

Scala Programming Projects

Build real world projects using popular Scala frameworks
like Play, Akka, and Spark



Packt

www.packt.com

Mikaël Valot and Nicolas Jorand

Scala Programming Projects

Build real world projects using popular Scala frameworks like Play, Akka, and Spark

Mikaël Valot
Nicolas Jorand

Packt

BIRMINGHAM - MUMBAI

Scala Programming Projects

Copyright © 2018 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(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Commissioning Editor: Aaron Lazar

Acquisition Editor: Alok Dhuri

Content Development Editor: Zeeyan Pinheiro

Technical Editor: Gaurav Gala

Copy Editor: Safis Editing

Project Coordinator: Vaidehi Sawant

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Graphics: Alishon Mendonsa

Production Coordinator: Nilesh Mohite

First published: September 2018

Production reference: 1280918

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78839-764-3

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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.

Contributors

About the authors

Mikaël Valot is Principal Software Engineer at IHS Markit in London, UK. He is the lead developer of a strategic market risk solution for banking regulation.

He has over 15 years of experience in the financial industry of the UK, Switzerland, and France. He has a Diplôme d'Ingénieur in Computing (equivalent to an M.Sc.) from Telecom Nancy, France.

After years of working with Java, he started developing professionally with Scala in 2010, and never looked back. He was a speaker at Scala Exchange 2015.

When he is not coding in Scala, Mikaël likes to dabble with Haskell, the Robotic Operating System, and deep learning.

He strongly believes that functional programming with strong typing skills is the best way to write safe and scalable programs.

I dedicate this book to my parents Yvette & Francis, who helped me pursue the education that made all of this possible.

I express my gratitude to my wife, Anne, my children, Charline & Lucile, for their continued support, and to Nicolas, for the great contribution and fun, he brought to this project.

*I thank all the people who helped write this book:
Christopher Chane-Yook, Qiong Lee, Gaurav Gala, Titos Matsakos, Zeeyan Pinheiro, Jean Zottner*

Nicolas Jorand is a senior developer. He worked for the finance industry for about 15 years before switching to the energy industry. He is a freelancer enjoying a partial time at Romande Energy, a Swiss utility company providing exclusively green electricity. Nicolas is a full-stack developer, playing with microcontrollers, developing standard web user and 3D interfaces on Unity, developing software to animate a humanoid robot (Nao) and finally, working with Scala on integration and backend software. All these projects are done with the same leitmotif; "In the dev process, get the issues as early as possible."

*I would like to thank my wife, Luisa, and my children Natsumi and Yuuta for the unconditional support,
Jacques Couvreur, François Leytens, Gaurav Gala and Zeeyan Pinheiro who helped in writing this book,*

last but not least, Mikaël for his kindness and the motivation he brought in this adventure.

About the reviewer

Vlad Patryshev is currently working as a Lead Data Engineer at Salesforce; he is also a professor of Logic at Santa Clara University.

He is an organizer of two meetups, Bay Area Categories and Types, and Scala Bay, the latter dedicated to Scala language.

Vlad's education is in math, but he spent most of his life working in software. 7 years at Borland, 3.5 years at Google.

I'd also like to thank the author for the opportunity to read and review this wonderful book, which, I hope, will serve the wide range of beginning and mid-level Scala developers around the world to get fast into our beautiful world of Scala.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Title Page
Copyright and Credits
Scala Programming Projects
Packt Upsell
Why subscribe?
Packt.com
Contributors
About the authors
About the reviewer
Packt is searching for authors like you
Preface
Who this book is for
What this book covers
To get the most out of this book
Download the example code files
Download the color images
Conventions used
Get in touch
Reviews
1. Writing Your First Program
Setting up your environment
Installing the Java SDK
Installing IntelliJ IDEA
Using the basic features
SBT synchronization
Build
Running the Scala Console
Using the Scala Console and Worksheet
Using the Scala Console
Declaring variables
Types
Declaring and calling functions
Side effects
If...else expression
Class
Using the worksheet
Class inheritance
Subclass assignment
Overriding methods

- Abstract class
- Trait
- Scala class hierarchy
- Case class
- Companion object

Creating my first project

- Creating the project
- Creating the Main object
- Writing the first unit test
- Implementing another feature
- Implementing the Main method

Summary

2. Developing a Retirement Calculator

- Project overview
- Calculating the future capital
 - Writing a unit test for the accumulation phase
 - Implementing futureCapital
 - Refactoring the production code
 - Writing a test for the decumulation phase
 - Simulating a retirement plan
 - Writing a failing unit test
 - Working with tuples
 - Implementing simulatePlan
- Calculating when you can retire
 - Writing a failing test for nbOfMonthsSaving
 - Writing the function body
 - Understanding tail-recursion
 - Ensuring termination
- Using market rates
 - Defining an algebraic data type
 - Filtering returns for a specific period
 - Pattern matching
 - Refactoring simulatePlan
 - Loading market data
 - Writing the unit test using the column selection mode
 - Loading the file with Source
 - Loading inflation data
 - Computing real returns
- Packaging the application
 - Creating the App object
 - Packaging the application

Summary

3. Handling Errors

Setup

Using exceptions

 Throwing exceptions

 Catching exceptions

 Using the finally block

Ensuring referential transparency

 Defining pure functions

 Best practices

 Showing how exceptions break referential transparency

Using Option

 Manipulating instances of Option

 Composing transformations with for... yield

 Refactoring the retirement calculator to use Option

Using Either

 Manipulating Either

 Refactoring the retirement calculator to use Either

 Refactoring nbOfMonthsSavings

 Refactoring monthlyRate

 Refactoring simulatePlan

 Refactoring SimulatePlanApp

Using ValidatedNel

 Adding the cats dependency

 Introducing NonEmptyList

 Introducing Validated

 Refactoring the retirement calculator to use ValidatedNel

 Adding unit tests

 Implementing parsing functions

 Implementing SimulatePlanApp.strSimulatePlan

 Refactoring SimulatePlanApp.strMain

Summary

Questions

Further reading

4. Advanced Features

Project setup

Strictness and laziness

 Strict val

 lazy val

 By-name parameters

 Lazy data structures

Covariance and contravariance

- InvariantDecoder
- CovariantDecoder
- Contravariant encoder
- Covariance in collections
- Currying and partially applied functions
 - Function value
 - Currying
 - Partially applied functions
- Implicits
 - Implicit parameters
 - Implicit parameter usage
 - Passing a timeout
 - Passing an application context
 - Examples in the SDK
 - breakOut
 - executionContext
 - Implicit conversion
 - Implicit class
 - How are implicits resolved?
- Summary

5. Type Classes

- Understanding type classes
 - Type class recipe
 - Common type classes
 - scala.math.Ordering
 - org.scalactic.Equality
 - cats.Semigroup
 - Laws
 - Usage examples
 - cats.Monoid
 - Laws
 - Usage examples
 - Higher-kinded types
 - Arity
 - Higher-order function
 - Higher-kinded types
 - cats.Functor
 - Laws
 - Usage examples
 - cats.Apply
 - Laws
 - Usage examples
 - cats.Applicative
 - Laws

Usage examples
cats.Monad

Laws

Usage examples

Summary

6. Online Shopping - Persistence

Creating the project

Persistence

Setting up Slick

Setting up the database

Database test

Product test

Cart test

Future

Getting a concrete value

Waiting on a Future

Callback

Composing Futures

Using for-comprehension

Execution context

Rounding up Futures

Database creation

Data Access Object creation

Running the test

Deploying the application

Setting up an account

Deploying your application

Heroku configuration

Summary

7. Online Shopping - REST API

The REST API

Writing the tests

Defining the routes

Running the test

Checking the API

Implementing the API with persistence

Completing the product tests

Implementing the product API

Product list

Encoding JSON with Circe

Action definition

Adding a product

Completing the cart test

Logging in

- Unit test
- Implementation
- Passing the cookie
- Listing products in cart
- Swagger
 - Installing Swagger
 - Declaring endpoints
 - Running the application
- Login
- List of products
- Cart endpoints

 Deploying on Heroku

 Summary

8. Online Shopping - User Interface

- Learning objectives
- Setting up
- Defining the layout
- Creating the layout
- Building the page
- Main layout
- Product list panel
- Cart panel
- Introducing the UI manager
 - Adding jQuery to our project
 - Calling our API
 - Setting the main method
 - Initializing the user interface
- Implementing UI actions
 - Adding a product to the cart
 - Removing a product from the cart
 - Updating the quantity
- Deploying the user interface
- Debugging the interface

 Summary

9. Interactive Browser

- Actors
- Setting up
- Implementing the server side
 - Creating the web socket route
 - Implementing BrowserManager
 - Handling WebSocket

A diagram of Actors
Implementing the client side

Adding the web socket

Notifying the user

Extending jQuery

Summary

10. Fetching and Persisting Bitcoin Market Data

Setting up the project

Understanding Apache Spark

RDD, DataFrame, and Dataset

Spark SQL

Dataframe

Dataset

Exploring the Spark API with the Scala console

Transforming rows using map

Transforming rows using select

Execution model

Implementing the transaction batch producer

Calling the Bitstamp REST API

Parsing the JSON response

Unit testing jsonToHttpTransaction

Implementing jsonToHttpTransaction

Unit testing httpToDomainTransactions

Implementing httpToDomainTransactions

Saving transactions

Introducing the Parquet format

Writing transactions in Parquet

Using the IO Monad

Putting it all together

Testing processOneBatch

Implementing processOneBatch

Implementing processRepeatedly

Implementing BatchProducerApp

Running the application with spark-submit

Installing Apache Spark

Packaging the assembly JAR

Running spark-submit

Summary

11. Batch and Streaming Analytics

Introduction to Zeppelin

Installing Zeppelin

Starting Zeppelin

Testing Zeppelin

Structure of a notebook
Writing a paragraph
Drawing charts
Analyzing transactions with Zeppelin
 Drawing our first chart
 Drawing more charts
Introducing Apache Kafka
 Topics, partitions, and offsets
 Producing data into Kafka
 Consuming data from Kafka
 Consumer group
 Offset management
 Connecting to Kafka
Streaming transactions to Kafka
 Subscribing with Pusher
 Deserializing live transactions
 Converting to transaction and serializing
 Putting it all together
 Running StreamingProducerApp
Introducing Spark Streaming
Analyzing streaming transactions with Zeppelin
 Reading transactions from Kafka
 Writing to an in-memory sink
 Drawing a scatter chart
 Aggregating streaming transactions
Summary
Other Books You May Enjoy
Leave a review - let other readers know what you think

Preface

Scala is a type-safe JVM language that incorporates **object-oriented programming (OOP)** and **functional programming (FP)** aspects. This book gets you started with the essentials of software development by guiding you through the different aspects of Scala programming, helping you bridge the gap between learning and implementing. You will learn about the unique features of Scala through diverse applications and encounter simple yet powerful approaches for software development. You will see how to use the basic tools, set up the environment, and write Scala programs.

Scala Programming Projects will help you build a number of applications, beginning with a simple project, such as a financial independence calculator, and advancing to other projects, such as a shopping application or a Bitcoin transaction analyzer. You will be able to use various Scala features, such as its OOP and FP capabilities, and learn ways to write concise, reactive, and concurrent applications in a type-safe manner. You will also learn how to use top-notch libraries, such as Akka and Play, and integrate Scala applications with Kafka, Spark, and Zeppelin; plus, you'll explore deploying applications on cloud platforms.

By the end of the book, you will be empowered by knowing the ins and outs of Scala, being able to apply Scala to solve a variety of real-world problems.

Who this book is for

If you are an amateur programmer who wishes to learn how to use Scala, this book is for you. Knowledge of Java would be beneficial for, but is not necessary to, understanding the concepts covered in this book.

What this book covers

[Chapter 1](#), *Writing Your First Program*, explains how to set up your environment to start programming in Scala, covering the basic tools that are required and what the simplest Scala application might look like.

[Chapter 2](#), *Developing a Retirement Calculator*, puts into practice the features of the Scala language seen in the first chapter. We will also introduce other elements of the Scala language and SDK to develop the model and logic for a retirement calculator. This calculator will help people work out how long they'll need to save and how much they'll need to save to have a comfortable retirement.

[Chapter 3](#), *Handling Errors*, has you continuing to work on the retirement calculator from the previous chapter, handling the errors that riddle it. For instance, the calculator works correctly as long as the right arguments are passed to it but fails badly with a horrible stack trace if any of the parameters are wrong.

[Chapter 4](#), *Advanced Features*, explores the more advanced features of Scala. As with any programming language, some advanced constructs might be seldom used in practice or can obfuscate code. We will aim to only explain features that we have encountered in real projects that have been deployed to production.

[Chapter 5](#), *Type Classes*, you will learn what a type class is, what are the most common type classes in the `cats` library, and how to use them in your projects.

[Chapter 6](#), *Online Shopping – Persistence*, explains how to persist data in a relational database. The data will be the contents of a cart for a shopping website.

[Chapter 7](#), *Online Shopping – REST API*, covers how to develop a REST API using Play Framework. **API** is an acronym for **Application Programming Interface**, and the acronym **REST** stands for **Representational State Transfer**. Basically, we will provide an interface for our application, so that other programs can interact with it.

[Chapter 8](#), *Online Shopping – User Interface*, gets you using Scala.js to build the user interface for the online shopping application. In this interface, you will be able to select a product to add to your cart, update the number of products that you wish to buy, and remove them from the cart if needed.

[Chapter 9](#), *Interactive Browser*, introduces the actor model by extending our shopping project. The extension will consist of a notification, provided to anyone connected to the website, about who is adding/removing a product to/from the cart.

[Chapter 10](#), *Fetching and Persisting Bitcoin Market Data*, look at developing a data pipeline to fetch, store, and analyze Bitcoin transaction data. We will use Apache Spark in batch mode to do so.

[Chapter 11](#), *Batch and Streaming Analytics*, focuses on how to use Zeppelin and Spark to query our historical data store. We will use Zeppelin's capability to plot interesting graphs and Apache Kafka with Spark streaming to analyze live transactions.

To get the most out of this book

Basic knowledge of any programming language would be helpful to you during the course of this book. Additional knowledge of Java would also be useful for understanding some concepts covered in this book. There are several levels of involvement to make the most of this book, from the quickest to the most effective:

- You can just read it and look at the code in your IDE.
- While you read it, you can copy and paste the code samples in your IDE and run them.
- Same as before, but this time you re-type all the code samples. Using the auto-completion will make you discover more functions of the API. Typing the code will also make you remember it more. You make use of your visual and kinesthetic memory.
- The **Benjamin Franklin** method. You read a whole chapter or a section in one go, then close the book. After that, try to re-write the code samples from memory. If you are stuck you can reopen the book. This will force your brain to have a complete picture of a project. You will memorize and understand the concepts in much more depth.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

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
- Ziipeg/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/Scala-Programming-Projects>. In case there is an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/9781788397643_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`codeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
| class LazyDemo {  
|   lazy val lazyVal = {  
|     println("Evaluating lazyVal")  
|   }  
| }
```

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

```
| def lazyEvenPlusOne(xs: Vector[Int]): Vector[Int] =  
|   xs.withFilter { x => println(s"filter $x"); x % 2 == 0 }
```

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

```
| $ mkdir css  
| $ cd css
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Writing Your First Program

In 2001, Martin Odersky started to design the Scala language – it took him three years to release the first public version. The name comes from Scalable language. This was chosen because Scala is designed to grow with the requirements of its users – you can use Scala for small scripts or for large enterprise applications.

Scala has been constantly evolving ever since, with a growing popularity. As a general purpose language, it is used in many different industries such as finance, telecoms, retail, and media. It is particularly compelling in distributed scalable systems and big data processing. Many leading open source software projects have been developed in Scala, such as Apache Spark, Apache Kafka, Finagle (by Twitter), and Akka. A large number of companies use Scala in production, such as Morgan Stanley, Barclays, Twitter, LinkedIn, The Guardian, and Sony.

Scala is not an extension of Java but is fully interoperable with it. You can call Java code from Scala, and you can call Scala code from Java. There is also a compiler to JavaScript, which we will explore later on in this book. You can, therefore, run Scala code in your browser.

Scala is a blend of object-oriented and functional programming paradigms, and it is statically typed. As such, it can serve as a bridge for people from an object-oriented or imperative background to move gradually to functional programming.

In this chapter, we will cover the following topics:

- Setting up your environment
- Using the basic features
- Running the Scala Console
- Using the Scala Console and Worksheet
- Creating my first project

Setting up your environment

First things first, we need to set up our work environment. In this section, we will get all the tools and libraries, and then install and configure them on your computer.

Scala programs are compiled to Java bytecode, which is a kind of assembly language that can be executed using a **Java Virtual Machine (JVM)**. You will, therefore, need to have a Java compiler and a JVM installed on your computer. The **Java Development Kit (JDK)** provides both components, alongside other tools.

You could develop in Scala using a simple text editor and compile your programs using the Scala **Simple Build Tool (SBT)**. However, this would not be a pleasant nor productive experience. The majority of professional Scala developers use an **Integrated Development Environment (IDE)**, which provides many helpful features such as syntax highlighting, autocompletion, code navigation, integration with SBT, and many more. The most widely used IDE for Scala is IntelliJ Idea from JetBrains, and this is the one we are going to install and use in this book. The other options are Scala IDE for Eclipse and ENSIME. ENSIME is an open source project that brings IDE-like features to popular text editors such as Emacs, Vim, Atom, Sublime, and VSC.

Installing the Java SDK

We are going to install the Oracle JDK, which includes a JVM and a Java compiler. On many Linux distributions, the open source OpenJDK is preinstalled. OpenJDK is fully compatible with the Oracle JDK, so if you already have it you do not need to install anything else to follow this book.

You might already have a Java SDK installed on your computer. We are going to check if this is the case. If you are using Windows, open a DOS Command Prompt. If you are using macOS or Linux, open a Terminal. After the prompt, type the following:

```
| javac -version
```

If you have a JDK installed, the version of the installed compiler will be printed:

```
| javac 1.8.0_112
```

If the version installed is greater than or equal to 1.8.0_112, you can skip the JDK installation. The version of Scala that we are going to use is compatible with JDK version 1.8 or 1.9.

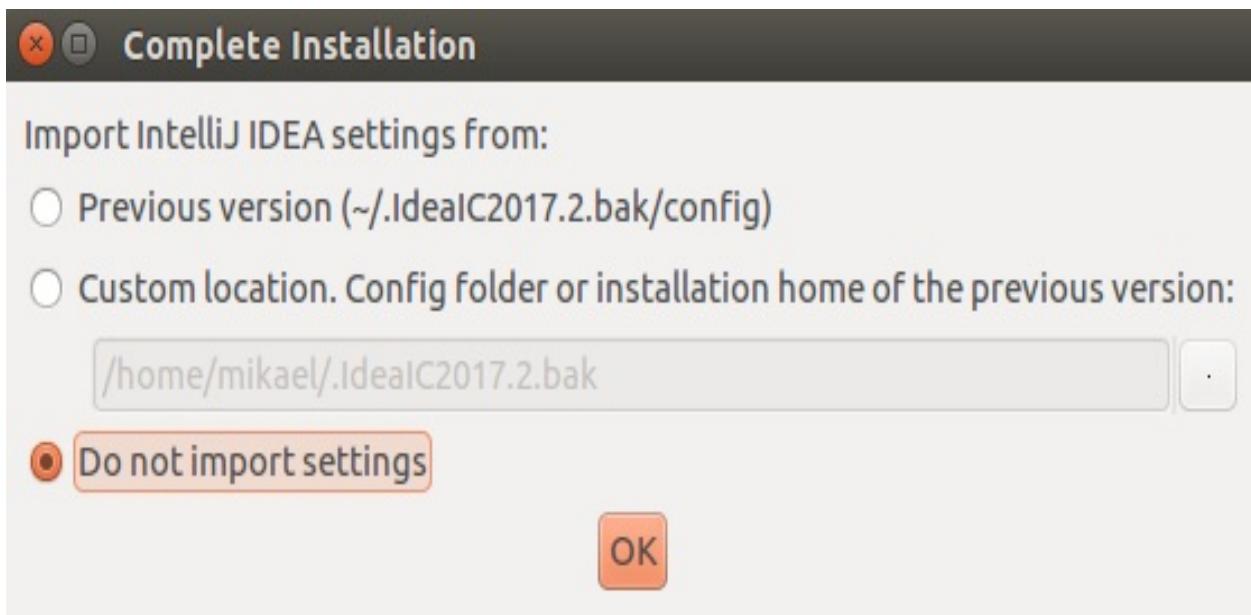
If not, open the following URL, download the SDK for your platform, and follow the installation instructions given: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

Installing IntelliJ IDEA

Go to <https://www.jetbrains.com/idea/download>. Download the community edition for your platform. The ultimate edition offers more features, but we will not use them in this book.

The following are the steps to install IntelliJ IDEA:

1. Run IntelliJ Idea.
2. Select the Do not import settings option:



3. Choose a UI theme. I personally prefer Dracula, since a dark background saves battery on a laptop and is more gentle on the eyes:

Customize IntelliJ IDEA

UI Themes → Desktop Entry → Launcher Script → Default plugins → Featured plugins

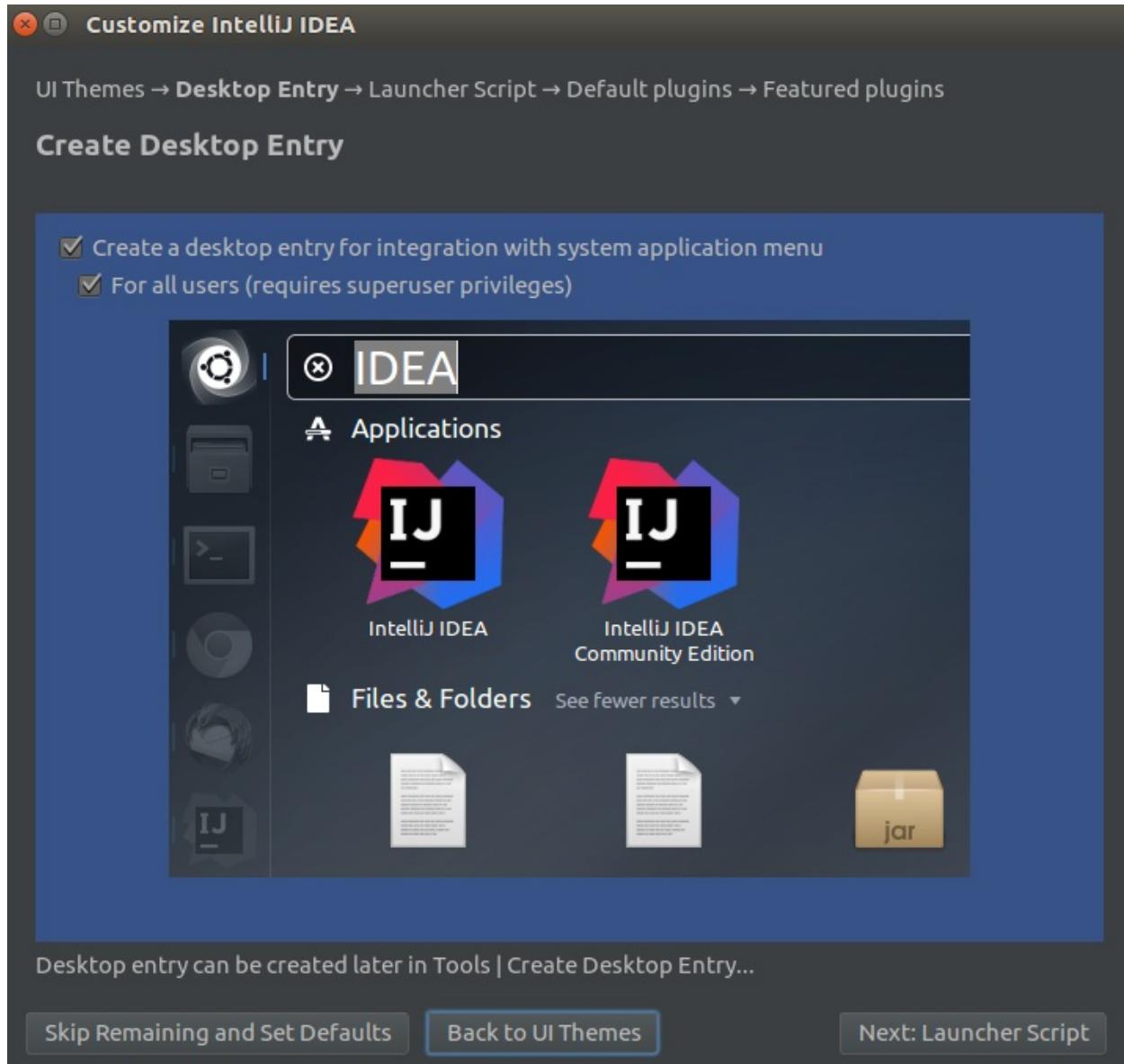
Set UI theme

The screenshot displays the IntelliJ IDEA interface with three theme options: IntelliJ (selected), Darcula, and GTK+. The IntelliJ theme is shown with a blue header bar and a light blue sidebar. The Darcula theme is shown with a dark gray header bar and a dark gray sidebar. The GTK+ theme is shown with a white header bar and a white sidebar. The main workspace shows a Java file named 'HelloWorld.java' with code related to creating a JFrame and setting its title to 'Hello world'. A 'Breakpoints' tool window is open on the right, showing a list of breakpoints, including 'Line 6 in HelloWorld.java' and 'Any exception'.

UI theme can be changed later in Settings | Appearance & Behavior | Appearance

[Skip Remaining and Set Defaults](#) [Next: Desktop Entry](#)

4. Create a desktop entry by checking the options given:



5. In the Create Launcher Script dialog window, check the create a script... checkbox. It will let you open files in IntelliJ from the command line:

The screenshot shows the 'Customize IntelliJ IDEA' dialog with the title 'UI Themes → Desktop Entry → Launcher Script → Default plugins → Featured plugins'. The main section is titled 'Create Launcher Script' with a checked checkbox labeled 'Create a script for opening files and projects from the command line'. Below it is a text input field containing '/usr/local/bin/idea'. A note at the bottom says 'Launcher script can be created later via Tools | Create Command-Line Launcher...'. At the bottom are three buttons: 'Skip Remaining and Set Defaults', 'Back to Desktop Entry', and 'Next: Default plugins' (which is highlighted).

6. Customize the plugins. For each component, click on Customize... or Disable All. We will not need most of the plugins. You can only select the following:

- Build Tools: Disable All.
- Version Controls: Only keep Git and GitHub.
- Test Tools: Disable All.
- Swing: Disable.
- Android: Disable.
- Other Tools: Disable All and keep Bytecode viewer, Terminal, and YAML.
- Plugin Development: Disable.

You can see the aforementioned plugins in the following screenshot:

Customize IntelliJ IDEA

UI Themes → Desktop Entry → Launcher Script → **Default plugins** → Featured plugins

Tune IDEA to your tasks

IDEA has a lot of tools enabled by default. You can set only ones you need or leave them all.

 Build Tools <i>Ant, Maven, Gradle</i> Customize... Enable All	 Version Controls <i>CVS, Git, GitHub, Mercurial, Subversion</i> Customize... Disable All	 Test Tools <i>JUnit, TestNG-J, Coverage</i> Customize... Enable All
 Swing <i>UI Designer</i> Enable	 Android <i>Android</i> Enable	 Other Tools <i>Bytecode Viewer, Eclipse, Java Stream Debugger...</i> Customize... Disable All
 Plugin Development <i>Plugin DevKit</i> Enable		

[Skip Remaining and Set Defaults](#) [Back to Launcher Script](#) [Next: Featured plugins](#)

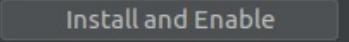
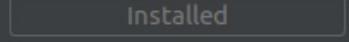
7. Install the featured plugins—some additional plugins are proposed for you to install, such as the Scala plugin and a tool to learn the essential features of IntelliJ.
8. Click on the Install button for Scala and for the IDE Features Trainer, as shown in the following screenshot, and then proceed by clicking on Start using IntelliJ IDEA:

 **Customize IntelliJ IDEA**

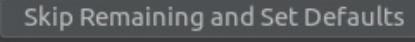
UI Themes → Desktop Entry → Launcher Script → Default plugins → **Featured plugins**

Download featured plugins

We have a few plugins in our repository that most users like to download. Perhaps, you need them too?

Scala Custom Languages Plugin for Scala language support 	Live Edit Tool Web Development Provides live edit HTML/CSS/JavaScript 	IdeaVim Editor Emulates Vim editor   Recommended only if you are familiar with Vim.
NodeJS JavaScript Node.js integration 	IDE Features Trainer Code tools Learn basic shortcuts and essential IDE features with quick interactive exercises 	

New plugins can also be downloaded in [Settings | Plugins](#)



If you are already a Vim aficionado, you can install IdeaVim. Otherwise, I would recommend that you avoid it. I personally use it daily, but it took me some time to get used to it.

