should look something like this: @(products:Seq[Product])(implicit flash: Flash) @main("Products") { // ... rest of the ui code ... Add Product Products Report }

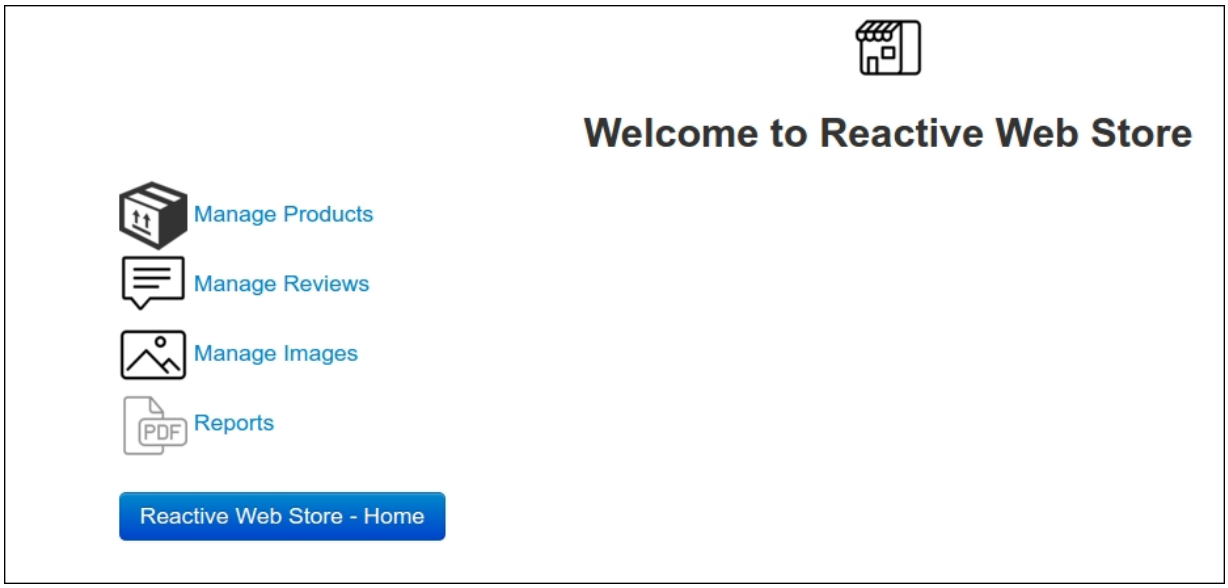
As you can see here, we added a new button pointing to the new report function. We will need to do the same for the review and the image UI.

Your `review_index.scala.html` file should look something like this: @(reviews:Seq[Review])(implicit flash: Flash) @main("Reviews") { // ... rest of the ui code ... Add Review Review Report }

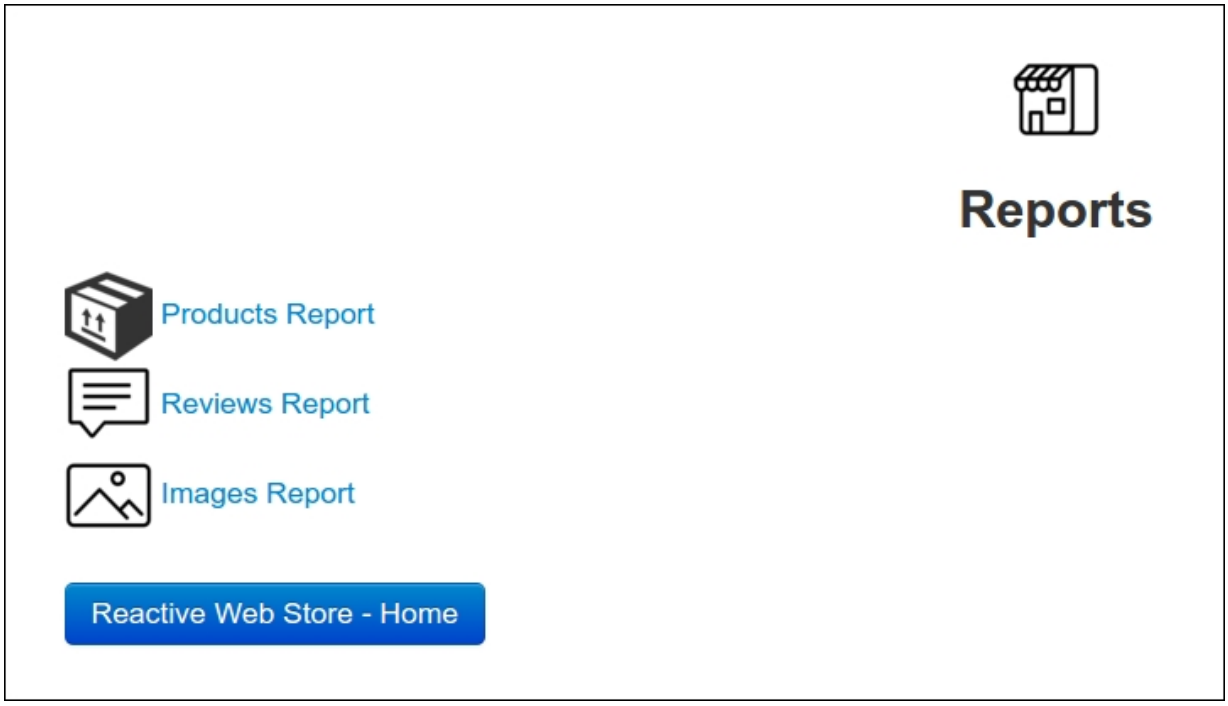
Now we can add the final button to the image view.

Your `image_index.scala.html` file should look something like this: @(images:Seq[Image])(implicit flash:Flash) @main("Images") { // ... rest of the ui template ... Add Image Images Report }

All set! Now we can run `$ activator run` and see the new UI and report buttons. Go to `http://localhost:9000/`:



If you go to `http://localhost:9000/reports`, or click on **Reports**, you will see the following:



That's it! We have all the reports working on the Play framework application.

Summary

In this chapter, you learned how to create custom reports using Jaspersoft Studio and JasperReports. Additionally, you also changed your application in order to integrate the Play framework and JasperReports.

In the next chapter, you will learn how to use the Akka framework. We will continue building our application and embrace the actor model for a new killer feature for your application.

Chapter 8. Developing a Chat with Akka

In the previous chapters, we persisted data into MySQL using Slick and wrote PDF reports using Jasper reports. Now we will add more features in our app using Akka.

In this chapter, you will learn how to create Actors using the Akka framework. We will use Actors in combination with the Play framework and WebSockets in order to have a chat capability.

We will cover the following topics in this chapter:

- Understanding the Actor model
- Actor systems, Actor routing, and dispatchers
- Mailboxes, Actor configuration, and persistence
- Creating our Chat Application
- Testing our Actors

Adding the new UI introduction to Akka

Akka (http://akka.io/) is a framework to build concurrent, distributed, and resilient message-driven applications in Scala, Java, and .NET. Building applications with Akka has several advantages, which are as follows:

- **High performance**: Akka delivers up to 50 million messages per second on a commodity hardware having ~2.5 million Actors per GB of RAM.
- **Resilient by design**: Akka systems have self-healing properties for local and remote Actors.

- **Distributed and elastic**: Akka has all the mechanisms to scale your application, such as cluster, load balancing, partitioning, and sharding. Akka lets you grow or shrink your Actors on demand.

The Akka framework provides good abstractions for concurrent, asynchronous, and distributed programming, such as Actors, Streams, and Futures. There are plenty of great success cases in production, such as BBC, Amazon, eBay, Cisco, The Guardian, Blizzard, Gilt, HP, HSBC, Netflix, and so many others.

Akka is a truly reactive framework because everything, in the sense of sending and receiving messages to Actors, is lockless, non-blocking IO, and asynchronous.

Introduction to the Actor model

The key for concurrency programming is to avoid a shared mutable state. A shared state often requires locks and synchronization, which makes your code less concurrent and more complex. Actors share nothing; they have internal state, but they don't share their internal state.

Actors have location transparency; they can run in a local or remote system and a cluster. It's also possible to mix local and remote actors - this is great for scalability and fits perfectly into a cloud environment. Actors can run anywhere, from your local box, the cloud, bare-metal datacenter, and Linux containers.

What is an Actor?

Actors can be alternatives to threads, callback listeners, singleton services, **Enterprise Java Beans** (**EJB**), routers, load balancer or pool, and a **finite-state machine** (**FSM**). The Actor model concept is not new at all; it was created by Carl Hewitt in 1973. The Actor model is heavily used in the telecom industry in rock-solid technologies such as Erlang. Erlang and the Actor model had immense success with companies such as Ericsson and Facebook.

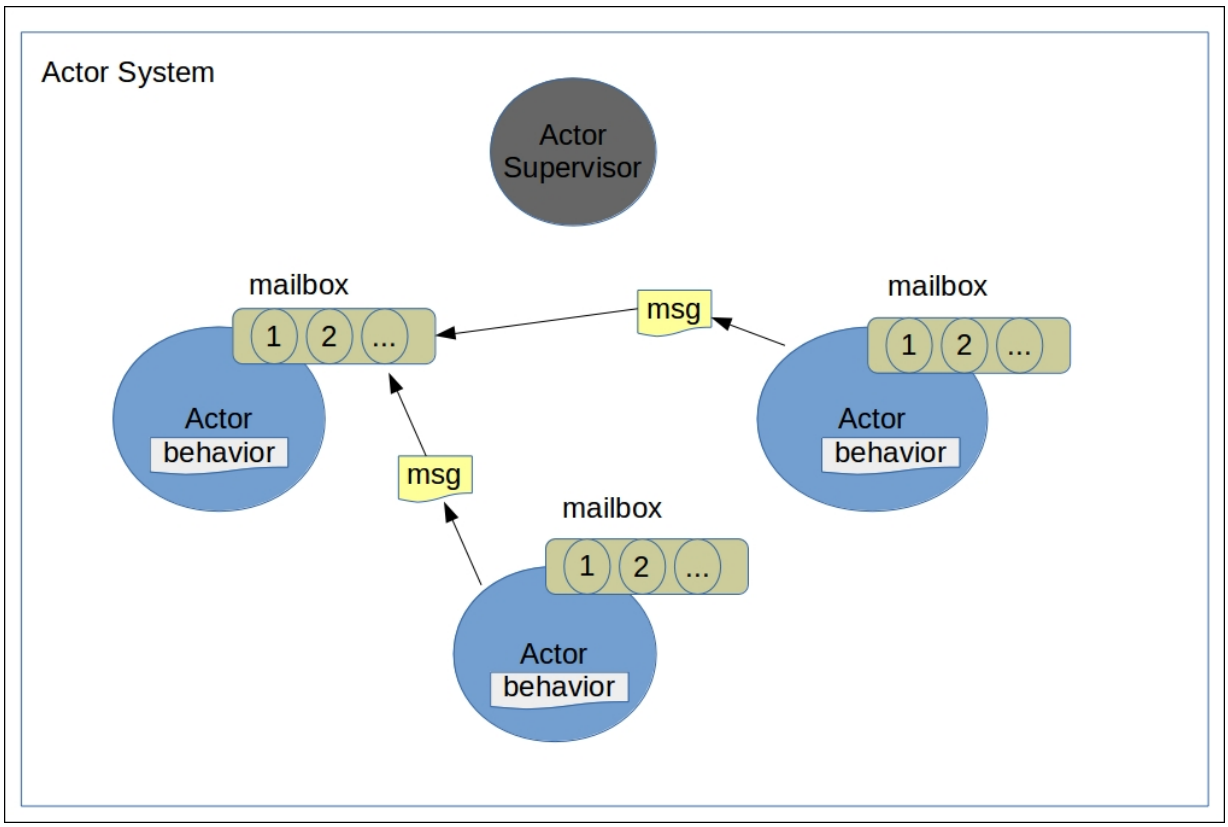
Actors have a simple way of working:

- Unit of code organization:
    - Processing
    - Storage
    - Communication
        - They manage the internal states
        - They have a mailbox
        - They communicate with other actors using messages
        - They can change the behavior at runtime

Message exchange and mailboxes

Actors talk with each other via messaging. There are two patterns: one is called ask and the other is called fire and forget. Both methods are asynchronous and non-blocking IO. When an Actor sends a message to another Actor, it does not send the message directly to the other Actor; it actually sends it to the Actor's mailbox.

Messages are enqueued in the Actor mailbox in a time-ordered fashion. There are different mailboxes implementations in Akka. The default is **First In First Out** (**FIFO**) based. This is a good default; however, you might need a different algorithm, which is fine as you can change the mailbox if you need to. More details can be found in the official documentation (http://doc.akka.io/docs/akka/2.4.9/scala/mailboxes.html#mailboxes-scala). Actors live in an Actor system. You can have multiple Actor systems in a cluster:



Akka encapsulates the actor state in mailbox and decouples it from the Actor behavior. The Actor behavior is the code you will have inside your Actor. You will need to see Actors and Akka as a protocol. So, basically, you will need to define how many Actors you will have and what each Actor will do in the sense of code, responsibility, and behavior. The Actor system has Actors and supervisors. Supervisors are one of the Akka mechanisms to deliver fault tolerance and resiliency. Supervisors take care of the Actor instances, and they can restart, kill, or create more Actors as needed.

The Actor model is great for concurrency and scalability; however, like every single thing in computer science, there are tradeoffs and cons. For instance, Actors require a new mindset and a different way of thinking.

There is no silver bullet. Once you have your protocol, it might be hard to reuse your Actors outside your protocol. In general, Actors

can be harder to compose, as compared to object-oriented classes or functions in Functional Programming, for instance.

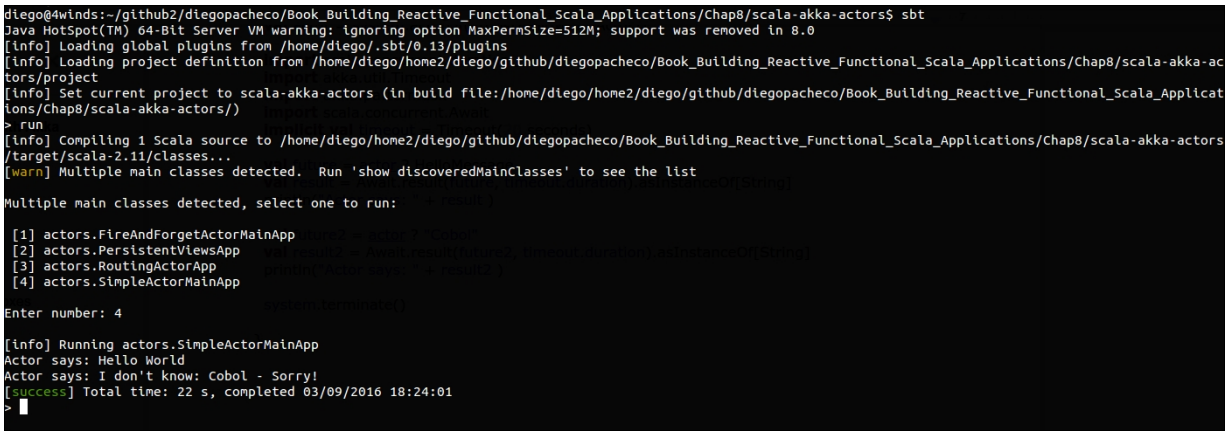
Coding actors with Akka

Let's take a look at the following Actor code using the Akka framework and Scala:

```
import akka.actor._
    case object HelloMessage
    class HelloWorldActor extends Actor {
      def receive = {
        case HelloMessage => sender() ! "Hello
World"
        case a:Any => sender() ! "I don't know: "
+ a + " - Sorry!"
      }
    }
    object SimpleActorMainApp extends App{
      val system =
ActorSystem("SimpleActorSystem")
      val actor =
system.actorOf(Props[HelloWorldActor])
      import scala.concurrent.duration._
      import akka.util.Timeout
      import akka.pattern.ask
      import scala.concurrent.Await
      implicit val timeout = Timeout(20 seconds)
      val future = actor ? HelloMessage
      val result = Await.result(future,
      timeout.duration).asInstanceOf[String]
      println("Actor says: " + result )
      val future2 = actor ? "Cobol"
      val result2 = Await.result(future2,
      timeout.duration).asInstanceOf[String]
      println("Actor says: " + result2 )
      system.terminate()
    }
```

If you run this Akka code on `sbt` in your console, you will see an output similar to this:

```
$ sbt run
```



Let's take a closer look at this Akka code we just wrote, in which we defined a Scala class called `HelloWorldActor`. In order for this class be an Actor, we will need to extend Actor. Actors are reactive by default, which means that they are waiting to receive messages to react to the messages. You will need to code your behavior in an event loop. In Akka, this is done by coding the `receive` function with a pattern matcher in Scala.

The pattern matcher will define what the actor can do. You will need to code all the possible kinds of messages you want that actor to handle. As I mentioned earlier, you will need to have a protocol; so your protocol has a message called `HelloMessage`. It's a common practice in Akka to use Scala objects as messages. However, you can pass pretty much all types as messages. It's even possible to send case classes with parameters.

Alright, we have our protocol, which is our Actors, and the messages they can exchange. Now we will need to create an Actor system and start our application. As you can see, we will use the `ActorSystem` object to create an Actor system. Actor systems need to have a name, which can be any string you like, as long as it contains any letter [a-z, A-Z, 0-9] and non-leading '-' or '_'.

After creating the system, you can create Actors. The system has a function called `actorOf`, which can be used to create Actors. You will

need to use a special object called `Props` and pass the `actor` class. Why do we need it this way? It's because Akka manages the Actor state. You should not try to manage the Actor instance by yourself. This is dangerous because you can break referential transparency and your code might not work.

For this code, we are using the ask pattern. We will use this to send messages to the Actor, and we want to know what the Actor will return. Akka does everything in an async and non-blocking way , as mentioned previously. However, sometimes you want to get the answer now and then, unfortunately, you will need to block.

In order to get the answer now, we will need to define a timeout and use the `Await` object. When you send a message to an Actor using `?` (the ask pattern), Akka will return a Future for you. Then, you can pass the Future with a timeout to `Await`, and if the answer comes back before the timeout, you will have the response from the Actor.

Again, we are blocking here because we want to get the answer now, and we are outside the Actor system. Keep in mind that when an Actor talks with another Actor inside the Actor system, it should not block ever. So be careful with the usage of `Await`.

Another important thing in this code is that the `sender()` method inside of the Actor receives a function. This means that you want to get the reference of the Actor who sends the message to you. As we are performing `sender() !` method, we are sending an answer back to the caller. The `sender()` function is an Akka abstraction to deal with response messages to other Actors or function callers.

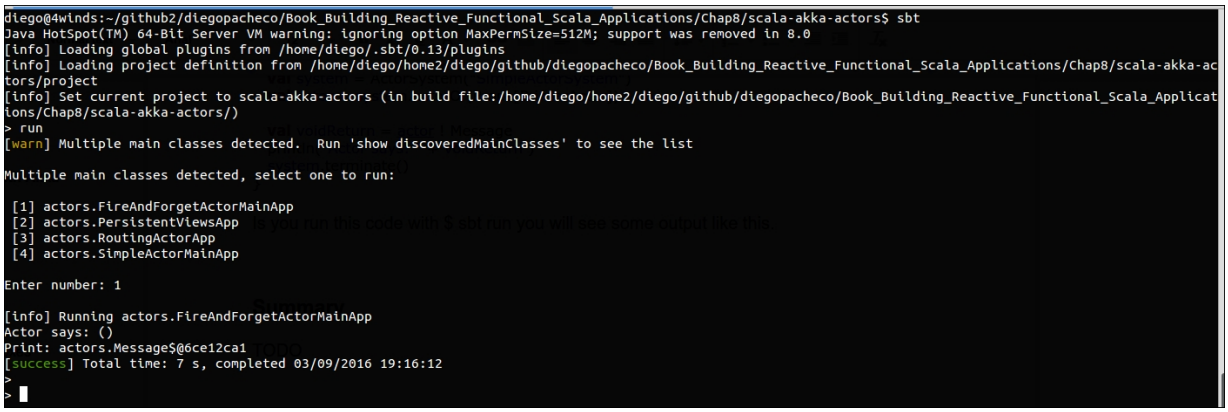
We also have another case, with `Any`, which means all other messages will be handled by that case code.

The ask pattern is one way to send messages to Actors. There is another pattern called `FireAndForget "!"`. Fire and forget will send a message and will not block and wait for the answer. So, there is no answer - in other words, Unit.