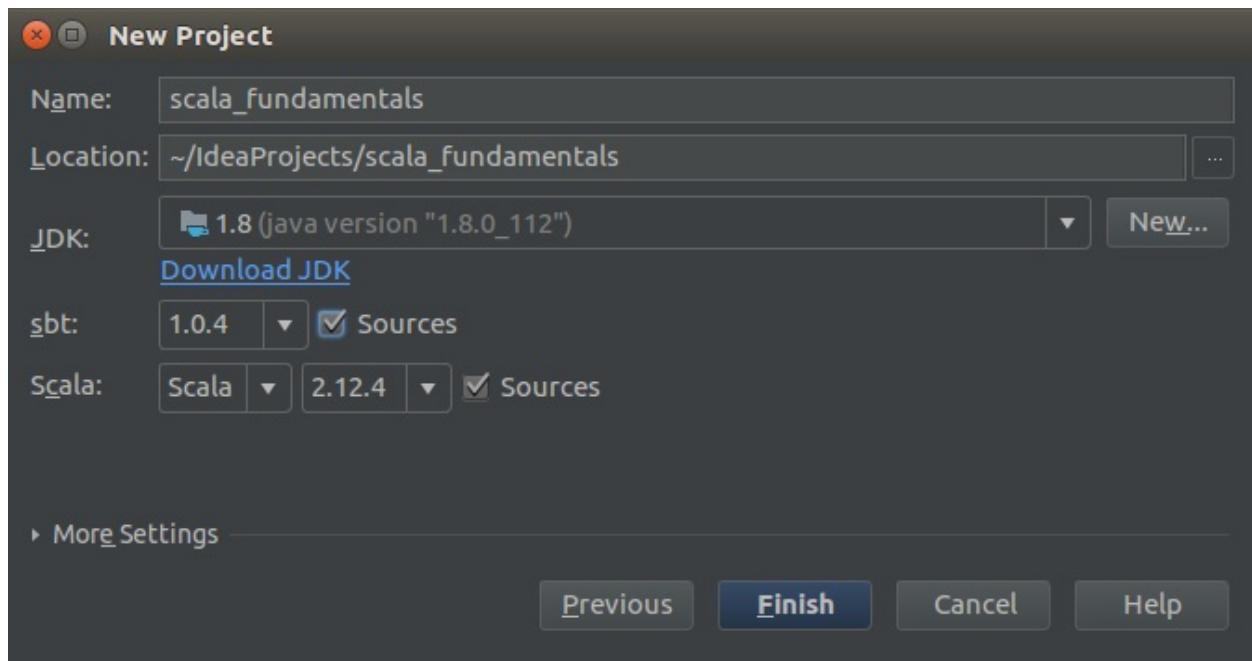
8. Click on Create New Project | Scala | sbt:



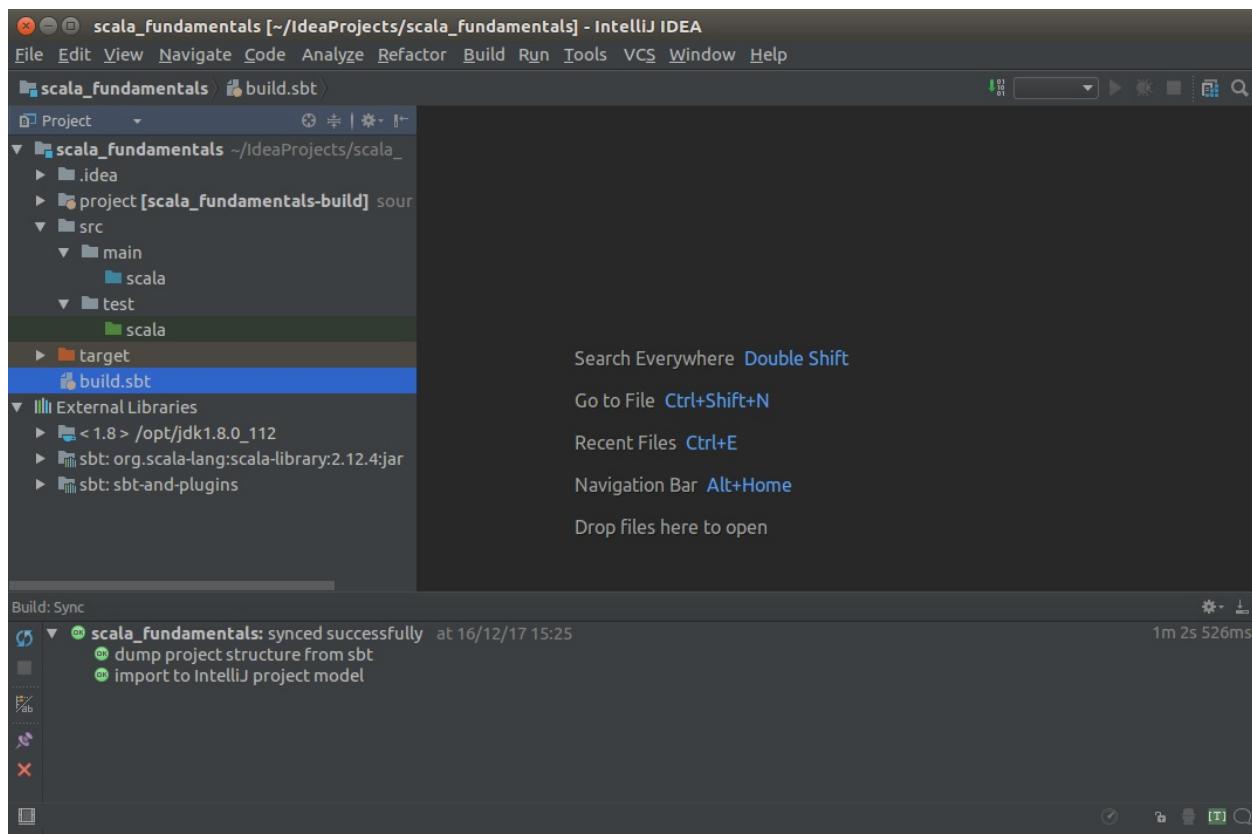
9. Fill in the following details, as shown in the following screenshot:

- Name: `scala_fundamentals`.
- JDK: Click on New and then select the installation directory of the Oracle JDK.
- sbt: Choose the version 1.0.4, check Sources.
- Scala: Choose the latest version 2.12.x, for instance 2.12.4 (IntelliJ lists all the possible versions and will download the one you choose), and check Sources.
- Click on Finish.

It is going to take some time depending on your internet connection's speed:



10. You should see the following project structure:



Using the basic features

In this section, and in the rest of this book, we will highlight some key shortcuts in *italics*. I strongly encourage you to use and remember these shortcuts. They save a tremendous amount of time and will keep you focused on the task at hand. If you cannot remember a shortcut, you can use the mother of all shortcuts, *Ctrl + Shift + A* (Windows/Linux) or *cmd + shift + A* (macOS), and type the name of the action you are looking for.

If you are using IntelliJ for the first time, I find it useful to display all tool buttons. Go to the View menu, and check Toolbar and Tool buttons.

SBT synchronization

Now, let's have a look at our build configuration. **SBT** (short for **Simple Build Tool**) is the *de facto* build tool in the Scala community. Double-click on `build.sbt`:

```
name := 'scala_fundamentals'  
version := "0.1"  
scalaVersion := "2.12.4"
```

This file describes how SBT will compile, test, and deploy our project. For now, it is fairly simple.

One important thing to keep in mind is that IntelliJ manages its own set of files to define a project structure. They are located in the `.idea` directory of your project.

Whenever you change `build.sbt`, IntelliJ has to interpret the changes and translate them.

For instance, If I change the Scala version to `2.12.3` and save (*Ctrl + S* or *cmd + S*), IntelliJ will propose to synchronize the changes or enable autoimport:

```
name := "scala_fundamentals"
version := "0.1"
scalaVersion := "2.12.3"
```

On a small project, it is ok to use autoimport, but on a large one, it can be a bit annoying. The synchronization can take time and it might kick off too often.

When you program in Scala using IntelliJ, you therefore have two ways of compiling your project:

- SBT, in which case you would only use IntelliJ as an advanced text editor
- IntelliJ

You could, in theory, mix and match: start building with SBT and continue with IntelliJ or the other way around. However, I strongly discourage you to do so, as you may get some unexpected compilation errors. When you want to switch to one tool or the other, it is best to clean all compiled files first.

We will further expand on SBT later in this book, but for now, we are only going to use IntelliJ's own build.

Build

The project has been created and ready to be built. The build process does the following:

- Compiles the source files present at the source path and the test path
- Copies any resource files needed in the output path
- Reports any errors/warnings in the Message tool window

There are two ways to build the project:

- If you want to build your project incrementally, go to Build | Build Project (*Ctrl + F9* or *cmd + F9*)
- If you want to delete all files and rebuild everything, go to Build | Rebuild All

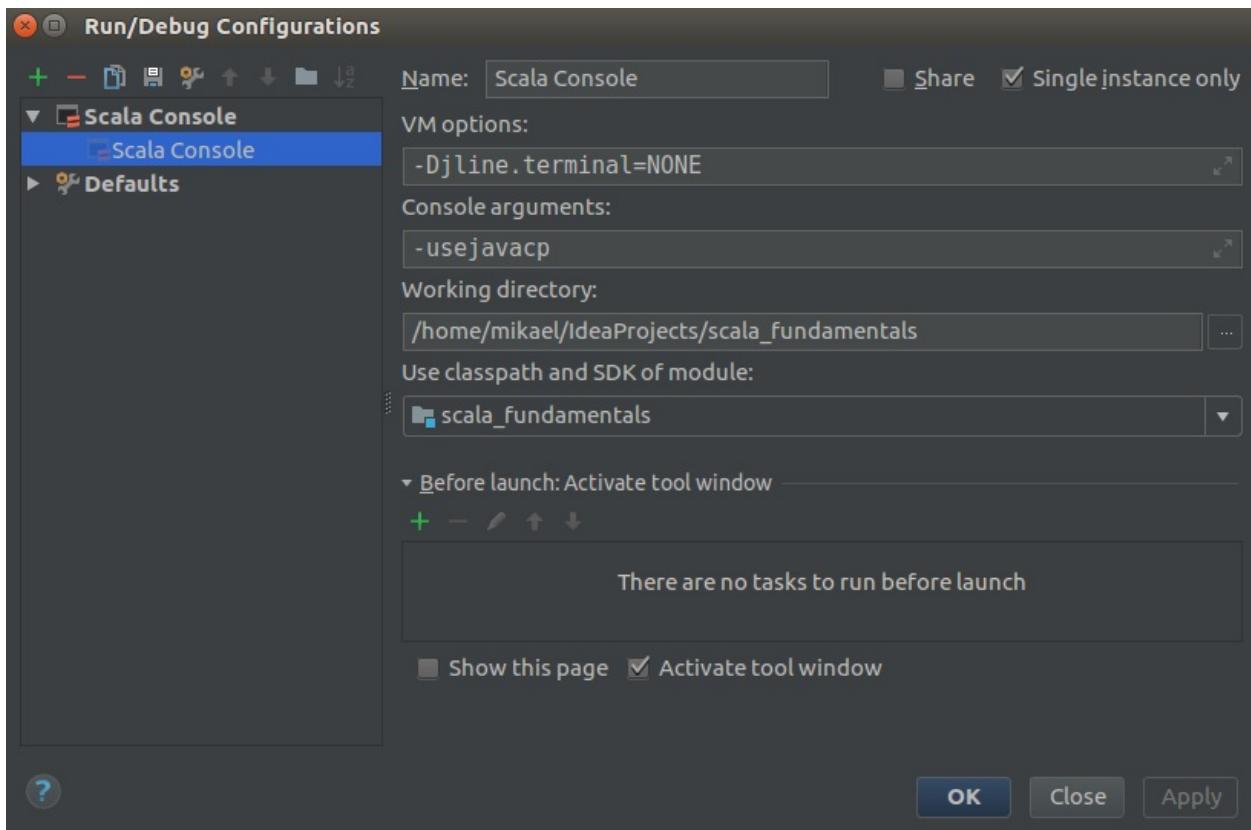
As we do not have a source yet, the build is fast and no errors should appear in the Message tool window.

Running the Scala Console

In IntelliJ, you need to have a run configuration whenever you want to run something: a program, a unit test, an external tool. A run configuration sets up the classpath, arguments, and environment variables that you need to run your executable.

We need to create a run configuration the first time we want to run the Scala console:

1. Go to Run | Edit Configurations. Click on the green + button, and select Scala Console. You should see the following screen:



2. Make the following changes and click OK:

- Name: `scala Console`.
- Check Single instance only box – we rarely need to have two consoles

- running at the same time.
- In, Before launch, click on Build and then click the Remove button. This way, you will always be able to quickly run a console, even if your code does not compile.
- Following that, click on OK.
- On the top toolbar, you should see that IntelliJ created a new Scala Console run configuration:



- Click on the green arrow to run the console. You should see the following at the bottom of the screen, in the Run window. We can now type our first Scala expression after the Scala prompt:

A screenshot of the IntelliJ IDEA Run window titled 'Scala Console'. The window shows the Scala REPL output:

```
/opt/jdk1.8.0_112/bin/java ...
Welcome to Scala 2.12.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112).
Type in expressions for evaluation. Or try :help.

scala>
```

The left sidebar contains various run configurations and settings. The Scala REPL prompt 'scala>' is visible at the bottom of the window.

Using the Scala Console and Worksheet

By now, all the necessary tools and libraries should be installed. Let's start to play with the basics of Scala by experimenting in different environments. The simplest way to try Scala is to use the Scala Console. Subsequently, we will introduce the Scala Worksheet, which allows you to keep all the instructions that are entered in a file.

Using the Scala Console

The Scala console, also called Scala **REPL** (short for **Read-Eval-Print-Loop**), allows you to execute bits of code without having to compile them beforehand. It is a very convenient tool to experiment with the language or when you want to explore the capabilities of a library.

In the console, type `1+1` after the `scala>` prompt and hit *Ctrl + Enter* or *cmd + Enter*:

```
| scala> 1+1
```

The console displays the result of the evaluation, like so:

```
| res0: Int = 2
```

What happened here? The REPL compiled, evaluated the expression `1+1`, and automatically assigned it to a variable named `res0`. This variable is of type `Int`, and its value is `2`.

Declaring variables

In Scala, a variable can be declared using `val` or `var`. A `val` is **immutable**, which means you can never change its value. A `var` is **mutable**. It is not mandatory to declare the type of the variable. If you do not declare it, Scala will **infer** it for you.

Let's define some immutable variables:



In all the following code examples, you only need to type the code that is after the Scala Command Prompt, and hit Ctrl + Enter or cmd + return to evaluate. We show the result of the evaluation underneath the prompt, as it would appear on your screen.

```
scala> val x = 1 + 1
x: Int = 2

scala> val y: Int = 1 + 1
y: Int = 2
```

In both cases, the type of the variable is `Int`. The type of `x` was inferred by the compiler to be `Int`. The type of `y` was explicitly specified with `: Int` after the name of the variable.

We can define a mutable variable and modify it as follows:

```
scala> var x = 1
x: Int = 1

scala> x = 2
x: Int = 2
```

It is a good practice to use `val` in most situations. Whenever I see a `val` declared, I know that its content will never change subsequently. It helps to reason about a program, especially when multiple threads are running. You can share an immutable variable across multiple threads without fearing that one thread might see a different value at some point. Whenever you see a Scala program using `var`, it should make you raise an eyebrow: **the programmer should have a good reason to use a mutable variable, and it should be documented.**

If we attempt to modify a `val`, the compiler will raise an error message:

```
scala> val y = 1
```

```
| y: Int = 1
| scala> y = 2
| <console>:12: error: reassignment to val
|       y = 2
|             ^
```

This is a good thing: the compiler helps us make sure that no piece of code can ever modify a `val`.

Types

We saw in the previous examples that Scala expressions have a **type**. For instance, the `value` `i` is of type `Int`, and the expression `i+1` is also of type `Int`. A type is a classification of data and provides a finite or infinite set of values. An expression of a given type can take any of its provided values.

Here are a few examples of types available in Scala:

- `Int` provides a finite set of values, which are all the integers between -2^{31} and $2^{31}-1$.
- `Boolean` provides a finite set of two values: `true` and `false`.
- `Double` provides a finite set of values: all the 64 bits and IEEE-754 floating point numbers.
- `String` provides an infinite set of values: all the sequence of characters are of an arbitrary length. For instance, "Hello World" or "Scala is great!".

A type determines the operations that can be performed on the data. For instance, you can use the `+` operator with two expressions of type `Int` or `String`, but not with expressions of type `Boolean`:

```
scala> val str = "Hello" + "World"
str: String = HelloWorld

scala> val i = 1 + 1
i: Int = 2

scala> val b = true + false
<console>:11: error: type mismatch;
 found   : Boolean(false)
```

When we attempt to use an operation on a type that does not support it, the Scala compiler complains of a type mismatch error.

An important feature of Scala is that it is a statically typed language. This means that the type of a variable or expression is known at compile time. The compiler will also check that you do not call an operation or function that is not legal for this type. This helps tremendously to reduce the number of bugs that can occur at **runtime** (when running a program).

As we saw earlier, the type of an expression can be specified explicitly with `:` followed by the name of the type, or in many cases, it can be automatically inferred by the compiler.

If you are not used to working with statically typed languages, you might get frustrated to have to fight with the compiler to make it accept your code, but you will gradually get more accustomed to the kind of errors thrown at you and how to resolve them. You will soon find that the compiler is not an enemy that prevents you from running your code; it is acting more like a good friend that shows you what logical errors you have made and gives you some indication on how to resolve them.

People coming from dynamically typed languages such as Python, or people coming from not-as-strongly statically typed language such as Java or C++, are often astonished to see that a Scala program that compiles has a much higher probability of being correct on the first run.



IntelliJ can automatically add the inferred type to your definitions.

For instance, type `val a = 3` in the Scala console, then move the cursor at the beginning of the

a. You should see a light bulb icon. When you click on it, you will see a hint add type annotation to value definition. Click on it, and IntelliJ will add `: Int` after the a.

Your definition will become `val a: Int = 3`.

Declaring and calling functions

A Scala function takes 0 to n **parameters** and returns a value. The type of each parameter must be declared. The type of the returned value is optional, as it is inferred by the Scala compiler when not specified. However, it is a good practice to always specify the return type, as it makes the code more readable:

```
scala> def presentation(name: String, age: Int): String =  
    "Hello, my name is " + name + ". I am " + age + " years old."  
presentation: (name: String, age: Int)String  
  
scala> presentation(name = "Bob", age = 25)  
res1: String = Hello, my name is Bob. I am 25 years old.  
  
scala> presentation(age = 25, name = "Bob")  
res2: String = Hello, my name is Bob. I am 25 years old.
```

We can call a function by passing arguments in the right order, but we can also name the arguments and pass them in any order. It is a good practice to name the arguments when some of them have the same type, or when a function takes many arguments. It avoids passing the wrong argument and improves readability.

Side effects

A function or expression is said to have a side effect when it modifies some state or has some action in the outside world. For instance, printing a string to the console, writing to a file, and modifying a `var`, are all side effects.

In Scala, all expressions have a type. A statement which performs a side effect is of type `Unit`. The only value provided by the type `Unit` is `()`:

```
scala> val x = println("hello")
hello
x: Unit = ()

scala> def printName(name: String): Unit = println(name)
printName: (name: String)Unit

scala> val y = {
    var a = 1
    a = a+1
}
y: Unit = ()

scala> val z = ()
z: Unit = ()
```

A pure function is a function whose result depends only on its arguments, and that does not have any observable side effect. Scala allows you to mix side-effecting code with pure code, but it is a good practice to push side-effecting code to the boundaries of your application. We will talk about this later in more detail in the *Ensuring referential transparency* section in [Chapter 3, Handling Errors](#).



Good practice: When a function with no parameters has side effects, you should declare it and call it with empty brackets `()`. It informs users of your function that it has side effects.

Conversely, a pure function with no parameters should not have empty brackets, and should not be called with empty brackets. IntelliJ helps you in keeping some consistency: it will display a warning if you call a parameterless function with `()`, or if you omit the `()` when you call a function declared with `()`.

Here is an example of a method call with a side effect where we have to use empty brackets, and an example of a pure function:

```
scala> def helloworld(): Unit = println("Hello world")
helloworld: ()Unit
```

```
scala> helloWorld()
Hello world

scala> def helloWorldPure: String = "Hello world"
helloWorldPure: String

scala> val x = helloWorldPure
x: String = Hello world
```

If...else expression

In Scala, `if (condition) ifExpr else if ifExpr2 else elseExpr` is an expression, and has a type. If all sub-expressions have a type `A`, the type of the `if ... else` expression will be `A` as well:

```
scala> def agePeriod(age: Int): String = {
    |   if (age >= 65)
    |     "elderly"
    |   else if (age >= 40 && age < 65)
    |     "middle aged"
    |   else if (age >= 18 && age < 40)
    |     "young adult"
    |   else
    |     "child"
  }
agePeriod: (age: Int)String
```

If sub-expressions have different types, the compiler will infer a common super-type, or widen the type if it is a numeric type:

```
scala> val ifElseWiden = if (true) 2: Int else 2.0: Double
ifElseWiden: Double = 2.0

scala> val ifElseSupertype = if (true) 2 else "2"
ifElseSupertype: Any = 2
```

In the first expression present in the preceding code, the first sub-expression is of type `Int` and the second is of type `Double`. The type of `ifElseWiden` is widened to be `Double`.

In the second expression, the type of `ifElseSupertype` is `Any`, which is the common super-type for `Int` and `String`.

An `if` without an `else` is equivalent to `if (condition) ifExpr else ()`. It is better to always specify the `else` expression, otherwise, the type of the `if/else` expression might not be the one we expect:

```
scala> val ifWithoutElse = if (true) 2
ifWithoutElse: AnyVal = 2

scala> val ifWithoutElseExpanded = if (true) 2: Int else (): Unit
ifWithoutElseExpanded: AnyVal = 2

scala> def sideEffectingFunction(): Unit = if (true) println("hello world")
```

```
| sideEffectingFunction: ()Unit
```

In the preceding code, the common super-type between `Int` and `Unit` is `AnyVal`. This can be a bit surprising. In most situations, you would want to avoid that.

Class

We mentioned earlier that all Scala expressions have a type. A `class` is a sort of template that can create objects of a specific type. When we want to obtain a value of a certain type, we can **instantiate** a new **object** using `new` followed by the class name:

```
scala> class Robot
defined class Robot

scala> val nao = new Robot
nao: Robot = Robot@78318ac2
```

The instantiation of an object allocates a portion of **heap** memory in the JVM. In the preceding example, the value `nao` is actually a **reference** to the portion of heap memory that keeps the content of our new `Robot` object. You can observe that when the Scala console printed the variable `nao`, it outputted the name of the class, followed by `@78318ac2`. This hexadecimal number is, in fact, the memory address of where the object is stored in the heap.

The `eq` operator can be handy to check if two references are equal. If they are equal, this means that they point to the same portion of memory:

```
scala> val naoBis = nao
naoBis: Robot = Robot@78318ac2

scala> nao eq naoBis
res0: Boolean = true

scala> val johnny5 = new Robot
johnny5: Robot = Robot@6b64bf61

scala> nao eq johnny5
res1: Boolean = false
```

A class can have zero to many **members**. A member can be either:

- An **attribute**, also called a **field**. It is a variable whose content is unique to each instance of the class.
- A **method**. This is a function that can read and/or write the attributes of the instance. It can have additional parameters.

Here is a class that defines a few members:

```
| scala> class Rectangle(width: Int, height: Int) {  
|   val area: Int = width * height  
|   def scale(factor: Int): Rectangle = new Rectangle(width * factor, height * factor)  
| }  
defined class Rectangle
```

The attributes declared inside the brackets () are a bit special: they are **constructor arguments**, which means that their value must be specified when we instantiate a new object of the class. The other members must be defined inside the curly brackets {}. In our example, we defined four members:

- Two attributes that are constructor arguments: `width` and `height`.
- One attribute, `area`. Its value is defined when an instance is created by using the other attributes.
- One method, `scale`, which uses the attributes to create a new instance of the class `Rectangle`.

You can call a member on an instance of a class by using the **postfix** notation `myInstance.member`. Let's create a few instances of our class and try to call the members:

```
| scala> val square = new Rectangle(2, 2)  
| square: Rectangle = Rectangle@2af9a5ef  
  
| scala> square.area  
| res0: Int = 4  
  
| scala> val square2 = square.scale(2)  
| square2: Rectangle = Rectangle@8d29719  
  
| scala> square2.area  
| res1: Int = 16  
  
| scala> square.width  
<console>:13: error: value width is not a member of Rectangle  
      square.width
```

We can call the members `area` and `scale`, but not `width`. Why is that?

This is because, by default, constructor arguments are not accessible from the outside world. They are private to the instance and can only be accessed from the other members. If you want to make the constructor arguments accessible, you need to prefix them with `val`:

```
| scala> class Rectangle(val width: Int, val height: Int) {  
|   val area: Int = width * height  
|   def scale(factor: Int): Rectangle = new Rectangle(width * factor, height * factor)  
| }  
defined class Rectangle
```

```
scala> val rect = new Rectangle(3, 2)
rect: Rectangle = Rectangle@3dbb7bb

scala> rect.width
res3: Int = 3

scala> rect.height
res4: Int = 2
```

This time, we can get access to the constructor arguments. Note that you can declare attributes using `var` instead of `val`. This would make your attribute modifiable. However, in functional programming, we avoid mutating variables. A `var` attribute in a class is something that should be used cautiously in specific situations. An experienced Scala programmer would flag it immediately in a code review and its usage should be always justified in a code comment.

If you need to modify an attribute, it is better to return a new instance of the class with the modified attribute, as we did in the preceding `Rectangle.scale` method.

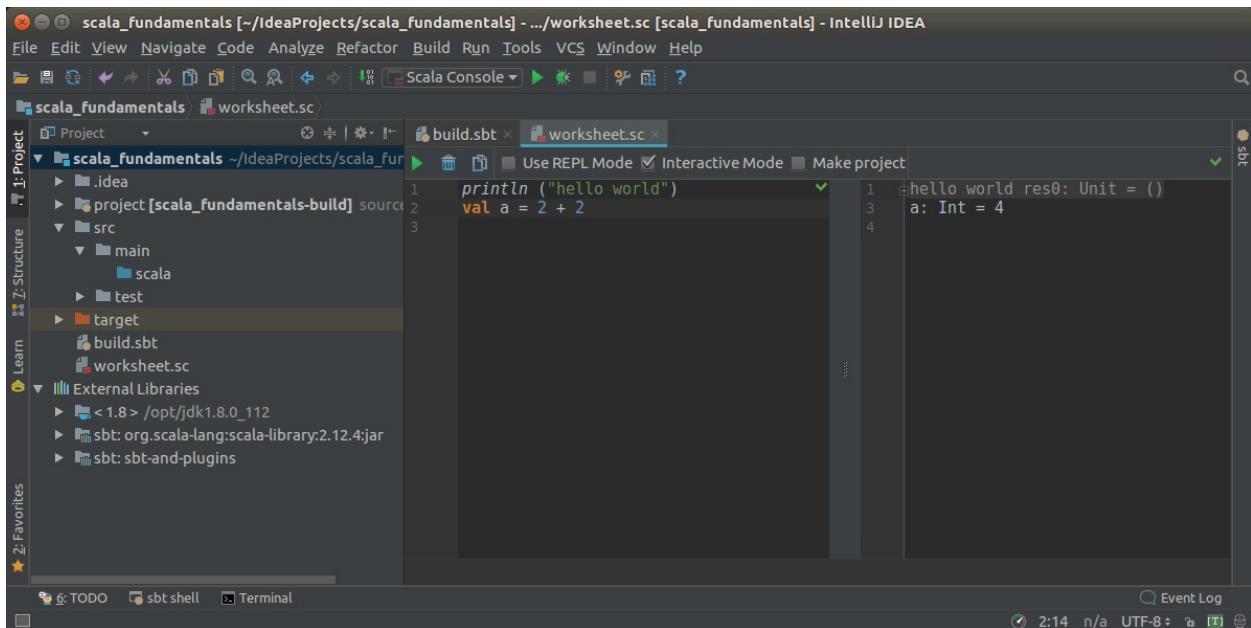


*You might worry that all these new objects will consume too much memory. Fortunately, the JVM has a mechanism known as the **garbage collector**. It automatically frees up the memory used by objects that are not referenced by any variable.*

Using the worksheet

IntelliJ offers another handy tool to experiment with the language: the Scala worksheet.

Go to File | New | Scala Worksheet. Name it `worksheet.sc`. You can then enter some code on the left-hand side of the screen. A red/green indicator in the top right corner shows you if the code you are typing is valid or not. As soon as it compiles, the results appear on the right-hand side:



You will notice that nothing gets evaluated until your whole worksheet compiles.

Class inheritance

Scala classes are extensible. You can extend an existing class to inherit from all its members. If `B` extends `A`, we say that `B` is a **subclass** of `A`, a **derivation** of `B`, or a **specialization** of `B`. `A` is a **superclass** of `B` or a **generalization** of `B`.

Let's see how it works in an example. Type the following code in the worksheet:

```
class Shape(val x: Int, val y: Int) {
    val isAtOrigin: Boolean = x == 0 && y == 0
}

class Rectangle(x: Int, y: Int, val width: Int, val height: Int)
    extends Shape(x, y)

class Square(x: Int, y: Int, width: Int)
    extends Rectangle(x, y, width, width)

class Circle(x: Int, y: Int, val radius: Int)
    extends Shape(x, y)

val rect = new Rectangle(x = 0, y = 3, width = 3, height = 2)
rect.x
rect.y
rect.isAtOrigin
rect.width
rect.height
```

The classes `Rectangle` and `Circle` are subclasses of `Shape`. They inherit from all the members of `Shape`: `x`, `y`, and `isAtOrigin`. This means that when I instantiate a new `Rectangle`, I can call members declared in `Rectangle`, such as `width` and `height`, and I can also call members declared in `Shape`.

When declaring a subclass, you need to pass the constructor arguments of the superclass, as if you were instantiating it. As `Shape` declares two constructor parameters, `x` and `y`, we have to pass them in the declaration `extends Shape(x, y)`. In this declaration, `x` and `y` are themselves the constructor arguments of `Rectangle`. We just passed these arguments up the chain.

Notice that in the subclasses, the constructor parameters `x` and `y` are declared without `val`. If we had declared them with `val`, they would have been promoted as publicly available attributes. The problem is that `Shape` also has `x` and `y` as public attributes. In this situation, the compiler would have raised a compilation error to highlight the conflict.

Subclass assignment

Consider two classes, `A` and `B`, with `B extends A`.

When you declare a variable of type `A`, you can assign it to an instance of `B`, with
`val a: A = new B.`

On the other hand, if you declare a variable of type `B`, you cannot assign it to an instance of `A`.

Here is an example that uses the same `Shape` and `Rectangle` definitions that were described earlier:

```
| val shape: Shape = new Rectangle(x = 0, y = 3, width = 3, height = 2)
| val rectangle: Rectangle = new Shape(x = 0, y = 3)
```

The first line compiles because `Rectangle` **is a** `Shape`.

The second line does not compile, because not all shapes are rectangles.

Overriding methods

When you derive a class, you can override the members of the superclass to provide a different implementation. Here is an example that you can retype in a new worksheet:

```
class Shape(val x: Int, val y: Int) {
  def description: String = s"Shape at (" + x + "," + y + ")"
}

class Rectangle(x: Int, y: Int, val width: Int, val height: Int)
  extends Shape(x, y) {
  override def description: String = {
    super.description + s" - Rectangle " + width + " * " + height
  }
}

val rect = new Rectangle(x = 0, y = 3, width = 3, height = 2)
rect.description
```

When you run the worksheet, it evaluates and prints the following `description` on the right-hand side:

```
| res0: String = Shape at (0,3) - Rectangle 3 * 2
```

We defined a method `description` on the class `Shape` that returns a String. When we call `rect.description`, the method called is the one defined in the class `Rectangle`, because `Rectangle` overrides the method `description` with a different implementation.

The implementation of `description` in the class `Rectangle` refers to `super.description`. `super` is a keyword that lets you use the members of the superclass without taking into account any overriding. In our case, this was necessary so that we could use the `super` reference, otherwise, `description` would have called itself in an infinite loop!

On the other hand, the keyword `this` allows you to call the members of the same class. Change `Rectangle` to add the following methods:

```
class Rectangle(x: Int, y: Int, val width: Int, val height: Int)
  extends Shape(x, y) {
  override def description: String = {
    super.description + s" - Rectangle " + width + " * " + height
  }
}
```

```
def descThis: String = this.description
def descSuper: String = super.description
}
val rect = new Rectangle(x = 0, y = 3, width = 3, height = 2)
rect.description
rect.descThis
rect.descSuper
```

When you evaluate the worksheet, it prints the following strings:

```
res0: String = Shape at (0,3) - Rectangle 3 * 2
res1: String = Shape at (0,3) - Rectangle 3 * 2
res2: String = Shape at (0,3)
```

The call to `this.description` used the definition of `description`, as declared in the class `Rectangle`, whereas the call to `super.description` used the definition of `description`, as declared in the class `Shape`.

Abstract class

An abstract class is a class that can have many abstract members. An **abstract member** defines only a signature for an attribute or a method, without providing any implementation. You cannot instantiate an abstract class: you must create a subclass that implements all the abstract members.

Replace the definition of `Shape` and `Rectangle` in the worksheet as follows:

```
abstract class Shape(val x: Int, val y: Int) {  
    val area: Double  
    def description: String  
}  
  
class Rectangle(x: Int, y: Int, val width: Int, val height: Int)  
    extends Shape(x, y) {  
  
    val area: Double = width * height  
  
    def description: String =  
        "Rectangle " + width + " * " + height  
}
```

Our class `Shape` is now abstract. We cannot instantiate a `Shape` class directly anymore: we have to create an instance of `Rectangle` or any of the other subclasses of `Shape`. `Shape` defines two concrete members, `x` and `y`, and two abstract members, `area` and `description`. The subclass, `Rectangle`, implements the two abstract members.



*You can use the prefix `override` when implementing an abstract member, but it is not necessary. I recommend **not** adding it to keep the code less cluttered. Also, if you subsequently implement the abstract method in the superclass, the compiler will help you find all subclasses that had an implementation. It will not do this if they use `override`.*

Trait

A trait is similar to an abstract class: it can declare several abstract or concrete members and can be extended. It cannot be instantiated. The difference is that a given class can only extend one abstract class, however, it can **mixin** one to many traits. Also, a trait cannot have constructor arguments.

For instance, we can declare several traits, each declaring different abstract methods, and mixin them all in the `Rectangle` class:

```
trait Description {
    def description: String
}

trait Coordinates extends Description {
    def x: Int
    def y: Int

    def description: String =
        "Coordinates (" + x + ", " + y + ")"
}

trait Area {
    def area: Double
}

class Rectangle(val x: Int,
               val y: Int,
               val width: Int,
               val height: Int)
extends Coordinates with Description with Area {

    val area: Double = width * height

    override def description: String =
        super.description + " - Rectangle " + width + " * " + height
}

val rect = new Rectangle(x = 0, y = 3, width = 3, height = 2)
rect.description
```

The following string gets printed when evaluating `rect.description`:

```
| res0: String = Coordinates (0, 3) - Rectangle 3 * 2
```

The class `Rectangle` mixes in the traits `Coordinates`, `Description`, and `Area`. We need to use the keyword `extends` before `trait` or `class`, and the keyword `with` for all subsequent traits.

Notice that the `coordinates` trait also mixes the `description` trait, and provides a default implementation. As we did when we had a `shape` class, we override this implementation in `Rectangle`, and we can still call `super.description` to refer to the implementation of `description` in the trait `Coordinates`.

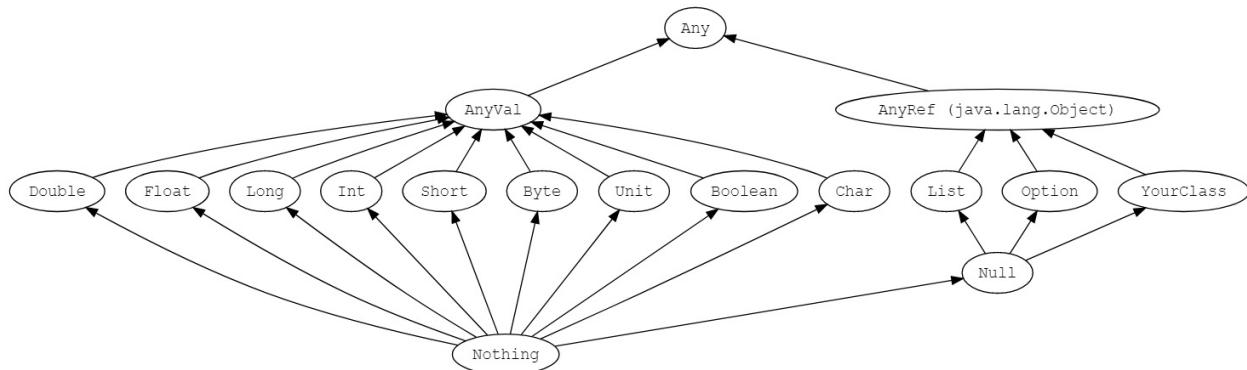
Another interesting point is that you can implement an abstract method with `val` – in trait `Area`, we defined `def area: Double`, and implemented it in `Rectangle` using `val area: Double`. It is a good practice to define abstract members with `def`. This way, the implementer of the trait can decide whether to define it by using a method or a variable.

Scala class hierarchy

All Scala types extend a built-in type called `Any`. This type is the root of the hierarchy of all Scala types. It has two direct subtypes:

- `AnyVal` is the root class of all value types. These types are represented as primitive types in the JVM.
- `AnyRef` is the root class of all object types. It is an alias for the class `java.lang.Object`.
- A variable of type `AnyVal` directly contains the value, whereas a variable of type `AnyRef` contains the address of an object stored somewhere in memory.

The following diagram shows a partial view of this hierarchy:



When you define a new class, it indirectly extends `AnyRef`. This being an alias for `java.lang.Object`, your class inherits from all the default methods implemented in `Object`. Its most important methods are as follows:

- `def toString: String` returns a string representation of an object. This method is called whenever you print an object using `println`. The default implementation returns the class's name followed by the address of the object in memory.
- `def equals(obj: Object): Boolean` returns `true` if the object is equal to another object, and `false` otherwise. This method is called whenever you compare two objects using `=`. The default implementation only compares the objects' references, and hence is equivalent to `eq`. Fortunately, most classes from the Java and Scala SDK override this method to provide a good comparison.

For instance, the class `java.lang.String` overrides the `equals` method to compare the content of the strings, character by character. Therefore, when you compare two strings with `==`, the result will be `true` if the strings are the same, even if they are stored in different places in memory.

- `def hashCode: Int` is called whenever you put an object in `Set` or if you use it as a key in `Map`. The default implementation is based on the address of the object. You can override this method if you want to have a better distribution of the data in `Set` or `Map`, which can improve the performance of these collections. However, if you do so, you must make sure that `hashCode` is consistent with `equals`: if two objects are equal, their `hashCodes` must also be equal.

It would be very tedious to have to override these methods for all your classes. Fortunately, Scala offers a special construct called `case class` that will automatically override these methods for us.

Case class

In Scala, we define most data structures using case classes. `case class` has one to many immutable attributes and provides several built-in functions compared to a standard class.

Type the following into the worksheet:

```
case class Person(name: String, age: Int)
val mikaelNew = new Person("Mikael", 41)
// 'new' is optional
val mikael = Person("Mikael", 41)
// == compares values, not references
mikael == mikaelNew
// == is exactly the same as .equals
mikael.equals(mikaelNew)

val name = mikael.name

// a case class is immutable. The line below does not compile:
//mikael.name = "Nicolas"
// you need to create a new instance using copy
val nicolas = mikael.copy(name = "Nicolas")
```

In the preceding code, the text following `//` is a comment that explains the preceding statement.

When you declare a class as `case class`, the Scala compiler automatically generates a default constructor, an `equals` and `hashCode` method, a `copy` constructor, and an accessor for each attribute.

Here is a screenshot of the worksheet we have. You can see the results of the evaluations on the right-hand side:

The screenshot shows a Scala worksheet interface. On the left, the code is written in a text editor-like area. On the right, the results of the evaluations are displayed in a table format.

Code Line	Evaluation Result
1	defined class Person
2	mikaelNew: Person = Person(Mikael,41)
3	mikael: Person = Person(Mikael,41)
4	res0: Boolean = true
5	res1: Boolean = true
6	name: String = Mikael
7	nicolas: Person = Person(Nicolas,41)

Companion object

A class can have a companion object. It must be declared in the same file as the class, using the keyword `object` followed by the name of the class it is accompanying. A companion object is a **singleton** – there is only one instance of this object in the JVM. It has its own type and is not an instance of the accompanied class.

This object defines static functions or values that are closely related to the class it is accompanying. If you are familiar with Java, it replaces the keyword `static`: in Scala, all static members of a class are declared inside the companion object.

Some functions in the companion object have a special meaning. Functions named `apply` are constructors of the class. The name `apply` can be omitted when we call them:

```
case class City(name: String, urbanArea: Int)
object City {
    val London = City("London", 1738)
    val Lausanne = City("Lausanne", 41)
}

case class Person(firstName: String, lastName: String, city: City)
object Person {
    def apply(fullName: String, city: City): Person = {
        val splitted = fullName.split(" ")
        new Person(firstName = splitted(0), lastName = splitted(1), city = city)
    }
}

// Uses the default apply method
val m1 = Person("Mikael", "Valot", City.London)
// Call apply with fullName
val m2 = Person("Mikael Valot", City.London)
// We can omit 'apply'
val n = Person("Nicolas Jorand", City.Lausanne)
```

In the preceding code, we defined a companion object for the class `city`, which defines some **constants**. The convention for constants is to have the first letter in uppercase.

The companion object for the class `Person` defines an additional `apply` function that acts as a constructor. Its implementation calls the method `split(" ")`, which splits a string separated by spaces to produce an array of type `String`. It allows us to

construct a `Person` instance using a single string where the first name and last name are separated by a space. We then demonstrated that we can either call the default `apply` function that comes with the case class, or the one we implemented.

Creating my first project

As you now know the basics of running code in the REPL and the worksheet, it is time to create your first 'Hello World' project. In this section, we are going to filter a list of people and print their name and age into the console.

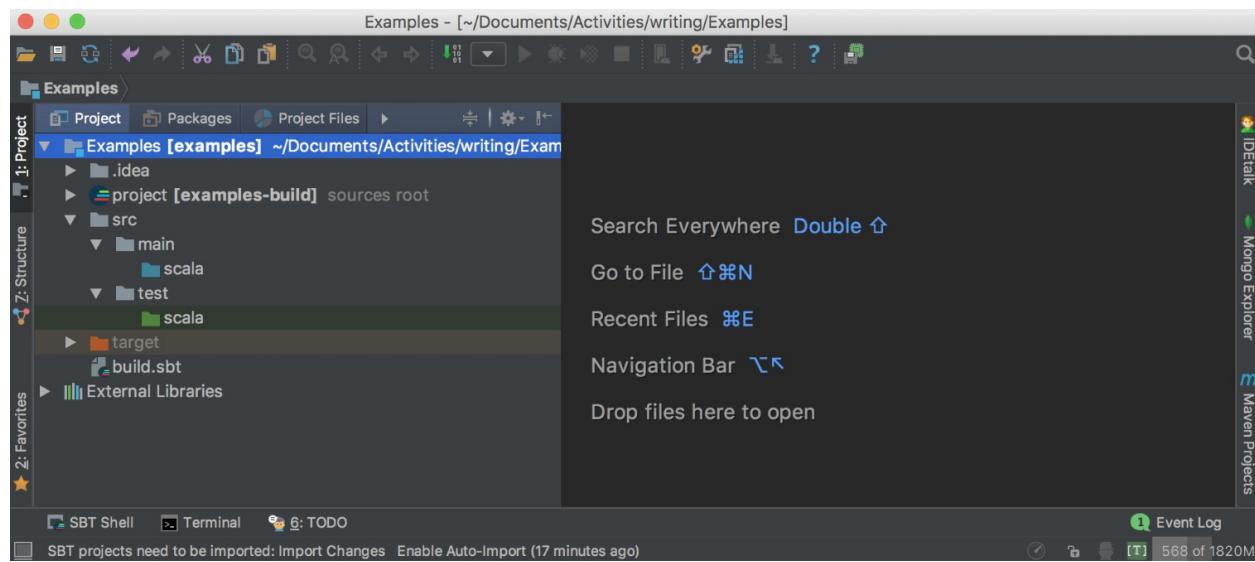
Creating the project

Repeat the same recipe that you completed in the *Installing IntelliJ* section to create a new project. Here is a summary of the tasks you must complete:

1. Run IntelliJ and select Create New Project
2. Select Scala and sbt
3. Input the name of the project, such as `Examples`
4. If the selected directory doesn't exist, IntelliJ will ask you if you want to create it – select OK

As soon as you accept that you are going to create the directory, IntelliJ is going to download all the necessary dependencies and build the project structure. Be patient, as this could take a while, especially if you do not have a good internet connection.

Once everything is downloaded, you should have your IDE in the following state:

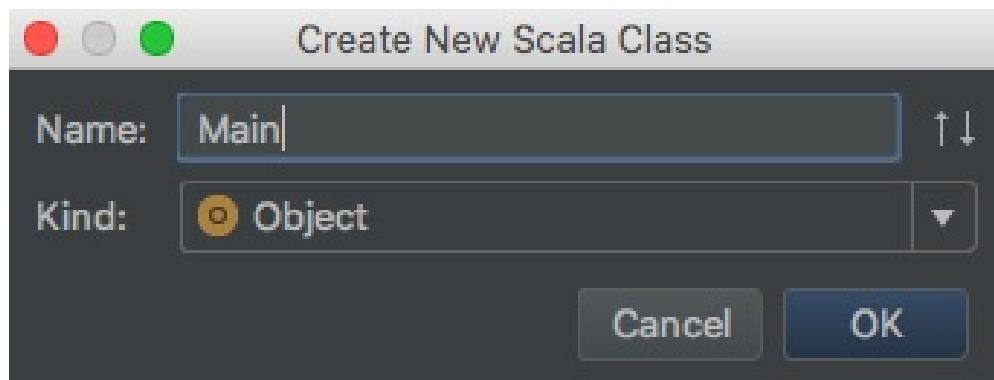


Notice the folder structure. The source code is under `src/main/scala` and the test code is under `src/test/scala`. If you have used Maven before, this structure should sound familiar.

Creating the Main object

Here we are! Let's create our first application. First, create the entry point for the program. If you are coming from Java, it would be equivalent to defining the `public static void main(String[] args)`.

Right-click on the `src/main/scala` folder and select New | Scala Class. Give `Main` as the class name and `Object` as the Kind:



We have created our first object. This object is a singleton. There can be only one instance of it in the JVM. The equivalent in Java would be a static class with static methods.

We would like to use it as the main entry point of our program. Scala provides a convenient class named `App` that needs to be extended. Let's extend our `Main` object with that class:

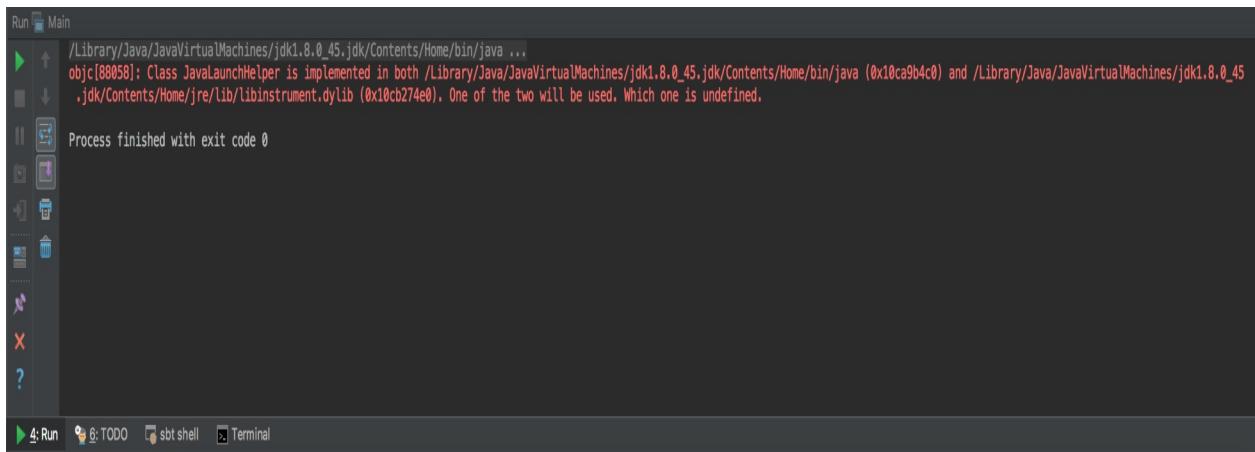
```
| object Main extends App {  
| }
```

The `App` superclass defines a static `main` method that will execute all the code defined inside your `Main` object. That's all – we created our first version, which does nothing!

We can now run the program in IntelliJ. Click on the small green triangle in the gutter of the object definition, as follows:

```
1 ► object Main extends App {  
2  
3 }  
4
```

The program gets compiled and executed, as shown in the following screenshot:



It is not spectacular, but let's improve it. To get the right habits, we are going to use the **TDD** technique to proceed further.

Writing the first unit test

TDD is a very powerful technique to write efficient, modular, and safe programs. It is very simple, and there are only three rules to play this game:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail, and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.



See the full article from Uncle Bob here: <http://butunclebob.com/Articles.UncleBob.TheThreeRulesOfTdd>.

There are multiple testing frameworks in Scala, but we chose ScalaTest (<http://www.scalatest.org/>) for its simplicity.

In order to add the ScalaTest library in the project, follow these steps:

1. Edit the `build.sbt` file.
2. Add a new repository resolver to search for Scala libraries.
3. Add the ScalaTest library:

```
name := "Examples"
version := "0.1"
scalaVersion := "2.12.4"
resolvers += "Artima Maven Repository" at "http://repo.artima.com/releases"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.4" % "test"
```



Notice the information bar on the top of the screen. It tells you that your file has changed and asks for multiple choices. As this is a small project, you can select **enable autoimport**.

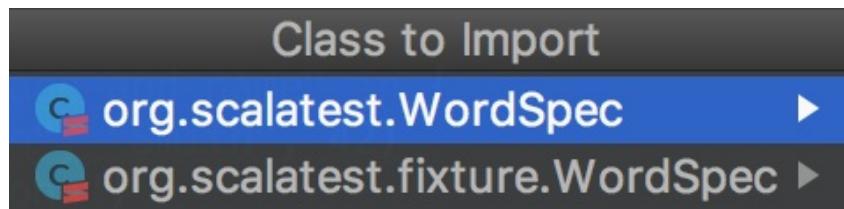
4. Create the test class by right-clicking on the `test/scala` folder and clicking on create a new class. Name it `MainSpec`.

ScalaTest offers multiple ways to define your test – the full list can be found on the official website (http://www.scalatest.org/at_a_glance/WordSpec). We are going to use the `WordSpec` style since it is quite prescriptive, offers a hierarchical structure, and is commonly used on large Scala projects.

Your `MainSpec` should extend the `WordSpec` class and the `Matchers` class, like so:

```
class MainSpec extends WordSpec with Matchers {  
}  
  
i The class Matchers is providing the word should as a keyword to perform the comparison on a test.
```

`WordSpec` and `Matchers` are underlined in red, which means that the class is not resolved. To make it resolved, go with the cursor on the class and press *Alt + Enter* of your keyboard. If you are positioned on the `WordSpec` word, a popup should appear. This is normal, as there are several classes named `WordSpec` in different packages:



Select the first option and IntelliJ will automatically add the import on the top of your code. On the `Matchers` class, as soon as you type *Alt + Enter*, the import will be added directly.

The final code should be as follows:

```
import org.scalatest.{WordSpec, Matchers}  
class MainSpec extends WordSpec with Matchers {  
}
```

Our class skeleton is now ready for our first test. We would like to create the `Person` class and test its constructor.

Let's explain what we would like to test using simple sentences. Complete the test class with the following code:

```
class MainSpec extends WordSpec with Matchers {  
  "A Person" should {  
    "be instantiated with a age and name" in {  
      val john = Person(firstName = "John", lastName = "Smith", 42)  
      john.firstName should be("John")  
      john.lastName should be("Smith")  
      john.age should be(42)  
    }  
  }  
}
```

```
| } }
```

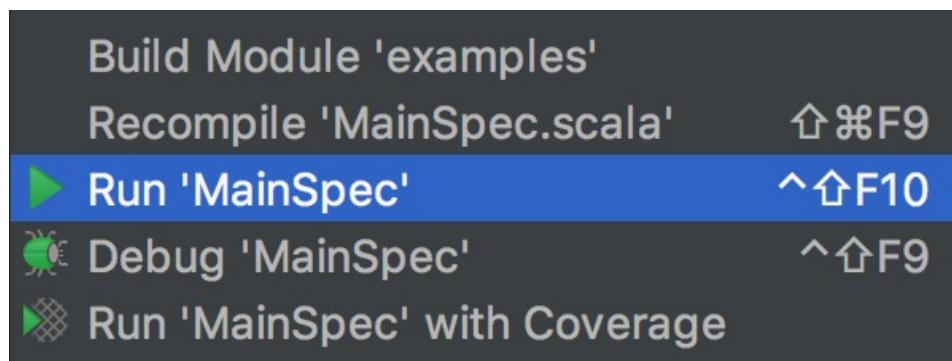
IntelliJ is complaining that it cannot resolve the symbols `Person`, `name`, `surname`, and `age`. This is expected since the `Person` class does not exist. Let's create it in the folder `src/main/scala`. Right-click on the folder and create a new class named `Person`.

Transform it in the case of the class by adding the `case` keyword and defining the constructor with the `name`, `surname`, and `age`:

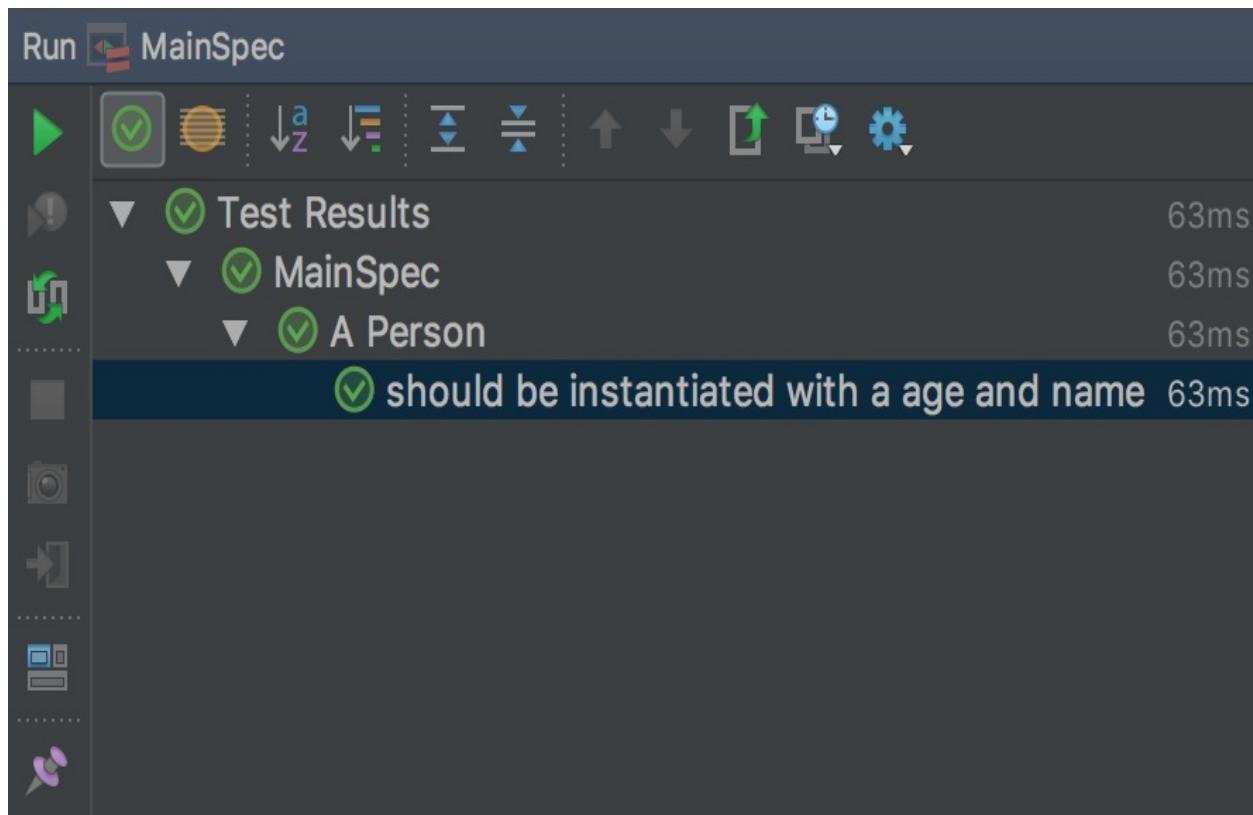
```
| case class Person(firstName: String, lastName: String, age: Int)
```

If you go back to the `MainSpec.scala` file, you'll notice that the class is now compiled without any error and warning. The green tick (勾) on the top-right of the code window confirms this.