Let's look at some code with the `FireAndForget` message exchange:

```
import akka.actor._
    object Message
    class PrinterActor extends Actor {
      def receive = {
        case a:Any =>
        println("Print: " + a)
      }
    }
    object FireAndForgetActorMainApp extends App{
      val system =
ActorSystem("SimpleActorSystem")
      val actor =
system.actorOf(Props[PrinterActor])
      val voidReturn = actor ! Message
      println("Actor says: " + voidReturn )
      system.terminate()
    }
```

If you run this code with `$ sbt run`, you will see an output as follows:



Here, we have a `PrinterActor` method, which accepts pretty much anything and prints on the console. Then, we will create an Actor system and just send a message to our Actor with the fire and forget pattern, a.k.a "!", and as you can see, we will receive Unit; finally, we

will await the shutdown of the Actor system using the `terminate` option.

Actor routing

Akka provides routing functionality. This is useful from a business point of view because you can route to the right Actor in the sense of business logic and behavior. For architecture, we can use this as load balancing and route messages to more Actors to achieve fault tolerance and scalability.

Akka has several options for routing, which are as follows:

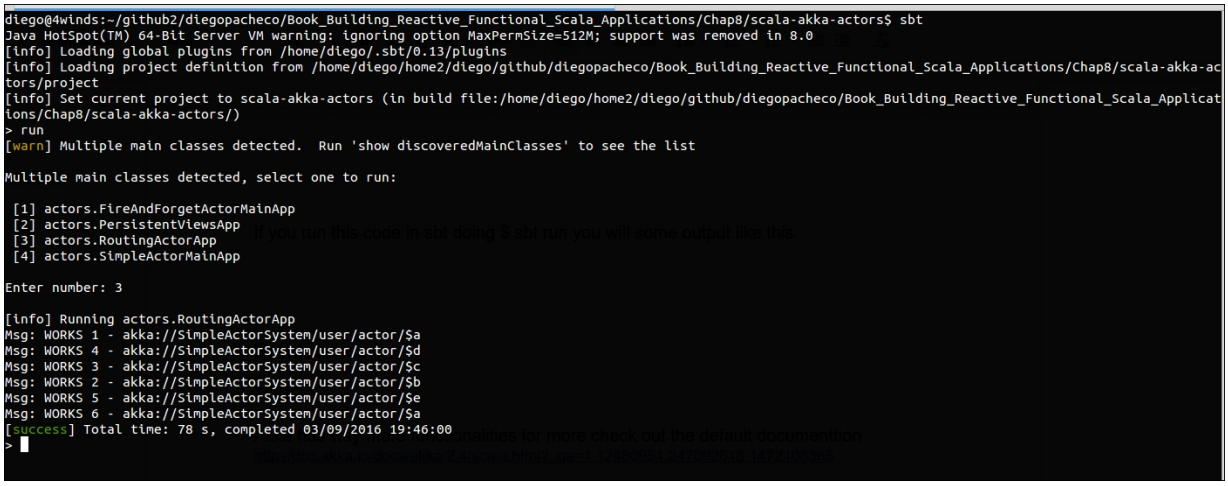
- **RoundRobin**: This is a random logic to every different Actor on the pool.
- **SmallestMailbox**: This sends the message to the Actor with fewer messages.
- **Consistent Hashing**: This partitions the Actors per hash ID.
- **ScatterGather**: This sends message to all actors, and the first to reply wins.
- **TailChopping**: This sends to a route randomly, and if a reply doesn't come back in a second, it chooses a new route and sends again, and so on.

Let's see the following code in practice:

```
import akka.actor._
    import akka.routing.RoundRobinPool
    class ActorUpperCasePrinter extends Actor {
      def receive = {
        case s:Any =>
        println("Msg: " +
s.toString().toUpperCase() + " - " +
        self.path)
      }
    }
    object RoutingActorApp extends App {
      val system =
ActorSystem("SimpleActorSystem")
      val actor:ActorRef = system.actorOf(

RoundRobinPool(5).props(Props[ActorUpperCasePrinte
r]),name =
```

```
        "actor")
        try{
          actor ! "works 1"
          actor ! "works 2"
          actor ! "works 3"
          actor ! "works 4"
          actor ! "works 5"
          actor ! "works 6"
        }catch{
          case e:RuntimeException =>
println(e.getMessage())
        }
        system.terminate()
      }
```

If you run this code in sbt doing `$ sbt run`, you will get an output as follows:



So, here we have an `ActorUppercasePrinter` function that prints whatever it receives and calls the `toString` function, and then `toUpperCase`. Finally, it also prints the `self.path` Actor, which will be the address of the Actor. Actors are structured in a hierarchical structure, similar to a file system.

There are multiple ways to use Akka - Akka supports code or configuration (`application.conf` file). Here, we are creating a round-robin pool actor that has five routes. We are passing the target Actor to the router that will be our printer Actor.

As you can see, when we send messages using the fire and forget pattern, every message is delivered to a different Actor.

Persistence

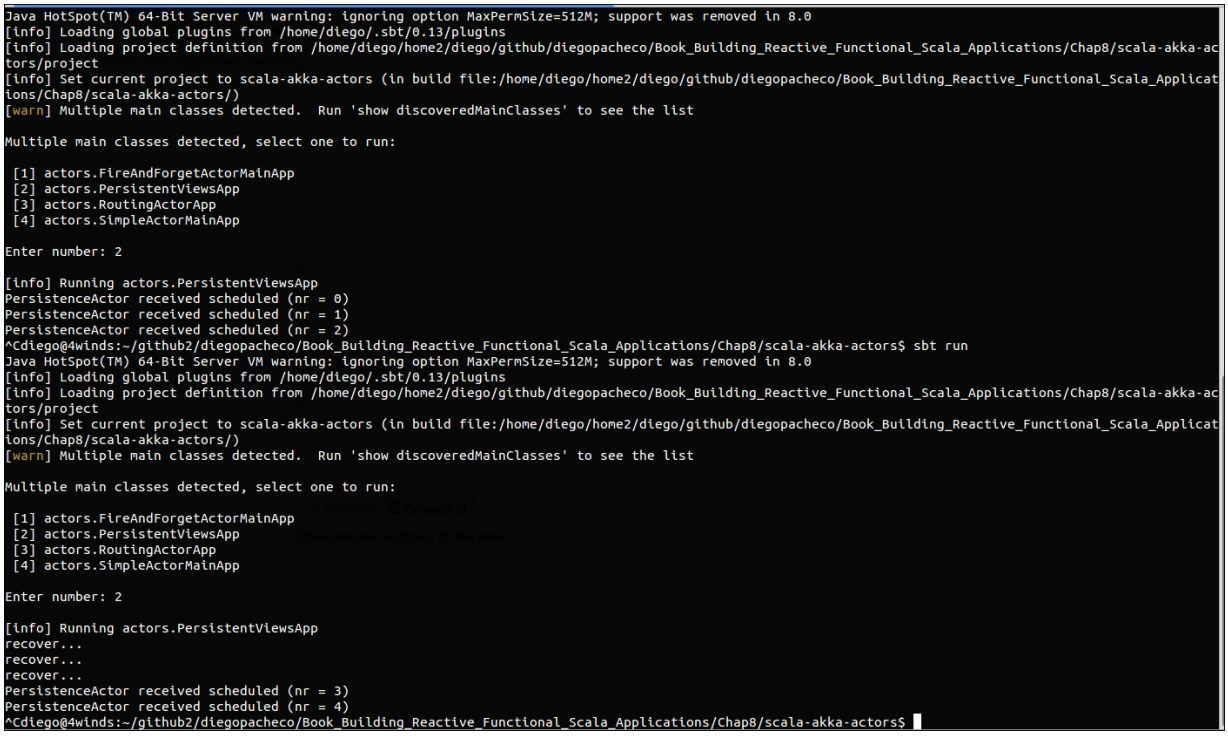
Akka works on memory. However, it is possible to use persistence. Persistence is still kind of experimental in Akka. However, it is stable. For production, you can use advanced persistence plugins, such as Apache Cassandra. For the sake of development and education, we will use Google leveldb in our file system. Akka has multiple persistence options, such as views and persistent Actors.

Let's take a look at a persistent actor using the Google leveldb and file system:

```
import akka.actor._
    import akka.persistence._
    import scala.concurrent.duration._
    class PersistenceActor extends
PersistentActor{
      override def persistenceId = "sample-id-1"
      var state:String = "myState"
      var count = 0
      def receiveCommand: Receive = {
        case payload: String =>
        println(s"PersistenceActor received
${payload} (nr =
        ${count})")
        persist(payload + count) { evt =>
          count += 1
        }
      }
      def receiveRecover: Receive = {
        case _: String =>
        println("recover...")
        count += 1
      }
    }
    object PersistentViewsApp extends App {
      val system =
ActorSystem("SimpleActorSystem")
      val persistentActor =

system.actorOf(Props(classOf[PersistenceActor]))
      import system.dispatcher
```

```
        system.scheduler.schedule(Duration.Zero,
    2.seconds,
        persistentActor, "scheduled")
      }
```

Executing the `$ sbt run` command will give you the following output:



If you run this code with `$ sbt run`, and stop and run again, you will see the data is being stored and recovered every time you stop and start again.

As you can see, your Actor needs to extend `PersistentActor` in order to have persistence support. You will also need to provide a `persistenceID`.

Here, you will need to implement two receive functions. One is for commands (also known as messages), and the other one is for

recovery. The command's receive loop will be activated when this Actor receives messages, while the recover one will be activated when the Actor boots up and will read the persistent data from the database.

So, this Actor here has a counter to count each message it receives, and prints every message it gets on the console. That's it; as you can see, it is pretty simple. In order to use this functionality, you will also need to configure your `application.conf`.

Your `application.conf` file should look something like this:

```
akka {
      system = "SimpleActorSystem"
      remote {
        log-remote-lifecycle-events = off
        netty.tcp {
          hostname = "127.0.0.1"
          port = 0
        }
      }
    }
    akka.cluster.metrics.enabled=off
    akka.persistence.journal.plugin =
    "akka.persistence.journal.leveldb"
    akka.persistence.snapshot-store.plugin =
    "akka.persistence.snapshot-store.local"
    akka.persistence.journal.leveldb.dir =
"target/persistence/journal"
    akka.persistence.snapshot-store.local.dir =
    "target/persistence/snapshots"
    # DO NOT USE THIS IN PRODUCTION !!!
    # See also
https://github.com/typesafehub/activator/issues/28
7
    akka.persistence.journal.leveldb.native =
false
```

So, here we are defining a simple Akka system (local mode), and we are configuring the persistence for Google leveldb. As you can see, we will need to provide a path for persistence, and this path must exist on the OS.

As we are using an additional functionality, we will also need to change `build.sbt` in order to import all jars that we will need in the sense of Akka, persistence, and leveldb.

Your `build.sbt` file should look something like this:

```
// rest of the build.sbt file ...
    val akkaVersion = "2.4.9"
    libraryDependencies += "com.typesafe.akka" %%
"akka-actor" %
    akkaVersion
    libraryDependencies += "com.typesafe.akka" %%
"akka-kernel" %
    akkaVersion
    libraryDependencies += "com.typesafe.akka" %%
"akka-remote" %
    akkaVersion
    libraryDependencies += "com.typesafe.akka" %%
"akka-cluster" %
    akkaVersion
    libraryDependencies += "com.typesafe.akka" %%
"akka-contrib" %
    akkaVersion
    libraryDependencies += "com.typesafe.akka" %%
"akka-persistence" %
    akkaVersion
    libraryDependencies += "org.iq80.leveldb" %
"leveldb" % "0.7"
    libraryDependencies += "org.iq80.leveldb" %
"leveldb-api" % "0.7"
    libraryDependencies +=
"org.fusesource.leveldbjni" % "leveldbjni" %
    "1.8"
    libraryDependencies +=
"org.fusesource.leveldbjni" % "leveldbjni-
    linux64" % "1.8"
    libraryDependencies += "org.fusesource" %
"sigar" % "1.6.4"
    libraryDependencies += "org.scalatest" %
"scalatest_2.11" % "2.2.6"
```

That's it. That's all we need to persist the Actor's state.

Note

Akka has way more functionalities. For more, check out the default documentation at http://doc.akka.io/docs/akka/2.4/scala.html?_ga=1.12480951.247092618.1472108365.

Creating our chat application

Now that we know Akka better, we will continue to develop our application. Akka has a great integration with the Play framework. We will use Actors with the Akka and Play framework right now. Let's build a simple chat feature for our app. We will change the code to add a new UI and, using the Akka testkit, we will test out actors.

The Play framework already includes Akka on the classpath for us, so we don't need to worry about it. However, we will need to add the Akka testkit dependency to the `build.sbt` file in order to have the classes in our classpath.

Your `build.sbt` should look something like this:

```
// rest of the build.stb ...
    libraryDependencies ++= Seq(
      "com.typesafe.akka" %% "akka-testkit" %
"2.4.4" % Test,
    // rest of the deps ...
    )
    // rest of the build.stb ...
```

Okay, now you can go to the console and type `$ activator`, `$ reload`, and then `$ compile`. This will force sbt to download the new dependency.

Now we will need to create a package called Actors. This package needs to be located at `ReactiveWebStore/app/`. We will start creating an `ActorHelper` utility object in order to have a generic function for the ask pattern that we saw earlier. It is an Actor helper generic ask pattern utility.

Your `ActorHelper.scala` file should look something like this:

```
package actors
    object ActorHelper {
      import play.api.libs.concurrent.
```

```
        Execution.Implicits.defaultContext
        import scala.concurrent.duration._
        import akka.pattern.ask
        import akka.actor.ActorRef
        import akka.util.Timeout
        import scala.concurrent.Future
        import scala.concurrent.Await
        def get(msg:Any,actor:ActorRef):String = {
          implicit val timeout = Timeout(5 seconds)
          val result = (actor ?
msg).mapTo[String].map { result =>
          result.toString }
          Await.result(result, 5.seconds)
        }
      }
```

The `ActorHelper` has just one function: `get`. This function will get an answer from any Actor given in any message. However, as you can see, we have a timeout of five seconds. If the result does not come back in this time, an exception will be raised.

In this code, we are also mapping the Actor result to a String calling the `toString` function in the result future. This is not a lot of code; however, there are lots of imports, and it makes the code cleaner and we can get answers from Actors with less code and fewer imports.

The chat protocol

Now we will need to define our protocol. For this functionality, we will need three Actors. The Actors that we create will be as follows:

- **ChatRoom**: This will have a reference for all users in the chat room
- **ChatUser**: This will have one instance per user (active browser)
- **ChatBotAdmin**: This simple Bot Admin will provide stats about the chat room

`ChatUserActor` will need to join `JoinChatRoom` object in order to start chatting. `ChatUserActor` will also need to send messages to `ChatMessage` class to the `ChatRoomActor` that will broadcast messages to all users. The `ChatBotAdmin` will get a report from `GetStats` object from `ChatRoomActor`.

Let's start coding this protocol. First, we will need to define the messages that will be exchanged between these Actors, as shown in the following piece of code:

```
package actors
    case class ChatMessage(name:String,text:
String)
    case class Stats(users:Set[String])
    object JoinChatRoom
    object Tick
    object GetStats
```

As you can see here, we have a `ChatMessage` class with a name and a text. This will be the message each user will send on the chat. Then, we will have a stats class, which has a set of users--this will be all the users logged into the chat application.

Finally, we have some action messages, such as `JoinChatRoom`, `Tick`, and `GetStats`. So, `JoinChatRoom` will be sent by `ChatUserActor` to `ChatRoomActor` in order to join the chat. `Tick` will be a scheduled

message that will happen from time to time in order to make `ChatBotAdmin` send stats about the chat room to all logged users. `GetStats` is the message that `ChatBotAdminActor` will send to `ChatRoomActor` in order to get information about who is in the room.

Let's code our three actors now.

The `ChatRoomActor.scala` file should look something like this:

```
package actors
    import akka.actor.Props
    import akka.actor.Terminated
    import akka.actor.ActorLogging
    import akka.event.LoggingReceive
    import akka.actor.Actor
    import akka.actor.ActorRef
    import play.libs.Akka
    import akka.actor.ActorSystem
    class ChatRoomActor extends Actor with
ActorLogging {
      var users = Set[ActorRef]()
      def receive = LoggingReceive {
        case msg: ChatMessage =>
        users foreach { _ ! msg }
        case JoinChatRoom =>
        users += sender
        context watch sender
        case GetStats =>
        val stats:String = "online users[" +
users.size + "] - users["
        + users.map( a =>
a.hashCode().mkString("|") + "]"
        sender ! stats
        case Terminated(user) =>
        users -= user
      }
    }
    object ChatRoomActor {
      var room:ActorRef = null
      def apply(system:ActorSystem) = {
        this.synchronized {
          if (room==null) room =
system.actorOf(Props[ChatRoomActor])
          room
        }
```

```
    }
  }
```

`ChatRoomActor` has a var called `users`, which is a set of `ActorRef`. `ActorRef` is a generic reference to any actor. We have the receive function with three cases: `ChatMessage`, `JoinChatRoom`, and `GetStats`.

A `JoinChatRoom` will be sent by the `ChatUserActor` method in order to join the room. As you can see, we are getting the `ActorRef` method from the sender Actor using the `sender()`function, and we are adding this reference to the set of users. In this way, the set of `ActorRef` represents the online logged-in users in the chat room right now.

The other case is with the `ChatMessage` method. Basically, we will broadcast the message to all users in the chat. We do this because we have the reference for all actors in users. Then, we will call the `foreach` function in order to iterate all users one by one, and then we will send the message using `FireAndForget "!"` to each user Actor represented by the operator underscore `_`.

The `GetStats` case creates a string with all chat room stats. For now, the stats are just the number of online users, which is computed by calling the size function on the users object. We are also showing all the hash codes that identify all Actors logged in, just for fun.

That's our `ChatRoomActor` implementation. As you can see, it is hard to talk about one Actor without describing the other, as the protocol will always be kind of coupled. You might also be wondering why we have a companion object for the `ChatRoomActor` method.

This object is to provide an easy way to create Actor instances. We are creating a single room for our design; we don't want to have multiple chat rooms, so that's why we will need to control the creation of the room Actor.

If the room is null, we will create a new room; otherwise, we will return the cached instance of the room that we already got in the memory. We will need an instance of the Actor system in order to

create actors, so that's why we are receiving the system on the `apply` function. The `apply` function will be called when someone writes a code like `ChatRoomActor(mySystem)`.

Now, let's move to the ChatUserActor implementation.

The `ChatUserActor.scala` file should look something like this:

```
package actors
    import akka.actor.ActorRef
    import akka.actor.Actor
    import akka.actor.ActorLogging
    import akka.event.LoggingReceive
    import akka.actor.ActorSystem
    import akka.actor.Props
    class ChatUserActor(room:ActorRef,
out:ActorRef) extends Actor with
    ActorLogging {
      override def preStart() = {
        room ! JoinChatRoom
      }
      def receive = LoggingReceive {
        case ChatMessage(name, text) if sender ==
room =>
        val result:String = name + ":" + text
        out ! result
        case (text:String) =>
        room ! ChatMessage(text.split(":")(0),
text.split(":")(1))
        case other =>
        log.error("issue - not expected: " +
other)
      }
    }
    object ChatUserActor {
      def props(system:ActorSystem)(out:ActorRef)
= Props(new
      ChatUserActor(ChatRoomActor(system), out))
    }
```

This Actor is a little bit easier than the previous one. `ChatUserActor` receives, as a parameter, the room actor reference and also an out actor. The room will be an instance of the room that the user will use to communicate with other users. The `ActorRef` method called out is

the Play framework Actor responsible for sending the answer back to the controllers and UI.

We pretty much just have two cases: one where we receive a `ChatMessage` and the other is the `ChatUserActors` method in the chat room. So, we will just need to send back to the UI using the out Actor. That's why there is a fire and forget message for the out Actor with a result. Using a new Actor model can be dangerous, please read more at http://doc.akka.io/docs/akka/current/scala/actors.html.

There is another case that just receives a string that will be the message from that Actor itself. Remember that each Actor represents a user and a browser full of duplex connections via WebSockets. Don't worry about WebSockets now; we will cover it in more detail later in this chapter.

For this case function, we are sending the `ChatMessage` method to the room. We will split the messages in two parts: the username and the text, which is split by `:`.

Here, we also have a companion object for the sake of good practice. So, you can call `ChatUserActor`, passing the Actor system and a curried parameter for the out actor.