Run the test by right-clicking on the `MainSpec.scala` file and selecting `Run 'MainSpec'`, or use the keyboard shortcut `Ctrl + Shift + F10` or `Ctrl + Shift + R`:



The test contained in `MainSpec` runs and the results appear in the Run window:

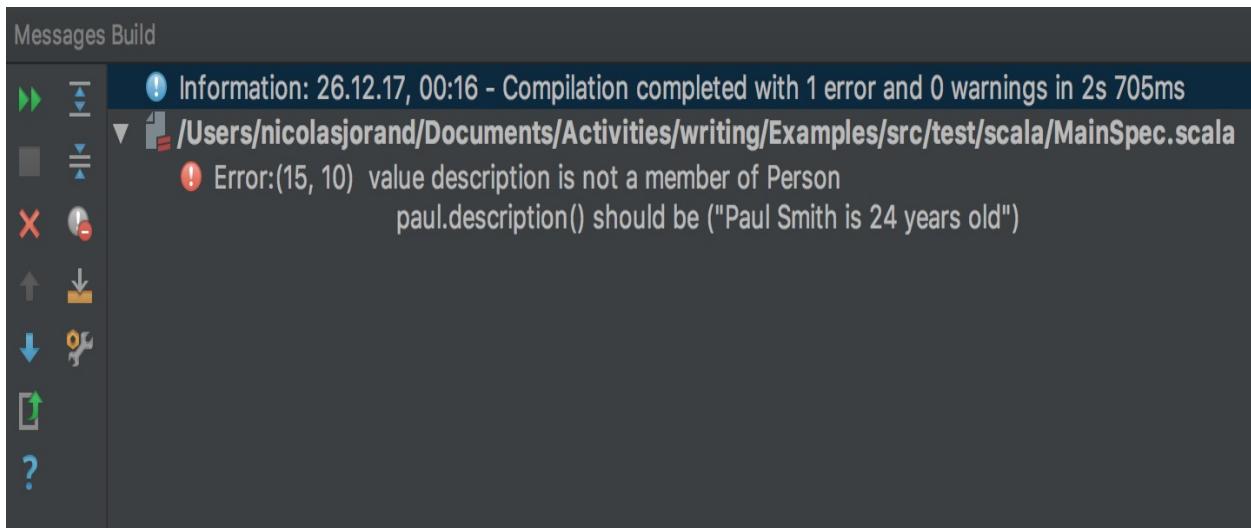


Implementing another feature

Now, we would like to have a nice representation of the person by stating his/her name and age. The test should look like the following:

```
"Get a human readable representation of the person" in {  
    val paul = Person(firstName = "Paul", lastName = "Smith", age = 24)  
    paul.description should be("Paul Smith is 24 years old")  
}
```

Run the test again. We will get a compilation error:



This is expected as the function doesn't exist on the `Person` class. To implement it, add the expected implementation by setting the cursor on the `description()` error in the `MainSpec.scala` class, hitting `Alt + Enter`, and selecting the create method `description`.

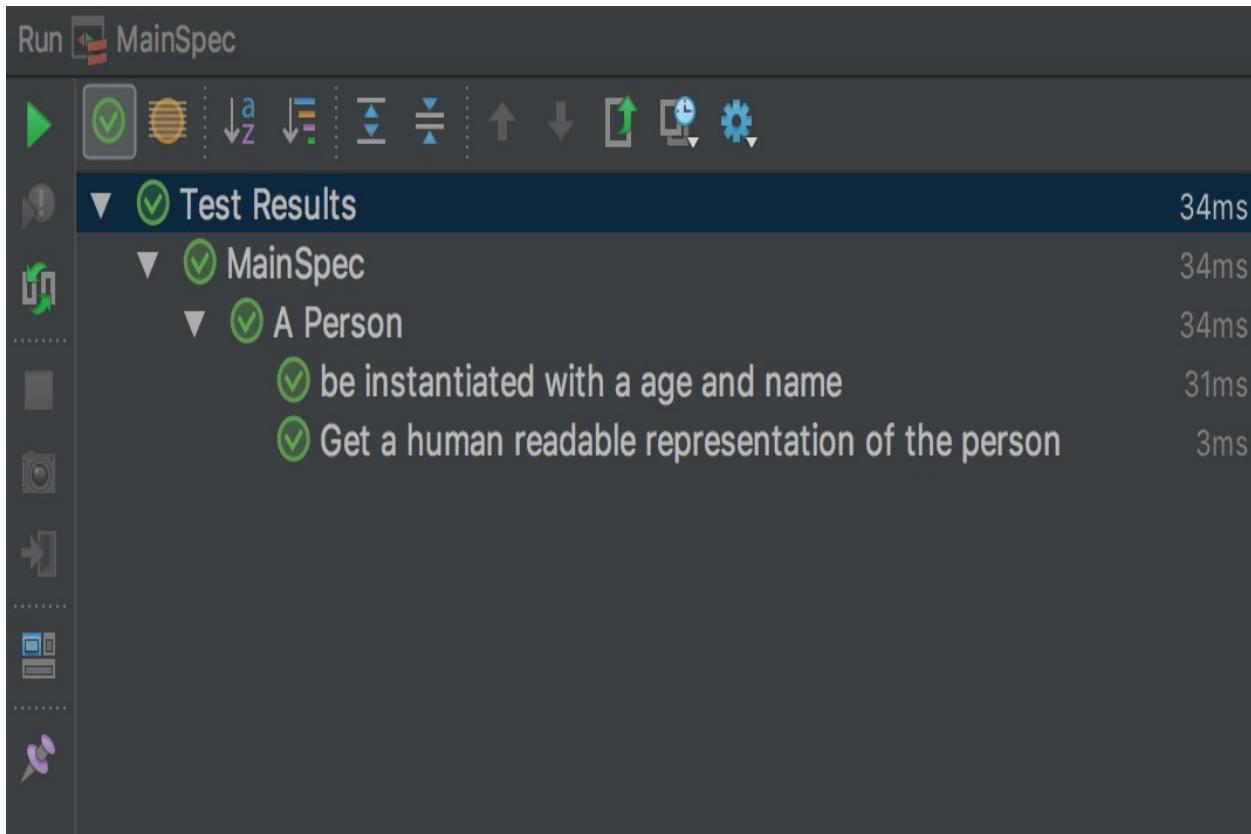
IntelliJ generates the method for you and sets the implementation to `???`. Replace `???` with the expected code:

```
| def description = s"$firstName $lastName is $age ${if (age <= 1) "year"
```

By doing so, we defined a method that does not take any parameter and return a string representing `Person`. In order to simplify the code, we are using a **string interpolation** to build the string. To use string interpolation, you just have to prepend an `s` before the first quote. Inside the quote, you can use the wildcard

\$ so that we can use an external variable and use the bracket after the dollar sign to enter more code than just a variable name.

Execute the test and the result should be green:



The next step is to write a utility function that, given a list of people, returns only the adults.

For the tests, two cases are defined:

```
"The Person companion object" should {
    val (akira, peter, nick) = (
        Person(firstName = "Akira", lastName = "Sakura", age = 12),
        Person(firstName = "Peter", lastName = "Müller", age = 34),
        Person(firstName = "Nick", lastName = "Tagart", age = 52)
    )
    "return a list of adult person" in {
        val ref = List(akira, peter, nick)
        Person.filterAdult(ref) should be(List(peter, nick))
    }
    "return an empty list if no adult in the list" in {
        val ref = List(akira)
        Person.filterAdult(ref) should be(List.empty[Person])
    }
}
```

Here, we used a tuple to define three variables. This is a convenient way to define multiple variables. The scope of the variables is bounded by the enclosing curly brackets.

Use IntelliJ to create the `filterAdult` function by using the *Alt+ Enter* shortcut. The IDE understands that the function should be in the `Person` companion object and generates it for you.



If you didn't use the named parameters and would like to use them, IntelliJ can help you: hit Alt + Enter when the cursor is after the parenthesis and select "used named arguments ...".

We implement this method using the `for` **comprehension** Scala feature:

```
object Person {  
    def filterAdult(persons: List[Person]): List[Person] = {  
        for {  
            person <- persons  
            if (person.age >= 18)  
        } yield (person)  
    }  
}
```

It is a good practice to define the return type of the method, especially when this method is exposed as a public API.

The `for` comprehension has been used only for demonstration purposes. We can simplify it using the `filter` method on `List`. `filter` is part of the Scala Collections API and is available for many kinds of collections:

```
def filterAdult(persons: List[Person]): List[Person] = {  
    persons.filter(_.age >= 18)  
}
```

Implementing the Main method

Now that all our tests are green, we can implement the `main` method. The implementation becomes trivial as all the code is already in the test:

```
object Main extends App {
    val persons = List(
        Person(firstName = "Akira", lastName = "Sakura", age = 12),
        Person(firstName = "Peter", lastName = "Müller", age = 34),
        Person(firstName = "Nick", lastName = "Tagart", age = 52))

    val adults = Person.filterAdult(persons)
    val descriptions = adults.map(p => p.description).mkString("\n\t")
    println(s"The adults are \n\t$descriptions")
}
```

The first thing is to define a list of `Person`, so that `Person.filterAdult()` is used to remove all the persons, not the adults. The `adults` variable is a list of `Person`, but I would like to transform this list of `Person` into a list of the description of the `Person`. To perform this operation, the `map` function of the collection is used. The `map` function transforms each element of the list by applying the function in the parameter.

The notation inside the `map()` function defines an anonymous function that takes `p` as the parameter. The body of the function is `p.description`. This notation is commonly used whenever a function takes another function as an argument.

Once we have a list of descriptions, we create a string with the `mkString()` function. It concatenates all the elements of the list using the special character `\n\t`, which are respectively the carriage return and the tab character.

Finally, we perform the side effect, which is the print on the console. To print in the console, the `println` alias is used. It is a syntactic sugar for `System.out.println`.

Summary

We have finished the first chapter, and you should now have the basics to start a project on your own. We covered the installation of an IDE to code in Scala with the basic usage of the dedicated build tool named SBT. Three ways to explore Scala have been demonstrated, including the REPL to test simple Scala features, the IntelliJ worksheet to play with a small environment, and lastly a real project.

To code our first project, we used ScalaTest and the TDD methodology so that we had good code quality from the beginning.

In the next chapter, we will write a complete program. It is a financial application that allows its users to estimate when they can retire. We will keep using the TDD technique and will further explore the Scala language, its development kit, and their best practices.

Developing a Retirement Calculator

In this chapter, we will put into practice the features of the Scala language seen in the first chapter. We will also introduce other elements of the Scala language and SDK to develop the model and logic for a retirement calculator. This calculator will help people work out how long and how much to save to have a comfortable retirement.

We will use the **test-driven development (TDD)** technique to develop the different functions. I encourage you to try writing the body of the functions yourself before looking at the solution. Also, it would be better to retype the code rather than copy/pasting it. You will remember it more and will have a sense of what it feels like to use IntelliJ's completion and editor. Do abuse the autocompletion with *Ctrl + spacebar*. You will not only type faster, but you will also discover what functions are available in a given class.

You are going to get a strong foundation for building more complex applications using the Scala language.

In this chapter, we will cover the following topics:

- Calculating the future capital
- Calculating when you can retire
- Using market rates
- Packaging the application

Project overview

Using some parameters such as your net income, your expenses, your initial capital, and so on, we will create functions to calculate the following:

- Your future capital at retirement
- Your capital after a number of years in retirement
- How long you need to save to be able to retire

We will first use a fixed interest rate for these calculations. After that, we will load market data from `.tsv` files, then refactor the previous functions to simulate what could happen during the investment period.

Calculating the future capital

The first thing you need to know when planning for retirement is how much capital you can get at your chosen retirement date. For now, we will assume that you invest your savings every month at a constant rate. To simplify things, we will ignore the effects of inflation, hence the capital calculated will be in today's money and the interest rate will be calculated as *real rate = nominal interest rate - inflation rate.*

We intentionally do not mention any currency in the rest of this chapter. You can consider that the amounts are in USD, EUR, or any other currency. It will not change the results as long as all the amounts are expressed in the same currency.

Writing a unit test for the accumulation phase

We want a function that behaves similarly to the `FV` function in Excel: it calculates the future value of an investment based on a constant interest rate. As we follow a TDD approach, the first thing to do is create a failing test:

1. Create a new Scala project called `retirement_calculator`. Follow the same instructions as in [Chapter 1](#), *Writing Your First Program*.
2. Right-click on the directory `src/main/scala` and select New | Package. Name it `retcalc`.
3. Right-click on the new package and select New | Scala class. Name it `RetCalcSpec`.
4. Enter the following code:

```
package retcalc

import org.scalactic.{Equality, TolerantNumerics, TypeCheckedTripleEquals}
import org.scalatest.{Matchers, WordSpec}

class RetCalcSpec extends WordSpec with Matchers with TypeCheckedTripleEquals {

    implicit val doubleEquality: Equality[Double] =
        TolerantNumerics.tolerantDoubleEquality(0.0001)

    "RetCalc.futureCapital" should {
        "calculate the amount of savings I will have in n months" in {
            val actual = RetCalc.futureCapital(
                interestRate = 0.04 / 12, nbOfMonths = 25 * 12,
                netIncome = 3000, currentExpenses = 2000,
                initialCapital = 10000)
            val expected = 541267.1990
            actual should ===(expected)
        }
    }
}
```

As seen in [Chapter 1](#), *Writing Your First Program*, in the section *Creating my first project*, we used the `WordSpec` ScalaTest style. We also used a handy feature called `TypeCheckedTripleEquals`. It provides a powerful assertion, `should ===`, that ensures at compile time that both sides of the equality have the same type. The default ScalaTest assertion `should` verifies the type equality at runtime. We encourage you to always use `should ===`, as it will save a lot of time when

refactoring code.

Besides, it lets us use a certain amount of tolerance when comparing double values. Consider the following declaration:

```
| implicit val doubleEquality: Equality[Double] =  
|   TolerantNumerics.tolerantDoubleEquality(0.0001)
```

It will let `a double1 should === (double2)` assertion pass if the absolute difference between `double1` and `double2` is lower than `0.0001`. This allows us to specify expected values to only the fourth digit after the decimal point. It also avoids hitting floating point calculation issues. For instance, enter the following code in the Scala Console:

```
| scala> val double1 = 0.01 -0.001 + 0.001  
| double1: Double = 0.01000000000000002  
  
| scala> double1 == 0.01  
| res2: Boolean = false
```

It can be a bit surprising, but this is a well-known problem in any language that encodes floating point numbers in binary. We could have used `BigDecimal` instead of `Double` to avoid this kind of issue, but for our purposes, we do not need the additional precision offered by `BigDecimal`, and `BigDecimal` computations are much slower.

The body of the test is quite straightforward; we call a function and expect a value. As we wrote the test first, we had to work out what would be the expected result before writing the production code. For non-trivial calculations, I generally use Excel or LibreOffice. For this function, the expected value can be obtained by using the formula `=-FV(0.04/12, 25*12, 1000, 10000, 0)`. We assume that the user saves the full difference between his or her income and his/her expenses every month. Hence, the PMT parameter in the `FV` function is `1,000 = netIncome - currentExpenses`.

We now have a failing test, but it does not compile, as the `RetCalc` object and its `futureCapital` function do not exist yet. Create a `RetCalc` object in a new package `retcalc` in `src/main/scala`, then select the red `futureCapital` call in the unit test and hit *Alt + Enter* to generate the function body. Fill in the names and types of the parameters. You should end up with the following code in `RetCalc.scala`:

```
| package retcalc
```

```
| object RetCalc {  
|   def futureCapital(interestRate: Double, nbOfMonths: Int, netIncome:  
|     Int, currentExpenses: Int, initialCapital: Double): Double = ???  
| }
```

Open `RetcalcSpec`, and type *Ctrl + Shift + R* to compile and run it. Everything should compile and the test should fail.

Implementing futureCapital

We now have a failing test, so it is time to make it pass by writing the production code. If we use `initialCapital = 10,000` and `monthlySavings = 1,000`, the computation we need to perform can be decomposed as follows:

- For month 0, before any savings, we have `capital0 = initialCapital = 10,000`.
- For month 1, our initial capital generated some interest. We also saved 1,000 more. We therefore have `capital1 = capital0 * (1 + monthlyInterestRate) + 1,000`
- For month 2, we have `capital2 = capital1 * (1 + monthlyInterestRate) + 1,000`

There is a mathematical formula to compute `capitalN` from the parameters, but we will not use it here. This formula works well for fixed interest rates, but we will use variable interest rates later in this chapter.

Here is the body of the function:

```
def futureCapital(interestRate: Double, nbOfMonths: Int, netIncome: Int, currentExpenses: Int): Double =  
  val monthlySavings = netIncome - currentExpenses  
  
  def nextCapital(accumulated: Double, month: Int): Double =  
    accumulated * (1 + interestRate) + monthlySavings  
  
  (0 until nbOfMonths).foldLeft(initialCapital)(nextCapital)
```

We first generate a collection of integers using `0` to `nbOfMonths`, and we then iterate through it using `foldLeft`. `foldLeft` is one of the most powerful functions in the Scala collection library. Many other functions in the `collections` library could be implemented just by using `foldLeft`, such as `reverse`, `last`, `contains`, `sum`, and so on.

In the Scala SDK, the signature of `foldLeft` is as follows:

```
| def foldLeft[B](z: B)(op: (B, A) => B): B
```

You can see its definition in IntelliJ by pointing at it with the mouse and using cmd + left-click. This introduces some new syntax:

- `[B]` means that the function has a **type parameter** named `B`. When we call

the function, the compiler automatically infers what `B` is, depending on the type of the `z: B` argument. In our code, the `z` argument is `initialCapital`, of type `Double`. Therefore, our call to `foldLeft` in `futureCapital` will behave as if the function was defined with `B = Double`:

```
def foldLeft(z: Double)(op: (Double, A) => Double): Double.
```

- The function has two parameter lists. Scala allows you to have many parameter lists. Each list can have one or many parameters. This does not change the behavior of the function; it is just a way of separating the concerns of each parameter list.
- `op: (B, A) => B` means that `op` must be a function that has two parameters of type `B` and `A` and returns a value of type `B`. Since `foldLeft` is a function that takes another function as an argument, we say that `foldLeft` is a **higher order function**.

If we consider a `coll` collection, `foldLeft` works as follows:

1. It creates a `var acc = z` accumulator then calls the `op` function:

```
acc = op(acc, coll(0))
```

2. It carries on calling `op` with each element of the collection:

```
acc = op(acc, coll(i))
```

3. It returns `acc` once it has iterated through all elements of the collection

In our `futureCapital` function, we pass `op = nextCapital`. The `foldLeft` will iterate through all `Int` between `1` and `nbofMonths`, each time computing the capital using the previous capital. Note that, for now, we do not use the `month` parameter in `nextCapital`. We must declare it, though, because the `op` function in `foldLeft` must have two parameters.

You can now run the `RetCalcSpec` unit test again. It should pass.

Refactoring the production code

In the TDD approach, it is common to refactor the code once we have passing tests. If our test coverage is good, we should not have any fear of changing the code, because any mishap should be flagged by a failing test. This is known as a **Red-Green-Refactor** cycle.

Change the body of `futureCapital` with the following code:

```
| def futureCapital(interestRate: Double, nbOfMonths: Int, netIncome: Int, currentExpenses
|   val monthlySavings = netIncome - currentExpenses
|   (0 until nbOfMonths).foldLeft(initialCapital)(
|     (accumulated, _) => accumulated * (1 + interestRate) +
|       monthlySavings)
| }
```

Here, we have inlined the `nextCapital` function in the `foldLeft` call. In Scala, we can define an **anonymous function** using the syntax:

```
| (param1, param2, ..., paramN) => function body.
```

We saw earlier that the `month` parameter in `nextCapital` was not used. In an anonymous function, it is a good practice to name any unused parameter with `_`. A parameter named `_` cannot be used in the function body. If you try to replace the `_` character with a name, IntelliJ will underline it. If you hover the mouse over it, you will see a popup stating `Declaration is never used`. You can then hit *Alt + Enter* and choose to remove the unused element to automatically change it back to `_`.

Writing a test for the decumulation phase

Now you know how much capital you can expect at your retirement date. It turns out you can reuse the same `futureCapital` function, to work out how much capital will be left for your heirs.

Add the following test in `RetCalcSpec`, underneath the previous unit test, and run it. It should pass:

```
"RetCalc.futureCapital" should {
    "calculate how much savings will be left after having taken a pension
    for n months" in {
        val actual = RetCalc.futureCapital(
            interestRate = 0.04/12, nbOfMonths = 40 * 12,
            netIncome = 0, currentExpenses = 2000, initialCapital =
            541267.1990)
        val expected = 309867.53176
        actual should ===(expected)
    }
}
```

So, if you live for 40 years after your retirement date, spend the same amount every month, and don't have any other income, you will still have a significant capital left for your heirs. If the remaining capital was negative, that would have meant that you would have run out of money at some point during your retirement and it is an outcome we want to avoid.

Feel free to call the function from the Scala Console and try different values that would match more closely to your personal situation. Try different values for the interest rate and observe how you can end up with a negative capital if the rate is low.

Note that, in a production system, you would certainly add more unit tests to cover some other edge cases and make sure that the function will not crash. As we will cover error handling in [Chapter 3, Handling Errors](#), we can assume that the test coverage of `futureCapital` is good enough for now.

Simulating a retirement plan

Now that we know how to calculate the capital at retirement and after death, it would be useful to combine the two calls in a single function. This function will simulate a retirement plan in one go.

Writing a failing unit test

Here is the unit test you need to add to `RetCalcSpec`:

```
"RetCalc.simulatePlan" should {
    "calculate the capital at retirement and the capital after death" in {
        val (capitalAtRetirement, capitalAfterDeath) =
            RetCalc.simulatePlan(
                interestRate = 0.04 / 12,
                nbOfMonthsSaving = 25 * 12, nbOfMonthsInRetirement = 40 * 12,
                netIncome = 3000, currentExpenses = 2000,
                initialCapital = 10000)
        capitalAtRetirement should === (541267.1990)
        capitalAfterDeath should === (309867.5316)
    }
}
```

Select the call to `simulatePlan`, and hit *Alt + Enter* to let IntelliJ create the function for you in `RetCalc`. It should have the following signature:

```
def simulatePlan(interestRate: Double,
                 nbOfMonthsSavings: Int, nbOfMonthsInRetirement: Int,
                 netIncome: Int, currentExpenses: Int, initialCapital:
                 Double) : (Double, Double) = ???
```

Now compile the project with `cmd + F9`, and run `RetCalcSpec`. It should fail since the `simulatePlan` function must return two values. The simplest way of modeling the return type is to use `Tuple2`. In Scala, a tuple is an immutable data structure which holds several objects of different types. The number of objects contained in a tuple is fixed. It is akin to a case class, which does not have specific names for its attributes. In type theory, we say that a tuple or a case class is a **product type**.

Working with tuples

Type the following in the Scala Console to get more familiar with tuples. Feel free to experiment with different types and sizes of tuples:

```
scala> val tuple3 = (1, "hello", 2.0)
tuple3: (Int, String, Double) = (1,hello,2.0)

scala> tuple3._1
res1: Int = 1

scala> tuple3._2
res2: String = hello

scala> val (a, b, c) = tuple3
a: Int = 1
b: String = hello
c: Double = 2.0
```

You can create tuples of any length up to 22, and access their elements using `_1`, `_2`, and so on. You can also declare several variables in one go for each element of the tuple.

Implementing simulatePlan

The implementation for `simulatePlan` is straightforward; we call `futureCapital` twice with different arguments:

```
def simulatePlan(interestRate: Double,
                 nbOfMonthsSaving: Int, nbOfMonthsInRetirement: Int,
                 netIncome: Int, currentExpenses: Int, initialCapital:
                 Double) : (Double, Double) = {
    val capitalAtRetirement = futureCapital(
        interestRate = interestRate, nbOfMonths = nbOfMonthsSaving,
        netIncome = netIncome, currentExpenses = currentExpenses,
        initialCapital = initialCapital)

    val capitalAfterDeath = futureCapital(
        interestRate = interestRate, nbOfMonths = nbOfMonthsInRetirement,
        netIncome = 0, currentExpenses = currentExpenses,
        initialCapital = capitalAtRetirement)

    (capitalAtRetirement, capitalAfterDeath)
}
```

Run `RetCalcSpec` again, and it should pass now. Feel free to experiment calling `simulatePlan` from the Scala Console with different values.

Calculating when you can retire

If you have tried to call `simulatePlan` from the Scala Console, you probably tried different values for `nbOfMonths` and observed the resulting capital at retirement and after death. It would be useful to have a function that finds the optimal `nbOfMonths` so that you have enough capital to never run out of money during your retirement.

Writing a failing test for nbOfMonthsSaving

As usual, let's start with a new unit test to clarify what we expect from this function:

```
"RetCalc.nbOfMonthsSaving" should {
  "calculate how long I need to save before I can retire" in {
    val actual = RetCalc.nbOfMonthsSaving(
      interestRate = 0.04 / 12, nbOfMonthsInRetirement = 40 * 12,
      netIncome = 3000, currentExpenses = 2000, initialCapital = 10000)
    val expected = 23 * 12 + 1
    actual should ===(expected)
  }
}
```

In this test, the expected value can be a bit difficult to figure out. One way would be to use the `NPM` function in Excel. Alternatively, you could call `simulatePlan` many times in the Scala Console, and increase `nbOfMonthsSaving` to gradually find out what the optimal value is.

Writing the function body

In functional programming, we avoid mutating variables. In an imperative language, you would typically implement `nbOfMonthsSaving` by using a `while` loop. It is also possible to do so in Scala, but it is a better practice to only use immutable variables. One way of solving this problem is to use recursion:

```
def nbOfMonthsSaving(interestRate: Double, nbOfMonthsInRetirement: Int,
                     netIncome: Int, currentExpenses: Int, initialCapital: Double): Int
def loop(months: Int): Int = {
    val (capitalAtRetirement, capitalAfterDeath) = simulatePlan(
        interestRate,
        nbOfMonthsSaving = months, nbOfMonthsInRetirement =
        nbOfMonthsInRetirement,
        netIncome = netIncome, currentExpenses = currentExpenses,
        initialCapital = initialCapital)

    val returnValue =
        if (capitalAfterDeath > 0.0)
            months
        else
            loop(months + 1)
    returnValue
}
loop(0)
```

We declare the recursive function inside the body of the function, as it is not meant to be used anywhere else. The `loop` function increments `months` by 1 until the calculated `capitalAfterDeath` is positive. The `loop` function is initiated in the body of `nbMonthsSaving` with `months = 0`. Note that IntelliJ highlights the fact that the `loop` function is recursive with a kind of @ sign.

Now, we can run our unit test again, and it should pass. However, we are not quite done yet. What happens if you can never reach a month that would satisfy the condition `capitalAfterDeath > 0.0`? Let's find out by writing another test.

Understanding tail-recursion

Add the following tests underneath the previous one:

```
"RetCalc.nbOfMonthsSaving" should {
    "calculate how long I need to save before I can retire" in {...}

    "not crash if the resulting nbOfMonths is very high" in {
        val actual = RetCalc.nbOfMonthsSaving(
            interestRate = 0.01 / 12, nbOfMonthsInRetirement = 40 * 12,
            netIncome = 3000, currentExpenses = 2999, initialCapital = 0)
        val expected = 8280
        actual should ===(expected)
    }

    "not loop forever if I enter bad parameters" in pending
```

We will implement the `not loop forever` later on. It is a good practice to write pending tests as soon as you think about new edge cases or other use cases for your function. It helps to keep a direction toward the end goal, and gives some momentum—as soon as you have made the previous test pass, you know exactly what test you need to write next.

Run the test, and it will fail with `StackOverflowError`. This is because each time `loop` is called recursively, local variables are saved onto the JVM stack. The size of the stack is quite small, and as we can see, it is quite easy to fill it up. Fortunately, there is a mechanism in the Scala compiler to automatically transform tail-recursive calls into a `while` loop. We say that a recursive call (the call to the `loop` function inside the `loop` function) is tail-recursive when it is the last instruction of the function.

We can easily change our previous code to make it tail-recursive. Select `returnValue`, and hit cmd + O to inline the variable. The body of the `loop` function should look like this:

```
@tailrec
def loop(months: Int): Int = {
    val (capitalAtRetirement, capitalAfterDeath) = simulatePlan(
        interestRate = interestRate,
        nbOfMonthsSaving = months, nbOfMonthsInRetirement =
        nbOfMonthsInRetirement,
        netIncome = netIncome, currentExpenses = currentExpenses,
        initialCapital = initialCapital)

    if (capitalAfterDeath > 0.0)
```

```
    months
else
  loop(months + 1)
```

IntelliJ changed the small symbol in the editor's margin to highlight the fact that our function is now tail-recursive. Run the test again and it should pass. It is a good practice to also put an annotation `@tailrec` before the function definition. This tells the compiler that it must verify that the function is indeed tail-recursive. If you annotate a `@tailrec` function that is not tail-recursive, the compiler will raise an error.