Now, we will move to the last Actor: the Bot Admin Actor, which should look something like this:

```
package actors
    import akka.actor.ActorRef
    import akka.actor.Actor
    import akka.actor.ActorLogging
    import akka.event.LoggingReceive
    import akka.actor.ActorSystem
    import akka.actor.Props
    import scala.concurrent.duration._
    class ChatBotAdminActor(system:ActorSystem)
extends Actor with
    ActorLogging {
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
      val room:ActorRef = ChatRoomActor(system)
```

```
      val cancellable =
system.scheduler.schedule(0 seconds,
      10 seconds, self , Tick)
      override def preStart() = {
        room ! JoinChatRoom
      }
      def receive = LoggingReceive {
        case ChatMessage(name, text) => Unit
        case (text:String) => room !
ChatMessage(text.split(":")(0),
        text.split(":")(1))
        case Tick =>
        val response:String = "AdminBot:" +
ActorHelper.get
        (GetStats, room)
        sender() ! response
        case other =>
        log.error("issue - not expected: " +
other)
      }
    }
    object ChatBotAdminActor {
      var bot:ActorRef = null
      def apply(system:ActorSystem) = {
        this.synchronized {
          if (bot==null) bot =
system.actorOf(Props
          (new ChatBotAdminActor(system)))
          bot
        }
      }
    }
```

As you can see, this Actor receives the reference of the chat room as a parameter. Using the Actor system, it gets the reference of the chat room Actor. This Actor receives an `ActorSystem` message by now.

Using the Actor system variable called system, we will also schedule a `Tick` for this Actor for every ten seconds. This time, the window interval will be the time in which the bot will notify the chat room about the current status.

We will also override the `preStart` function. Akka will call this function when the Actor is created on the actor system. This implementation

will send a message to the room, which is `JoinChatRoom`.

Like all Actors, there is the receive function implementation. First case with `ChatMessage` is returning Unit. If you want to make this bot respond to people, remove Unit and write the proper Scala code as you wish.

In the second case, we will have the String message that will be sent to the chat room. Finally, after this case, we will have the `Tick` method, which will appear every ten seconds. So, we will use the `ActorHelper` to get the stats from the room, and then we will send a string message with the information about the room. This will trigger the second case and broadcast the message to the whole room.

Finally, we have a companion object. We don't want to have two instances of the bot, which is why we will control this object creation by design. We're done with the actors implementations. Next, we will need to work a new controller for the chat actors.

The chat controller

We will need to create a new controller. This controller will be located at `ReactiveWebStore/app/controllers`.

Implementing the chat controller

`ChatController.scala` should look something like this:

```
package controllers
    import akka.actor.ActorSystem
    import akka.stream.Materializer
    import javax.inject._
    import play.api._
    import play.api.mvc._
    import play.api.libs.streams._
    import actors.ChatUserActor
    import actors.ChatBotAdminActor
    @Singleton
    class ChatController @Inject() (implicit val
system: ActorSystem,
    materializer: Materializer)
    extends Controller {
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
      ChatBotAdminActor(system)
      def index_socket = Action { request =>
        Ok(views.html.chat_index()(Flash(Map())))
      }
      def ws = WebSocket.accept[String, String] {
request =>
        ActorFlow.actorRef(out =>
ChatUserActor.props(system)(out))
      }
    }
```

The `ChatController` method will use Google Guice to get injected instances of Actor System and an Actor materializer instance. A materializer is needed because it will provide the instance of the out Actor for each user in the system.

As you can see, we will create an instance of the `ChatBotAdmin` method passing through the actor system, which Google Guice injected for us. For this controller, we will just have two functions: one function to render the chat UI, and the other one to serve the WebSocket.

The Play framework already provides built-in integration with Akka and WebSockets. So, we will just need to use the `ActorFlow` method using the `actorRef` function in order to obtain an out Actor.

Here, we will call the `ChatUserActor` companion object and create a chat user for the websocket passing out the Actor system the controller has. As you can see, this returns `WebSocket.accept`, which is a full duplex connection between the web browser and the backend.

Configuring the routes

Next, we will need to expose our controller functions to the UI. We will need to add more routes to the `ReactiveWebStore/conf/routes` file:

```
    routes
    #
    # Akka and Websockets
    #
    GET /chat/index_socket
controllers.ChatController.index_socket
    GET /chat/ws
controllers.ChatController.ws
```

The routes are done now.

Working on the UI

Now, it is time to code on the UI on both the HTML layout and the WebSocket code in JavaScript. We will need to create a new file, located at `ReactiveWebStore/app/views`.

Your `chat_index.scala.html` file should look something like this:

```
@()(implicit flash:Flash)
    @main("Chat"){
    <!DOCTYPE html>
      <meta charset="utf-8" />
      <title>Chat Room</title>
      <script type="text/javascript">
        var output;
        var websocket = new
WebSocket("ws://localhost:9000/chat/ws");
        function init(){
          output =
document.getElementById("output");
          websocket.onmessage = function(evt) {
            writeToScreen('<span
style="color:blue;">' + evt.data+
            '</span>');
          };
          websocket.onerror = function(evt) {
            writeToScreen('<span style="color:
red;">ERROR:</span> ' +
            evt.data);
          };
        }
        function doSend(message){
          websocket.send(message);
        }
        function writeToScreen(message){
          var pre = document.createElement("p");
          pre.style.wordWrap = "break-word";
          pre.innerHTML = message;
          $('#output').prepend(pre);
        }
        window.addEventListener("load", init,
false);
      </script>
      <h3>Messages</h3>
```

```
      <div id="output" style="width: 800px;
height: 250px; overflow-y:
      scroll;" >
      </div>
      <div id="contentMessage">
        <BR>
        user:
      
        <input type="text" name="txtUser"
id="txtUser" /> <BR><BR>
        message: <input type="text"
name="txtMessage"
        id="txtMessage" />
        <BR>
        <BR>
        <a href="#" class="btn btn-success"
        onclick="doSend(
document.getElementById('txtUser').value + ':'
        +
document.getElementById('txtMessage').value );">
        <i class="icon-plus icon-white"></i>Send
Message</a>
      </div>
    }
```

The UI is very simple. There is an input text for you to put your name and there is another one for the text message itself, and a send button. As you can see in the JavaScript code, the first thing that we will do is open a WebSocket connection to the `ws://localhost:9000/chat/ws` URL. Then, we will register the `init` function to run once the browser is ready.

The `init` function in JavaScript will create two functions for our WebSocket. One function will run when any error occurs and the other function will run for each message emitted by the Akka backend.

We will have a `doSend` function in JavaScript in order to send a message to the WebSocket. This message will be delivered to the controller and then to the Akka actors. You can also see some JQuery and HTML code in order to create new elements on the UI. This is done in order to display the message in the chat room.

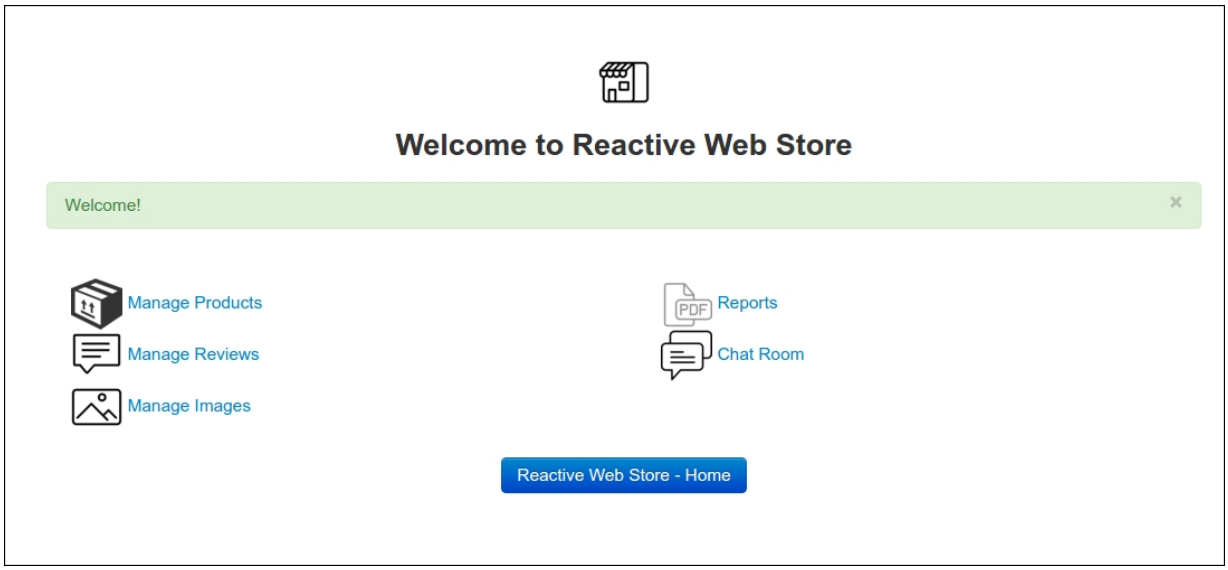
OK, there is one more thing that we will need to do - add a reference to the chat UI on the main page of our application.

Your `Index.scala.html` should look something like this:

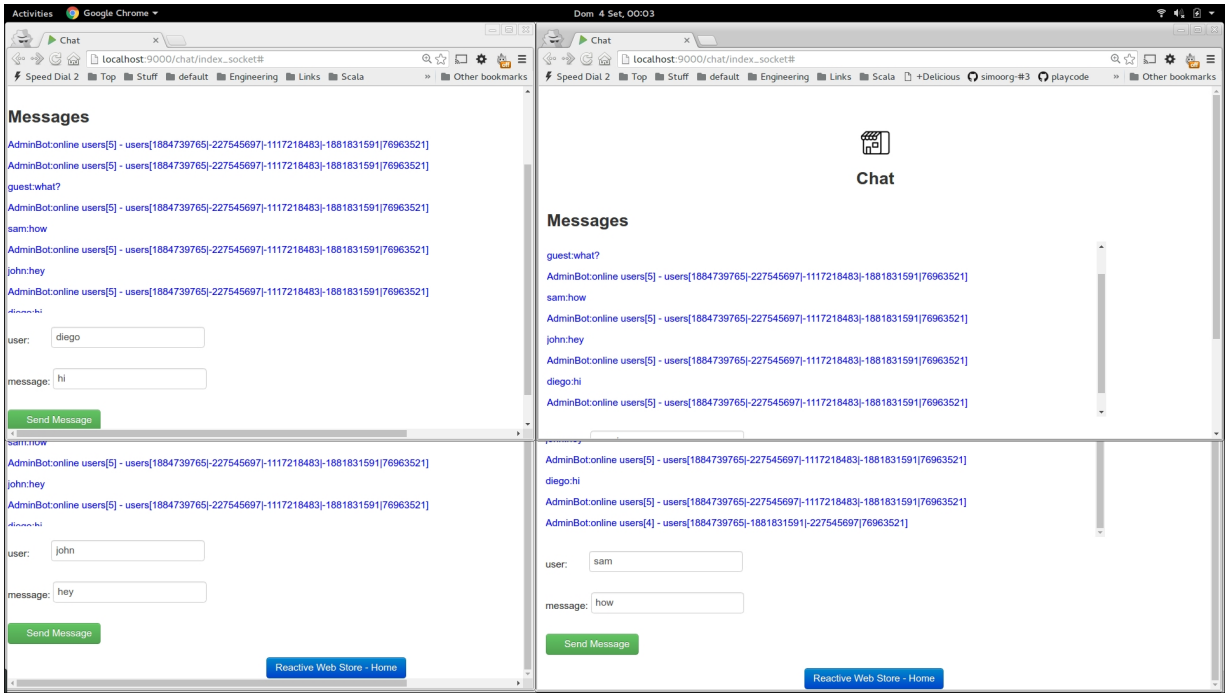
```
@(message: String)(implicit flash:Flash)
    @main("Welcome to Reactive Web Store"){
    <div class="row-fluid">
      <BR>
      <div class="span12">
        <div class="row-fluid">
          <div class="span6">
            <a href="/product"><img height="42"
width="42"

src="@routes.Assets.at("images/product.png")">
Manage
            Products</a><BR>
            <a href="/review"><img height="42"
width="42"

src="@routes.Assets.at("images/review.png")">
Manage
            Reviews</a><BR>
            <a href="/image"><img height="42"
width="42"

src="@routes.Assets.at("images/image.png")">
Manage
            Images</a><BR>
          </div>
          <div class="span6">
            <a href="/reports"><img height="42"
width="42"

src="@routes.Assets.at("images/reports.png")">
Reports </a>
            <BR>
            <a href="/chat/index_socket"><img
height="42" width="42"

src="@routes.Assets.at("images/chat.png")"> Chat
Room </a>
              <BR>
            </div>
          </div>
```

```
        </div>
      </div>
    }
```

We will also use the opportunity to improve a little bit of the UI design using Twitter Bootstrap column design. In the last row, you can see our link to the chat UI. Now, we can run the application and see our chat working. Run `$ activator run`:



As you can see, our new chat UI link is there. Now, let's have fun with this new feature. Open four new browsers (simulate four different users), then go to the `http://localhost:9000/chat/index_socket` url and let's have a little chat. You should see something similar to this:



Almost done. Our chat feature works; however, we will need to do more than just a functional black box test on the UI. We will need unit tests. Luckily for us, we have the Akka testkit, which allows us to easily test actors.

Adding Akka tests

We will create three more tests: one for each actor that we have. They are located at `ReactiveWebStore/test/`.

Scala test for Akka Actor

`ChatUserActorSpec.scala` should look something like this:

```
      class OutActor extends Actor {
            def receive = {
              case a:Any => Unit
            }
          }
          class ChatUserActorSpec extends PlaySpec {
            class Actors extends
      TestKit(ActorSystem("test"))
            "ChatUserActor" should {
              "joins the chat room and send a message"
      in new Actors {
                val probe1 = new TestProbe(system)
                val actorOutRef = TestActorRef[OutActor]
      (Props[OutActor])
                val actorRef =
      TestActorRef[ChatUserActor]
                (ChatUserActor.props(system)
      (actorOutRef))
                val userActor = actorRef.underlyingActor
                assert(userActor.context != null)
                val msg = "testUser:test msg"
                probe1.send(actorRef,msg)
                actorRef.receive(msg)
                receiveOne(2000 millis)
              }
            }
          }
```

The Akka testkit is very cool as it allows us to test actors with a very easy **Domain Specific Language** (**DSL**). It's possible to check the Actor mailbox, the Actor internal state, and so much more. There is one trick that we will need to do because we need to extend one

class; in order to have Play working with the Scala test, we will need to use PlaySpec. However, we will also need to extend one class to make the Akka testkit work, which is `TestKit`. We can't extend both at the same time, but no worries, there is always a workaround.

The workaround here is to create a case class, make that case class extend `TestKit`, and then use it in a spec context, that is, in a new `Actor {}`.

Here, we are checking if `ChatUserActor` can join the chat room properly. This is done by simply creating the Actor, as the Actor has a `preStart` method that will auto-join the room.

We will need to create a fake implementation of the out Actor here, which is why we have the `OutActor` implementation. We will create a probe to test the actor system, and we will also use a special function to test the Actors, called `TestActorRef`. This abstraction provides a way to access the Actor's state via `actorRef.underlyingActor`, and this is useful because you can check the Actor internal state to validate the code. The rest of the code is normal Akka and Scala test code. Let's move to the next test.

Chat room Actor test

The `ChatRoonActorSpec.scala` file should look something like this:

```
class ChatRoomActorSpec extends PlaySpec {
      class Actors extends
TestKit(ActorSystem("test"))
      "ChatRoomActor" should {
        "accept joins the chat rooms" in new
Actors {
          val probe1 = new TestProbe(system)
          val probe2 = new TestProbe(system)
          val actorRef =
TestActorRef[ChatRoomActor]
          (Props[ChatRoomActor])
          val roomActor = actorRef.underlyingActor
          assert(roomActor.users.size == 0)
          probe1.send(actorRef, JoinChatRoom)
          probe2.send(actorRef, JoinChatRoom)
```

```
          awaitCond(roomActor.users.size == 2, 100
millis)

assert(roomActor.users.contains(probe1.ref))

assert(roomActor.users.contains(probe2.ref))
        }
        "get stats from the chat room" in new
Actors {
          val probe1 = new TestProbe(system)
          val actorRef =
TestActorRef[ChatRoomActor]
          (Props[ChatRoomActor])
          val roomActor = actorRef.underlyingActor
          assert(roomActor.users.size == 0)
          probe1.send(actorRef, JoinChatRoom)
          awaitCond(roomActor.users.size == 1, 100
millis)

assert(roomActor.users.contains(probe1.ref))
          probe1.send(actorRef, GetStats)
          receiveOne(2000 millis)
        }
        "and broadcast messages" in new Actors {
          val probe1 = new TestProbe(system)
          val probe2 = new TestProbe(system)
          val actorRef =
TestActorRef[ChatRoomActor]
          (Props[ChatRoomActor])
          val roomActor = actorRef.underlyingActor
          probe1.send(actorRef, JoinChatRoom)
          probe2.send(actorRef, JoinChatRoom)
          awaitCond(roomActor.users.size == 2, 100
millis)
          val msg = ChatMessage("sender", "test
message")
          actorRef.receive(msg)
          probe1.expectMsg(msg)
          probe2.expectMsg(msg)
        }
        "and track users ref and counts" in new
Actors {
          val probe1 = new TestProbe(system)
          val probe2 = new TestProbe(system)
          val actorRef =
TestActorRef[ChatRoomActor]
          (Props[ChatRoomActor])
          val roomActor = actorRef.underlyingActor
```

```
            probe1.send(actorRef, JoinChatRoom)
            probe2.send(actorRef, JoinChatRoom)
            awaitCond(roomActor.users.size == 2, 100
millis)
            probe2.ref ! PoisonPill
            awaitCond(roomActor.users.size == 1, 100
millis)
          }
        }
    }
```

So, here we have the same concepts as the other test. However, we have more usage of the Akka testkit DSL. For instance, we are using `expectMsg` on the probe to check if an Actor received a specific message. We are also using `awaitCond` to check the Actor's internal state in an assertion.

Now is the time to test the last Actor method.

Chat Bot Admin Actor test

`ChatBotAdminActorSpec.scala` file should look something like this:

```
class ChatBotAdminActorSpec extends
TestKit(ActorSystem("test"))
    with ImplicitSender
    with WordSpecLike with Matchers with
BeforeAndAfterAll {
      "ChatBotAdminActor" should {
        "be able to create Bot Admin in the Chat
Room and Tick" in {
          val probe1 = new TestProbe(system)
          val actorRef =
TestActorRef[ChatBotAdminActor](Props(new
          ChatBotAdminActor(system)))
          val botActor = actorRef.underlyingActor
          assert(botActor.context != null)
          awaitCond(botActor.room != null )
        }
      }
    }
```

For this test, we will check if the actor context was not null, and also if the room was created and the scheduler was also not null. All good to go.

Alright, that's it! This is the last actor test. Now we are completely done. You can run this test with `$ activator test`, or, if you prefer the activator, then use `"test-only TESTCLASSNAME" -Dsbt.task.forcegc=false` to run a specific test case.

Summary

In this chapter, you learned how to work with Akka actors and created a web chat using Akka, the Play framework, and WebSockets. Akka is a really powerful solution that can be used with or without the Play framework. Additionally, you learned about the Actor model, mailboxes, routing, persistence, Akka configuration, message patterns, and how to write code with actors in Scala and Play.

In the next chapter, you will learn more about REST, JSON, and how to model a REST API, as well as how to create a Scala client for your REST services.

Chapter 9. Design Your REST API

In the previous chapter, we added a new chat feature in our app using Akka. Our web application is close to the end. This chapter will add the REST API in our Play framework application.

We will also create a Scala client using the `ws` library from the Play framework in order to call our REST API. Later in this chapter, we will add Swagger support and embed the Swagger UI in our app.

In this chapter, we will cover the following topics:

- REST and API design
- Creating our API with REST and JSON
- Creating a Scala client
- Adding validations
- Adding back pressure
- Adding Swagger support

Introduction to REST

Representational State Transfer (**REST**) is an architectural style. It was defined by Roy Fielding in his doctoral dissertation. REST happens over the HTTP 1.1 protocol using HTTP verbs, such as `GET`, `POST`, `DELETE`, `PUT`, and **Uniform Resource Identifier** (**URI**), for instance, `/users/profile/1` or `sales/cart/add/2`.

The REST architecture has the following properties:

- **Simplicity**: Pretty much all languages have libraries to manipulate HTTP URIs.
- **Interoperability**: REST is language, platform, and OS agnostic.

- **Scalable and Reliable**: As REST is based on HTTP, you can use the HTTP server to scale up your application in conjunction with HTTP load balancer, the HTTP caches, and HTTP DNS.
- **Separation of Concerns** (**SOC**): As you have a URI, that's your contract, not the code, underlying backend, or database. This means that you can change the database or language without affecting the code.
- **Client/Server**: There is a server that provides the REST interface and the clients, which call the REST endpoints.

Web services that embrace the REST principles are often called RESTful.