When you know that the depth of a recursive call can be high (more than 100), always make sure it is tail-recursive.

When you write a tail-recursive function, always annotate it with `@tailrec` to let the compiler verify it.

Ensuring termination

We are not quite done yet because our function might loop indefinitely. Imagine that you always spend more than what you earn. You will never be able to save enough to retire, even if you live a million years! Here is a unit test highlighting this:

```
"RetCalc.nbOfMonthsSaving" should {
    "calculate how long I need to save before I can retire" in {...}

    "not crash if the resulting nbOfMonths is very high" in {...}

    "not loop forever if I enter bad parameters" in {
        val actual = RetCalc.nbOfMonthsSavingV2(
            interestRate = 0.04 / 12, nbOfMonthsInRetirement = 40 * 12,
            netIncome = 1000, currentExpenses = 2000, initialCapital = 10000)
        actual should === (Int.MaxValue)
    }
}
```

We decided to use a special value `Int.MaxValue` to indicate that `nbOfMonths` is infinite. This is not very pretty, but we will see in the next chapter how we can model this better with `Option` or `Either`. This is good enough for now.

To make the test pass, we just need to add an `if` statement:

```
def nbOfMonthsSaving(interestRate: Double, nbOfMonthsInRetirement: Int,
                     netIncome: Int, currentExpenses: Int, initialCapital: Double): Int
  @tailrec
  def loop(nbOfMonthsSaving: Int): Int = {...}

  if (netIncome > currentExpenses)
    loop(0)
  else
    Int.MaxValue
}
```

Using market rates

In our calculations, we have always assumed that the interest rate of return was constant, but the reality is more complex. It would be more accurate to use real rates from market data to gain more confidence with our retirement plan. For this, we first need to change our code to be able to perform the same calculations using variable interest rates. Then, we will load real market data to simulate regular investments in a fund by tracking the S & P 500 index.

Defining an algebraic data type

In order to support variable rates, we need to change the signature of all functions that accept `interestRate: Double`. Instead of a double, we need a type that can represent either a constant rate or a sequence of rates.

Considering two types `A` and `B`, we previously saw how to define a type that can hold a value of type `A` and a value of type `B`. This is a product type, and we can define it using a tuple, such as `ab: (A, B)`, or a case class, such as `case class MyProduct(a: A, b: B)`.

On the other hand, a type that can hold either `A` **or** `B` is a sum type, and in Scala, we declare it using a `sealed trait` inheritance:

```
sealed trait Shape
case class Circle(diameter: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape
```

An **Algebraic Data Type (ADT)** is a type that composes sum types and product types to define a data structure. In the preceding code, we defined a `Shape` ADT, which composes a sum type (a `Shape` can be a `Circle` or a `Rectangle`), with a product type `Rectangle` (a `Rectangle` holds a width and a height).

The `sealed` keyword indicates that all subclasses of the trait must be declared in the same `.scala` file. If you attempt to declare a class that extends a `sealed trait` in another file, the compiler will reject it. This is used to guarantee that our inheritance tree is complete, and as we will see later on, it has interesting benefits when it comes to using pattern matching.

Going back to our problem, we can define a `Returns` ADT as follows. Create a new Scala class in the `retcalc` package in `src/main/scala`:

```
package retcalc
sealed trait Returns
case class FixedReturns(annualRate: Double) extends Returns
case class VariableReturns(returns: Vector[VariableReturn]) extends Returns
case class VariableReturn(monthId: String, monthlyRate: Double)
```

For `VariableReturn`, we keep the monthly interest rate and an identifier `monthId` that

will have the form `2017.02`, for February 2017. I recommend that you use `vector` whenever you need to model a sequence of elements. `vector` is faster than `List` for appending/inserting elements or accessing an element by index.

Filtering returns for a specific period

When we have `variableReturns` over a long period, for instance, 1900 to 2017, it can be interesting to use a smaller period to simulate what would happen if the historical returns in a smaller period, say 50 years, would be repeated.

We are going to create a method in the `VariableReturns` class that will keep only the returns contained in a specific period. Here is the unit test in `ReturnsSpec.scala`:

```
"VariableReturns.fromUntil" should {
  "keep only a window of the returns" in {
    val variableReturns = VariableReturns(Vector.tabulate(12) { i =>
      val d = (i + 1).toDouble
      VariableReturn(f"2017.$d%02.0f", d)
    })
    variableReturns.fromUntil("2017.07", "2017.09").returns should ===
    (Vector(
      VariableReturn("2017.07", 7.0),
      VariableReturn("2017.08", 8.0)
    ))
    variableReturns.fromUntil("2017.10", "2018.01").returns should ===
    (Vector(
      VariableReturn("2017.10", 10.0),
      VariableReturn("2017.11", 11.0),
      VariableReturn("2017.12", 12.0)
    ))
  }
}
```

First, we generate a sequence of returns and assign them to `variableReturns` using the function `vector.tabulate`. It generates 12 elements, and each element is produced by an anonymous function, taking a parameter `i` that will go from 0 to 11. In the call to the `VariableReturn` constructor, for the `monthId` argument, we use the `f` interpolator to generate a string in the form `2017.01` when `d = 1`, `2017.02` when `d = 2`, and so on.

The function `fromUntil` that we are specifying will return a `VariableReturns` type that contains a specific window inside the original returns. For now, we assume that the arguments passed to `fromUntil` are valid months that are present in `variableReturns`. Ideally, we should add unit tests to specify what should happen if they are not.

Here is the implementation in `Returns.scala`:

```
case class VariableReturns(returns: Vector[VariableReturn]) extends Returns {
  def fromUntil(monthIdFrom: String, monthIdUntil: String):
    VariableReturns =
    VariableReturns(
      returns
        .dropWhile(_.monthId != monthIdFrom)
        .takeWhile(_.monthId != monthIdUntil))
}
```

We use the higher order function `dropWhile` to drop elements until we reach the condition `monthId == monthIdFrom`. Then, we call `takeWhile` on the resulting collection to keep all elements until `monthId == monthIdUntil`. This will return a collection that keeps only the elements in a window that starts at `monthIdFrom` and ends just before `monthIdUntil`.

Run `ReturnsSpec`, and it should pass.

Pattern matching

Now that we have a way of expressing variable returns, we need to change our `futureCapital` function to accept a `Returns` type instead of a monthly interest rate of type `Double`. Change the tests first:

```
"RetCalc.futureCapital" should {
    "calculate the amount of savings I will have in n months" in {
        // Excel =-FV(0.04/12,25*12,1000,10000,0)
        val actual = RetCalc.futureCapital(FixedReturns(0.04),
            nbOfMonths = 25 * 12, netIncome = 3000,
            currentExpenses = 2000, initialCapital = 10000).right.value
        val expected = 541267.1990
        actual should ===(expected)
    }

    "calculate how much savings will be left after having taken a
    pension for n months" in {
        val actual = RetCalc.futureCapital(FixedReturns(0.04),
            nbOfMonths = 40 * 12, netIncome = 0, currentExpenses = 2000,
            initialCapital = 541267.198962).right.value
        val expected = 309867.5316
        actual should ===(expected)
    }
}
```

Then, change the `futureCapital` function in `RetCalc` as follows:

```
def futureCapital(returns: Returns, nbOfMonths: Int, netIncome: Int, currentExpenses: Int,
    initialCapital: Double): Double = {
    val monthlySavings = netIncome - currentExpenses
    (0 until nbOfMonths).foldLeft(initialCapital) {
        case (accumulated, month) =>
            accumulated * (1 + Returns.monthlyRate(returns, month)) +
            monthlySavings
    }
}
```

Here, instead of just using the `interestRate` in the formula, we introduced a new function called `Returns.monthlyRate` which we must now create. As we follow a rather strict TDD approach, we will only create its signature first, then write the unit test, and finally implement it.

Write the function signature in `Returns.scala`:

```
object Returns {
    def monthlyRate(returns: Returns, month: Int): Double = ???
```

Create a new unit test `ReturnsSpec` in the `retcalc` package in `src/test/scala`:

```
class ReturnsSpec extends WordSpec with Matchers with TypeCheckedTripleEquals {  
    implicit val doubleEquality: Equality[Double] =  
        TolerantNumerics.tolerantDoubleEquality(0.0001)  
  
    "Returns.monthlyRate" should {  
        "return a fixed rate for a FixedReturn" in {  
            Returns.monthlyRate(FixedReturns(0.04), 0) should ===(0.04 / 12)  
            Returns.monthlyRate(FixedReturns(0.04), 10) should ===(0.04 / 12)  
        }  
  
        val variableReturns = VariableReturns(Vector(  
            VariableReturn("2000.01", 0.1),  
            VariableReturn("2000.02", 0.2)))  
        "return the nth rate for VariableReturn" in {  
            Returns.monthlyRate(variableReturns, 0) should ===(0.1)  
            Returns.monthlyRate(variableReturns, 1) should ===(0.2)  
        }  
  
        "roll over from the first rate if n > length" in {  
            Returns.monthlyRate(variableReturns, 2) should ===(0.1)  
            Returns.monthlyRate(variableReturns, 3) should ===(0.2)  
            Returns.monthlyRate(variableReturns, 4) should ===(0.1)  
        }  
    }  
}
```

These tests act as a specification for our `monthlyRate` function. For `variableRate`, the `monthlyRate` must return the n^{th} rate stored in the returned `vector`. If n is greater than the number of rates, we decide that `monthlyRate` should go back to the beginning of `vector`, as if the history of our variable returns would repeat itself infinitely. We could have made a different choice here, for instance, we could have taken a mirror of the returns, or we could have just returned some error if we reached the end. To implement this rotation, we are taking the month value and applying the modulo (`%` in Scala) of the length of the vector.

The implementation introduces a new element of syntax, called **pattern matching**:

```
def monthlyRate(returns: Returns, month: Int): Double = returns match {  
    case FixedReturns(r) => r / 12  
    case VariableReturns(rs) => rs(month % rs.length).monthlyRate  
}
```

You can now run `ReturnsSpec`, and all tests should pass. Pattern matching allows you to deconstruct an ADT and evaluate some expression when it matches one of the patterns. You can also assign variables along the way and use them in the expression. In the preceding example, `case FixedReturns(r) => r/12` can be interpreted as "if the variable `returns` is of type `FixedReturns`, assign `r` =

`returns.annualRate`, and return the result of the expression `r/12`".

This is a simple example, but you can use much more complicated patterns. This feature is very powerful, and can often replace lots of `if/else` expressions. You can try some more complex patterns in the Scala Console:

```
scala> Vector(1, 2, 3, 4) match {
  case head +: second +: tail => tail
}
res0: scala.collection.immutable.Vector[Int] = Vector(3, 4)

scala> Vector(1, 2, 3, 4) match {
  case head +: second +: tail => second
}

scala> ("0", 1, (2.0, 3.0)) match {
  case ("0", int, (d0, d1)) => d0 + d1
}
res2: Double = 5.0

scala> "hello" match {
  case "hello" | "world" => 1
  case "hello world" => 2
}
res3: Int = 1

scala> def present(p: Person): String = p match {
  case Person(name, age) if age < 18 => s"$name is a child"
  case p => s"${p.name} is an adult"
}
present: (p: Person)String
```

It is a good practice to exhaustively match all possible patterns for your value. Otherwise, if no pattern matches the value, Scala will raise a runtime exception, and it might crash your program. However, when you use `sealed traits`, the compiler is aware of all the possible classes for a trait and will issue a warning if you do not match all cases.

In `Returns.scala`, try to comment out this line with cmd + /:

```
// def monthlyRate(returns: Returns, month: Int): Double = returns match {
//   case FixedReturns(r) => r / 12
//   case VariableReturns(rs) => rs(month % rs.length).monthlyRate
// }
```

Recompile the project with cmd + F9. The compiler will warn you that you are doing something wrong:

```
| Warning:(27, 59) match may not be exhaustive.
| It would fail on the following input: FixedReturns(_)
| def monthlyRate(returns: Returns, month: Int): Double = returns match {
```

If you then try to remove the `sealed` keyword and recompile, the compiler will not issue any warning.

We now have a good grasp of how to use pattern matching. Keep the `sealed` keyword, revert the comment in `monthlyRate`, and run `ReturnsSpec` to make sure everything is green again.

If you are coming from an object-oriented language, you might wonder why we did not implement `monthlyRate` using an abstract method with implementations in `FixedRate` and `VariableRate`. This is perfectly feasible in Scala, and some people might prefer this design choice.

However, as I am an advocate of a functional programming style, I prefer using pure functions in objects:

- They are easier to reason about, as the whole dispatching logic is in one place.
- They can be moved to other objects easily, which facilitates refactoring.
- They have a more limited scope. In class methods, you always have all the attributes of the class in scope. In a function, you only have the parameters of the function. This helps unit testing and readability, as you know that the function cannot use anything else but its parameters. Also, it can avoid side effects when the class has mutable attributes.
- Sometimes in object-oriented design, when a method manipulates two objects, `A` and `B`, it is not clear if the method should be in class `A` or class `B`.

Refactoring simulatePlan

As we changed the signature of `futureCapital`, we also need to change the callers of that function. The only caller is `simulatePlan`. Before introducing variable rates, the implementation was straightforward: we just had to call `futureCapital` for the accumulation and decumulation phases with the same fixed rate argument. However, with variable rate, we must make sure that the decumulation phase uses the rates that follow the rates of the accumulation phase.

For instance, consider that you started saving in 1950, and retired in 1975. For the accumulation phase, you need to use the returns from 1950 to 1975, and for the decumulation, you must use the returns from 1975. We created a new unit test to make sure that we are using different returns for the two phases:

```
val params = RetCalcParams(
    nbOfMonthsInRetirement = 40 * 12,
    netIncome = 3000,
    currentExpenses = 2000,
    initialCapital = 10000)

"RetCalc.simulatePlan" should {
  "calculate the capital at retirement and the capital after death" in {
    val (capitalAtRetirement, capitalAfterDeath) =
      RetCalc.simulatePlan(
        returns = FixedReturns(0.04), params, nbOfMonthsSavings = 25*12)

    capitalAtRetirement should === (541267.1990)
    capitalAfterDeath should === (309867.5316)
  }

  "use different returns for capitalisation and drawdown" in {
    val nbOfMonthsSavings = 25 * 12
    val returns = VariableReturns(
      Vector.tabulate(nbOfMonthsSavings + params.nbOfMonthsInRetirement)(i =>
        if (i < nbOfMonthsSavings)
          VariableReturn(i.toString, 0.04 / 12)
        else
          VariableReturn(i.toString, 0.03 / 12)))
    val (capitalAtRetirement, capitalAfterDeath) =
      RetCalc.simulatePlan(returns, params, nbOfMonthsSavings)
    capitalAtRetirement should ===(541267.1990)
    capitalAfterDeath should ===(-57737.7227)
  }
}
```

Since `simulatePlan` has quite a lot of parameters apart from the `returns` parameter, we decided to put them in a case class called `RetCalcParams`. This way, we are able

to reuse the same parameters for different unit tests. We will also be able to reuse it in `nbMonthsSaving`. As seen previously, we use the function `tabulate` to generate values for our variable returns.

The expected value for `capitalAtRetirement` can be obtained with Excel by using -
 $FV(0.04/12, 25*12, 1000, 10000)$. The expected value for `capitalAfterDeath` can be obtained by using - $FV(0.03/12, 40*12, -2000, 541267.20)$.

Here is the implementation in `RetCalc`:

```
case class RetCalcParams(nbOfMonthsInRetirement: Int,
                       netIncome: Int,
                       currentExpenses: Int,
                       initialCapital: Double)

object RetCalc {
  def simulatePlan RETURNS: Returns, params: RetCalcParams,
  nbOfMonthsSavings: Int)
  : (Double, Double) = {
    import params._

    val capitalAtRetirement = futureCapital(
      returns = returns,
      nbOfMonths = nbOfMonthsSavings,
      netIncome = netIncome, currentExpenses = currentExpenses,
      initialCapital = initialCapital)

    val capitalAfterDeath = futureCapital(
      returns = OffsetReturns(returns, nbOfMonthsSavings),
      nbOfMonths = nbOfMonthsInRetirement,
      netIncome = 0, currentExpenses = currentExpenses,
      initialCapital = capitalAtRetirement)

    (capitalAtRetirement, capitalAfterDeath)
  }
}
```

The first line, `import params._`, brings all the parameters of `RetCalcParams` into scope. This way, you can directly use, for instance, `netIncome` without having to prefix it with `params.netIncome`. In Scala, you can not only import classes from a package, but also functions or values from an object.

In the second call to `futureCapital`, we introduce a new subclass called `OffsetReturns`, which will shift the starting month. We need to write a new unit test for it in `ReturnsSpec`:

```
"Returns.monthlyReturn" should {
  "return a fixed rate for a FixedReturn" in {...}

  val variableReturns = VariableReturns(
    Vector(VariableReturn("2000.01", 0.1), VariableReturn("2000.02", 0.2)))

  "return the nth rate for VariableReturn" in {...}
}
```

```

    "return an error if n > length" in {...}

  "return the n+offset th rate for OffsetReturn" in {
    val returns = OffsetReturns(variableReturns, 1)
    Returns.monthlyRate(returns, 0).right.value should ===(0.2)
  }
}

```

And the corresponding implementation in `Returns.scala` is as follows:

```

sealed trait Returns
case class FixedReturns(annualRate: Double) extends Returns
case class VariableReturn(monthId: String, monthlyRate: Double)
case class OffsetReturns(orig: Returns, offset: Int) extends Returns

object Returns {
  def monthlyRate(returns: Returns, month: Int): Double = returns match {
    case FixedReturns(r) => r / 12
    case VariableReturns(rs) => rs(month % rs.length).monthlyRate
    case OffsetReturns(rs, offset) => monthlyRate(rs, month + offset)
  }
}

```

For an offset return, we call `monthlyRate` recursively and add the offset to the requested month.

Now, you can compile everything with cmd + F9 and rerun the unit tests. They should all pass.

Loading market data

In order to calculate the real monthly returns of our investments in a fund tracking the S & P 500, we are going to load a tab-separated file containing the S & P 500 prices and dividends, and another file containing the consumer price index. This will let us calculate a real rate of return, stripped out of inflation.

Writing the unit test using the column selection mode

First, copy `sp500_2017.tsv` from <https://github.com/PacktPublishing/Scala-Programming-Projects/blob/master/Chapter02/retirement-calculator/src/main/resources/sp500.tsv> to `src/test/resources`. Then, create a new unit test called `EquityDataspec` in the `retcalc` package. If you retype this example, try the column selection mode (*Alt + Shift + Insert*). Copy the content of the `.tsv` file in the test, then select the first column with *Shift + Down* 13 times, and then type `EquityData("")`. Finally, edit the rest of the lines using the arrow keys, delete, comma, and so on:

```
package retcalc

import org.scalatest.{Matchers, WordSpec}

class EquityDataSpec extends WordSpec with Matchers {
    "EquityData.fromResource" should {
        "load market data from a tsv file" in {
            val data = EquityData.fromResource("sp500_2017.tsv")
            data should ===(Vector(
                EquityData("2016.09", 2157.69, 45.03),
                EquityData("2016.10", 2143.02, 45.25),
                EquityData("2016.11", 2164.99, 45.48),
                EquityData("2016.12", 2246.63, 45.7),
                EquityData("2017.01", 2275.12, 45.93),
                EquityData("2017.02", 2329.91, 46.15),
                EquityData("2017.03", 2366.82, 46.38),
                EquityData("2017.04", 2359.31, 46.66),
                EquityData("2017.05", 2395.35, 46.94),
                EquityData("2017.06", 2433.99, 47.22),
                EquityData("2017.07", 2454.10, 47.54),
                EquityData("2017.08", 2456.22, 47.85),
                EquityData("2017.09", 2492.84, 48.17)
            ))
        }
    }

    "EquityData.monthlyDividend" should {
        "return a monthly dividend" in {
            EquityData("2016.09", 2157.69, 45.03).monthlyDividend should ===
                (45.03 / 12)
        }
    }
}
```

The first lines of `sp500_2017.tsv` look like this:

month	SP500	dividend
2016.09	2157.69	45.03

2016.10	2143.02	45.25
2016.11	2164.99	45.48

Loading the file with Source

Our implementation must drop the first line which contains the headers, then for each line, split and create a new instance of `EquityData`:

```
package retcalc

import scala.io.Source

case class EquityData(monthId: String, value: Double, annualDividend: Double) {
    val monthlyDividend: Double = annualDividend / 12
}

object EquityData {
    def fromResource(resource: String): Vector[EquityData] =
        Source.fromResource(resource).getLines().drop(1).map { line =>
            val fields = line.split("\t")
            EquityData(
                monthId = fields(0),
                value = fields(1).toDouble,
                annualDividend = fields(2).toDouble)
        }.toVector
}
```

This code is quite compact, and you might lose a sense of what types are returned by intermediate calls. In IntelliJ, you can select a portion of code and hit *Alt + =* to show the inferred type of the expression.

We first load the `.tsv` file using `scala.io.Source.fromResource`. This takes the name of a file located in a `resource` folder and returns a `source` object. It can be in `src/test/resources` OR `src/main/resources`. When you run a test, both folders will be searched. If you run the production code, only the files in `src/main/resources` will be accessible.

`getLines` returns `Iterator[String]`. An **iterator** is a **mutable** data structure that allows you to iterate over a sequence of elements. It provides many functions that are common to other collections. Here, we drop the first line, which contains the header, and transforms each line using an anonymous function passed to `map`.

The anonymous function takes `line` of type `String`, transforms it into `Array[String]` using `split`, and instantiates a new `EquityData` object.

Finally, we convert the resulting `Iterator[EquityData]` into `Vector[EquityData]` using

.toVector. This step is very important: we convert the mutable, unsafe, iterator into an immutable, safe vector. Public functions should, in general, not accept or return mutable data structures:

- It makes the code harder to reason about, as you have to remember the state the mutable structure is in.
- The program will behave differently depending on the order/repetition of the function calls. In the case of an iterator, it can be iterated only once. If you need to iterate again, you won't get any data:

```
scala> val iterator = (1 to 3).iterator
iterator: Iterator[Int] = non-empty iterator

scala> iterator foreach println
1
2
3

scala> iterator foreach println

scala>
```

Loading inflation data

Now that we can load some equity data, we need to load inflation data so that we're able to compute inflation-adjusted returns. It is very similar to the loading of equity data.

First, copy `cpi_2017.tsv` from <https://github.com/PacktPublishing/Scala-Programming-Project/blob/master/Chapter02/retirement-calculator/src/main/resources/cpi.tsv> to `src/test/resources`. Then, create a new unit test called `InflationDataSpec` in the `retcalc` package:

```
package retcalc

import org.scalatest.{Matchers, WordSpec}

class InflationDataSpec extends WordSpec with Matchers {
    "InflationData.fromResource" should {
        "load CPI data from a tsv file" in {
            val data = InflationData.fromResource("cpi_2017.tsv")
            data should ===(Vector(
                InflationData("2016.09", 241.428),
                InflationData("2016.10", 241.729),
                InflationData("2016.11", 241.353),
                InflationData("2016.12", 241.432),
                InflationData("2017.01", 242.839),
                InflationData("2017.02", 243.603),
                InflationData("2017.03", 243.801),
                InflationData("2017.04", 244.524),
                InflationData("2017.05", 244.733),
                InflationData("2017.06", 244.955),
                InflationData("2017.07", 244.786),
                InflationData("2017.08", 245.519),
                InflationData("2017.09", 246.819)
            ))
        }
    }
}
```

Then, create the corresponding `InflationData` class and companion object:

```
package retcalc

import scala.io.Source

case class InflationData(monthId: String, value: Double)

object InflationData {
    def fromResource(resource: String): Vector[InflationData] =
        Source.fromResource(resource).getLines().drop(1).map { line =>
            val fields = line.split("\t")
            InflationData(monthId = fields(0), value = fields(1).toDouble)
        }
}
```

```
| }    }.toVector
```

Computing real returns

For a given month, n , the real return is $return_n - inflationRate_n$, hence the following formula:

$$realReturn_n = \frac{price_n + dividends_n}{price_{n-1}} - \frac{inflation_n}{inflation_{n-1}}$$

We are going to create a new function `in Returns` that creates `variableReturns` using `vector[EquityData]` and `vector[InflationData]`. Add the following unit test to `ReturnsSpec`:

```
"Returns.fromEquityAndInflationData" should {
  "compute real total returns from equity and inflation data" in {
    val equities = Vector(
      EquityData("2117.01", 100.0, 10.0),
      EquityData("2117.02", 101.0, 12.0),
      EquityData("2117.03", 102.0, 12.0))

    val inflations = Vector(
      InflationData("2117.01", 100.0),
      InflationData("2117.02", 102.0),
      InflationData("2117.03", 102.0))

    val returns = Returns.fromEquityAndInflationData(equities,
      inflations)
    returns should ===(VariableReturns(Vector(
      VariableReturn("2117.02", (101.0 + 12.0 / 12) / 100.0 - 102.0 /
        100.0),
      VariableReturn("2117.03", (102.0 + 12.0 / 12) / 101.0 - 102.0 /
        102.0))))
  }
}
```

We create two small `vector` instances of `EquityData` and `InflationData`, and calculate the expected value using the preceding formula.

Here is the implementation of `fromEquityAndInflationData` in `Returns.scala`:

```
object Returns {
  def fromEquityAndInflationData(equities: Vector[EquityData],
    inflations: Vector[InflationData]): VariableReturns = {
    VariableReturns(equities.zip(inflations).sliding(2).collect {
      case (prevEquity, prevInflation) +: (equity, inflation) +:
        Vector() =>
        val inflationRate = inflation.value / prevInflation.value
    })
  }
}
```

```

    val totalReturn =
      (equity.value + equity.monthlyDividend) / prevEquity.value
    val realTotalReturn = totalReturn - inflationRate

    VariableReturn(equity.monthId, realTotalReturn)
  }.toVector
}

```

Firstly, we `zip` the two `vectors` to create a collection of tuples, `(EquityData, InflationData)`. This operation brings our two collections together as if we were zipping a jacket. It is a good habit to play around with the Scala Console to get a sense of what it does:

```

scala> Vector(1,2).zip(Vector("a", "b", "c"))
res0: scala.collection.immutable.Vector[(Int, String)] = Vector((1,a), (2,b))

```

Note that the resulting `vector` has a size that is the minimum size of the two arguments. The last element, "c", is lost, because there is nothing to zip it with!

This is a good start, as we could now iterate through a collection that can give us *priceⁿ*, *dividendsⁿ*, and *inflationⁿ*. But in order to calculate our formula, we also need the previous data on *n-1*. For this, we use `sliding(2)`. I encourage you to read the documentation on `sliding`. Let's try it in the console:

```

scala> val it = Vector(1, 2, 3, 4).sliding(2)
it: Iterator[scala.collection.immutable.Vector[Int]] = non-empty iterator

scala> it.toVector
res0: Vector[scala.collection.immutable.Vector[Int]] =
Vector(Vector(1, 2), Vector(2, 3), Vector(3, 4))

scala> Vector(1).sliding(2).toVector
res12: Vector[scala.collection.immutable.Vector[Int]] = Vector(Vector(1))

```

`sliding(p)` creates an `Iterator` which will produce collections of size `p`. Each collection will have a new iterated element plus all the previous `p-1` elements. Notice that if the collection size `n` is lower than `p`, the produced collection will have a size of `n`.

Next, we iterate through the sliding collections using `collect`. `collect` is similar to `map`: it allows you to transform the elements of a collection, but with the added capability of filtering them. Basically, whenever you want to `filter` and `map` a collection, you can use `collect` instead. The filtering is performed using pattern matching. Anything that does not match any pattern is filtered out:

```

scala> val v = Vector(1, 2, 3)
v: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)

```

```
| scala> v.filter(i => i != 2).map(_ + 1)
| res15: scala.collection.immutable.Vector[Int] = Vector(2, 4)
|
| scala> v.collect { case i if i != 2 => i + 1 }
| res16: scala.collection.immutable.Vector[Int] = Vector(2, 4)
```

Notice that, in the preceding code, we used `map(_ + 1)` instead of `map(i => i + 1)`. This is a shorthand notation for an anonymous function. Whenever you use a parameter once in your anonymous function, you can replace it with `_`.

Finally, we pattern match on our zipped and sliding elements using the following:

```
| case (prevEquity, prevInflation) +: (equity, inflation) +: Vector() =>
```

This has the benefit of filtering out sliding elements of size 0 or 1, if we were passing equities or inflation arguments of size 0 or 1. We will not write a unit test for this edge case in this book, but I encourage you to do so as an excercise.

The rest of the function is straightforward: we use the matched variables to compute the formula and create a new `VariableReturn` instance. The resulting iterator is converted to `vector`, and we instantiate a `VariableReturns` case class using it.

Packaging the application

We have now implemented some useful building blocks, and it is time to create an executable so that end users can use our calculator with their own parameters.

Creating the App object

We are going to build a simple executable around `RetCalc.simulatePlan`. It will take a list of parameters separated by spaces, and print the results on the console.