REST API design

When you are working with REST, there are some principles that you should keep in mind, and these principles should provide guidance for your design choices when you are doing API design.

HTTP verbs design

These are the following verbs found in HTTP:

- `GET`: This is often used to answer queries
- `PUT`: This is often used to insert data
- `POST`: This is often used to update data
- `DELETE`: This is often used to remove data

Why do we keep saying often? Well, there are some exceptions in regards of size limitations. For instance, for the `GET` verb, we can't have a request bigger than 8192 bytes or 8 KB. If you need to send a bigger payload, we will need to use the `POST` verb.

Uniform API

REST uses a uniform API. For example, consider the following piece of code: GET /users/1 = List information about user id 1 PUT /users/1 = Insert user 1 POST /users/1 = Update user 1 DELETE /users/1 = Delete user 1 GET /users/ = Lists All users

If we change the resource from users to sales, the API would almost be the same. Retrieving data is done using `GET` and update is done via `POST`, so it's a uniform API.

Response with HTTP status codes

REST runs the error handler using the HTTP 1.1 status codes. For instance:

- **200 -> OK**: This is often used with the `GET` verb
- **201 -> Created**: This is often used by the `PUT/POST` verbs
- **204 -> No Content**: This is often for the `DELETE` verb
- **400 -> Invalid Request**: This often means an invalid request for the `POST/PUT` verbs
- **404 -> Not Found**: This is often used with the `GET` verb
- **500 -> Internal Server Error - Unexpected Server Error**: This is often used by all the verbs

REST API patterns

There are some commons patterns for good and clear REST API designs, as follows:

- **Use nouns; do not use verbs**: Often, you can use standard URIs, such as `/cars/` or `/members/`. You should not use `/getCars/` or `getMembers/` because you are using the URI with a verb, and the verb already tells the actions.
- **GET method should not change state**: If you want to change the state of the server, you will need to use verbs such as `PUT`, `POST`, or `DELETE`. `GET` should not change the state of the server, so it should always be safe calling `GET` as many times as you want. This is called idempotent.
- **Prefer sub-resource relation**: Let's say we have a resource called `/users/`, and a user has projects. It's always a good idea to use sub-resources, such as `/users/1/projects/2`, because we have a relationship between users and projects.
- **Use HTTP headers**: HTTP headers should be used for serialization, security, and all the kinds of metadata your application needs. The HTTP Headers are often used for content negotiation. For instance, you might do the following:

```
HTTP HEADER Content-Type = XML - GET /cars/1
    HTTP HEADER Content-Type = JSON - GET /cars/1
```

- The URI is the same; however, based on the header type, it will return data in XML or JSON format.

- **Filter, Sorting and Pagination**: Sometimes, your data may be big. It's always a good idea to provide mechanisms to sort, filter, and paginate as follows:

```
GET /projects/1/tasks?by=priority                -> Sorting
    GET /projects/1/tasks?status=done            -> Filter
    GET /projects/1/tasks?limit=30&offset=5      -> Pagination
```

API versioning

There are two ways to perform API versioning. First strategy is to version by the endpoint explicit such as `/v1/cars`. The second strategy is based on metadata such as `/cars/`, but then you will pass an HTTP HEADER version as v1.

Both strategies have pros and cons. Explicit versioning is more clear, and you can always create a new version and don't break your consumers. Header strategy is more elegant; however, it can get tricky to manage.

Some anti-patterns to be avoided

There are several traps in the REST API design, but the following things need to be avoided:

- `GET` verb for everything
- Ignoring HTTP headers such as MIME-types
- Returning 200 when an error happens
- Returning 500 for an invalid parameter or a missing parameter

Creating our API with REST and JSON

Alright, now is the time to design a REST API for your Play framework application. We will create an API to export all data in the system. This API will be READ only; however, you can add write operations if you like.

Later on in this chapter, we will add some back pressure to limit the API REST rate for consumers and create a Scala client application for our REST API. So, first of all, let's get started with the Play framework (server) first.

We don't need any extra library in order to create a REST API in our Play framework application. We will just need a new controller and new routes. Additionally, we will leverage most of the code we made in the previous chapters.

RestApiContoller

Let's create a new controller located at `ReactiveWebStore/app/controllers`.

REST API Front Controller implementation

`RestApiController.scala` file should look something like this:

```
package controllers
    @Singleton
    class RestAPIController @Inject()
    (val productService:IProductService,
      val reviewService:IReviewService,
    val imageService:IImageService) extends
Controller {
      import play.api.libs.concurrent.Execution.
      Implicits.defaultContext
```

```
  def listAllProducts = Action {
    val future = productService.findAll()
    val products = Awaits.get(5,future)
    val json = ProductsJson.toJson(products)
    Ok(json)
  }
  def listAllReviews = Action {
    val future = reviewService.findAll()
    val reviews = Awaits.get(5,future)
    val json = ReviewsJson.toJson(reviews)
    Ok(json)
  }
  def processImages = {
    val future = imageService.findAll()
    val images = Awaits.get(5,future)
    val json = ImagesJson.toJson(images)
    json
  }
  def listAllImages = Action {
    Ok(processImages)
  }
}
```

Basically, we have three functions here. These functions list all products, images, and reviews. As you can see at the top of the controller, we are injecting the three services that we have for products, images, and reviews.

The code is pretty much straightforward for all functions. First, we will call the proper service, and then we will wait for the result with the `await` object. Once we have the data, we will call a function to convert the data to JSON.

Let's take a look at the JSON helpers objects that we used here.

JSON mapping

Our REST controller used JSON helper objects to map objects to JSON. First, we will start with the Products JSON helper.

`ProductsJson` is located at `ReactiveWebStore/app/controllers/Product.scala`:

```
object ProductsJson {
      import play.api.libs.json._
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val productWrites: Writes[Product]
= (
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "name").write[String] and
        (JsPath \ "details").write[String] and
        (JsPath \ "price").write[BigDecimal]
      )(unlift(Product.unapply))
      implicit val productReads: Reads[Product] =
(
        (JsPath \ "id").readNullable[Long] and
        (JsPath \ "name").read[String] and
        (JsPath \ "details").read[String] and
        (JsPath \ "price").read[BigDecimal]
      )(Product.apply _)
      def toJson(products:Option[Seq[Product]]) =
Json.toJson(products)
    }
```

Basically, there are three important concepts here. First, we have `productsWrites` , which maps from JSON to `model`, and `Product` for writes, which is also known as deserialization. We have another mapping for serialization called `productsReads`, which converts objects to JSON.

As you can see, we need to map all fields existing in our model, such as ID, name, details, and price. This mapping must match proper types as well. ID mapping uses `readNullable` because ID is optional.

Finally, we have a function to convert from JSON to object, called `toJson`, which uses a generic Play framework library called JSON. Let's move for the next helper--the review.

`ReviewsJson` is located at `ReactiveWebStore/app/controllers/Review.scala` and should look something like this:

```
object ReviewsJson {
      import play.api.libs.json._
```

```
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val reviewWrites: Writes[Review] =
(
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "productId").write[Option[Long]]
and
        (JsPath \ "author").write[String] and
        (JsPath \ "comment").write[String]
      )(unlift(Review.unapply))
      implicit val reviewReads: Reads[Review] = (
        (JsPath \ "id").readNullable[Long] and
        (JsPath \ "productId").readNullable[Long]
and
        (JsPath \ "author").read[String] and
        (JsPath \ "comment").read[String]
      )(Review.apply _)
      def toJson(reviews:Option[Seq[Review]]) =
Json.toJson(reviews)
    }
```

Here, we have the same concepts that we saw earlier in the Products JSON helper. We have a mapping for reads and writes and a function which converts a `model.Review` to JSON. Let's move to the last helper, the `ImageJson`.

`ImagesJson` is located at `ReactiveWebStore/app/controllers/Image.scala`, which should look something like this:

```
object ImagesJson {
      import play.api.libs.json._
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val imagesWrites: Writes[Image] = (
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "productId").write[Option[Long]]
and
        (JsPath \ "url").write[String]
      )(unlift(Image.unapply))
      implicit val imagesReads: Reads[Image] = (
        (JsPath \ "id").readNullable[Long] and
        (JsPath \ "productId").readNullable[Long]
and
```

```
        (JsPath \ "url").read[String]
      )(Image.apply _)
      def toJson(images:Option[Seq[Image]]) =
Json.toJson(images)
    }
```

Just as with the other two mappers, we have reads, writes, mappings, and the `toJson` function. We are done with mappers, so now the next step is to create the new routes.

Configuring new routes

We need to add the following three new routes for our REST API, which is located at `ReactiveWebStore/conf/routes`:

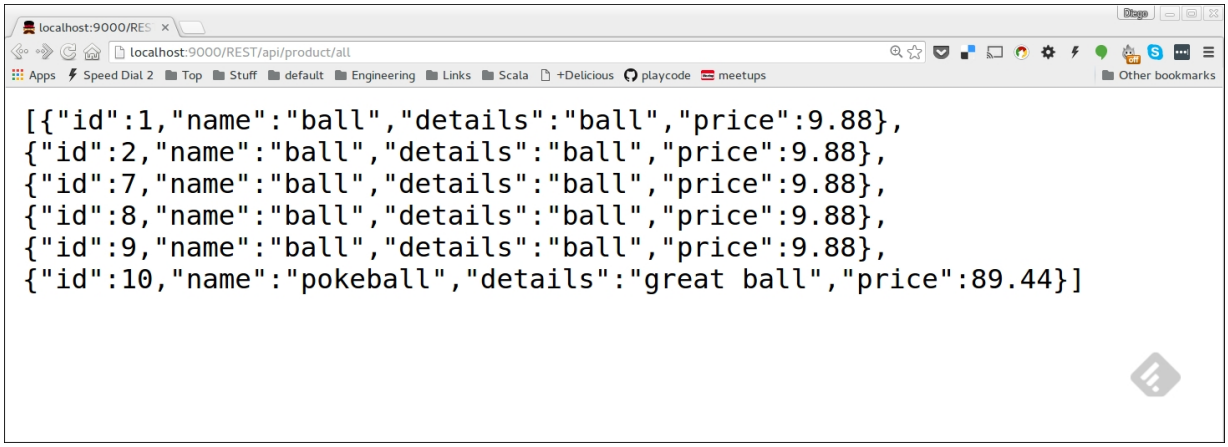
```
#
    # REST API
    #
    GET /REST/api/product/all
    controllers.RestAPIController.listAllProducts
    GET /REST/api/review/all
    controllers.RestAPIController.listAllReviews
    GET /REST/api/image/all
controllers.RestAPIController.listAllImages
```

As you can see, we mapped all the list operations we just created.

Testing the API using the browser

Now we can run `$ activator run` and test our new REST API using our web browser.

Go to `http://localhost:9000/REST/api/product/all`; you should see something similar to the following screenshot:



Let's look at the review API.

Go to `http://localhost:9000/REST/api/review/all`; you should see results similar to the following screenshot:

Finally, let's check out the image of REST API.

Go to `http://localhost:9000/REST/api/image/all`; you should see results similar to the following screenshot:

OK. Now we will continue to work with REST. We just finished the server; however, it is important to create a REST client to consume these REST APIs.

Creating a Scala client

First, you will need to create a new project. Go to your file system and create a folder called `rest-client`. Then, create another folder inside `rest-client` called `project`. Inside `project`, you will need to add the following two files:

- `build.properties`: This contains an SBT configuration, such as version
- `Plugins.sbt`: This contains an SBT plugins configuration

Let's start with `build.properties`:

```
sbt.version=0.13.11
```

As you can see here, we are configuring this project to use SBT version 0.13.11. Now, we can move to the plugins file.

Configuring plugins.sbt

Your `plugins.sbt` file should look something like this:

```
addSbtPlugin("com.typesafe.sbteclipse" %
"sbteclipse-plugin" %
      "2.5.0")
    addSbtPlugin("com.github.mpeltonen" % "sbt-
idea" % "1.6.0")
```

Here, we are adding Eclipse and IntelliJ support. For this book, we are using Eclipse, but feel free to use anything you like.

Outside of the `project` folder, under `rest-client`, we will need to configure the `build.sbt` file.

Configuring build.sbt

Your `build.sbt` file should look something like this:

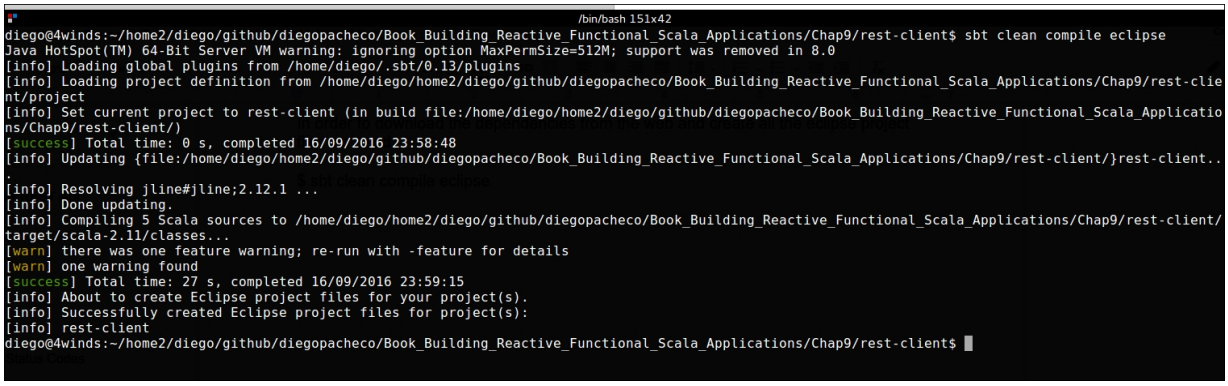
```
name := "rest-client"
    version := "1.0"
    scalaVersion := "2.11.7"
    scalaVersion in ThisBuild := "2.11.7"
    resolvers += DefaultMavenRepository
    resolvers += JavaNet1Repository
    resolvers += "OSSSonatype" at

"https://oss.sonatype.org/content/repositories/rel
eases"
    resolvers += "Sonatype OSS Snapshots" at

"https://oss.sonatype.org/content/repositories/sna
pshots"
    resolvers += "Sonatype OSS Snapshots" at

"https://oss.sonatype.org/content/repositories/sna
pshots"
    resolvers += "amateras-repo" at
    "http://amateras.sourceforge.jp/mvn/"
    libraryDependencies += "com.typesafe.play" %
"play-ws_2.11" %
    "2.5.6"
    libraryDependencies += "org.scalatest" %
"scalatest_2.11" % "2.2.6"
    % Test
```

So here, we are using Scala version 2.11.7, and we are declaring just two dependencies. One dependency is for tests, which is `scala-test`, and the other dependency is on the Play framework `ws` library, which we will use to call our REST APIs.

Let's also create two source folders, as follows:

- `src/main/scala`: This is the Scala source code
- `src/test/scala`: This is the Scala test source code

OK. Now we can run `$ sbt clean compile eclipse` in order to download the dependencies from the web and create all the Eclipse project files that we need.



Now we can import this code in Eclipse and move on.

Scala client code

First of all, we will need to create a `Factory` to instantiate the `WS` Play framework library to call `webservices`. Under the `rest-client/src/main/scala` location, let's create a package called `client` and add the following code under `WSFactory.scala`:

```
package client
    object WSFactory {
      import akka.actor.ActorSystem
      import akka.stream.ActorMaterializer
      def ws = {
        implicit val system = ActorSystem()
        implicit val materializer =
ActorMaterializer()
        import com.typesafe.config.ConfigFactory
        import play.api._
        import play.api.libs.ws._
        import play.api.libs.ws.ahc.{AhcWSClient,
AhcWSClientConfig}
        import
play.api.libs.ws.ahc.AhcConfigBuilder
        import
org.asynchttpclient.AsyncHttpClientConfig
        import java.io.File
        val configuration =
Configuration.reference ++
```

```
        Configuration(ConfigFactory.parseString(
          """
          |ws.followRedirects = true
        """.stripMargin))
        val parser = new
WSConfigParser(configuration,
        play.api.Environment.simple(
        new File("/tmp/"), null))
        val config = new
AhcWSClientConfig(wsClientConfig =
        parser.parse())
        val builder = new AhcConfigBuilder(config)
        val logging = new

AsyncHttpClientConfig.AdditionalChannelInitializer
() {
          override def initChannel(channel:
io.netty.channel.Channel):
          Unit = {
            channel.pipeline.addFirst("log", new

io.netty.handler.logging.LoggingHandler("debug"))
          }
        }
        val ahcBuilder = builder.configure()

ahcBuilder.setHttpAdditionalChannelInitializer(log
ging)
        val ahcConfig = ahcBuilder.build()
        new AhcWSClient(ahcConfig)
      }
    }
```

The preceding code is just technical. These are the steps needed to instantiate the WSClient outside the Play framework. If this client was a web application using the Play framework, it would be way easier as we can just use Google Guice and inject what we need.

The main idea you need to keep in mind is that you need to use Akka and ActorSystem in order to use this feature. As you can see, all this code is locked inside an object in a single function called `ws`.

We will need some utility class to work with futures. As we use `ws` library to call REST APIs, it returns Future. So, let's create a new

package called `utils`:

Your `Awaits.scala` file should look like something like this:

```
package utils
    import scala.concurrent.Future
    import scala.concurrent.duration._
    import scala.concurrent.Await
    object Awaits {
      def get[T](sec:Int,f:Future[T]):T = {
        Await.result[T](f, sec seconds)
      }
    }
```

The preceding code is pretty simple. We used the `Await` object and then used a generic `T` in order to convert the result to a generic parameterized type. By using this parameter, we will also receive how many seconds we should wait before the timeout.

Creating our REST client proxies

We will now make REST calls; however, we will create a Scala API. So, the developers who use our `rest-client` won't need to deal with REST and just execute the Scala code. This is good for many reasons, some of which are as follows:

- **SOC**: We still have separation of concerns between the Play framework and the client application
- **Isolation**: If the REST API changes, we will need to deal with it on the proxy layer
- **Abstraction**: The rest of the client code just uses Scala and does not know anything about REST or HTTP calls

These techniques are very common nowadays with microservices. These techniques can also be known as drivers or thick clients. Right now, we will need to create three proxies, one for each resource, that we have on the REST API. Let's create a new package called `proxy`.

Your `ProductProxy.scala` file should look something like this:

```
package proxy
    import client.WSFactory
    import utils.Awaits
    case class Product
    ( var id:Option[Long],
      var name:String,
      var details:String,
    var price:BigDecimal ) {
      override def toString:String = {
        "Product { id: " + id.getOrElse(0) +
",name: " + name + ",
        details: "+ details + ", price:
        " + price + "}"
      }
    }
```

```
    object ProductsJson {
      import play.api.libs.json._
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val productWrites: Writes[Product]
= (
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "name").write[String] and
        (JsPath \ "details").write[String] and
        (JsPath \ "price").write[BigDecimal]
      )(unlift(Product.unapply))
      implicit val productReads: Reads[Product] =
(
        (JsPath \ "id").readNullable[Long] and
        (JsPath \ "name").read[String] and
        (JsPath \ "details").read[String] and
        (JsPath \ "price").read[BigDecimal]
      )(Product.apply _)
      def toJson(products:Option[Seq[Product]]) =
Json.toJson(products)
    }
    object ProductProxy {
      import scala.concurrent.Future
      import play.api.libs.json._
      import ProductsJson._
      val url =
"http://localhost:9000/REST/api/product/all"
      implicit val context =

play.api.libs.concurrent.Execution.Implicits.defau
ltContext
      def listAll():Option[List[Product]] = {
        val ws = WSFactory.ws
        val
futureResult:Future[Option[List[Product]]] =
        ws.url(url).withHeaders("Accept" ->
        "application/json").get().map(
          response =>

Json.parse(response.body).validate[List[Product]].
asOpt
        )
        val products = Awaits.get(10,
futureResult)
        ws.close
        products
```

```
  }
}
```

We have three big concepts in this code. First of all, we have a `case` class that represents the product. The preceding code is very similar to the code we have on the Play framework application. However, if you pay attention, you will see it is much cleaner because we don't have any metadata around persistence.

You might think, this is duplicated code! It is, and it is 100% okay. Duplicate code is decoupled. Remember that we have a REST interface and also a proxy between the rest of the client code, so we have at least two layers of indirection that we can deal with changes. If these two code bases share the same class, we would have coupling and less space to accommodate changes.

The second big concept here is mapping. We will receive JSON, and we will want to convert JSON to our case class, so we will have similar mapping that we did in the Play framework application.

Finally, we have the proxy implementation. We will instantiate the Play framework `WS` library using our factory and call the `ws` function. Then, we will use the `url` function passing the REST API URI for products and define a header in order to accept JSON. We are also doing this using the HTTP verb, `GET`. The response is mapped with `Json.parse` passing `response.body`. Additionally, we will call the validate function to make sure this JSON matches our `case` class. This validation is important because then we can be sure that the format did not change, and that everything works fine. `WS` will return this as a Future, so we will use our `Awaits` helper to get the result.

Let's move to the next proxy, the review.

Your `ReviewProxy.scala` file should look something like this:

```
package proxy
    import client.WSFactory
    import utils.Awaits
    case class Review
```

```
    (var id:Option[Long],
      var productId:Option[Long],
      var author:String,
    var comment:String)
    {
      override def toString:String = {
        "Review { id: " + id + " ,productId: " +
productId.getOrElse(0)
        + ",author: " + author +
        ",comment: " + comment + " }"
      }
    }
    object ReviewsJson {
      import play.api.libs.json._
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val reviewWrites: Writes[Review] =
(
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "productId").write[Option[Long]]
and
        (JsPath \ "author").write[String] and
        (JsPath \ "comment").write[String]
      )(unlift(Review.unapply))
      implicit val reviewReads: Reads[Review] = (
        (JsPath \ "id").readNullable[Long] and
        (JsPath \ "productId").readNullable[Long]
and
        (JsPath \ "author").read[String] and
        (JsPath \ "comment").read[String]
      )(Review.apply _)
      def toJson(reviews:Option[Seq[Review]]) =
Json.toJson(reviews)
    }
    object ReviewProxy {
      import scala.concurrent.Future
      import play.api.libs.json._
      import ReviewsJson._
      val url =
"http://localhost:9000/REST/api/review/all"
      implicit val context =

play.api.libs.concurrent.Execution.Implicits.defau
ltContext
      def listAll():Option[List[Review]] = {
        val ws = WSFactory.ws
        val
futureResult:Future[Option[List[Review]]] =
```

```
        ws.url(url).withHeaders("Accept" ->
        "application/json").get().map(
          response =>

Json.parse(response.body).validate[List[Review]].a
sOpt
        )
        val reviews = Awaits.get(10, futureResult)
        ws.close
        reviews
      }
    }
```

Here, we have the same principles that we had on the product proxy, but this time for review. As you can see, we will call a different URI. Now, let's move to the last proxy--the `ImageProxy.scala` file.

Your `ImageProxy.scala` file should look something like this:

```
package proxy
    import client.WSFactory
    import utils.Awaits
    case class Image
    (var id:Option[Long],
      var productId:Option[Long],
    var url:String)
    {
      override def toString:String = {
        "Image { productId: " +
productId.getOrElse(0) + ",url: " + url
        + "}"
      }
    }
    object ImagesJson {
      import play.api.libs.json._
      import play.api.libs.json.Reads._
      import play.api.libs.functional.syntax._
      implicit val imagesWrites: Writes[Image] = (
        (JsPath \ "id").write[Option[Long]] and
        (JsPath \ "productId").write[Option[Long]]
and
        (JsPath \ "url").write[String]
      )(unlift(Image.unapply))
      implicit val imagesReads: Reads[Image] = (
        (JsPath \ "id").readNullable[Long] and
```

```
        (JsPath \ "productId").readNullable[Long]
and
        (JsPath \ "url").read[String]
      )(Image.apply _)
      def toJson(images:Option[Seq[Image]]) =
Json.toJson(images)
    }
    object ImageProxy {
      import scala.concurrent.Future
      import play.api.libs.json._
      import ImagesJson._
      val url =
"http://localhost:9000/REST/api/image/all"
      implicit val context =

play.api.libs.concurrent.Execution.Implicits.defau
ltContext
      def listAll():Option[List[Image]] = {
        val ws = WSFactory.ws
        val
futureResult:Future[Option[List[Image]]] =
        ws.url(url).withHeaders("Accept" ->
        "application/json").get().map(
          response =>

Json.parse(response.body).validate[List[Image]].as
Opt
        )
        val images = Awaits.get(10, futureResult)
        ws.close
        images
      }
    }
```

That's it. We have the same concepts as product and review. We have finished all our proxies. Now, it is time to test our proxy implementation. The best way to do this is via tests, so let's create Scala tests for these three implementations.

Creating ScalaTest tests for the proxies

Under the `/src/test/scala` source folder, we will need to create a package called `proxy.test`.

Your `ProductProxtTestSpec.scala` should look something like this:

```
package proxy.test
    import org.scalatest._
    import proxy.ProductProxy
    class ProductProxtTestSpec extends FlatSpec
with Matchers {
      "A Product Rest proxy " should "return all
products" in {
        val products = ProductProxy.listAll().get
        products shouldNot(be(null))
        products shouldNot(be(empty))
      }
    }
```

The test is quite simple; we will just have to call the `listAll` operation in our product proxy and then add some assertions to make sure the result is not null. We will also show all the products in the console.

Now, we will need to create tests for the review proxy, which will be similar to the product.

Your `ReviewProxyTestSpec.scala` file should look something like this:

```
package proxy.test
    import org.scalatest._
    import proxy.ReviewProxy
    class ReviewProxyTestSpec extends FlatSpec
with Matchers {
      "A Review REST Proxy " should "return all
reviews" in {
        val reviews = ReviewProxy.listAll().get
        reviews shouldNot(be(null))
        reviews shouldNot(be(empty))
        for( r <- reviews){
          println(r)
        }
```

```
    }
  }
```

Here, we used the proxy ideas to test the review. We called the proxy using the `listAll` function to get all the reviews. Later, we will check to see if the review is not null. We will print all the reviews. Finally, it's time to move to the last proxy test--the image proxy.

Your `ImageProxyTestSpec.scala` should look something like this:

```
package proxy.test
    import org.scalatest._
    import proxy.ImageProxy
    import scala.concurrent.Future
    import
play.api.libs.concurrent.Execution.Implicits.defau
ltContext
    import java.util.concurrent.CountDownLatch
    class ImageProxyTestSpec extends FlatSpec with
Matchers {
      "A Image REST Proxy " should "return all
images" in {
        val images = ImageProxy.listAll().get
        images shouldNot(be(null))
        images shouldNot(be(empty))
        for( i <- images){
          println(i)
        }
      }
    }
```

Same deal goes for the image proxy. We have all our tests; now, we can run the tests. You will need to make sure our `ReactiveWebStore` Play framework app is up and running.

Let's run this test with sbt:

open your console and type in `$ sbt test`

```
> test
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Review { id: Some(1) ,productId: 1,author: diego,comment: yeah ok }
Review { id: Some(2) ,productId: 3,author: diego,comment: ok ok  }
Review { id: Some(3) ,productId: 3,author: whoknows,comment: coool }
[info] ReviewProxyTestSpec:
[info] A Review REST Proxy
[info] - should return all reviews
Image { productId: 1,url: http://wwww.google.com.be/img.png}
Product { id: 1,name: ball, details: ball, price: 9.88}
Product { id: 2,name: ball, details: ball, price: 9.88}
Product { id: 7,name: ball, details: ball, price: 9.88}
Product { id: 8,name: ball, details: ball, price: 9.88}
Product { id: 9,name: ball, details: ball, price: 9.88}
Product { id: 10,name: pokeball, details: great ball, price: 89.44}
[info] ProductProxtTestSpec:
[info] A Product Rest proxy
[info] - should return all products
Image { productId: 1,url: http://wwww.google.com.be/img.png}
[info] ImageProxyTestSpec:
[info] A Image REST Proxy
[info] - should return all images
[info] A Image REST Proxy
[info] - should suffer backpressure
[info] Run completed in 18 seconds, 935 milliseconds.
[info] Total number of tests run: 4
[info] Suites: completed 3, aborted 0
[info] Tests: succeeded 4, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 22 s, completed 17/09/2016 01:01:06
>
```

Alright, it all works! Our next step will be to add back pressure.

Adding back pressure

Back pressure is a well-known concept in the automotive industry. Nowadays, this term is used in software engineering as well. Back pressure in the automotive world refers to the pressure opposed to the desired flow of gasses in a confined place, such as a pipe. For software engineering, it is often related to slowing down a producer, which can be an application, a stream processing engine, or even the user itself.

When we are executing REST, it's easy to reach a situation where the client can saturate the server. This can be a security breach too,

which is also known as the **Denial Of Service** (**DOS**) attack.

There are two architectural scenarios. In the first scenario, your REST API is internal, and you just have consumers in your company. In the second scenario, you are making the REST API a public API so that it will be open to the whole Internet. For this scenario, you really should have back pressure, also known as throttling.

It's possible to scale our architecture in order to handle more users. We will discuss this, and the scalability techniques, in Chapter 10, *Scaling Up*.

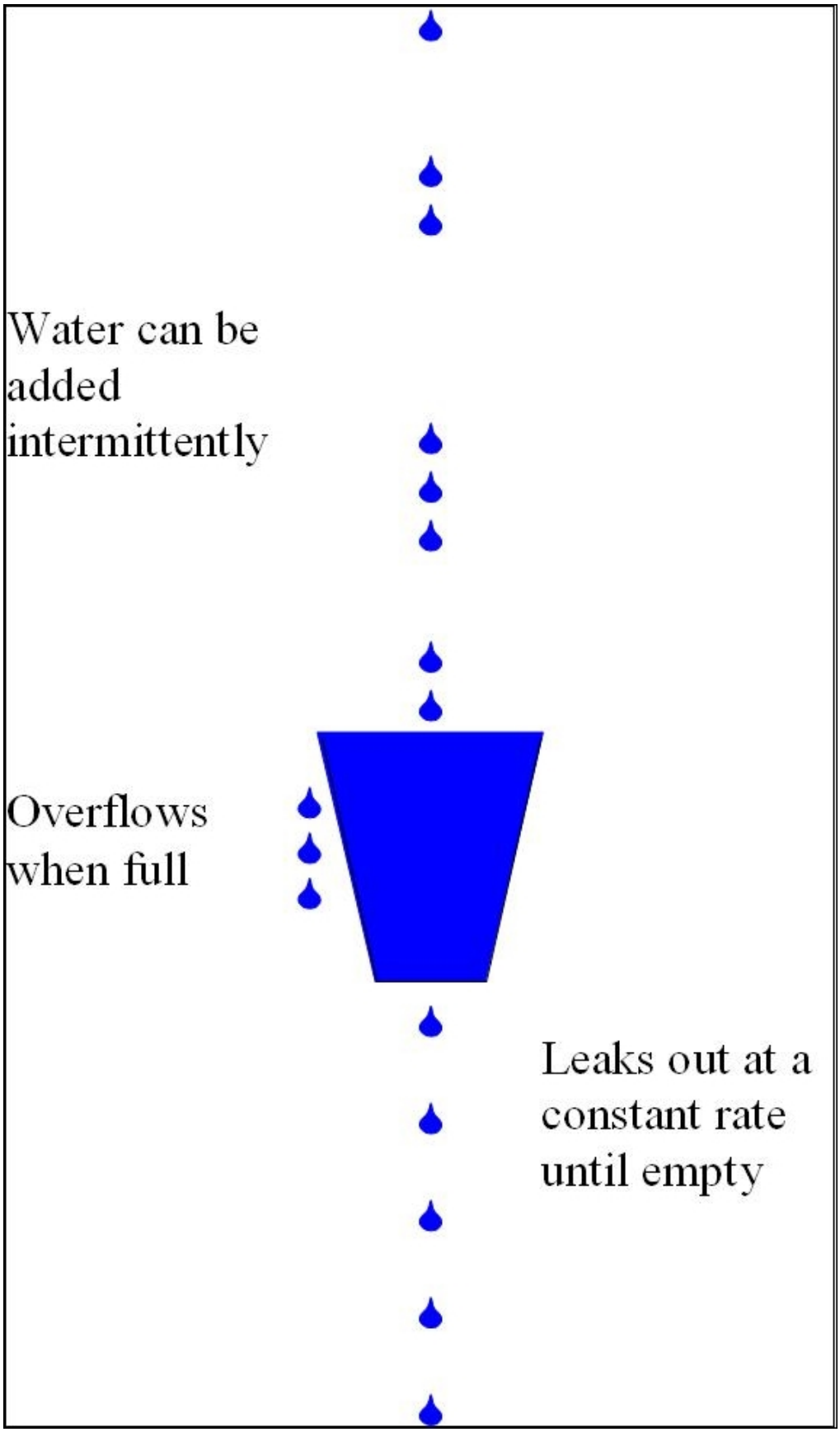
Currently, there are several ways to apply back pressure. For instance, if our code is pure RxScala/RxJava, we can apply back pressure on observables. More details can be found at https://github.com/ReactiveX/RxJava/wiki/Backpressure.

As we are exposing a REST interface, we will add back pressure on the controller, so we will need to create a new class with the back pressure code.

There are some algorithms for back pressure; we will use the leaky bucket algorithm. The algorithm itself is very simple--just 30 lines of Scala code.

The leaky bucket algorithm

The leaky bucket metaphor is pretty simple. Let's take a look at it in the following diagram:


Water can be added intermittently
Overflows when full
Leaks out at a constant rate until empty

The metaphor behind the algorithm is based around a bucket with holes. The water flows or drips into the bucket and leaks through the holes of the bucket. If there is too much water, and the bucket is full of water, the water will spill out of the bucket--in other words, it will be discarded.

This algorithm is used in network programing, and also by the telecommunication industry. The API manager solutions are also use cases for this algorithm.

This concept allows RATE limit constraints. We can express the back pressure rate limits in requests per time. Time, in this case, is often measured in seconds or minutes, so we have **Requests Per Second** (**RPS**) or **Requests Per Minute** (**RPM**).

You can implement this algorithm with a queue. However, in our implementation, we will not use a queue; we will use time in order to control the flow. Our implementation will also be lock free, or non-blocking, as we won't use threads or external resources.

Now is the time to code a leaky bucket in Scala. First of all, we will create this code for the `ReactiveWebStore` application. We will need to create a new package located at `ReactiveWebStore/app`. The new package name will be `backpressure`.

Scala leaky bucket implementation

Your `LeakyBucket.scala` file should have the following files:

```
package backpresurre
    import scala.concurrent.duration._
    import java.util.Date
    class LeakyBucket(var rate: Int, var
perDuration: FiniteDuration) {
      var numDropsInBucket: Int = 0
      var timeOfLastDropLeak:Date = null
      var msDropLeaks = perDuration.toMillis
      def dropToBucket():Boolean = {
```

```
        synchronized {
          var now = new Date()
          if (timeOfLastDropLeak != null) {
            var deltaT = now.getTime() -
timeOfLastDropLeak.getTime()
            var numberToLeak:Long = deltaT /
msDropLeaks
              if (numberToLeak > 0) {
                if (numDropsInBucket <=
numberToLeak) {
                  numDropsInBucket -=
numberToLeak.toInt
                } else {
                numDropsInBucket = 0
              }
              timeOfLastDropLeak = now
            }
          }else{
            timeOfLastDropLeak = now
          }
          if (numDropsInBucket < rate) {
            numDropsInBucket = numDropsInBucket +
1
            return true;
          }
          return false;
        }
      }
    }
```

As you can see here, we created a Scala class that receives two parameters: `rate` and `perDuration`. Rate is an integer, which shows how many requests we are able to handle before applying back pressure. `PerDuration` is a Scala `FiniteDuration`, which can be any measure of time, such as milliseconds, seconds, minutes, or hours.

This algorithm keeps track of the time for the last drop in the bucket. As you can see, the code is synchronized, but it is fine because we won't call either external resources or threads.

First, we will get the current time with new a `Date()`. The first time we run the algorithm, we will fail on the `else` statement, and we will get the current time as last leak.

The second time it runs, it will enter on the first `If` statement. Then, we will calculate the delta (diff) between the last leak and now. This delta will be divided by the time in milliseconds that you passed on `perDuration`. If the delta is greater than 0, then we leak; otherwise we drop. Then, we will capture the time again for the last leak.

Finally, we will check the drop rate. If the rate is smaller, we will increment and return true, which means the request can proceed; otherwise, we will return false, and the request should not proceed.

Now that we have this algorithm coded in Scala, we can call for one of our controllers. We will add this back pressure on the image REST API.

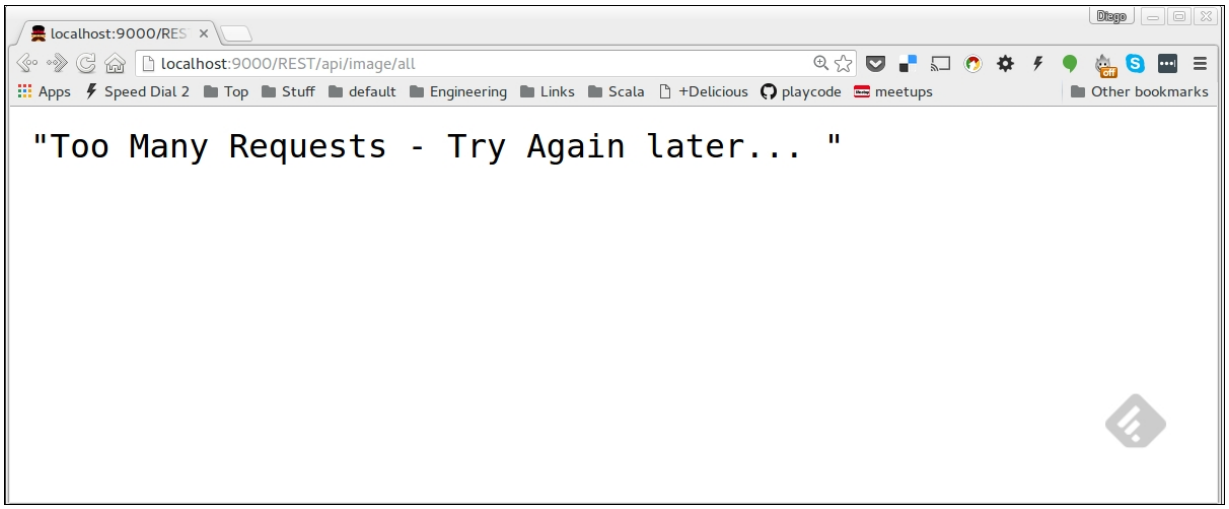
Your `RestApiController.scala` should look something like this:

```
package controllers
    class RestAPIController @Inject()
    (val productService:IProductService,
      val reviewService:IReviewService,
      val imageService:IImageService) extends
Controller {
        import

play.api.libs.concurrent.Execution.Implicits.defau
ltContext
        // REST of the Controller...
        import scala.concurrent.duration._
        var bucket = new LeakyBucket(5, 60
seconds)
        def processImages = {
          val future = imageService.findAll()
          val images = Awaits.get(5,future)
          val json = ImagesJson.toJson(images)
          json
        }
        def processFailure = {
          Json.toJson("Too Many Requests - Try
Again later... ")
        }
        def listAllImages = Action {
          bucket.dropToBucket() match {
            case true => Ok(processImages)
```

```
                case false =>

        InternalServerError(processFailure.toString())
                }
            }
        }
```

Here, we will create a leaky bucket with five requests per minute. We have two functions: One to process images that will call the service and convert the objects to JSON, and the other to process failures. The `processFailure` method will just send a message saying that there are too many requests, and we can't accept requests right now.

So, for the `listAllImages` function, we will just call the bucket trying to drop and use the Scala pattern matcher in order to process the proper response. If the response is true, we will return JSON with a 200 HTTP code. Otherwise, we will return a 500 internal error and deny that request. Here, we implemented a global RATE limiter; however, most of the time, people perform this operation per user. Now, let's open the web browser and try more than five requests within a minute. You should see something like the following screenshot at `http://localhost:9000/REST/api/images/all`:



Alright, it works! If you wait a minute and make requests again, you will see that the flow gets back to normal. The next step is to add a new client test, because we know that if we call an image too much in our REST API, we will be throttled.

We will need to add one more test in the `rest-client` Scala project. For that, we will need to change the `ImageProxyTestSpec`.

Testing back pressure

Your `ImageProxyTestSpec.scala` should look something like this:

```
package proxy.test
    class ImageProxyTestSpec extends FlatSpec with
Matchers {
      // REST of the tests...
      "A Image REST Proxy " should "suffer
backpressure" in {
        val latch = new CountDownLatch(10)
        var errorCount:Int = 0
        for(i <- 1 to 10){
          Future{
            try{
              val images =
ImageProxy.listAll().get
              images shouldNot(be(null))
              for( i <- images){
                println(i)
              }
            }catch{
              case t:Throwable => errorCount += 1
            }
            latch.countDown()
          }
        }
        while( latch.getCount >= 1 )
        latch.await()
        errorCount should be >=5
      }
    }
```

So, for this test, we will call `ImageProxy` ten times. We know that not all requests will be served as we have back pressure on the server. Here, we can call the proxy with a try...catch block and have an error

counter. Each time it fails, we can increment it. So, here, we are expected to fail at least five times.