The test we are going to write integrates several components together and will use a full market data set. As such, it is not really a unit test anymore; it is an integration test. For this reason, we suffixed it with IT instead of Spec.

First, copy `sp500.tsv` and `cpi.tsv` from <https://github.com/PacktPublishing/Scala-Programming-Projects/blob/master/Chapter02/retirement-calculator/src/main/resources/sp500.tsv> and <https://github.com/PacktPublishing/Scala-Programming-Projects/blob/master/Chapter02/retirement-calculator/src/main/resources/cpi.tsv> to `src/main/resources`, then create a new unit test called `simulatePlanIT` in `src/test/scala`:

```
package retcalc

import org.scalactic.TypeCheckedTripleEquals
import org.scalatest.{Matchers, WordSpec}

class SimulatePlanAppIT extends WordSpec with Matchers with TypeCheckedTripleEquals {
  "SimulatePlanApp.strMain" should {
    "simulate a retirement plan using market returns" in {
      val actualResult = SimulatePlanApp.strMain(
        Array("1997.09,2017.09", "25", "40", "3000", "2000", "10000"))

      val expectedResult =
        s"""
          |Capital after 25 years of savings:    499923
          |Capital after 40 years in retirement: 586435
          |""".stripMargin
      actualResult should === (expectedResult)
    }
  }
}
```

We call a function called `strMain` which will return a string instead of printing it to the console. This way, we can assert on the content printed to the console. To keep things simple, we assume that the arguments are passed in a specific order. We will develop a more user-friendly interface in the next chapter. The arguments are as follows:

1. A period that we will use in the variables `returns`, separated by a comma
2. The number of years of savings

3. The number of years in retirement
4. Income
5. Expenses
6. Initial capital

The expected value is a string that we define using triple quotes. In Scala, a string enclosed in triple quotes lets you enter special characters such as a quote or newline. It is very convenient to enter multiline strings while keeping a good indentation. The `|` characters allow you to mark the beginning of each line, and the `.stripMargin` function removes the white spaces before the `|`, as well as the `|` itself in order. In IntelliJ, when you type `"""` and then hit *Enter*, it automatically adds the `|` and `.stripMargin` after the closing triple quote.

The implementation calls the different functions we implemented earlier. Notice that IntelliJ can autocomplete the name of the files using *Ctrl + spacebar*, for instance, after `EquityData.fromResource("Create a new object SimulatePlanApp in the package retdcalc:`

```
package retdcalc

object SimulatePlanApp extends App {
  println(strMain(args))

  def strMain(args: Array[String]): String = {
    val (from :+: until :+: Nil) = args(0).split(",").toList
    val nbOfYearsSaving = args(1).toInt
    val nbOfYearsInRetirement = args(2).toInt

    val allReturns = Returns.fromEquityAndInflationData(
      equities = EquityData.fromResource("sp500.tsv"),
      inflations = InflationData.fromResource("cpi.tsv"))
    val (capitalAtRetirement, capitalAfterDeath) =
      RetCalc.simulatePlan(
        returns = allReturns.fromUntil(from, until),
        params = RetCalcParams(
          nbOfMonthsInRetirement = nbOfYearsInRetirement * 12,
          netIncome = args(3).toInt,
          currentExpenses = args(4).toInt,
          initialCapital = args(5).toInt,
          nbOfMonthssSavings = nbOfYearsSaving * 12)

    s"""
      |Capital after $nbOfYearsSaving years of savings:
      ${capitalAtRetirement.round}
      |Capital after $nbOfYearsInRetirement years in retirement:
      ${capitalAfterDeath.round}
      """".stripMargin
  }
}
```

The only code executed when we run our executable will be `println(strMain(args))`.

It is a good practice to keep this code as short as possible because it is not covered by any test. Our function `strMain` is covered, so we are fairly sure there won't be any unexpected behavior from a single `println`. `args` is an `Array[String]` containing all the arguments passed to the executable.

The first line of `strMain` uses a pattern matching on `List` to assign `from` and `until`. The variables will be assigned only if the split first argument, in our test `"1997.09, 2017.09"`, is a `List` of two elements.

Then, we load the equity and inflation data from our `.tsv` files. They contain data from 1900.01 until 2017.09. We then call `Returns.fromEquityAndInflationData` to compute the real returns.

After having assigned the returns to `allReturns`, we call `simulatePlan` with the right arguments. The returns are filtered for a specific period using `from` and `until`. Finally, we return `String` using a string interpolation and triple quotes.

This implementation is the first draft and is quite brittle. It will indeed crash with a horrible `ArrayIndexOutOfBoundsException` if we do not pass enough parameters to our executable, or with a `NumberFormatException` if some strings cannot be converted to `Int` or `Double`. We will see in the next chapter how we can handle these error cases gracefully, but for now, our calculator does the job as long as we feed it with the right arguments.

You can now run `simulatePlanIT`, and it should pass.

Since we built an application, we can also run it as such. Move your cursor to `simulatePlanApp`, and hit *Ctrl + Shift + R*. The app should run and crash with an exception because we did not pass any arguments. Click on the launcher for `simulatePlanApp` (underneath the Build menu), then click Edit Configurations. Put the following in program arguments:

```
| 1997.09, 2017.09 25 40 3000 2000 10000
```

Then, click OK, and run `simulatePlanApp` again. It should print the same content as what we had in our unit test. You can try calling the application with different parameters and observe the resulting calculated capitals.

Packaging the application

So far so good, but what if we want to send this application to uncle Bob so that he can plan for his retirement too? It would not be very convenient to ask him to download IntelliJ or SBT. We are going to package our application in a `.jar` file so that we can run it with a single command.

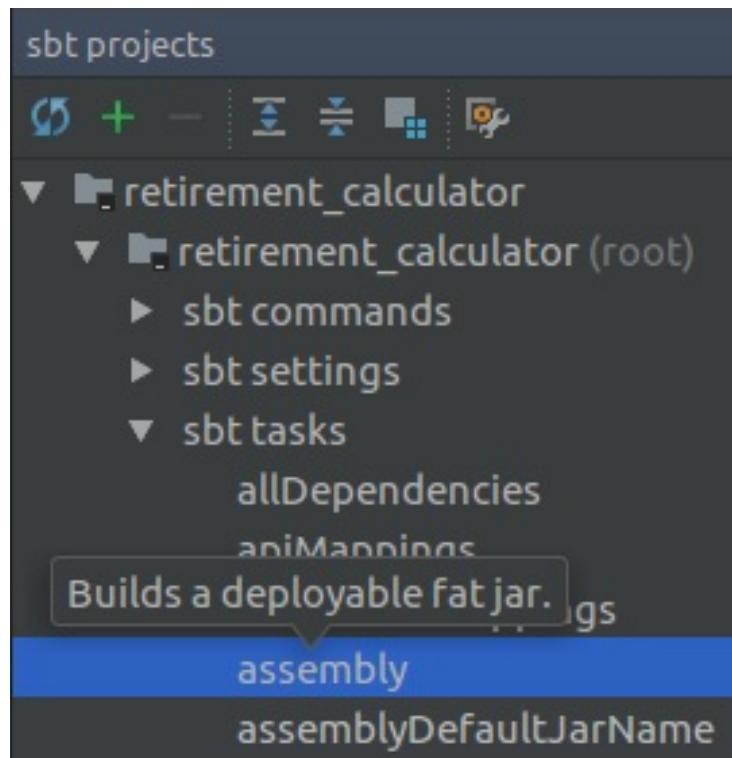
SBT provides a package task that can create a `.jar` file, but this file will not contain the dependencies. In order to package our own classes as well as the classes coming from the dependent libraries, we are going to use the `sbt-assembly` plugin. Create a new file called `project/assembly.sbt` containing the following:

```
| addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.6")
```

Then, edit `build.sbt` to define the name of our main class:

```
name := "retirement_calculator"
version := "0.1"
scalaVersion := "2.12.4"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.4" % "test"
mainClass in Compile := Some("retcalc.SimulatePlanApp")
```

Click on the SBT tab on the top-right and then on the Refresh button. Then, expand the project and double-click on the assembly task:



The assembly task will compile all the classes, run all the tests, and if they all pass, package a fat JAR. You should see an output similar to this at the bottom of the SBT console:

```
[info] Checking every *.class/*.jar file's SHA-1.
[info] Merging files...
[warn] Merging 'META-INF/MANIFEST.MF' with strategy 'discard'
[warn] Strategy 'discard' was applied to a file
[info] SHA-1: 7b7710bf370159c549a11754bf66302a76c209b2
[info] Packaging /home/mikael/projects/scala_fundamentals/retirement_calculator/target/s
[info] Done packaging.
[success] Total time: 11 s, completed 14-Jan-2018 12:23:39
```

Copy the location of the .jar file in your clipboard. Now, you can use a Unix terminal or a Windows command prompt and run the application as follows:

```
$ java -jar <path to your .jar file> 1997.09,2017.09 25 40 3000 2000 10000
Capital after 25 years of savings: 499923
Capital after 40 years in retirement: 586435
```

It is now easier to try different parameters. It is interesting to see that some periods are much more lucrative than others:

```
$ java -jar <path to your .jar file> 2000.01,2010.01 25 40 3000 2000 10000
Capital after 25 years of savings: 225209
```

```
Capital after 40 years in retirement: -510074
$ java -jar <path to your .jar file> 1950.01,1960.01 25 40 3000 2000 10000
Capital after 25 years of savings:    4505196
Capital after 40 years in retirement: 2077953853
```

Summary

We have covered how to create a small project in Scala from scratch to packaging. We used TDD along the way so that it is guaranteed that our code was well-designed and robust. It gives us confidence when we refactor code: as long as all tests pass, we know that our code stills works. We modeled our domain with immutable data structures and processed them using pure functions that have no side effects.

We used some basic features of the language that are used in most projects, and you should now be familiar with enough building blocks to implement a wide variety of projects.

As a further exercise, you could enhance this calculator with one or many of these features:

- Create an app for the function `RetCalc.nbOfMonthssaving`, which calculates how long you need to save before you can retire.
- Create a function called `RetCalc.annualizedTotalReturn`, which calculates the geometric average of a sequence of returns. See <https://www.investopedia.com/terms/a/annualized-total-return.asp> for more details.
- Create a function called `monthlyPension`, which calculates how much you will get in retirement if you save a given amount every month for a given number of months.
- Incorporate other streams of income. Maybe you will get a state pension after a number of years, or you might get an inheritance.
- Load a different index, such as STOXX Europe 600, MSCI World Index, and so on.
- Most people do not invest all their savings in the stock market, but wisely diversify with bonds, that is, typically 60% stocks, 40% bonds. You could add a new function in `Returns.scala` to calculate the returns of a mixed portfolio.
- We observed that some periods have much higher returns than others. As it is difficult to predict the future, you could run multiple simulations using different periods, and calculate a probability of success.

In the next chapter, we will improve our calculator further by handling errors in a functional way.

Handling Errors

In this chapter, we will continue working on the retirement calculator that we implemented in [Chapter 2, Developing a Retirement Calculator](#). Our calculator worked correctly as long as we passed the right arguments, but would fail badly with a horrible stack trace if any of the parameters were wrong. Our program only worked for what we call the *happy path*.

The reality of writing production software is that all kinds of error scenarios can occur. Some of them are recoverable, some of them must be presented to the user in an attractive way, and, for some hardware-related errors, we might need to let the program crash.

In this chapter, we will introduce exception handling, explain what referential transparency is, and try to convince you that exceptions are not the best way to deal with errors. Then, we will explain how to use functional programming constructs to effectively handle the possibility of an error.

In each section, we will briefly introduce a new concept, and then use it in a Scala worksheet to get a sense of how to use it. After that, we will apply this new knowledge to improve the retirement calculator.

In this chapter, we will look at the following topics:

- Using exceptions when necessary
- Understanding referential transparency
- Using `Option` to represent optional values
- Using `Either` to handle errors sequentially
- Using `validated` to handle errors in parallel

Setup

If you have not completed [Chapter 2, Developing a Retirement Calculator](#), then you can check out the retirement calculator project at GitHub. If you are not already familiar with Git, I would advise that you read the documents at <https://guides.github.com/introduction/git-handbook/> first.

To begin the setup, go through the following steps:

1. Create an account at <https://github.com/> if you do not have one already.
2. Go to the retirement calculator project at <https://github.com/PacktPublishing/Scala-Programming-Projects>. Click on Fork in the top-right corner to fork the project into your account.
3. Once the project is forked, click on Clone or download, and copy the URL into the clipboard.
4. In IntelliJ, go to File | New | Project from Version Control | GitHub and make the following edits:
 - Git repository URL: Paste the URL of your forked repository
 - Parent directory: Choose a location
 - Directory name: Keep `retirement_calculator`
 - Click on Clone
5. The project should be imported in IntelliJ. Click on git: master in the bottom-right of the screen and select Remote branches | origin/chapter2 | Checkout as new local branch. Name the new branch `chapter3_yourusername` to distinguish it from the final solution, which is in the `origin/chapter3` branch.
6. Build the project with *Ctrl + F9*. Everything should compile.

Using exceptions

Exceptions are one of the mechanisms that we can use in Scala to handle error scenarios. It consists of two statements:

- The `throw exceptionObject` statement stops the current function and passes the exception up to the caller.
- The `try { myFunc() } catch { case pattern1 => recoverExpr1 }` statement catches any exception thrown by `myFunc()` if the exception matches one of the patterns inside the `catch` block:
 - If an exception is thrown by `myFunc`, but no pattern matches the exception, the function stops, and the exception is passed up to the caller again. If there is no `try...catch` block in the call chain that can catch the exception, the whole program stops.
 - If an exception is thrown by `myFunc`, and the `pattern1` pattern matches the exception, the `try...catch` block will return the `recoverExpr1` expression at the right of the arrow.
 - If no exception is thrown, the `try...catch` block returns the result returned by `myFunc()`.

This mechanism comes from Java, and since the Scala SDK sits on top of the Java SDK, many function calls to the SDK can throw exceptions. If you are familiar with Java, the Scala exception mechanism differs slightly. Exceptions in Scala are always *unchecked*, which means that the compiler will never force you to catch an exception or declare that a function can throw an exception.

Throwing exceptions

The following is a code snippet that demonstrates how exceptions can be thrown. You can paste it in the Scala console or in a Scala worksheet:

```
case class Person(name: String, age: Int)
case class AgeNegativeException(message: String) extends Exception(message)

def createPerson(description: String): Person = {
    val split = description.split(" ")
    val age = split(1).toInt
    if (age < 0)
        throw AgeNegativeException(s"age: $age should be > 0")
    else
        Person(split(0), age)
```

The `createPerson` function creates the `Person` object if the string passed in an argument is correct, but throws different types of exceptions if it is not. In the preceding code, we also implemented our own `AgeNegativeException` instance, which is thrown if the age passed in the string is negative, as shown in the following code:

```
scala> createPerson("John 25")
res0: Person = Person(John, 25)

scala> createPerson("John25")
java.lang.ArrayIndexOutOfBoundsException: 1
  at .createPerson(<console>:17)
  ... 24 elided

scala> createPerson("John -25")
AgeNegativeException: age: -25 should be > 0
  at .createPerson(<console>:19)
  ... 24 elided

scala> createPerson("John 25.3")
java.lang.NumberFormatException: For input string: "25.3"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
  at scala.collection.immutable.StringLike.toInt(StringLike.scala:301)
  at scala.collection.immutable.StringLike.toInt$(StringLike.scala:301)
  at scala.collection.immutable.StringOps.toInt(StringOps.scala:29)
  at .createPerson(<console>:17)
  ... 24 elided
```

Since the exceptions are not caught by any `try...catch` block, the Scala console shows a **stack trace**. The stack trace shows all the nested function calls that led to the point where the exception was thrown. In the last example, the `val age =`

`split(1).toInt` line in `createPerson` called `scala.collection.immutable.StringOps.toInt`, which called `scala.collection.immutable.StringLike.toInt$`, and so on, until finally the `java.lang.Integer.parseInt` function threw the exception at line 580 in `Integer.java`.

Catching exceptions

In order to illustrate how exceptions bubble up the call stack, we are going to create a new `averageAge` function, which calculates the average age of a list of `Person` instances, using their string descriptions, as shown in the following code:

```
| def averageAge(descriptions: Vector[String]): Double = {  
|   val total = descriptions.map(createPerson).map(_.age).sum  
|   total / descriptions.length  
| }
```

This function calls our previously implemented `createPerson` function, and therefore will throw any exception that is thrown by `createPerson` because there is no `try...catch` block in `averageAge`.

Now, we can implement another function on top that will parse an input containing several `Person` descriptions and return a summary in a string. It will print an error message in case the input cannot be parsed, as shown in the following code:

```
| import scala.util.control.NonFatal  
| def personsSummary(personsInput: String): String = {  
|   val descriptions = personsInput.split("\n").toVector  
|   val avg = try {  
|     averageAge(descriptions)  
|   } catch {  
|     case e: AgeNegativeException =>  
|       println(s"one of the persons has a negative age: $e")  
|       0  
|     case NonFatal(e) =>  
|       println(s"something was wrong in the input: $e")  
|       0  
|   }  
|   s"${descriptions.length} persons with an average age of $avg"  
| }
```

In this function, we declare an `avg` value, which will get the return value of `averageAge` if no exception is thrown. If one of the descriptions contains a negative age, our catch block will print an error message, and `avg` will be assigned the value of `0`. If another type of exception is thrown, and this exception is `NonFatal`, then we print another message and `avg` will also be assigned the value of `0`. A fatal exception is an exception that cannot be recovered, such as `OutOfMemoryException`. You can look at the implementation of `scala.util.control.NonFatal` for more details.

The following code shows a few sample calls to `personsSummary`:

```
scala> personsSummary(  
    """John 25  
    |Sharleen 45""".stripMargin)  
res1: String = 2 persons with an average age of 35.0  
  
scala> personsSummary(  
    """John 25  
    |Sharleen45""".stripMargin)  
something was wrong in the input: java.lang.ArrayIndexOutOfBoundsException: 1  
res2: String = 2 persons with an average age of 0.0  
  
scala> personsSummary(  
    """John -25  
    |Sharleen 45""".stripMargin)  
one of the persons has a negative age: $line5.$read$$iw$$iw$AgeNegativeException: age st  
res3: String = 2 persons with an average age of 0.0
```

As we can see, as soon as any of the descriptions cannot be parsed, an error is printed and the average age is set to 0.

Using the finally block

A `try...catch` block can optionally be followed by a `finally {}` block. The code inside the `finally` block is always executed, even if an exception is not matched by any pattern in the `catch` block. The `finally` block is typically used to close any resource that is accessed inside the `try` block, such as a file or a network connection.

The following code shows how to use a URL to read a web page into a string:

```
import java.io.IOException
import java.net.URL
import scala.annotation.tailrec

val stream = new URL("https://www.packtpub.com/").openStream()
val htmlPage: String =
  try {
    @tailrec
    def loop(builder: StringBuilder): String = {
      val i = stream.read()
      if (i != -1)
        loop(builder.append(i.toChar))
      else
        builder.toString()
    }
    loop(StringBuilder.newBuilder)
  } catch {
    case e: IOException => s"cannot read URL: $e"
  }
  finally {
    stream.close()
  }
```

The `finally` block allows us to close `InputStream`, whether the reading of the page succeeded or not. This way, we will not leave a dangling open connection in case there is a network issue or a thread interruption.

Note that the preceding code is for illustrative purposes only. In a real project, you should use the following code format:

```
| val htmlPage2 = scala.io.Source.fromURL("https://www.packtpub.com/").mkString
```

Now that you know how to use exceptions, we are going to define the concept of referential transparency and show how catching exceptions can break it. We will then explore better data structures that will let us manage errors without breaking

referential transparency.

Ensuring referential transparency

We say that an expression is **referentially transparent** when it can be replaced by its value without changing the program's behavior, in any context. When an expression is a function call, it means that we can always substitute this function call with the return value of the function. A function that guarantees this in any context is called a **pure function**.

A pure function is like a mathematical function—the return value depends only on the arguments passed to the function. You do not have to consider anything else about the context in which it is called.

Defining pure functions

In the following code, the `puresquare` function is pure:

```
def pureSquare(x: Int) = x * x
val pureExpr = pureSquare(4) + pureSquare(3)
// pureExpr: Int = 25

val pureExpr2 = 16 + 9
// pureExpr2: Int = 25
```

The functions called `pureSquare(4)` and `pureSquare(3)` are referentially transparent—when we replace them with the return value of the function, the program's behavior does not change.

On the other hand, the following function is impure:

```
var globalState = 1
def impure(x: Int) = {
    globalState = globalState + x
    globalState
}
val impureExpr = impure(3)
// impureExpr: Int = 4
val impureExpr2 = 4
```

We cannot replace the call to `impure(3)` with its return value because the return value changes depending on the context. In fact, any function that has **side effects** is impure. A side effect can be any of the following constructs:

- Mutating a global variable
- Printing to the console
- Opening a network connection
- Reading/writing data to/from a file
- Reading/writing data to/from a database
- More generally, any interaction with the outside world

The following is another example of an impure function:

```
import scala.util.Random
def impureRand(): Int = Random.nextInt()
impureRand()
//res0: Int = -528134321
val impureExprRand = impureRand() + impureRand()
//impureExprRand: Int = 681209667
```

```
| val impureExprRand2 = -528134321 + -528134321
```

You cannot substitute the result of `impureRand()` with its value because the value changes for every call. If you do so, the program's behavior changes. The call to `impureRand()` is not referentially transparent, and hence `impureRand` is impure. In fact, the `random()` function mutates a global variable to generate a new random number for every call. We can also say that this function is **nondeterministic**—we cannot predict its return value just by observing its arguments.

We can rewrite our impure function to make it pure, as shown in the following code:

```
| def pureRand(seed: Int) = new Random(seed).nextInt()
| pureRand(10)
| //res1: Int = -1157793070
| val pureExprRand = pureRand(10) + pureRand(10)
| //pureExprRand: Int = 1979381156
| val pureExprRand2 = -1157793070 + -1157793070
| //pureExprRand2: Int = 1979381156
```

You can substitute the call to `pureRand(seed)` with its value; the function will always return the same value given the same seed. The call to `pureRand` is referentially transparent, and `pureRand` is a pure function.

The following is another example of an impure function:

```
| def impurePrint(): Unit = println("Hello impure")
| val impureExpr1: Unit = impurePrint()
| val impureExpr2: Unit = ()
```

In the second example, the return value of `impurePrint` is of the `Unit` type. There is only one value of this type in the Scala SDK: the `()` value. If we replace the call to `impurePrint()` with `()`, then the program's behavior changes—in the first case, something will be printed on the console, but not in the second case.

Best practices

Referential transparency is a key concept in functional programming. If most of your program uses pure functions, then it becomes much easier to do the following:

- **Understand what a program is doing:** You know that the function's return value only depends on its arguments. You do not have to think about what the state of this or that variable is in this or that context. You just have to look at the arguments.
- **Test your functions:** I will restate this point—the function's return value only depends on its argument. Testing it is very simple; you can try different argument values and confirm that the return value is what you expect.
- **Write multithreaded programs:** Since a pure function's behavior does not depend on a global state, you can execute it in parallel on multiple threads or even on different machines. The return values will not change. As our CPUs are built with more and more cores these days, this will help you write faster programs.

However, it is not possible to only have pure functions in a program because, in essence, a program must interact with the outside world. It has to print something, read some user input, or save some state in a database. In functional programming, the best practice is to use pure functions in the majority of the code base and to push the impure side-effecting functions to the boundaries of the program.

For instance, in [chapter 2](#), *Developing a Retirement Calculator*, we implemented a retirement calculator that mostly uses pure functions. One side-effecting function was the `println` call in the `SimulatePlanApp` object, which was at the boundaries of the program. There were other side effects in `EquityData.fromFile` and `InflationData.fromFile`; these functions are for reading files. However, the resource files can never change for the duration of the program. For a given filename, we would always get the same file content, and we could substitute the return value of `fromFile` in all calls without changing the program's behavior. In this case, the side effect of reading a file is not observable, and we can consider these file-

reading functions as being pure. Another side-effecting function was `strMain` because it could throw exceptions. In the rest of this chapter, we will see why throwing exceptions break referential transparency, and we will learn how to replace it with better functional programming structures.

An impure function fulfills the following two criteria:



- *It returns unit.*
- *It does not take any arguments but returns a type. Since it returns something that cannot be obtained by using its arguments, it must be using a global state.*

Note that a pure function can use mutable variables or side effects inside the body of the function. As long as these effects are *not observable* by the caller, we consider the function pure. In the Scala SDK, many pure functions are implemented using mutable variables to improve performance. Look, for the instance, at the following implementation of `TraversableOnce.foldLeft`:

```
| def foldLeft[B](z: B)(op: (B, A) => B): B = {  
|   var result = z  
|   this foreach (x => result = op(result, x))  
|   result  
| }
```

Showing how exceptions break referential transparency

This might not seem obvious, but when a function throws an exception, it breaks referential transparency. In this section, I am going to show you why. First, create a new Scala worksheet and type the following definitions:

```
case class Rectangle(width: Double, height: Double)

def area(r: Rectangle): Double =
  if (r.width > 5 || r.height > 5)
    throw new IllegalArgumentException("too big")
  else
    r.width * r.height
```

Then, we will call `area` with the following arguments:

```
val area1 = area(3, 2)
val area2 = area(4, 2)

val total = try {
  area1 + area2
} catch {
  case e: IllegalArgumentException => 0
}
```

We get `total: Double = 14.0`. In the preceding code, the `area1` and `area2` expressions are referentially transparent. We can indeed *substitute* them with their value without changing the program's behavior. In IntelliJ, select the `area1` variable inside the `try` block and hit `Ctrl + Alt + N` (inline variable), as shown in the following code. Do the same for `area2`:

```
val total = try {
  area(3, 2) + area(4, 2)
} catch {
  case e: IllegalArgumentException => 0
}
```

The `total` is the same as before. The program's behavior did not change, hence `area1` and `area2` are referentially transparent. However, let's see what happens if we define `area1` in the following way:

```
val area1 = area(6, 2)
val area2 = area(4, 2)
```

```
| val total = try {
|   area1 + area2
| } catch {
|   case e: IllegalArgumentException => 0
| }
```

In this case, we get `java.lang.IllegalArgumentException: too big`, because our `area(...)` function throws an exception when the width is greater than five. Now let's see what happens if we inline `area1` and `area2` as before, as shown in the following code:

```
| val total = try {
|   area(6, 2) + area(4, 2)
| } catch {
|   case e: IllegalArgumentException => 0
| }
```

In this case, we get `total: Double = 0.0`. The program's behavior changed when substituting `area1` with its value, hence `area1` is *not* referentially transparent.

We demonstrated that exception handling breaks referential transparency, and hence functions that throw exceptions are impure. It makes a program more difficult to understand because you have to take into account *where* a variable is defined to understand how the program will behave. The behavior will change depending on whether a variable is defined inside or outside a `try` block. This might not seem to be a big deal in a trivial example, but when there are multiple chained function calls with `try` blocks along the line, matching different types of exceptions, it can become daunting.

Another disadvantage when you use exceptions is that the *signature* of the function does not indicate that it can throw an exception. When you call a function that can throw exceptions, you have to look at its implementation to figure out what type of exception it can throw, and under what circumstances. If the function calls other functions, it compounds the problem. You can accommodate this by adding comments or an `@throws` annotation to indicate what exception types can be thrown, but these can become outdated when the code is refactored. When we call a function, we should only have to consider its signature. A signature is a bit like a contract—given these arguments, I will return you a result. If you have to look at the implementation to know what exceptions are thrown, it means that the contract is not completed: some information is hidden.

We now know how to throw and catch exceptions, and why we should use them with caution. The best practice is to do the following:

- Catch recoverable exceptions as early as possible, and indicate the possibility of failure with a specific return type.
- Not catch exceptions that cannot be recovered, such as disk full, out of memory, or some other catastrophic failure. This will make your program crash whenever such exceptions happen, and you should then have a manual or automatic recovery process outside the program.

In the rest of this chapter, I will show you how to use the `Option`, `Either`, and `Validated` classes to model the possibility of a failure.

Using Option

The Scala `option` type is an **algebraic data type (ADT)** that represents an optional value. It can also be viewed as `List` that can contain either one or no elements. It is a safe replacement for a `null` reference that you might have used if you have programmed in Java, C++, or C#.