We are creating the code with Features because we want the requests to happen at the same time. We will need to use `CountDownLatch` function, which is a Java utility class that lets us wait for all Futures to finish before moving on. This is done by the `countDown` function. Every time we execute `countdown`, we decrement the internal counter. As you can see, we created the `CountDownLatch` function with ten.

Finally, we have a `while` block to wait for until the counter has pending Futures. Now we wait. Once it's all done, we can check the error count; it should be at least five. That's it. We have tested our back pressure mechanism and it all works!

Now, it is time to move to the next feature that we will implement in our application: Swagger--we will add Swagger support to our REST API.

Adding Swagger support

Swagger (http://swagger.io/) is a simple JSON and UI representation tool for REST APIs. It can generate code in several languages. It also creates a very nice documentation, which is also a runnable Swagger code that allows you to call REST web services from the documentation it generates. We will need to make some changes in our Play framework application in order to get Swagger up and running with the Play framework.

First of all, we will need to add the Swagger dependency to `build.sbt`:

```
// Rest of build file...
    libraryDependencies ++= Seq(
      // Rest of other deps...
      "io.swagger" %% "swagger-play2" % "1.5.2-
SNAPSHOT"
    )
```

As you can see, we are using a Snapshot version. Why use Snapshot? Right now, it is not supported on a stable version. In order to resolve this dependency, we will need to use Git and clone another project. You can get more details at https://github.com/CreditCardsCom/swagger-play. Basically, you will need to write a command as follows:

```
$ git clone
https://github.com/CreditCardsCom/swagger-play.git
$ cd swagger-play/
$ sbt publishLocal
```

Now, we will need to enable Swagger on `ReactiveWebStore/conf/application.conf`.

Your `application.conf` file should look something like this:

```
play.modules {
      enabled +=
"play.modules.swagger.SwaggerModule"
    }
```

Next, we can change our controller in order to add Swagger support. Swagger has annotation in order to map the REST operations.

Your `RestAPIController.scala` file should look something like this:

```
package controllers
    @Singleton
    @Api(value = "/REST/api", description = "REST
operations on
    Products, Images and Reviews. ")
    class RestAPIController @Inject()
    (val productService:IProductService,
      val reviewService:IReviewService,
      val imageService:IImageService) extends
Controller {
        import

play.api.libs.concurrent.Execution.Implicits.defau
ltContext
```

```
        @ApiOperation(
          nickname = "listAllProducts",
          value = "Find All Products",
          notes = "Returns all Products",
          response = classOf[models.Product],
          httpMethod = "GET"
        )
        @ApiResponses(Array(
          new ApiResponse(code = 500, message =
"Internal Server
          Error"),
          new ApiResponse(code = 200, message =
"JSON response with
          data")
        )
      )
      def listAllProducts = Action {
        val future = productService.findAll()
        val products = Awaits.get(5,future)
        val json = ProductsJson.toJson(products)
        Ok(json)
      }
      @ApiOperation(
        nickname = "listAllReviews",
        value = "Find All Reviews",
        notes = "Returns all Reviews",
        response = classOf[models.Review],
        httpMethod = "GET"
      )
      @ApiResponses(Array(
        new ApiResponse(code = 500, message =
"Internal Server Error"),
        new ApiResponse(code = 200, message =
"JSON response with
        data")
      )
      )
      def listAllReviews = Action {
        val future = reviewService.findAll()
        val reviews = Awaits.get(5,future)
        val json = ReviewsJson.toJson(reviews)
        Ok(json)
      }
      import scala.concurrent.duration._
      var bucket = new LeakyBucket(5, 60 seconds)
      def processImages = {
        val future = imageService.findAll()
        val images = Awaits.get(5,future)
```

```
        val json = ImagesJson.toJson(images)
        json
      }
      def processFailure = {
        Json.toJson("Too Many Requests - Try Again
later... ")
      }
      @ApiOperation(
        nickname = "listAllImages",
        value = "Find All Images",
        notes = "Returns all Images - There is
throttling of 5
        reqs/sec",
        response = classOf[models.Image],
        httpMethod = "GET"
      )
      @ApiResponses(Array(
        new ApiResponse(code = 500, message =
"Internal Server Error"),
        new ApiResponse(code = 200, message =
"JSON response with
        data")
      )
      )
      def listAllImages = Action {
        bucket.dropToBucket() match {
          case true => Ok(processImages)
          case false =>
InternalServerError(processFailure.toString())
        }
      }
    }
```

Here we have several annotations. First of all, we have an `@Api` annotation at the top of the class. With this annotation, we will define the root path of the REST API. Then, for each REST API operation, we have the `@ApiOperation` and `@ApiResponses` annotations. The `@ApiOperation` defines the REST API itself where you can define the parameters and the HTTP verb, and also put some notes (documentation). It's also possible to describe the result; in our case, it will be a JSON representation of the models.

That's it! We have the controller mapped to Swagger. The next step is to add a route for Swagger. This needs to be done by adding a

new line as shown in the following piece of code which, is located at `ReactiveWebStore/conf/routes`:

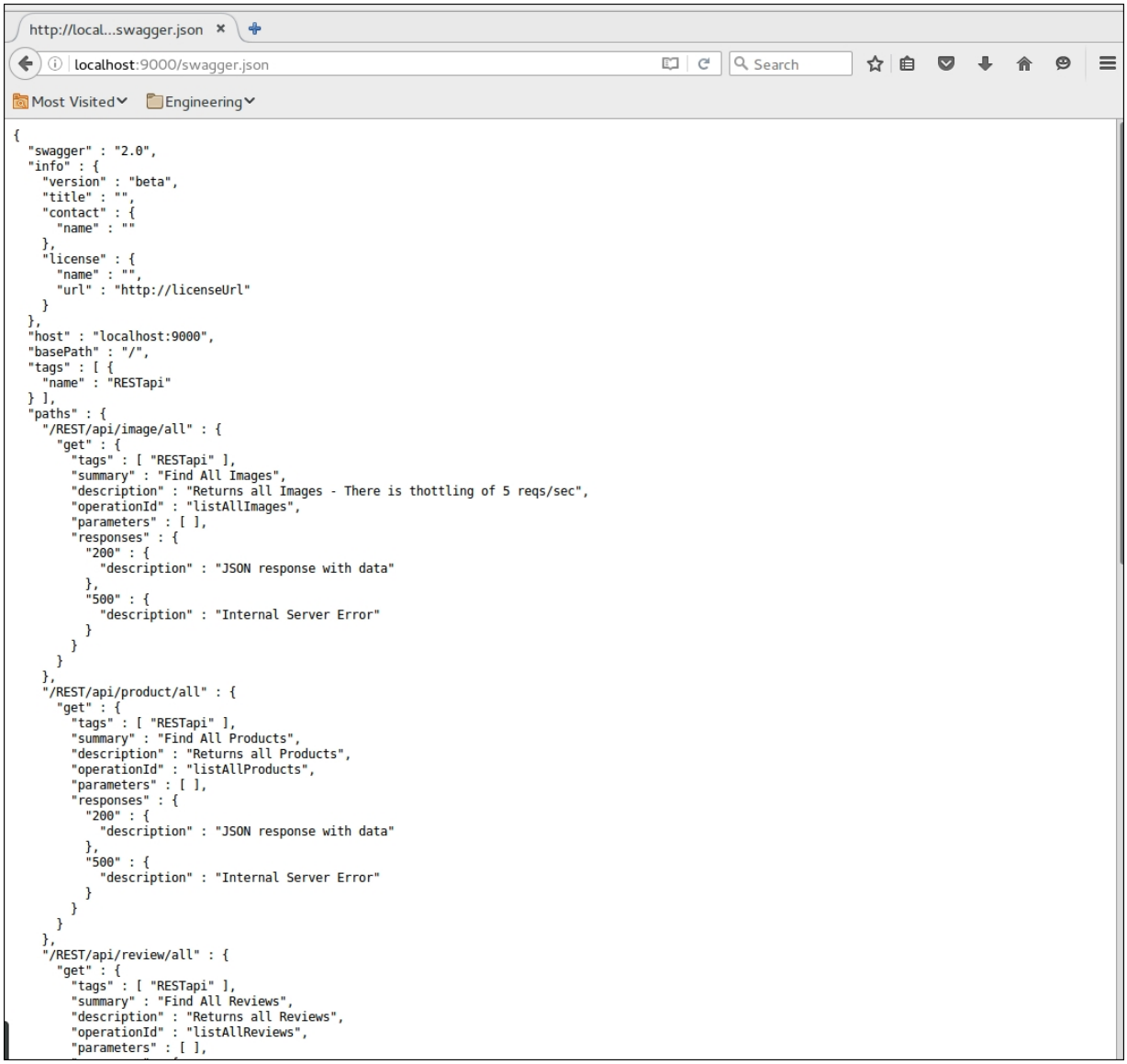
```
// REST of the other routes..
    GET     /swagger.json
controllers.ApiHelpController.getResources
```

Swagger UI

Swagger will generate a JSON response for our REST API. It's possible to use the Swagger UI, which is very nice and gives lots of facilities to the developer. There are two ways we can work with the Swagger UI: We can use it as a standalone or we can embed the Swagger UI into our Play framework application.

The strategy we will pick here is embedding the Swagger UI in our application. If you have multiple REST APIs with multiple Play applications or microservices, it is a good idea to have the standalone installation of the swagger UI.

In the previous steps, we enabled Swagger in our application. Open your browser and type `http://localhost:9000/swagger.json`. You can follow the instructions at http://swagger.io/swagger-ui/. In summary, you will get the following output:



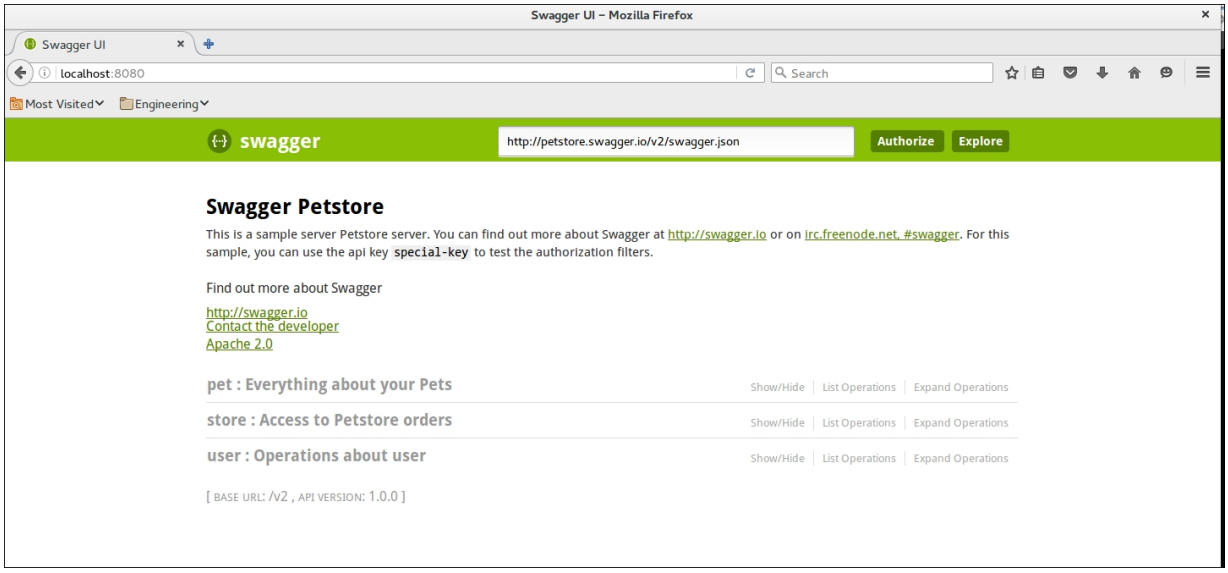
Build and install Swagger Standalone

Now we will download, build, and install Swagger Standalone. Let's get started by writing the following lines of code:

```
$ sudo apt-get update $ sudo apt-get install nodej
$ sudo apt-get install npm
$ git clone https://github.com/swagger-
api/swagger-ui.git $ cd swagger-ui/ $ sudo -E npm
```

```
install -g $ sudo -E npm run build $ npm run serve
$ GOTO: http://localhost:8080/
```

Once you start the Swagger UI, you can go to the browser, where you will see the following output:



Now, let's embed the Swagger UI into our Play framework application.

We will need to copy the content from `/swagger-ui/dist/` into our Play framework application under `ReactiveWebStore/public`. Then, we will create a folder called `swaggerui`.

We will need to edit one file in order to put our swagger JSON URI. Open `ReactiveWebStore/public/swaggerui/index.html` and change the 40 to the following code line:

```
url = "http://localhost:9000/swagger.json";
```

That's it. Now we will need to create a link from our application to embed the swagger UI. So, let's change `ReactiveWebStore/app/views/index.scala.html`.

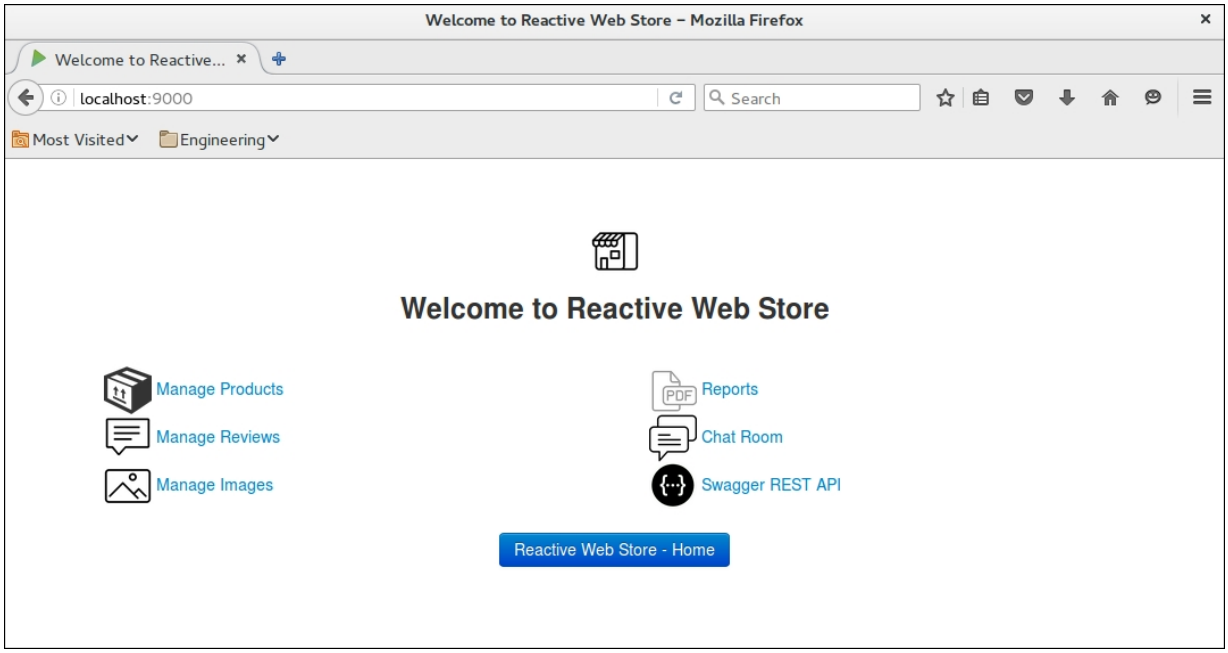
Your `index.scala.html` file should look something like this:

```
@(message: String)(implicit flash:Flash)
    @main("Welcome to Reactive Web Store"){
      <div class="row-fluid">
        <BR>
        <div class="span12">
          <div class="row-fluid">
            <div class="span6">
              <a href="/product"><img height="42"
width="42"

src="@routes.Assets.at("images/product.png")">
Manage
              Products</a><BR>
              <a href="/review"><img height="42"
width="42"

src="@routes.Assets.at("images/review.png")">
Manage
              Reviews</a><BR>
              <a href="/image"><img height="42"
width="42"

src="@routes.Assets.at("images/image.png")">
Manage
              Images</a><BR>
            </div>
            <div class="span6">
            <a href="/reports"><img height="42"
width="42"

src="@routes.Assets.at("images/reports.png")">
Reports </a>
            <BR>
            <a href="/chat/index_socket"><img
height="42" width="42"

src="@routes.Assets.at("images/chat.png")"> Chat
Room </a>
            <BR>
            <a
```

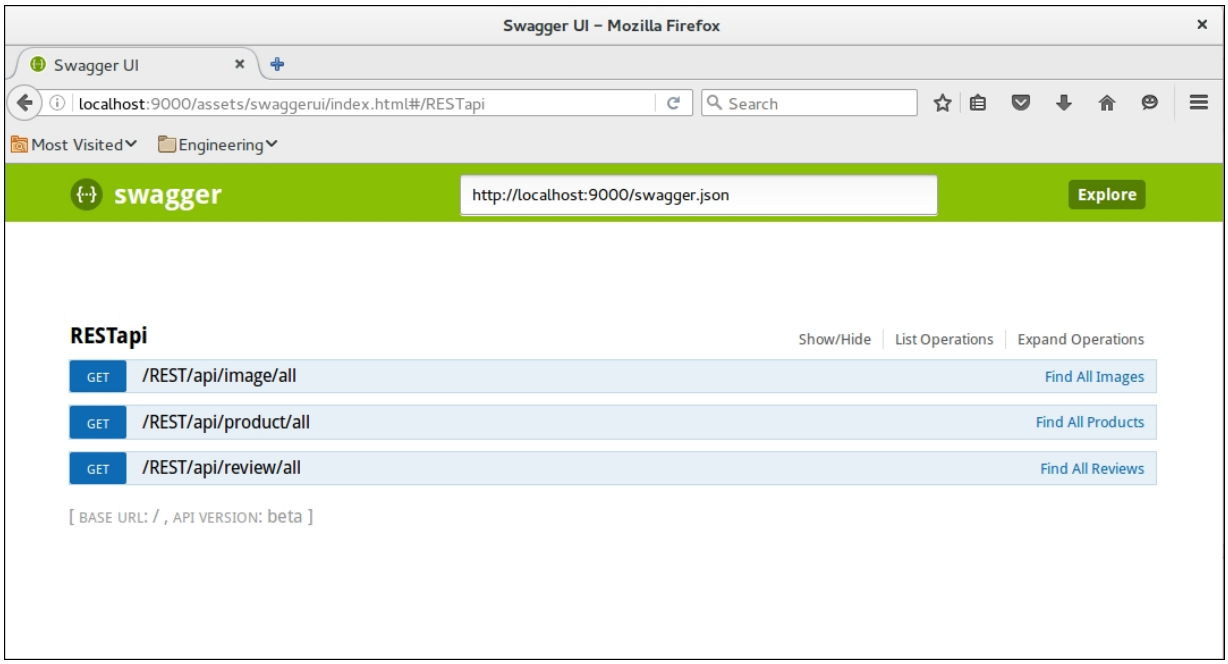
```
href="/assets/swaggerui/index.html"><img
height="42"
            width="42"

src="@routes.Assets.at("images/swagger.png")">
Swagger REST
            API </a><BR>
          </div>
        </div>
      </div>
    </div>}
```

Now we can run our Play application with `$ activator run`.

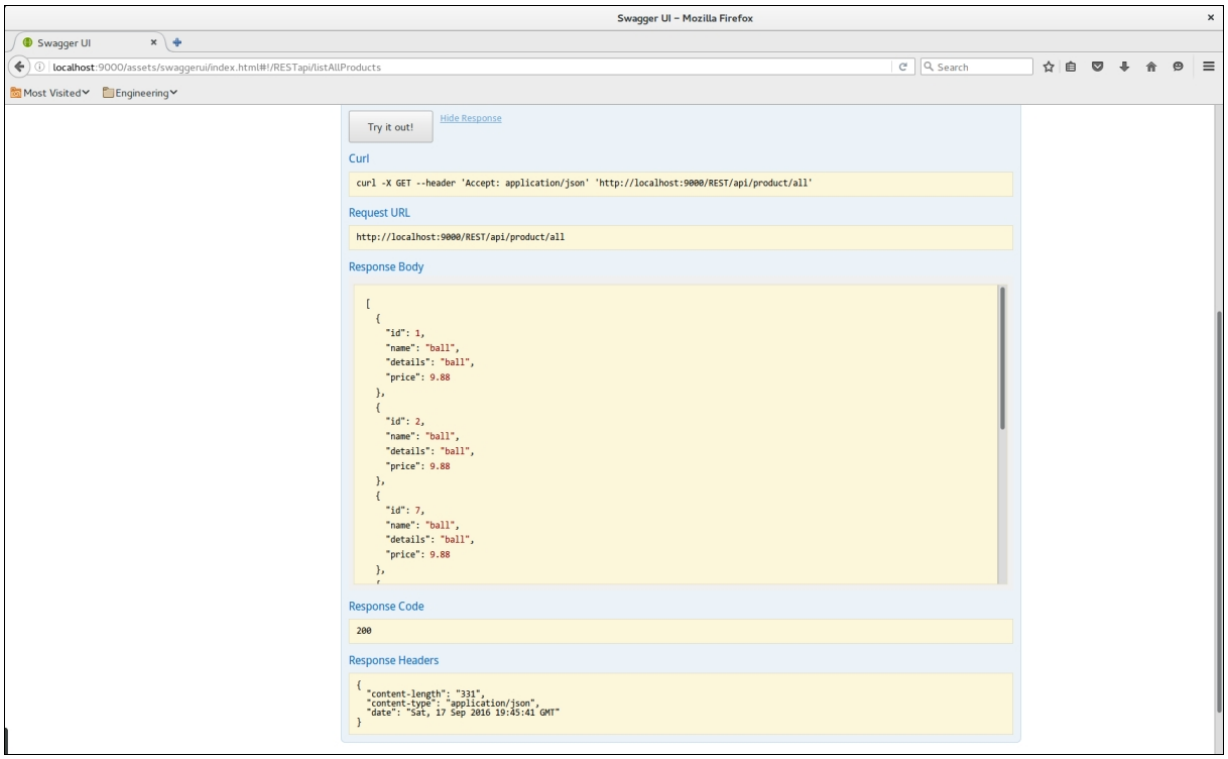
Open the browser and go to `http://localhost:9000/`. You will see the following screenshot:



Now we can open the Swagger UI by clicking on the Swagger REST API link or by just going to `http://localhost:9000/assets/swaggerui/index.html`. It should look something like this:



As you can see, the Swagger UI is very nice. You can click on each operation and see more details on how they work, which HTTP verb they use, and what the URI is. There is a **Try it out!** button. Let's click on the **Try it out!** button for products, which would look something like this:



As you can see, we have our JSON result and also some CURL samples as well.

Summary

In this chapter, you learned how to design a REST API and changed your Play framework application in order to have Swagger support. You created a Scala client library using proxy techniques, as well as Scala tests for the APIs. Additionally, you were introduced to back pressure using the leaky bucket algorithm.

In the next chapter, which will be the final chapter, you will learn about software architecture and scalability/resiliency techniques, such as discoverability, load balancers, caches, Akka Cluster, and the Amazon Cloud.

Chapter 10. Scaling up

In the previous chapters, we built a Scala and Play framework application. We used the most effective frameworks and tools around the Scala ecosystem, such as Play framework and Akka; and we used the Reactive and Functional Programming techniques using Futures and RxScala. Additionally, we created reports with Jasper and Chat with WebSockets. This is the final chapter, and we will learn how to deploy and scale our application.

In this chapter, we will cover the following topics:

- Standalone deploy
- Architecture principles
- Reactive drivers and discoverability
- Mid-Tier load-balancer, timeouts, Back pressure, and caching
- Scaling up microservices with an Akka cluster
- Scaling up the infrastructure with Docker and AWS cloud

Standalone deploy

Throughout this book, we used the Activator and SBT build and development tools. However, when we talk about production, we can't run the application with SBT or Activator; we need to do a standalone deploy.

What about standard Java Servlet containers, such as Tomcat? Tomcat is great; however, Play is greater. You won't get better performance by deploying your app on Tomcat. The standalone play uses Netty, which has superior network stack.

There are some small changes that we will need to make in order to deploy our application for Jasper reports. Don't worry; these changes are very simple and straightforward.

Reports folder

We need to move the reports template (JRXML files) from the source folder to the public folder. Why do we need to do this? Because when we generate the standalone deploy, they won't be included in the application jars. What is inside the public folder will be packed and deployed into a proper JAR file. That's why we need to make this change.

Create a folder called `reports` at `ReactiveWebStore/public/`. Then move all the JRXML files there.

Changing report builder

As our templates will be inside a JAR file, we need to change the loading logic in order to get the templates properly. Under `ReactiveWebStore/app/report/`, we need to change `ReportBuilder.scala`, which should look something like this after editing:

```
package reports
    object ReportBuilder {
      private var
reportCache:scala.collection.Map[String,Boolean] =
      new
scala.collection.mutable.HashMap[String,Boolean].e
mpty
      def generateCompileFileName(jrxml:String):
String =
      "/tmp/report_" + jrxml + "_.jasper"
      def compile(jrxml:String){
        if(reportCache.get(jrxml).getOrElse(true))
{
          val design:JasperDesign =
JRXmlLoader.load(

Play.resourceAsStream("/public/reports/" + jrxml)
          (Play.current).get )

JasperCompileManager.compileReportToFile(design,
          generateCompileFileName(jrxml))
          reportCache += (jrxml -> false)
        }
      }
      def toPdf(jrxml:String):ByteArrayInputStream
= {
        try {
          val os:OutputStream = new
ByteArrayOutputStream()
          val
reportParams:java.util.Map[String,Object] =
          new java.util.HashMap()
          val con:Connection =

DriverManager.getConnection("jdbc:mysql://localhos
```

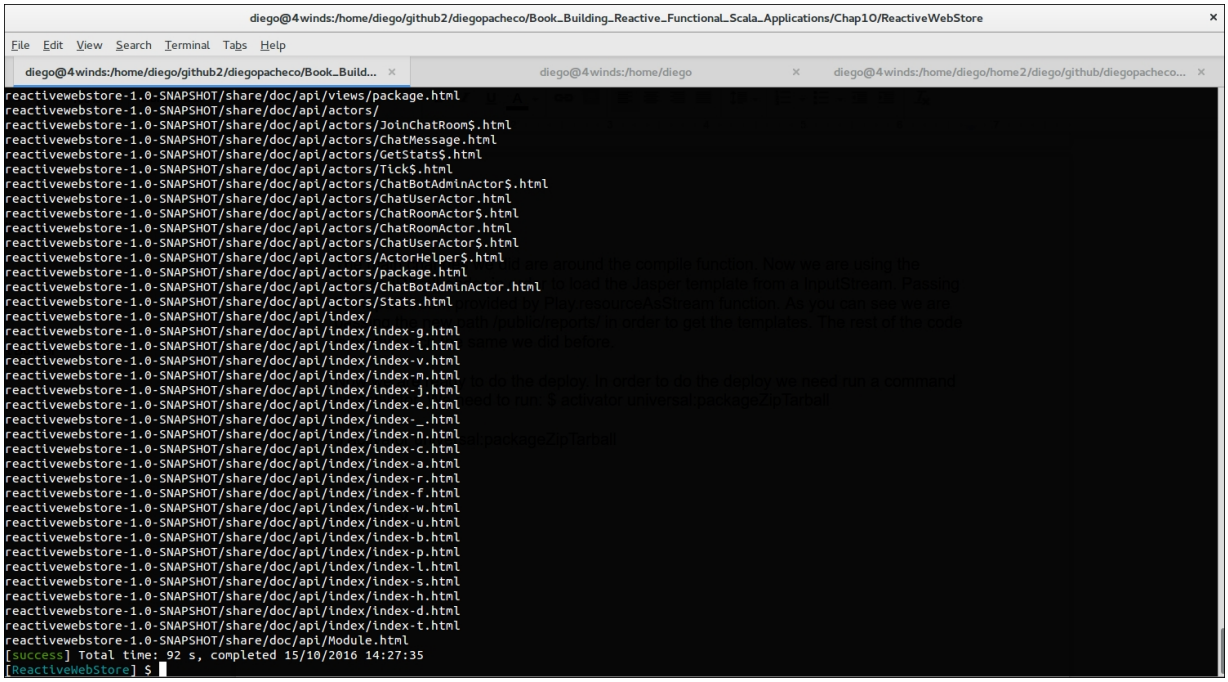
```
t/RWS_DB?
            user=root&password=&useUnicode=
            true&useJDBCCompliantTimezoneShift=

true&useLegacyDatetimeCode=false&serverTimezone=UT
C")
          compile(jrxml)
          val jrprint:JasperPrint =

JasperFillManager.fillReport(generateCompileFileNa
me(jrxml),
          reportParams, con)
          val exporter:JRPdfExporter = new
JRPdfExporter()
          exporter.setExporterInput(new
SimpleExporterInput(jrprint));
          exporter.setExporterOutput(
          new
SimpleOutputStreamExporterOutput(os));
          exporter.exportReport()
          new ByteArrayInputStream

((os.asInstanceOf[ByteArrayOutputStream]).toByteAr
ray())
        }catch {
          case e:Exception => throw new
RuntimeException(e)
          null
        }
      }
    }
```

The main changes that we made were around the `compile` function. Now, we are using the Jasper JRXmlLoader in order to load the Jasper template from an `InputStream` method. Passing the `InputStream` method provided by the `Play.resourceAsStream` function. As you can see, we are passing the new path, `/public/reports/`, in order to get the templates. The rest of the code is pretty much the same as the one we executed earlier.

Now we are ready to deploy. In order to do so, we will need to run a command on `activator`, which is as follows:

```
$ activator universal:packageZipTarball
```

You will see the following result:



As soon as the task is finished, our app will be packed into the `ReactiveWebStore/target/universal/` directory, and you will see a `reactivewebstore-1.0-SNAPSHOT.tgz` file.

Then you need to extract the file and you shall have the following directory:

- `reactivewebstore-1.0-SNAPSHOT`
- `bin`: Scripts to run the app
- `conf`: All config files: routes, logging, messages
- `bib`: All JARs including third-party dependencies
- `share`: All documentation about the app

Defining the secret

Before we run our standalone application, we will need to apply one more change. We need to change the default secret. Locate the `reactivewebstore-1.0-SNAPSHOT/conf/application.conf` file.

Change the secret to following in the `application.conf` file:
play.crypto.secret = "changeme"

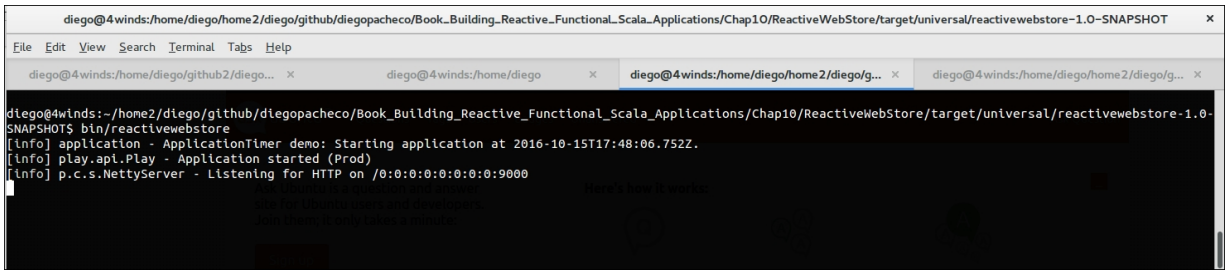
You will need to provide a different value. It can be anything, as long as you don't call it `changeme`. If you don't change this, your application will not boot up. You can get more information at http://www.playframework.com/documentation/latest/ApplicationSecret.

If you just want to test the deploy for now, let's call it `playworks`.

Now, we are all set to start the application.

Running the standalone deploy

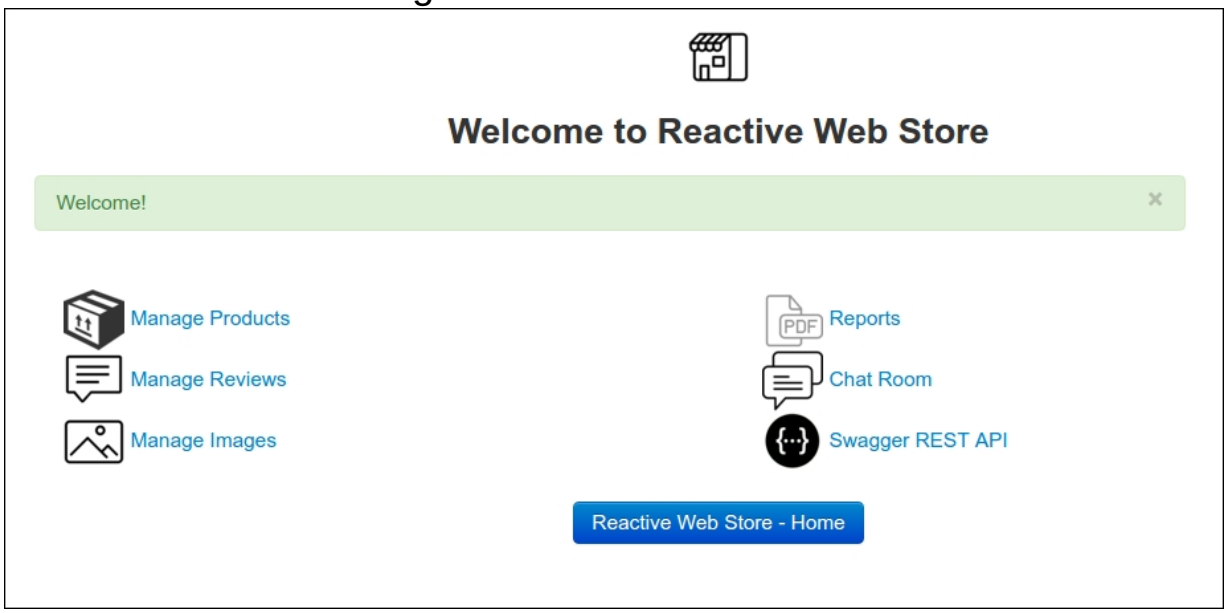
In order to run the app, we will use the script generated by the universal task. Go to the `reactivewebstore-1.0-SNAPSHOT` directory and then run `$ bin/reactivewebstore`, which would look something like this:



Now, you can open your browser and go to `http://localhost:9000/`.

That's it; we have our app up and running. Feel free to test all the features we built--they all work!

It should look something like this:



Architecture principles

Scalability is about handling more users, traffic, and data; and in order to do it, we will need to apply some principles and techniques. Our application is already using the most modern techniques and technologies, such as functional and ReactiveX programming, RxScala, Akka framework, Play, and much more. However, in order to scale, we will need to have an infrastructure in place and other kinds of system that will allow us to handle more users.

A good application architecture should be created around the following principles:

- **Separation of Concerns** (**SOC**) (more details at https://en.wikipedia.org/wiki/Separation_of_concerns)
- Service Orientation (SOA/microservices)
- Performance
- Scalability/Resiliency

Let's see these principles in detail.

Service orientation (SOA/microservices)

Service orientation is about having a higher level of abstraction, which is also called services or microservices. SOA is not about a specific technology, but about principles such as shared services, flexibility, and intrinsic operability. If you want to learn more about SOA, check out the SOA Manifesto at http://www.soa-manifesto.org/. Microservices is a particular flavor of SOA where the main difference is the focus on the granularity, autonomy, and isolation. If you want to learn more about microservices, you can check out https://www.linkedin.com/pulse/soa-microservices-isolation-evolution-diego-pacheco as well as http://microservices.io/.

Performance

The right algorithm might make your application work smoothly and the wrong algorithm might make your users have a poor experience. Performance is achieved by design--first of all, choose the right set of collections and the right set of algorithms and frameworks. However, performance needs to be measured and tuned eventually. The practice you should do in your application with regard of performance is stress testing. The best stress testing tools in the Scala ecosystem is Gatling. Gatling (http://gatling.io/#/) allows you to code in Scala using a very simple yet powerful DSL. Gatling focuses on HTTP and latency percentiles and distributions, which is the right thing to do nowadays. Latency is not only used for the sake of performance and scalability, but it is also heavily related to user experience as everything is online.

Scalability/Resiliency

Scalability is one of the main reasons why we do software architecture, because an architecture that does not scale does not have business value. We will continue talking about Scalability principles during this chapter. Resiliency is about how much the system can resist and keep operating under the most adverse situations, such as hardware failure or infrastructure failure. Resiliency is an old term. Currently, there is a new and more modern and accurate principle called antifragility. This principle was well developed and used in practice by Netflix. Antifragility is about systems and architecture that can adapt and fail over to other systems and other operational modes to keep working no matter what. If you want to know more about antifragility, you can visit http://diego-pacheco.blogspot.com.br/2015/09/devops-is-about-anti-fragility-not-only.html and http://diego-pacheco.blogspot.com.br/2015/11/anti-fragility-requires-chaos.html.

Scalability principles

Having architecture around these principles makes it possible to scale your application up. However, we will still need to rely on other principles and techniques to scale it.

There are several principles and techniques for scalability, which are as follows:

- Vertical and horizontal scaling (up and out)
- Caching
- Proxy
- Load balancer
- Throttling
- Database cluster
- Cloud computing/containers
- Auto Scaling
- Reactive drivers

Vertical and horizontal scaling (up and out)

You can add more resources, have better hardware, or you can add more boxes. These are the two basic ways to scale. You can always improve and tune your app to use fewer resources and get more from a single box. Recently, there were several improvements in this area around reactive programming that uses fewer resources and delivers more throughput. However, there are always limits to which a single box can provide in sense of scaling up, which is why we always need to be able to scale out.

Caching

Databases are great. However, there is a latency cost to call a traditional database. A great way to fix this is having a memory cache, which you can use as a subset of your data and get the benefit of fast retrieval. Play framework has cache support. If you want to learn more, check out https://www.playframework.com/documentation/2.5.x/ScalaCache.

There are other options in sense of caching. There are lots of companies that use the memory as a definitive data store nowadays. For this, you can consider tools such as Redis (http://redis.io/) and Memcached (https://memcached.org/). However, if you want to scale Redis and Memcached, you will need something like Netflix/Dynomite (https://github.com/Netflix/dynomite). Dynomite provides a cluster based on AWS Dynamo paper for Redis, which has the following benefits:

- High throughput and low latency
- Multi-region support (AWS cloud)
- Token aware
- Consistent hashing
- Replication
- Sharding
- High availability

> **Note**
>
> If you want to learn more about dynomite, check out https://github.com/Netflix/dynomite/wiki.

Load balancer

A load balancer is a key tool to scale servers. So, let's say, you have 10 boxes with our Play framework application or 10 Docker containers. We will need something in front of our application to distribute the traffic. There are several servers that can do this, such as NGINX (https://nginx.org/) and Apache HTTP Server

(https://httpd.apache.org/). If you want to scale your application, this is the easiest solution for it. Configuration and more details can be found at https://www.playframework.com/documentation/2.5.x/HTTPServer#Setting-up-a-front-end-HTTP-server.

Load balancers are often proxy servers as well. You can use them to have HTTPS support. If you want, you can have HTTPS on Play framework as well (https://www.playframework.com/documentation/2.5.x/ConfiguringHttps). Keep in mind that you will need to change swagger embedded installation as all the code that we have points to the HTTP interface. If you are doing deploys in the AWS cloud, you will need to change some of the configuration to forward the proxies, which you can find at https://www.playframework.com/documentation/2.5.x/HTTPServer#Setting-up-a-front-end-HTTP-server.

Throttling

This is also known as Back pressure. We covered throttling in Chapter 9, *Design Your REST API*. You can get more details there. However, the main idea is to limit the request for each user. This is also a way to make sure that a single user does not steal all computational resources. This is also important from the security point of view, especially for the services that are Internet-facing or also known as edge. Another great way to protect and have this capability is using Netflix/Zuul (https://github.com/Netflix/zuul).

Database cluster

Sometimes, the problem is not on the application side, but in the database. When we talk about scalability, we need be able to scale everything. We need to have the same concepts for databases that we have for the Mid-Tier. For databases, it is important to work with the following:

- Clustering
- Index
- Materialized views
- Data partition

For our application, we used the MySQL database. Here are some resources that can help you scale the database and apply the previous concepts:

- http://dev.mysql.com/doc/refman/5.7/en/faqs-mysql-cluster.html
- http://www.fromdual.com/mysql-materialized-views
- http://dev.mysql.com/doc/refman/5.7/en/optimization-indexes.html
- http://dev.mysql.com/doc/refman/5.7/en/partitioning.html
- https://dev.mysql.com/doc/refman/5.7/en/partitioning-overview.html

Cloud computing/containers

Scaling up application in traditional data centers is always hard because we need to have the hardware in place. This gets done by the practice of capacity planning. Capacity planning is great to make sure we don't spend money beyond our budget. However, it is very hard to get it done right. Software is hard to predict, and that's a great advantage of the cloud. Cloud is just another level of abstraction. Hardware and networks become logical, and they are encapsulated behind APIs. This makes it easier to scale our application as we can rely on cloud elasticity and scale on demand when we need to. However, the architecture needs to be ready for this moment and use the tools and techniques described in this chapter. Currently, there are several public clouds; the best options are as follows:

- AWS--Amazon Cloud (https://aws.amazon.com/)
- Google Cloud (https://cloud.google.com/)
- Microsoft Azure Cloud (https://azure.microsoft.com/en-us/)

Today, Cloud is not the only big elephant in the room. We also have the Linux containers, such as Docker (https://www.docker.com/) and LXC (https://linuxcontainers.org/). Containers provide another level of abstraction, and they can run on the cloud or on premises. This makes your application more portable and also more cost effective. Containers also scale. The main advantage around containers is speed and flexibility. It's way faster to boot up a container in comparison with a virtualized image in any public cloud. They are also portable and can run everywhere.

Auto Scaling

Currently, this is one of the greatest resources of cloud computing. Basically, you can define a base image, which is a state of an operational system such as Linux, and the cloud will create and destroy instances for you on demand. These instances can be created by the increase in computational resources, such as memory, CPU, network, or even based on custom rules. This is the key concern in order to have elasticity. If you want to learn more about Auto Scaling, check out https://aws.amazon.com/autoscaling/.

A note about automation

In order to use all these techniques and technologies at scale, we need to have full automation (https://en.wikipedia.org/wiki/List_of_build_automation_software) because it is impossible to handle all this with manual work. When we are using the cloud or containers, there is no other way around; everything needs to be automated. There are several tools that help us achieve this goal, such as Ansible (https://www.ansible.com/).

Don't forget about telemetry

When you have all infrastructures in place, you will also need to have monitoring, alerting, and proper dashboards. There are plenty

of great tools for containers and public clouds, such as Sensu (https://sensuapp.org/) and Prometheus (https://prometheus.io/).

Reactive Drivers and discoverability

Reactive Drivers: We talked a lot and did a lot of reactive code using Play framework and RxScala. However, to have full benefits of ReactiveX programming, you need to make sure everything is a non-blocking IO and reactive. In other words, we need to have all of our drivers reactive. Slick is great because it gives us reactivity with the MySQL database. We will need to apply the same principles everywhere we have a driver or connection point. There are lots of libraries becoming reactive these days. For instance, if you want to cache using Redis, you can use Lettuce (https://github.com/mp911de/lettuce), which is reactive.

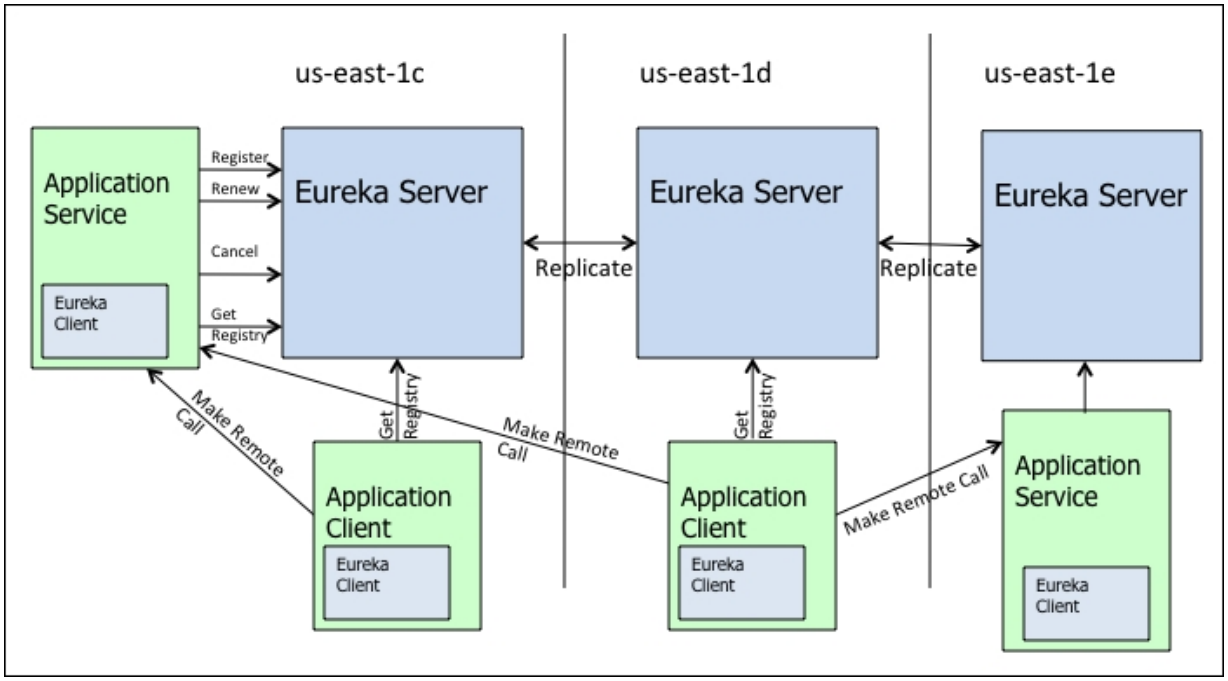
When we work with microservices, we tend to have hundreds of microservice instances. These microservices will run on containers and/or cloud computing units. You can't point to specific IPs because the code will not be managed and will not survive in a cloud/container environment. Cloud/container infrastructure is ephemeral, and you don't know when an instance will be terminated. That's why you need to be able to switch to another availability zone or region at any moment.

There are tools that can help us apply these changes in our code. These tools are Netflix/Eureka (https://github.com/Netflix/eureka) and Consul (https://www.consul.io/), or even Apache Zookeeper (https://zookeeper.apache.org/). Eureka has one advantage--it is easier to use and has tools around the JVM ecosystem, which was battle tested by Netflix.

Eureka is a central registry where microservices register their IP and metadata. Eureka has a REST API. Microservices can use the Eureka API to query and search existing applications. Eureka can run in a multi-vpc/multi-region environment. There are other JVM components, such as ribbon (https://github.com/Netflix/ribbon) and

karyon (https://github.com/Netflix/karyon), which can automatically register and retrieve eureka information and metadata.

Based on the Eureka information, you can perform microservice load balancing and fail over to other availability zones and regions automatically. Why use Eureka if I can use DNS? DNS for Mid-Tier load balancing is not the right choice as DNS is not flexible and the timeout is quite big. If you want know more about discoverability, check out http://microservices.io/patterns/service-registry.html.



Eureka overview - Eureka architecture overview on the AWS cloud

As you can see in the preceding diagram, you will deploy the Eureka server at least in three **Availability Zones** (**AZs**) in order to have availability. Then, Eureka data will be replicated to each server. Our applications or microservices will register in Eureka, and other applications/microservices can retrieve this metadata, such as IP

address, to them to the REST calls. If you want learn more, you can check out https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance.

Mid-Tier load balancer, timeouts, Back pressure, and caching

Eureka, Zookeeper, or Consul are only one part of the equation. We still need some software on the client side that can use the Eureka information in order to do Mid-Tier load balancing, fail over, and caching. The Netflix stack has a component for that, which is called ribbon (https://github.com/Netflix/ribbon). With ribbon, you can automatically resolve the microservice IPs from Eureka, do retries, and failover to other AZs and regions. Ribbon has a cache concept; however, it is on preloaded cache.

Ribbon ideas are simple. The great thing about ribbon is that everything is reactive, and you can use RxJava and RxScala in order to work with the stack. If you don't want to use ribbon, you can still create a simple integration layer with Scala and perform the same concerns, such as load balancing, failover, and caching.

What about Back pressure? Back pressure can be done with RxJava and Rxscala, and you will be able to do it on the client side as well. You can learn more about Back pressure in Rx at https://github.com/ReactiveX/RxJava/wiki/Backpressure.

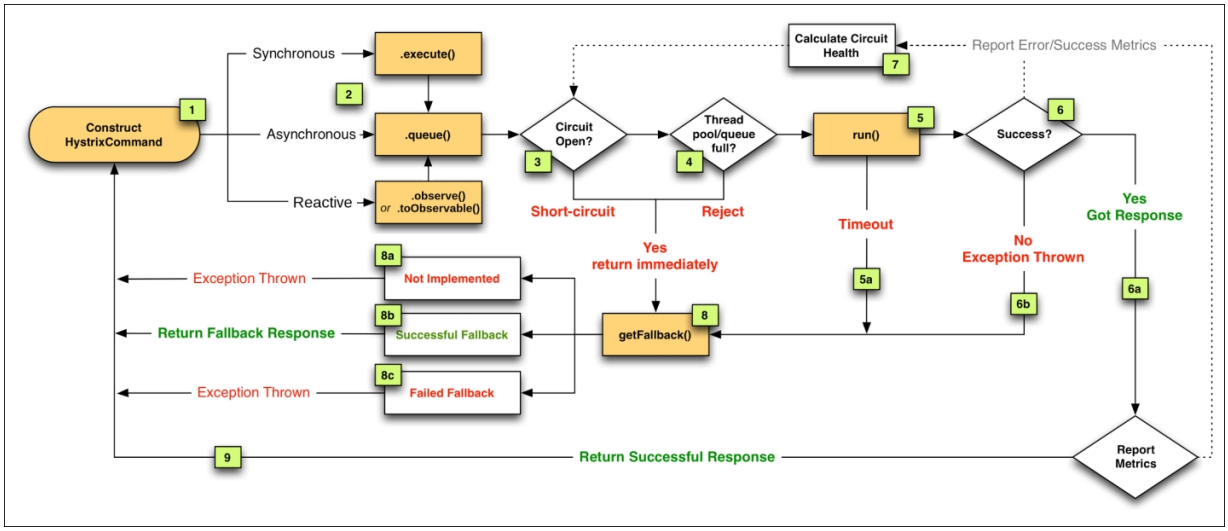
So, if I have client-side load balancing, failover, caching, and Back pressure, am I good to go? Yes, you are; however, we can always do better. Working with microservices is not easy as everything is a remote call, and remote calls can fail, hang, or timeout. These cons are hard and dangerous if not managed well. There is another solution that can help us a lot with this concept; it is called Hystrix (https://github.com/Netflix/Hystrix).

Hystrix is a library for the JVM designed for latency and fault tolerance protection. At a glance, Hystrix is a wrapper around any

remote code that can take time or go wrong.

Hystrix has thread isolation and provides a dedicated thread pool for each resource. This is great because it prevents you from running out of resources. It has an execution pattern called circuit breaker. Circuit breaker will prevent requests from tearing down the whole system. Additionally, it has a dashboard where we can visualize the circuits, so, at runtime, we can see what's going on. This capability is great not only for sense of telemetry, but also because it is easy to troubleshoot and visualize where the problem is.

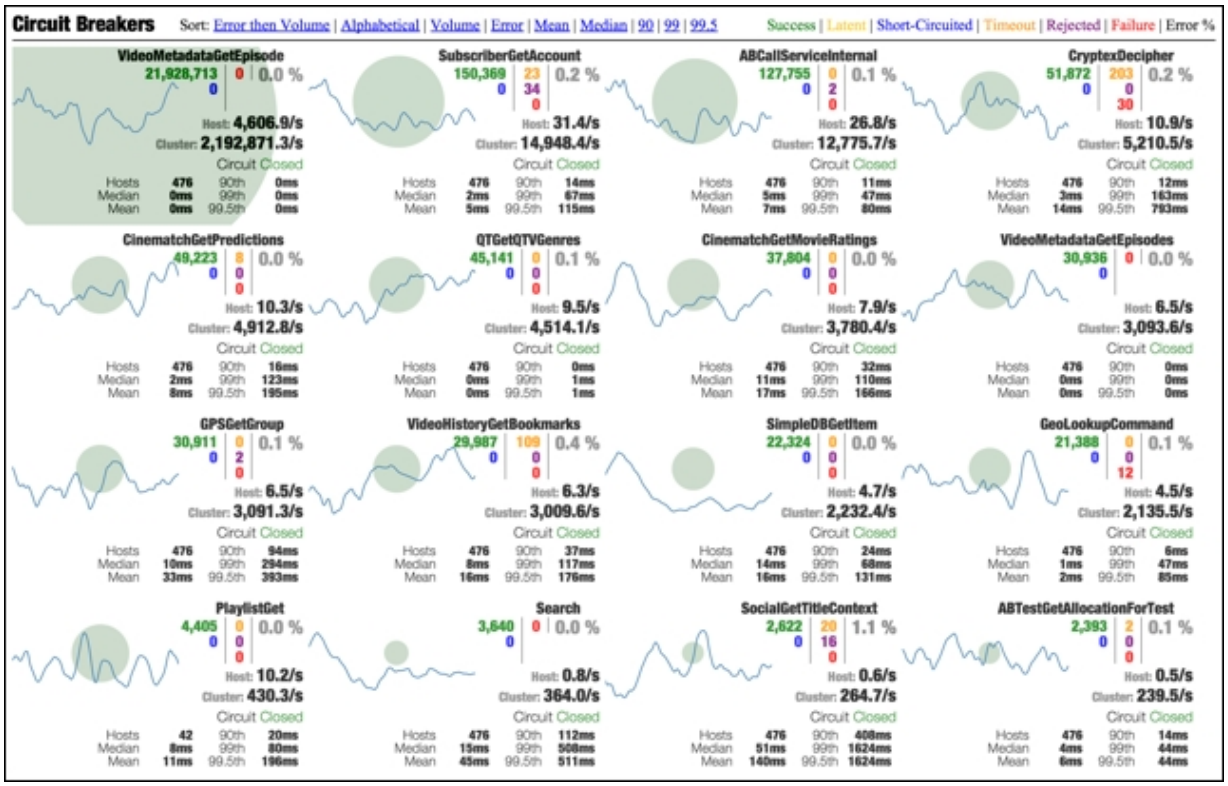
It can be further explained with the help of the following flowchart:



The code you want to protect will be around a Hystrix command. This command can be manipulated in sync or async programming models. The first thing that Hystrix will check is if the circuit is closed, which is good, and how it should be. Then, it checks if there are threads available for that command, and if there are available threads, then the command will be executed. If this fails, it tries to get a fallback code, which is a second option that you can provide in case of failure. This fallback should be static; however, you can be

loading data in the background and then return on the fallback. Another option is fallback to other AZ or Region.

Following is a snapshot of how a Hystrix dashboard circuit breaker view would work:



In the preceding image, we can see the Hystrix dashboard sample, where we can visualize critical information, such as success and error rate and if the circuit is open or closed. If you want learn more about the Hystrix dashboard, check out https://github.com/Netflix/Hystrix/wiki/Dashboard.

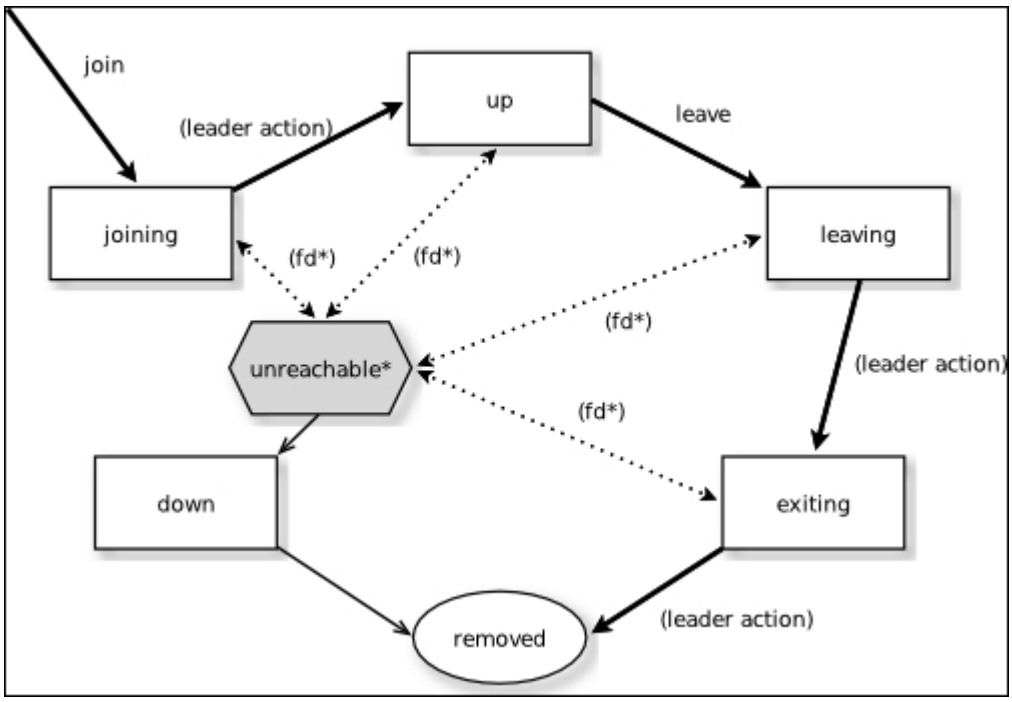
Scaling up microservices with an Akka cluster

Our application also uses Akka. In order to scale Akka, we will need to use an Akka cluster. The Akka cluster allows us to clusterize several Actor systems in several machines. It has special Actor routers that are cluster aware, and we can use these Actors to route requests to the whole cluster; more details can be found at http://doc.akka.io/docs/akka/2.4.9/java/cluster-usage.html#Cluster_Aware_Routers.

The Akka cluster provides membership protocol and life cycle. Basically, we can be notified by the cluster when a new member joins or when a member leaves the cluster. Given this capability, it is possible for us to code a scalable solution around these semantics. As we know when a member joins, we can deploy more nodes, and we can also drop nodes on demand.

A simple sample would be to create an Actor called frontend, and when we see this Actor, we could deploy three backend Actors across the cluster. If the frontend Actor leaves, we could undeploy the other Actors. All this logic can be archived using the membership protocol and clusters events that Akka generates for us. A frontend Actor is a not a UI or web application, it is just an Actor that receives work. So, let's say we want to generate analytics around our products catalog. We could have a frontend Actor who receives that request and delegates the work to backend Actors, which will be deployed across the cluster and deliver the analytical work.

The following image is the process view of an Akka cluster membership protocol:



As you can see in the preceding image, there is a set of states. First of all, the node is joining the cluster; then the node can be up. Once the node is up, it can leave the cluster. There are intermediate states, such as leaving and existing.

The Akka cluster provides many options to scale our Actor system. Another interesting option is to use the pattern of distributed Pub/Sub. If you are familiar with JMS Topics, it is almost the same idea. For those who are not familiar, you can check out http://doc.akka.io/docs/akka/2.4.9/java/distributed-pub-sub.html.

Note

If you want learn more about the Akka cluster, you can check out http://doc.akka.io/docs/akka/2.4.9/common/cluster.html.

Scaling up the infrastructure with Docker and AWS cloud

Scaling up with the AWS cloud is easy, as at any moment, with a simple click on the AWS console, you can change the hardware and use more memory, CPU, or better network. Scale-out is not hard; however, we need to have good automation in place. The key principle to scale is to have the Auto Scaling groups in place with good policies. You can learn more about it at http://docs.aws.amazon.com/autoscaling/latest/userguide/policy_creating.html.

There are other interesting services and components that can help you scale your application. However, you will need to keep in mind that this can lead to coupling. The IT industry is moving toward the container direction because it is faster, and it's easy to deploy in other public clouds.

We can scale out with Docker as well, because there are cluster managers that can help us scale our containers. Currently, there are several solutions. In the sense of capabilities and maturity, the following are the better solutions:

- Docker Swarm (https://docs.docker.com/swarm/overview/)
- Kubernetes (http://kubernetes.io/)
- Apache Mesos (http://mesos.apache.org/)

Docker Swarm: This is a cluster for Docker. Docker Swarm is very flexible and integrates well with other Docker ecosystem tools, such as Docker machine, Docker compose, and Consul. It can handle hundreds of nodes, and you can learn more about them at https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/.

Kubernetes: This was created by Google, and it is a full solution for development automation, operation, and scaling Docker containers. The Kubernetes cluster has two roles, a master node that coordinates the cluster, schedules applications, and keeps applications on a desired state; and there are nodes, that are workers that run applications. It can handle hundreds of containers and scale very well. To learn more about it, check out http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html.

Apache Mesos: This was created by Twitter. It is very interesting, as you can run a bare metal on a premises datacenter or on a public cloud. Mesos allows you to use Docker containers as well. If you want to learn more about mesos, check out the following paper:

http://mesos.berkeley.edu/mesos_tech_report.pdf

Summary

In this chapter, you learned how to deploy your Play framework application as a standalone distribution. Additionally, you learned several architectural principles, techniques, and tools, to help you scale out your application to thousands of users.

With this, we also reach the end of this book. I hope you enjoyed this journey. We built a nice application using Scala, Play Framework, Slick, REST, Akka, Jasper, and RxScala. Thank you for your time. I wish you the best in your coding career with the Scala language.