Manipulating instances of Option

The following is a simplified definition of the `option` ADT:

```
sealed trait Option[+A]
case class Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

The Scala SDK provides a more refined implementation; the preceding definition is just for illustrative purpose. This definition implies that `option` can be either of the following two types:

- `Some(value)`, which represents an optional value where the value is present
- `None`, which represents an optional value where the value is not present.

The + sign in front of the `A` type parameter in the `option[+A]` declaration means that `option` is covariant in `A`. We will explore contravariance in more details in chapter 4, Advanced Features. For now, you just have to know that if `B` is a subtype of `A`, then `option[B]` is a subtype of `option[A]`. Furthermore, you might notice that `None` actually extends `option[Nothing]` and not `option[A]`. This is because a case object cannot accept a type parameter. In Scala, `Nothing` is the bottom type, which means that it is a subtype of any other type. This implies that `None` is a subtype of `option[A]` for any `A`.

The following are some examples of the usage of the different types of `option` that you can paste in a Scala worksheet:

```
val opt0: Option[Int] = None
// opt0: Option[Int] = None

val opt1: Option[Int] = Some(1)
// opt1: Option[Int] = Some(1)

val list0 = List.empty[String]
list0.headOption
// res0: Option[String] = None
list0.lastOption
// res1: Option[String] = None

val list3 = List("Hello", "World")
list3.headOption
// res2: Option[String] = Some(Hello)
list3.lastOption
// res3: Option[String] = Some(World)
```

Explanation of the preceding code is as follows:

- The first two examples show how we can define an `option` type that can

optionally contain `Int`.

- The following examples use the `headOption` and `lastOption` methods in `List` to show that many safe functions of the SDK return `Option`. If `List` is empty, these functions always return `None`. Note that the SDK also provides *unsafe* equivalent `head` and `last` methods. The unsafe methods throw an exception if we call them with an empty `List`, which might crash our program if we do not catch the exception.



Many functions of the SDK provide equivalent safe (which return `option`) and unsafe functions (which throw an exception). It is a best practice to always use the safe alternative.

Since `Option` is an ADT, we can use pattern matching to test whether `option` is `None` or `Some`, as shown in the following code:

```
def personDescription(name: String, db: Map[String, Int]): String =  
  db.get(name) match {  
    case Some(age) => s"$name is $age years old"  
    case None => s"$name is not present in db"  
  }  
  
val db = Map("John" -> 25, "Rob" -> 40)  
personDescription("John", db)  
// res4: String = John is 25 years old  
personDescription("Michael", db)  
// res5: String = Michael is not present in db
```

The `get(key)` method in `Map` returns `Option`, containing the value associated with the key. If the key does not exist in `Map`, it returns `None`. When you start using `Option`, pattern matching is the most natural way of triggering different behaviors depending on the content of `Option`.

Another way is to use `map` and `getOrElse`, as shown in the following code:

```
def personDesc(name: String, db: Map[String, Int]): String = {  
  val optString: Option[String] = db.get(name).map(age => s"$name is  
$age years old")  
  optString.getOrElse(s"$name is not present in db")  
}
```

We saw earlier how to transform the elements of a vector using `map`. This works exactly the same for `Option`—we pass an anonymous function that will be called with the option's value if `Option` is not empty. Since our anonymous function returns a string, we obtain `Option[String]`. We then call `getOrElse`, which provides a value in case `Option` is `None`. The `getOrElse` phrase is a good way to safely extract the content of `Option`.



Never use the `.get` method on `option`—always use `.getorElse`. The `.get` method throws an exception if `option` is `None`, and hence is not safe.

Composing transformations with for... yield

Using the same `db: Map[String, Int]` phrase, containing the ages of different people, the following code is a naive implementation of a function that returns the average age of two people:

```
def averageAgeA(name1: String, name2: String, db: Map[String, Int]): Option[Double] = {
    val optOptAvg: Option[Option[Double]] =
        db.get(name1).map(age1 =>
            db.get(name2).map(age2 =>
                (age1 + age2).toDouble / 2))
    optOptAvg.flatten
}
val db = Map("John" -> 25, "Rob" -> 40)
averageAge("John", "Rob", db)
// res6: Option[Double] = Some(32.5)
averageAge("John", "Michael", db)
// res7: Option[Double] = None
```

The function returns `Option[Double]`. If `name1` or `name2` cannot be found in the `db` map, `averageAge` returns `None`. If both names are found, it returns `Some(value)`. The implementation uses `map` to transform the value contained in the option. We end up with a nested `Option[Option[Double]]`, but our function must return `Option[Double]`. Fortunately, we can use `flatten` to remove one level of nesting.

We managed to implement `averageAge`, but we can improve it using `flatMap`, as shown in the following code:

```
def averageAgeB(name1: String, name2: String, db: Map[String, Int]): Option[Double] =
    db.get(name1).flatMap(age1 =>
        db.get(name2).map(age2 =>
            (age1 + age2).toDouble / 2))
```

As its name suggests, `flatMap` is equivalent to composing `flatten` and `map`. In our function, we replaced `map(...).flatten` with `flatMap(...)`.

So far, so good, but what if we want to get the average age of three or four people? We would have to nest several instances of `flatMap`, which would not be very pretty or readable. Fortunately, Scala provides a syntactic sugar that allows us to simplify our function further, called the `for` comprehension, as shown in the

following code:

```
def averageAgeC(name1: String, name2: String, db: Map[String, Int]): Option[Double] =  
  for {  
    age1 <- db.get(name1)  
    age2 <- db.get(name2)  
  } yield (age1 + age2).toDouble / 2
```

When you compile a `for` comprehension, such as `for { ... } yield { ... }`, the Scala compiler transforms it into a composition of `flatMap/map` operations. Here is how it works:

- Inside the `for` block, there can be one or many expressions phrased as `variable <- context`, which is called a **generator**. The left side of the arrow is the name of a variable that is bound to the content of the **context** on the right of the arrow.
- Every generator except the last one is transformed into a `flatMap` expression.
- The last generator is transformed into a `map` expression.
- All context expressions (the right side of the arrow) must have the same context type.

In the preceding example, we used `Option` for the context type, but `for yield` can also be used with any class that has a `flatMap` and `map` operation. For instance, we can use `for..yield` with `Vector` to run nested loops, as shown in the following code:

```
for {  
  i <- Vector("one", "two")  
  j <- Vector(1, 2, 3)  
} yield (i, j)  
// res8: scala.collection.immutable.Vector[(String, Int)] =  
// Vector((one,1), (one,2), (one,3), (two,1), (two,2), (two,3))
```

 *Syntactic sugar* is syntax within a programming language that makes it easier to read or write. It makes it sweeter for the programmer.

Refactoring the retirement calculator to use Option

Now that we know what `option` can do for us, we are going to refactor one of the functions of the retirement calculator that we developed in [Chapter 2, Developing a Retirement Calculator](#), to improve the handling of some edge-case scenarios. If you have not done it yet, please follow the instructions at the beginning of the [Chapter 2, Developing a Retirement Calculator](#), to set up the project.

In `RetCalc.scala`, we are going to change the return type of `nbMonthsSaving`. In [Chapter 2, Developing a Retirement Calculator](#), we returned `Int.MaxValue` if `netIncome <= currentExpense` to avoid looping infinitely. This was not very robust, as this infinite result could then be used in another computation, which would lead to bogus results. It would be better to return `Option[Int]` to indicate that the function might not be computable and let the caller decide what to do. We would return `None` if it was not computable or `Some(returnValue)` if it was computable.

The following code is the new implementation for `nbMonthsSaving`, with the changed portions highlighted in bold:

```
def nbOfMonthsSaving(params: RetCalcParams,  
                     returns: Returns): Option[Int] = {  
    import params._  
    @tailrec  
    def loop(months: Int): Int = {  
        val (capitalAtRetirement, capitalAfterDeath) =  
            simulatePlan(returns, params, months)  
  
        if (capitalAfterDeath > 0.0)  
            months  
        else  
            loop(months + 1)  
    }  
  
    if (netIncome > currentExpenses)  
        Some(loop(0))  
    else  
        None  
}
```

Now try to compile the project. This change breaks many parts of our project, but the Scala compiler is a terrific assistant. It will help us identify the portions

of the code we need to change to make our code more robust.

The first error is in `RetCalcspec.scala`, as shown in the following code:

```
| Error:(65, 14) types Option[Int] and Int do not adhere to the type constraint selected f
|   actual should ===(expected)
```

This error means that the types in the `actual should === (expected)` expression do not match: `actual` is of the `option[Int]` type, whereas `expected` is of the `Int` type. We need to change the assertion, as follows:

```
| actual should ===(Some(expected))
```

You can apply the same fix for the second unit test. For the last unit test, we want to assert that `None` is returned instead of `Int.MaxValue`, as shown in the following code:

```
| "not loop forever if I enter bad parameters" in {
|   val actual = RetCalc.nbOfMonthsSaving(params.copy(netIncome = 1000), FixedReturns(0.04))
|   actual should ===(None)
| }
```

You can now compile and run the test. It should pass.

You are now able to model an optional value safely. However, sometimes it is not always obvious to know what `None` actually means. Why did this function return `None`? Was it because the arguments that were passed were wrong? Which argument was wrong? And what value would be correct? It would indeed be nice to have some explanation that comes along with the `None` in order to understand *why* there was no value. In the next section, we are going to use the `Either` type for this purpose.

Using Either

The `Either` type is an ADT that represents a value of either a `Left` type or a `Right` type. A simplified definition of `Either` would be the following:

```
sealed trait Either[A, B]
case class Left[A, B](value: A) extends Either[A, B]
case class Right[A, B](value: B) extends Either[A, B]
```

When you instantiate a `Right` type, you need to provide a value of a `B` type, and when you instantiate a `Left` type, you need to provide a value of an `A` type. Therefore, `Either[A, B]` can either hold a value of type `A` or a value of type `B`.

The following code shows an example of such a usage that you can type in a new Scala worksheet:

```
def divide(x: Double, y: Double): Either[String, Double] =
  if (y == 0)
    Left(s"$x cannot be divided by zero")
  else
    Right(x / y)

divide(6, 3)
// res0: Either[String,Double] = Right(2.0)
divide(6, 0)
// res1: Either[String,Double] = Left(6.0 cannot be divided by zero)
```

The `divide` function returns either a string or a double:

- If the function cannot compute a value, it returns an error `String` wrapped in a `Left` type
- If the function can compute a correct value, it returns the `Double` value wrapped in a `Right` type

By convention, we use `Right` to represent the correct or right value, and we use `Left` to represent an error.

Manipulating Either

Since `Either` is an ADT, we can use pattern matching to decide what to do when we get a `Left` or `Right` type.

The following is a modified version of the `personDescription` function that we showed earlier in the *Using Option* section:

```
def getPersonAge(name: String, db: Map[String, Int]): Either[String, Int] =  
  db.get(name).toRight(s"$name is not present in db")  
  
def personDescription(name: String, db: Map[String, Int]): String =  
  getPersonAge(name, db) match {  
    case Right(age) => s"$name is $age years old"  
    case Left(error) => error  
  }  
  
val db = Map("John" -> 25, "Rob" -> 40)  
personDescription("John", db)  
// res4: String = John is 25 years old  
personDescription("Michael", db)  
// res5: String = Michael is not present in db
```

The first `getPersonAge` function produces `Right(age)` if the `name` argument is present in `db`. If `name` is not present in `db`, it returns an error message wrapped in a `Left` type. For this purpose, we use the `Option.toRight` method. I encourage you to have a look at the documentation and implementation of this method.

The implementation of `personDescription` is straightforward—we pattern match using the result of `getPersonAge` and return an appropriate `String` depending on whether the result is a `Left` or `Right` type.

As with `option`, we can also use `map` and `flatMap` to combine several instances of `Either`, as shown in the following code:

```
def averageAge(name1: String, name2: String, db: Map[String, Int]): Either[String, Double] =  
  getPersonAge(name1, db).flatMap(age1 =>  
    getPersonAge(name2, db).map(age2 =>  
      (age1 + age2).toDouble / 2))  
  
averageAge("John", "Rob", db)  
// res4: Either[String,Double] = Right(32.5)  
averageAge("John", "Michael", db)  
// res5: Either[String,Double] = Left(Michael is not present in db)
```

Note how the body of the function is almost the same as `option`. This is because

`Either` is **right biased**, meaning that `map` and `flatMap` transform the right side of `Either`.

If you want to transform the `Left` side of `Either`, you need to call the `Either.left` method, as shown in the following code:

```
| getPersonAge("bob", db).left.map(err => s"The error was: $err")
| // res6: scala.util.Either[String,Int] = Left(The error was: bob is not present in db)
```

Since `Either` implements `map` and `flatMap`, we can refactor `averageAge` to use a `for` comprehension, as shown in the following code:

```
def averageAge2(name1: String, name2: String, db: Map[String, Int]): Either[String, Double] = {
  for {
    age1 <- getPersonAge(name1, db)
    age2 <- getPersonAge(name2, db)
  } yield (age1 + age2).toDouble / 2
```

Again, the code looks the same as it did with `option`.

Refactoring the retirement calculator to use Either

Now that we have a good understanding of how to manipulate `Either`, we are going to refactor our retirement calculator to take advantage of it.

Refactoring nbOfMonthsSavings

In the previous section, we changed the return type of `nbOfMonthsSavings` to return `option[Int]`. The function returned `None` if the `expenses` arguments were greater than `income`. We are now going to change it to return an error message wrapped in `Left`.

We could use a simple string for our error message, but the best practice when using `Either` is to create an ADT for all of the possible error messages. Create a new Scala class in `src/main/scala/retcalc` called `RetCalcError`, as shown in the following code:

```
package retcalc

sealed abstract class RetCalcError(val message: String)

object RetCalcError {
  case class MoreExpensesThanIncome(income: Double, expenses: Double)
    extends RetCalcError(
      s"Expenses: $expenses >= $income. You will never be able to save
        enough to retire !")
}
```

We define a `RetCalcError` trait that has only one `message` method. This method will produce the error message whenever we need to return it to the user. Inside the `RetCalcError` object, we define one case class per type of error message. We will then change the functions that need to return an error to return `Either[RetCalcError, A]`.

This pattern has many advantages over just using `String`, as shown in the following list:

- All of the error messages are located in one place. It allows you to immediately know what are all the possible error messages that can be returned to the user. You could also add different translations if your application uses multiple languages.
- Since `RetCalcError` is an ADT, you can use pattern matching to recover from a specific error and take action.
- It simplifies testing. You can test whether a function returns a specific type of error without having to assert on the error message itself. This way, you can change your error messages without having to change any test.

Now we can refactor our `RetCalc.nbOfMonthsSavings` function to return `Either[RetCalcError, Int]`, as shown in the following code:

```
| def nbOfMonthsSaving(params: RetCalcParams,  
|   returns: Returns): Either[RetCalcError, Int] = {  
|   import params._  
|   @tailrec  
|   def loop(months: Int): Int = {  
|     val (capitalAtRetirement, capitalAfterDeath) =  
|       simulatePlan(returns, params, months)  
|     if (capitalAfterDeath > 0.0)  
|       months  
|     else  
|       loop(months + 1)  
|   }  
|  
|   if (netIncome > currentExpenses)  
|     Right(loop(0))  
|   else  
|     Left(MoreExpensesThanIncome(netIncome, currentExpenses))  
| }
```

We also have to change the corresponding unit tests. ScalaTest provides convenient extensions to perform assertions on the `Either` type. To bring them in scope, extend `EitherValues` in `RetCalcSpec.scala`, as shown in the following code:

```
| class RetCalcSpec extends WordSpec with Matchers with TypeCheckedTripleEquals  
|   with EitherValues {
```

If you have a `myEither` variable of the `Either[A, B]` type in your test, then `EitherValues` will let us use the following methods:

- `myEither.left.value` returns the left value of type `A` or fails the test if `myEither` is `Right`
- `myEither.right.value` returns the right value of type `B` or fails the test if `myEither` is `Left`

We can now change our unit tests on `nbOfMonthsSaving` as follows:

```
"RetCalc.nbOfMonthsSaving" should {  
  "calculate how long I need to save before I can retire" in {  
    val actual = RetCalc.nbOfMonthsSaving(params,  
    FixedReturns(0.04)).right.value  
    val expected = 23 * 12 + 1  
    actual should ===(expected)  
  }  
  
  "not crash if the resulting nbOfMonths is very high" in {  
    val actual = RetCalc.nbOfMonthsSaving(  
      params = RetCalcParams(  
        nbOfMonthsInRetirement = 40 * 12,  
        netIncome = 3000, currentExpenses = 2999, initialCapital = 0),  
      returns = FixedReturns(0.01)).right.value
```

```
    val expected = 8280
    actual should ===(expected)
}

"not loop forever if I enter bad parameters" in {
    val actual = RetCalc.nbOfMonthsSaving(
        params.copy(netIncome = 1000), FixedReturns(0.04)).left.value
    actual should ===(RetCalcError.MoreExpensesThanIncome(1000, 2000))
}
```

Run the unit test. It should pass.

Refactoring monthlyRate

In [Chapter 2](#), *Developing a Retirement Calculator*, we implemented a `Returns.monthlyRate(returns: Returns, month: Int): Double` function that returned the monthly return rate for a given month. When we called it with a month exceeding the size of an instance of `VariableReturns`, we rolled over to the first month using a modulo operation.

This was not completely satisfying, as it could compute unrealistic simulations. Say that your `VariableReturns` instance contains data for 1950 to 2017. When you ask the monthly returns for 2018, `monthlyRate` would give you the returns that we had in 1950. The economic outlook of the fifties was very different compared to the current one, and it is unlikely that the returns in 2018 will mirror those in 1950.

Therefore, we are going to change `monthlyRate` to return an error if the `month` argument is outside the bounds of `VariableReturn`. First, open `RetCalcError.scala` and add the following error type:

```
| case class ReturnMonthOutOfBoundsException(month: Int, maximum: Int) extends RetCalcError(  
|   s"Cannot get the return for month $month. Accepted range: 0 to $maximum")
```

Next, we are going to change the unit tests to specify the function that we expect it to return. Open `ReturnsSpec.scala` and change the tests as follows:

```
"Returns.monthlyReturn" should {  
  "return a fixed rate for a FixedReturn" in {  
    Returns.monthlyRate(FixedReturns(0.04), 0).right.value should ===  
    (0.04 / 12)  
    Returns.monthlyRate(FixedReturns(0.04), 10).right.value should ===  
    (0.04 / 12)  
  }  
  
  val variableReturns = VariableReturns(  
    Vector(VariableReturn("2000.01", 0.1), VariableReturn("2000.02",  
      0.2)))  
  "return the nth rate for VariableReturn" in {  
    Returns.monthlyRate(variableReturns, 0).right.value should ===(0.1)  
    Returns.monthlyRate(variableReturns, 1).right.value should ===(0.2)  
  }  
  
  "return None if n > length" in {  
    Returns.monthlyRate(variableReturns, 2).left.value should ===(  
      RetCalcError.ReturnMonthOutOfBoundsException(2, 1))  
    Returns.monthlyRate(variableReturns, 3).left.value should ===(
```

```

        RetCalcError.ReturnMonthOutOfBounds(3, 1))
    }

    "return the n+offset th rate for OffsetReturn" in {
      val returns = OffsetReturns(variableReturns, 1)
      Returns.monthlyRate(returns, 0).right.value should ===(0.2)
    }
}

```

Then, open `Returns.scala` and change `monthlyRate` As follows:

```

def monthlyRate(returns: Returns, month: Int): Either[RetCalcError, Double] = returns match
  case FixedReturns(r) => Right(r / 12)

  case VariableReturns(rs) =>
    if (rs.isDefinedAt(month))
      Right(rs(month).monthlyRate)
    else
      Left(RetCalcError.ReturnMonthOutOfBounds(month, rs.size - 1))

  case OffsetReturns(rs, offset) => monthlyRate(rs, month + offset)
}

```

Try to compile the project now. Since `monthlyRate` is called by other functions, we will get some compilation errors, which is actually a good thing. We just have to fix the compilation errors to make our code deal with the possibility of an error. Every fix requires some thinking about what to do with that possibility.

On the other hand, if we had thrown an exception instead of returning `Either`, everything would have compiled, but the program would have crashed whenever the month was out of bounds. It would have been more difficult to achieve the desired behavior because the compiler would not have helped us.

The first compilation error is in `RetCalc.scala` in `futureCapital`, as shown in the following code:

```

Error:(55, 26) overloaded method value +
(...)

  cannot be applied to (Either[retcalc.RetCalcError,Double])
  accumulated * (1 + Returns.monthlyRate(returns, month)) +
  monthlySavings

```

This means that we cannot call the `+` method on `Either[RetCalcError, Double]`. If `monthlyRate` returns `Left`, we cannot calculate the accumulated capital. The best course of action would be to stop here and return the error. For this, we need to change the return type of `futureCapital` to `Either[RetCalcError, Double]` as well.

The following is the corrected version of the function:

```

def futureCapital(returns: Returns, nbOfMonths: Int, netIncome: Int, currentExpenses: Int,
                  initialCapital: Double): Either[RetCalcError, Double] = {
    val monthlySavings = netIncome - currentExpenses
    (0 until nbOfMonths).foldLeft[Either[RetCalcError, Double]](
      Right(initialCapital)) {
        case (accumulated, month) =>
        for {
          acc <- accumulated
          monthlyRate <- Returns.monthlyRate(returns, month)
        } yield acc * (1 + monthlyRate) + monthlySavings
    }
}

```

In the second line, we changed the initial element passed to `foldLeft`. We are now accumulating `Either[RetCalcError, Double]`. Note that we have to explicitly specify the type parameter for `foldLeft`. In the previous version of the function, when we were using `Double`, that type was inferred automatically.

If we do not specify the type parameter, the compiler will infer it to be of the type of the initial element. In our case, `Right(initialCapital)` is of the `Right[Nothing, Double]` type, which is a subclass of `Either[RetCalcError, Double]`. The problem is that, inside the anonymous function, we return `Either[RetCalcError, Double]`, not `Right[Nothing, Double]`. The compiler would complain that the types do not match.

Inside the anonymous function passed to `foldLeft`, we use a `for` comprehension to do the following:

- Return the accumulated result in `Right` if both `acc` and `monthlyRate` are `Right`
- Return `Left` if `acc` or `monthlyRate` is `Left`

Note that our implementation does not stop as soon as `monthlyRate` returns `Left`, which is a bit inefficient. There is no point in iterating through the other months when we get an error because this function should always return the first error it encounters. In [Chapter 4, Advanced Features](#), we will see how we can use lazy evaluation with `foldr` to stop the iteration early.

Compile the project again. Now we need to fix a compilation error in `simulatePlan`.

Refactoring simulatePlan

Since `simulatePlan` calls `futureCapital`, we need to change its implementation to take into account the new return type, as shown in the following code:

```
def simulatePlan(returns: Returns, params: RetCalcParams, nbOfMonthsSavings: Int,
                 monthOffset: Int = 0): Either[RetCalcError, (Double,
                                                       Double)] = {
    import params._

    for {
        capitalAtRetirement <- futureCapital(
            returns = OffsetReturns(returns, monthOffset),
            nbOfMonths = nbOfMonthsSavings, netIncome = netIncome,
            currentExpenses = currentExpenses,
            initialCapital = initialCapital)

        capitalAfterDeath <- futureCapital(
            returns = OffsetReturns(returns, monthOffset +
                nbOfMonthsSavings),
            nbOfMonths = nbOfMonthsInRetirement,
            netIncome = 0, currentExpenses = currentExpenses,
            initialCapital = capitalAtRetirement)
    } yield (capitalAtRetirement, capitalAfterDeath)
}
```

We moved the two calls to `futureCapital` inside a `for` comprehension. This way, if any of these calls return an error, `simulatePlan` will return it. If both calls succeed, `simulatePlan` will return the two double values in a tuple.

Compile the project. Now we need to fix a compilation error in `nbOfMonthsSaving`, which uses `simulatePlan`. The following code is the fixed version:

```
def nbOfMonthsSaving(params: RetCalcParams, returns: Returns): Either[RetCalcError, Int] =
    import params._
    @tailrec
    def loop(months: Int): Either[RetCalcError, Int] = {
        simulatePlan(returns, params, months) match {
            case Right((capitalAtRetirement, capitalAfterDeath)) =>
                if (capitalAfterDeath > 0.0)
                    Right(months)
                else
                    loop(months + 1)

            case Left(err) => Left(err)
        }
    }

    if (netIncome > currentExpenses)
        loop(0)
    else
        Left(MoreExpensesThanIncome(netIncome, currentExpenses))
```

```
| }
```

We had to change our recursive `loop` function to return `Either[RetCalcError, Int]`. The loop will stop as soon as we get an error or `if (capitalAfterDeath > 0.0)`. You might wonder why we did not use `flatMap` instead of using pattern matching. It would indeed have been more concise, but the `loop` function would not be tail recursive anymore, because the recursive call to `loop` would be inside an anonymous function. As an exercise, I encourage you to try changing the code so that it uses `flatMap` and observe the tail recursion compilation error.

Compile the project. The last compilation error in the production code is in `SimulatePlanApp.scala`.

Refactoring SimulatePlanApp

The entry point of our `simulatePlanApp` application calls `simulatePlan`. We need to change it to return the text of any error that could occur.

First, we need to change the integration test to add a new test case. Open `SimulatePlanIT.scala` and add the following test:

```
"SimulatePlanApp.strMain" should {
    "simulate a retirement plan using market returns" in {...}

    "return an error when the period exceeds the returns bounds" in {
        val actualResult = SimulatePlanApp.strMain(
            Array("1952.09,2017.09", "25", "60", "3000", "2000", "10000"))
        val expectedResult = "Cannot get the return for month 780.
Accepted range: 0 to 779"
        actualResult should === (expectedResult)
    }
}
```

Then, open `SimulatePlanApp.scala` and change the implementation of `SimulatePlanApp` as follows:

```
object SimulatePlanApp extends App {
    println(strMain(args))

    def strMain(args: Array[String]): String = {
        val (from +: until +: Nil) = args(0).split(",").toList
        val nbOfYearsSaving = args(1).toInt
        val nbOfYearsRetired = args(2).toInt

        val allReturns = Returns.fromEquityAndInflationData(
            equities = EquityData.fromResource("sp500.tsv"),
            inflations = InflationData.fromResource("cpi.tsv"))

        RetCalc.simulatePlan(
            returns = allReturns.fromUntil(from, until),
            params = RetCalcParams(
                nbOfMonthsInRetirement = nbOfYearsRetired * 12,
                netIncome = args(3).toInt,
                currentExpenses = args(4).toInt,
                initialCapital = args(5).toInt,
                nbOfMonthsSavings = nbOfYearsSaving * 12
            ) match {
                case Right((capitalAtRetirement, capitalAfterDeath)) =>
                    s"""
                        |Capital after $nbOfYearsSaving years of savings:
                        |${capitalAtRetirement.round}
                        |Capital after $nbOfYearsRetired years in retirement:
                        |${capitalAfterDeath.round}
                    """.stripMargin
                case Left(err) => err.message
            }
        )
    }
}
```

```
| } }
```

We just have to pattern match on the result of `simulatePlan`, and return a string explaining the result of the computation if it is a `Right` value, or return the error message if it is a `Left` value.

Compile the project. Now all the production code should compile, but there are still several compilation errors in the unit tests. As an exercise, I encourage you to try to fix them. In most cases, you have to make the test extend `EitherValues`, and call `.right.value` on an `Either` class to get its right value. Once you have fixed the remaining errors, compile and run all the tests of the project. They should all pass.

Now your code should look like the `Chapter03` branch in the Scala fundamentals GitHub project, apart from the `SimulatePlanApp` class, which we will improve in the next section. See <https://github.com/PacktPublishing/Scala-Programming-Projects> for more details.

Using ValidatedNel

In this chapter, we have seen how we can model the possibility of an optional value with `option` and the possibility of an error with `Either`. We demonstrated how these types can replace exceptions while guaranteeing referential transparency.

We also saw how we can combine several `option` or `Either` types using `flatMap`. This works well when we have to check for optional values or errors *sequentially*—call `function1`; if there is no error, call `function2`; if there is no error, call `function3`. If any of these functions return an error, we return that error and stop the call chain, as shown in the following code:

```
def sequentialErrorHandler(x: String): Either[MyError, String] =  
  for {  
    a <- function1(x)  
    b <- function2(a)  
    c <- function3(b)  
  } yield c
```

However, in some situations, we would want to call several functions in parallel and return all the errors that might have occurred. For instance, when you enter some personal details to purchase a product from an online shop, you expect the website to highlight all the mistakes in all the fields after you submit the details. It would be a bad user experience to have the website tell you that the last name is mandatory after submitting the details, and subsequently that your password is too short after submitting the details again. All the fields must be validated simultaneously and all errors must be returned to the user in one go.

The data structure that can help us address this use case is `validated`. Unfortunately, it is not part of the Scala SDK, and we have to use an external library called `cats` to bring it into our project.

Adding the cats dependency

The `cats` library provides abstractions for functional programming. Its name comes from the shortening of the phrase *category theory*. It is also a reference to the famous joke that managing developers is like herding cats—the truth is that you don't really have control—cats do whatever they want.

In this chapter, we are going to focus only on `validated` and `NonEmptyList`, but `cats` offers many more powerful abstractions that we will explore later in this book.

First, edit `built.sbt` and add the following lines:

```
| libraryDependencies += "org.typelevel" %% "cats-core" % "1.0.1"  
| scalacOptions += "-Ypartial-unification"
```

This brings the `cats` dependency to our project and also enables a compiler flag (`partial unification`) that is required by the library to infer types correctly.

Save the project with *Ctrl + S*. IntelliJ should offer to update the project to reflect the changes in the `build` file. Click on Refresh Project on top of `built.sbt`.

Introducing NonEmptyList

As its name implies, the `cats.data.NonEmptyList` type represents a `List` instance that has at least one element. In other words, it is a `List` instance that cannot be empty. The following are some examples of this usage that you can retype in a new Scala worksheet:

```
import cats.data.NonEmptyList

NonEmptyList(1, List(2, 3))
// res0: cats.data.NonEmptyList[Int] = NonEmptyList(1, 2, 3)
NonEmptyList.fromList(List(1, 2, 3))
// res3: Option[cats.data.NonEmptyList[Int]] = Some(NonEmptyList(1, 2, 3))
NonEmptyList.fromList(List.empty[Int])
// res4: Option[cats.data.NonEmptyList[Int]] = None
val nel = NonEmptyList.of(1, 2, 3)
// nel: cats.data.NonEmptyList[Int] = NonEmptyList(1, 2, 3)

nel.head
// res0: Int = 1
nel.tail
// res1: List[Int] = List(2, 3)
nel.map(_ + 1)
// res2: cats.data.NonEmptyList[Int] = NonEmptyList(2, 3, 4)
```

You can construct `NonEmptyList` with the following:

- `apply[A]`: You can pass a `head` element and a `List` as a tail.
- `fromList[A]`: You can pass `List`. You get back `Option[NonEmptyList[A]]` that will be `None` if the `List` argument is empty.
- `of[A]`: You can pass a `head` element and variable length `List` argument for the tail. This is the most convenient way of building `NonEmptyList` when you know its constituents.

Since `NonEmptyList` always contains at least one element, we can always call the `head` method without risking getting an exception. As a consequence, there is no `headOption` method. You can manipulate `NonEmptyList` with all the usual methods that you would use on `List`: `map`, `tail`, `flatMap`, `filter`, and `foldLeft`, to name a few.

Introducing Validated

The `cats.data.Validated[E, A]` type is very similar to `Either[E, A]`. It is an ADT that represents a value of either an `Invalid` type or a `Valid` type. A simplified definition would be the following:

```
sealed trait Validated[+E, +A]
case class Valid[+A](a: A) extends Validated[Nothing, A]
case class Invalid[+E](e: E) extends Validated[E, Nothing]
```

We will see what the `+` sign in front of a type parameter means in the section on covariance and contravariance in [Chapter 4, Advanced Features](#). For now, though, do not worry about it.



Similarly to the definition of `option`, the definitions use contravariance and `Nothing`. This way, `Valid[A]` is a subtype of `Validated[E, A]` for any `E`; and `Invalid[E]` is a subtype of `Validated[E, A]` for any `A`.

The main difference with `Either` is that we can accumulate the errors produced by several `Validated` instances. Here are some examples that you can retype in a new Scala worksheet. I advise you to uncheck the Type-aware highlighting box in the bottom-right corner of IntelliJ; otherwise, IntelliJ will underline some expressions in red, even though they compile fine:

```
import cats.data._
import cats.data.Validated._
import cats.implicits._

val valid1: Validated[NonEmptyList[String], Int] = Valid(1)
// valid1: cats.data.Validated[cats.data.NonEmptyList[String], Int] = Valid(1)

val valid2 = 2.validNel[String]
// valid2: cats.data.ValidatedNel[String, Int] = Valid(2)

(valid1, valid2).mapN { case (i1, i2) => i1 + i2 }
// res1: cats.data.ValidatedNel[String, Int] = Valid(3)

val invalid3: ValidatedNel[String, Int] = Invalid(NonEmptyList.of("error"))
val invalid4 = "another error".invalidNel[Int]
(valid1, valid2, invalid3, invalid4).mapN { case (i1, i2, i3, i4) => i1 + i2 + i3 + i4 }
// res2: cats.data.ValidatedNel[String, Int] = Invalid(NonEmptyList(error, another error))
```

We first define a `valid` value of `1`, with a `valid` type `Int` parameter, and an `Invalid` type `NonEmptyList[String]` parameter. Each error will be of a `String` type, and the `NonEmptyList` instance will force us to have at least one error when we produce an

`Invalid` value. This usage is so common that `cats` provide a type alias called `ValidatedNel` in the `cats.data` package, as shown in the following code :

```
| type ValidatedNel[+E, +A] = Validated[NonEmptyList[E], A]
```

Going back to our example, in the second line, we define a `valid` value of `2` using a handy `cats` method called `.validNel`. When calling `validNel`, we have to pass the type of error, because in this case, the compiler does not have any information to infer it. In our case, the error type is `String`. The resulting type of `valid2` is `ValidatedNel[String, Int]`, which is an alias for `Validated[NonEmptyList[String], Int]`.

In the third line, we *compose* the two valid values by putting them in a tuple and call `mapN`. The `mapN` phrase accepts an `f` function that takes as many arguments as there are elements in the tuple. If *all* of the elements of the tuple are `valid` values, `f` is called and its result will be wrapped in a `valid` value. If *any* of the elements inside the tuple are `invalid` values, then all the `invalid` values are *combined* together and wrapped in an `invalid` value.

We can observe that, when we compose `valid1` and `valid2`, which are all `valid`, `mapN` returns a `valid` value. When we compose `valid1`, `valid2`, `invalid3`, and `invalid4`, `mapN` returns an `invalid` value. This `invalid` value wraps a `NonEmptyList` that contains the errors of `invalid3` and `invalid4`.

We now know two mechanisms to represent the possibility of a failure:

- Either with `for...yield` can be used to validate *sequentially*, stopping at the first error encountered
- `validated` with `mapN` can be used to validate in parallel, accumulating all the errors in `NonEmptyList`

Refactoring the retirement calculator to use ValidatedNel

With this new knowledge, we are well equipped to improve our retirement calculator further. We are going to improve `SimulatePlanApp` to give the users more information if one or many arguments passed to the program are wrong.

When many arguments are wrong, for instance, if the user passes some random text instead of a parsable number, we want to report one error for every bad argument.

Adding unit tests

Firstly, we need to change the test associated with `SimulatePlanApp`. Open `SimulatePlanAppIT.scala` and change the content as follows:

```
package retcalc

import cats.data.Validated.{Invalid, Valid}
import org.scalactic.TypeCheckedTripleEquals
import org.scalatest.{Matchers, WordSpec}

class SimulatePlanAppIT extends WordSpec with Matchers with TypeCheckedTripleEquals {
  "SimulatePlanApp.strMain" should {
    "simulate a retirement plan using market returns" in {
      val actualResult = SimulatePlanApp.strMain(
        Array("1952.09,2017.09", "25", "40", "3000", "2000", "10000"))

      val expectedResult =
        s"""
          |Capital after 25 years of savings:    468925
          |Capital after 40 years in retirement: 2958842
          |""".stripMargin
      actualResult should ===(Valid(expectedResult))
    }

    "return an error when the period exceeds the returns bounds" in {
      val actualResult = SimulatePlanApp.strMain(
        Array("1952.09,2017.09", "25", "60", "3000", "2000", "10000"))
      val expectedResult = "Cannot get the return for month 780.  
Accepted range: 0 to 779"
      actualResult should ===(Invalid(expectedResult))
    }

    "return an usage example when the number of arguments is incorrect" in {
      val result = SimulatePlanApp.strMain(
        Array("1952.09:2017.09", "25.0", "60", "3'000", "2000.0"))
      result should ===(Invalid(
        """Usage:
          |simulatePlan from,until nbOfYearsSaving nbOfYearsRetired
          |netIncome currentExpenses initialCapital
          |
          |Example:
          |simulatePlan 1952.09,2017.09 25 40 3000 2000 10000
          |""".stripMargin))
    }

    "return several errors when several arguments are invalid" in {
      val result = SimulatePlanApp.strMain(
        Array("1952.09:2017.09", "25.0", "60", "3'000", "2000.0",
          "10000"))
      result should ===(Invalid(
        """Invalid format for fromUntil. Expected: from,until, actual:
          |1952.09:2017.09
          |Invalid number for nbOfYearsSaving: 25.0
          |Invalid number for netIncome: 3'000
          |Invalid number for currentExpenses: 2000.0""".stripMargin))
    }
  }
}
```

```
| } }
```

Let's have a look at the preceding code in detail:

- The first two tests do not change much—we just changed the expectations to be `Valid(expectedResult)`. We are going to change the return type of `simulatePlanApp.strMain`—instead of returning a string, we are going to change it to return `Validated[String, String]`. We expect `strMain` to return a `Valid` value containing the result if all arguments are correct. If some arguments are incorrect, it should return an `Invalid` value containing `String` explaining what the incorrect arguments are.
- The third test is a new test. If we do not pass the right number of arguments, we expect `strMain` to return an `Invalid` value containing a usage example.
- The fourth test checks that one error for every bad argument is reported.

Implementing parsing functions

The next step is to add new types of errors that will be returned in `validateNel` when some arguments are wrong. We need to change `RetCalcError.scala` as follows:

```
object RetCalcError {
    type RetCalcResult[A] = ValidatedNel[RetCalcError, A]

    case class MoreExpensesThanIncome(income: Double, expenses: Double)
        extends RetCalcError(...)

    case class ReturnMonthOutOfBounds(month: Int, maximum: Int) extends
        RetCalcError(...)

    case class InvalidNumber(name: String, value: String) extends
        RetCalcError(
            s"Invalid number for $name: $value")

    case class InvalidArgument(name: String,
                                value: String,
                                expectedFormat: String) extends
        RetCalcError(
            s"Invalid format for $name. Expected: $expectedFormat, actual:
            $value")
}
```

Here, we introduced an `InvalidNumber` error that will be returned when a string cannot be parsed into a number. The other error, `InvalidArgument`, will be returned when an argument is wrong. We will use it when the `from` and `until` parameters are wrong (see the preceding unit test). Also, as we are going to use many types of `ValidatedNel[RetCalcError, A]` form, we created a type alias, `RetCalcResult`. It will also help IntelliJ to autocomplete the functions of the `cats` library.

After that, we need to change `simulatePlanApp.strMain` to validate the arguments. For this, we start by writing a small function that parses one string argument to produce `Validated Int`.

 *Ideally, all the following parsing functions should be unit tested. We do have indirect test coverage for them in `simulatePlanAppIT`, but it is not sufficient. In test-driven development, whenever you need to write a new function, you should just define its signature and then write a test before implementing it. Unfortunately, we do not have enough space in this book to show all of the unit tests that you would expect to have in a production application. However, as an exercise, I encourage you to write them.*

We call this function `parseInt`. It takes the name of an argument and its value, and

returns `Validated[Int]`, as shown in the following code:

```
def parseInt(name: String, value: String): RetCalcResult[Int] =  
  Validated  
    .catchOnly[NumberFormatException](value.toInt)  
    .leftMap(_ => NonEmptyList.of(InvalidNumber(name, value)))
```

We first call the `Validated.catchOnly` method, which executes a block of code (in our case, `value.toInt`) and catches a specific type of exception. If the block does not throw any exception, `catchOnly` returns a `Valid` value with the result. If the block throws the exception type passed as an argument (in our case, `NumberFormatException`), then `catchOnly` returns an `Invalid` value containing the caught exception. The resulting expression type is `Validated[NumberFormatException, Int]`. However, our `parseInt` function must return `RetCalcResult[Int]`, which is an alias for `ValidatedNel[RetCalcError, Int]`. In order to transform the error or left type, we call the `Validated.leftMap` method to produce `NonEmptyList[RetCalcError]`.

Then, we write another function, `parseFromUntil`—that is in charge of parsing the `from` and `until` arguments. These two arguments are separated by a comma, as shown in the following code:

```
import cats.implicitly._  
def parseFromUntil(fromUntil: String): RetCalcResult[(String, String)] = {  
  val array = fromUntil.split(",")  
  if (array.length != 2)  
    InvalidArgument(  
      name = "fromUntil", value = fromUntil,  
      expectedFormat = "from,until"  
    ).invalidNel  
  else  
    (array(0), array(1)).validNel  
}
```

We create an `Array[String]` with the `String.split` method. If the array does not have exactly two elements, we return an `Invalid` value containing an `InvalidArgument` error. If the array has two elements, then we return them in a tuple inside a `Valid` value.

Finally, we write a `parseParams` function that accepts an array of arguments and produces `RetCalcResult[RetCalcParams]`. The `RetCalcParams` argument is one of the arguments required by `RetCalc.simulatePlan`, as shown in the following code:

```
def parseParams(args: Array[String]): RetCalcResult[RetCalcParams] =  
(  
  parseInt("nbOfYearsRetired", args(2)),  
  parseInt("netIncome", args(3)),  
  parseInt("currentExpenses", args(4)),
```

```
    parseInt("initialCapital", args(5))
).mapN { case (nbOfYearsRetired, netIncome, currentExpenses,
  initialCapital) =>
  RetCalcParams(
    nbOfMonthsInRetirement = nbOfYearsRetired * 12,
    netIncome = netIncome,
    currentExpenses = currentExpenses,
    initialCapital = initialCapital)
}
```

The function assumes that the `args` array has at least six elements. We create a tuple of four elements, each element being the result of `parseInt`, and hence, it has the `RetCalcResult[Int]` type. Then, we call the `mapN` method on our `Tuple4`, which will accumulate any error produced by the calls to `parseInt`. If all of the `parseInt` calls return a `Valid` value, the anonymous function passed to `mapN` is called. It takes `Tuple4 (Int, Int, Int, Int)` and returns a `RetCalcParams` instance.

Implementing SimulatePlanApp.strSimulatePlan

In order to keep `SimulatePlanApp.strMain` small and readable, we are going to extract the code that is in charge of calling `RetCalc.simulatePlan` and return a human-readable string detailing the result of the simulation. We call this new function `strSimulatePlan`, and use it as shown in the following code:

```
def strSimulatePlan(returns: Returns, nbOfYearsSaving: Int, params: RetCalcParams)
  : RetCalcResult[String] = {
  RetCalc.simulatePlan(
    returns = returns,
    params = params,
    nbOfMonthsSavings = nbOfYearsSaving * 12
  ).map {
    case (capitalAtRetirement, capitalAfterDeath) =>
      val nbOfYearsInRetirement = params.nbOfMonthsInRetirement / 12
      s"""
        |Capital after $nbOfYearsSaving years of savings:
        |${capitalAtRetirement.round}
        |Capital after $nbOfYearsInRetirement years in retirement:
        |${capitalAfterDeath.round}
        |""".stripMargin
  }.tovalidatedNel
}
```

The function takes the parsed arguments, calls `simulatePlan`, and transforms its result into a string. In order to keep the same type as our parsing functions, we declare the return type of the function to be `RetCalcResult[String]`. This is an alias for `ValidatedNel[RetCalcError, String]`, but `simulatePlan` returns `Either[RetCalcError, String]`. Fortunately, cats provide `.tovalidatedNel` method to easily convert `Either` to `ValidatedNel`.

Refactoring SimulatePlanApp.strMain

We have implemented some building blocks for parsing the whole arguments array. It is now time to refactor `SimulatePlanApp.strMain` to call them. First, we need to check that the arguments array has the right size, as shown in the following code:

```
def strMain(args: Array[String]): Validated[String, String] = {
  if (args.length != 6)
    """Usage:
      |simulatePlan from,until nbOfYearsSaving nbOfYearsRetired
      |netIncome currentExpenses initialCapital
      |
      |Example:
      |simulatePlan 1952.09,2017.09 25 40 3000 2000 10000
      |""".stripMargin.invalid
  else {
    val allReturns = Returns.fromEquityAndInflationData(
      equities = EquityData.fromResource("sp500.tsv"),
      inflations = InflationData.fromResource("cpi.tsv"))

    val vFromUntil = parseFromUntil(args(0))
    val vNbOfYearsSaving = parseInt("nbOfYearsSaving", args(1))
    val vParams = parseParams(args)

    (vFromUntil, vNbOfYearsSaving, vParams)
      .tupled
      .andThen { case ((from, until), nbOfYearsSaving, params) =>
        strSimulatePlan(allReturns.fromUntil(from, until),
                      nbOfYearsSaving, params)
      }
      .leftMap(nel => nel.map(_.message).toList.mkString("\n"))
  }
}
```

In order to match the assertions we made in the `SimulatePlanAppIT` integration test, we change the signature to return `Validated[String, String]`. If the arguments array is of the wrong size, we return an `Invalid` value, with a string explaining the correct usage for our program. Otherwise, when the arguments array is of the right size, we first declare the `allReturns` variable as before.

Then, we call the three parsing functions that we implemented earlier and assign them to `vFromUntil`, `vNbOfYearsSaving`, and `vParams`. Their types are `RetCalcResult[(String, String)]`, `RetCalcResult[Int]`, and `RetCalcResult[RetCalcParams]` respectively. After that, we put these three values in `Tuple3`, and call the `cats tupled` function, which

combines the three elements of the tuple to produce `RetCalcResult[((String, String), Int, RetCalcParams)]`.

At this point, we have a `ValidatedNel` instance containing all the required parameters to call the `strSimulatePlan` function that we implemented earlier. In this case, we need to check errors sequentially—first, we validate all the arguments, *and then* we call `strSimulatePlan`. If we had used `Either` instead of `ValidatedNel`, we would have used `flatMap` to do this. Fortunately, `ValidatedNel` provides an equivalent method in the form of `andThen`.



Unlike `Option` and `Either`, instances of `ValidatedNel` do not have a `flatMap` method because it is not a monad, it is an applicative functor. We will explain what these terms mean in [Chapter 4, Advanced Features](#). If you want to run validations in sequence, you need to use `andThen` or convert in to `Either` and use `flatMap`.

Before the call to `.leftMap`, we have an expression of the `RetCalcResult[String]` type, which is an alias for `Validated[NonEmptyList[RetCalcError], String]`. However, our function must return `Validated[String, String]`. Therefore, we transform the left `NonEmptyList[RetCalcError]` type to a string using the anonymous function passed to `.leftMap`.

Summary

In this chapter, we saw how we can handle optional values and how to handle errors in a purely functional way. You are now better equipped to write safer programs that do not throw exceptions and crash unexpectedly.

If you use Java libraries or some non-purely-functional Scala libraries, you will notice that they can throw exceptions. If you do not want your program to crash when exceptions are raised, I advise you to wrap them as early as possible inside `Either` OR `Validated`.

We saw how `Either` can be used to handle errors sequentially, and how `Validated` can handle errors in parallel. As these two types are very similar, I would advise you to use `Validated` most of the time. Instances of `Validated` can indeed handle errors in parallel using `mapN`, but they can also perform sequential validation using `andThen`.

This chapter went a bit further in the way of writing programs in a functional way. In the next chapter we will explore other features of the language that you will necessarily encounter in typical Scala projects: laziness, covariance and contravariance, and implicit.

Questions

Here are some questions to test your knowledge:

- What type can you use to represent an optional value?
- What type(s) can you use to represent the possibility of an error?
- What is referential transparency?
- Is it good practice to throw exceptions?

Here are some exercises:

- Write unit tests for `SimulatePlanApp`
- Use `RetCalcResult` instead of `Either[RetCalcError, x]` in `RetCalc.scala`
- Change `variableReturns.fromUntil` to return an error if `monthIdFrom` or `monthIdUntil` cannot be found in the `returns` vector

Further reading

The `cats` documentation on `Either` and `validated` at the following links provides other usage examples, as well as more details on their respective topics:

- <https://typelevel.org/cats/datatypes/either.html>
- <https://typelevel.org/cats/datatypes/validated.html>

Advanced Features

In this chapter, we are going to explore the more advanced features of Scala. As with any programming language, some advanced constructs might be seldom used in practice or can obfuscate code.

We will aim to only explain features that we have encountered in real projects that have been deployed to production. Some features are used more in libraries or in the SDK than in a typical project, but it is important to understand them in order to be able to use a library effectively.

As these features are varied and cover a large spectrum, we found it easier to explain them using ad hoc code examples rather than a complete project. You can, therefore, jump directly to any section of this chapter if you are already familiar with some of these concepts.

In this chapter, we will cover the following topics:

- Strictness and laziness, and their impact on performance
- Covariance and contravariance
- Currying and partially applied functions
- Implicit usage

Project setup

All the examples of this chapter have been committed to the following Git repository:

- <https://github.com/PacktPublishing/Scala-Programming-Projects>

If you want to run the code examples in this chapter, you need to clone this repository and import the project into IntelliJ. Each section has a corresponding Scala worksheet file—for instance, the next section's examples are in the `lazyness.sc` file.

The solutions to the exercises are given in these worksheets, and so it would be more profitable to you if you do not read them completely until you have tried to do the exercises.

Strictness and laziness

Scala's default evaluation strategy is strict. This means that if you don't do anything special, any variable declaration or arguments of a function call are immediately evaluated. The opposite of a strict evaluation strategy is a lazy evaluation strategy, which means that evaluation is performed only when needed.

Strict val

The following is a strict variable declaration:

```
class StrictDemo {  
    val strictVal = {  
        println("Evaluating strictVal")  
        "Hello"  
    }  
}  
val strictDemo = new StrictDemo  
//Evaluating strictVal  
//strictDemo: StrictDemo = StrictDemo@32fac009
```

We can see that `println` is called immediately. This means that the block at the right side of the assignment is evaluated as soon as the `strictDemo` class is instantiated. If we want to delay the block's evaluation, we have to use the `lazy` prefix.

lazy val

When we use the `lazy` prefix in front of `val` or `var` (as shown in the following code), it is evaluated only when needed:

```
class LazyDemo {  
    lazy val lazyVal = {  
        println("Evaluating lazyVal")  
        "Hello"  
    }  
}  
val lazyDemo = new LazyDemo  
//lazyDemo: LazyDemo = LazyDemo@13ca84d5
```

When we instantiate the class, the block at the right side of the assignment is not evaluated. It will only get evaluated when we use the variable, as shown in the following code:

```
lazyDemo.lazyVal + " World"  
//Evaluating lazyVal  
//res0: String = Hello World
```

This mechanism allows you to defer the evaluation of computationally expensive operations. For instance, you could use it to start an application quickly and run the initialization code only when the first user needs it. Scala guarantees that the evaluation will be performed only once, even if there are multiple threads trying to use the variable at the same time.

You can have chains of `lazy` values, which will get evaluated only when the last element of the chain is required, as shown in the following code:

```
class LazyChain {  
    lazy val val1 = {  
        println("Evaluating val1")  
        "Hello"  
    }  
    lazy val val2 = {  
        println("Evaluating val2")  
        val1 + " lazy"  
    }  
    lazy val val3 = {  
        println("Evaluating val3")  
        val2 + " chain"  
    }  
}  
val lazyChain = new LazyChain  
// lazyChain: LazyChain = LazyChain@4ca51fa
```

When we require `val3`, the three values get evaluated, as shown in the following code:

```
lazyChain.val3
// Evaluating val3
// Evaluating val2
// Evaluating val1
// res1: String = Hello lazy chain
```

By-name parameters

We can go a bit further and delay the evaluation of function arguments. A by-name parameter is like a function that does not take any argument. This way, it only gets evaluated when the function's body requires it.

Say that you have an application that loads its configuration from a file or a database. You want the application to start as fast as possible, and so you decide to use `lazy val` to load a greeting message on demand, as shown in the following code:

```
object AppConfig {  
    lazy val greeting: String = {  
        println("Loading greeting")  
        "Hello "  
    }  
}
```

Note that for the sake of brevity we haven't actually loaded anything here—just imagine that we did.

If we then want to use the greeting variable in a function but keep delaying its evaluation, we can use a by-name parameter:

```
def greet(name: String, greeting: => String): String = {  
    if (name == "Mikael")  
        greeting + name  
    else  
        s"I don't know you $name"  
}  
greet("Bob", AppConfig.greeting)  
// res2: String = I don't know you Bob  
greet("Mikael", AppConfig.greeting)  
// Loading greeting  
// res3: String = Hello Mikael
```

The `AppConfig.greeting` phrase is not evaluated the first time we call `greet` with "Bob", because the body of the function did not require it. It is only evaluated when we call `greet` with "Mikael".

In some cases, using by-name parameters can enhance the performance of a program, because the evaluation of an expensive operation can be skipped if it is not required.

Lazy data structures

Here is a function definition that calls the strict methods of `vector`:

```
def evenPlusOne(xs: Vector[Int]): Vector[Int] =  
  xs.filter { x => println(s"filter $x"); x % 2 == 0 }  
    .map    { x => println(s"map $x");      x + 1      }  
  
evenPlusOne(Vector(0, 1, 2))
```

It prints the following on the console:

```
filter 0  
filter 1  
filter 2  
map 0  
map 2  
res4: Vector[Int] = Vector(1, 3)
```

We can see that `vector` is iterated twice: once for `filter`, and once for `map`. But our `evenPlusOne` function would be faster if it could iterate only once. One way to do this would be to change the implementation and use `collect`. Another way would be to use the non-strict `withFilter` method, as shown in the following code:

```
def lazyEvenPlusOne(xs: Vector[Int]): Vector[Int] =  
  xs.withFilter { x => println(s"filter $x"); x % 2 == 0 }  
    .map    { x => println(s"map $x");      x + 1      }  
  
lazyEvenPlusOne(Vector(0, 1, 2))
```

The `withFilter` method prints the following:

```
filter 0  
map 0  
filter 1  
filter 2  
map 2  
res5: Vector[Int] = Vector(1, 3)
```

This time, `vector` is iterated only once. Each element is filtered and then mapped one by one. This is because `withFilter` is a lazy operation—it does not immediately create a new filtered `vector`, but instead creates a new `withFilter` object that will store the filter's predicate. This SDK collection `withFilter` type has a special implementation of `map` that will call the filter's predicate before calling the function passed to `map`.

This works pretty well as long as you only have one `map` or `flatMap` operation, or if you further refine your `filter` with another call to `withFilter`. However, if you call another `map` operation, the collection will be iterated twice, as shown in the following code:

```
def lazyEvenPlusTwo(xs: Vector[Int]): Vector[Int] =  
  xs.withFilter { x => println(s"filter $x"); x % 2 == 0 }  
    .map { x => println(s"map $x") ; x + 1 }  
    .map { x => println(s"map2 $x") ; x + 1 }
```

```
lazyEvenPlusTwo(Vector(0, 1, 2))
```

The preceding code prints the following:

```
filter 0  
map 0  
filter 1  
filter 2  
map 2  
map2 1  
map2 3  
res6: Vector[Int] = Vector(2, 4)
```

We can see that the calls to `map2` are made at the end, which means that we iterated a second time over `Vector(1, 3)`. We need a lazier data structure that will not iterate until we actually need each element.

In the Scala SDK, this collection type is `Stream`. If we replace `Vector` with `Stream` in our `lazyEvenPlusTwo` function, then we get the desired behavior, as shown in the following code:

```
def lazyEvenPlusTwoStream(xs: Stream[Int]): Stream[Int] =  
  xs.filter { x => println(s"filter $x") ; x % 2 == 0 }  
    .map { x => println(s"map $x") ; x + 1 }  
    .map { x => println(s"map2 $x") ; x + 1 }
```

```
lazyEvenPlusTwoStream(Stream(0, 1, 2)).toVector
```

After the call to our function, we convert the resulting `Stream` to `Vector`. It is this call to `toVector` that materializes the elements of the stream and calls the anonymous functions passed to `filter` and `map`. The following code is what gets printed on the console:

```
filter 0  
map 0  
map2 1  
filter 1  
filter 2  
map 2  
map2 3
```

```
| res7: Vector[Int] = Vector(2, 4)
```

We can see that `stream` is iterated only once. For each element, we call `filter`, then `map` and `map2`.

But there is more to it. Since `stream` is lazy, it can be used to represent infinite collections. The following code shows how we can get a `stream` of all the positive even integers:

```
| val evenInts: Stream[Int] = 0 #:: 2 #:: evenInts.tail.map(_ + 2)
| evenInts.take(10).toVector
| // res8: Vector[Int] = Vector(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

We use the `stream` operator `#::`, which builds `stream` with a head element and a tail. It works the same way as the `List` operator `::`, but in a lazy way. The following steps show how it works:

1. We build `stream 0 #:: 2`, which has `0` as its head and a tail with one element, `2`.
2. The third element will be `(0 #:: 2).tail.map(_ + 2)`. At this stage, the tail is only `Stream(2)`, and so the third element will be `4`.
3. The fourth element will be `(0 #:: 2 #:: 4).tail.map(_ + 2)`. The same process repeats for all subsequent elements.

Since our `stream` is infinite, we cannot convert all of it to `vector`, as this would go on forever. We just take the first 10 elements with `take(10)` and then convert them to `vector`.

Covariance and contravariance

When an F type accepts a type parameter of A , we can add a + or - sign in front of the parameter declaration to indicate the **variance** of F on A :

- $F[+A]$ makes F **covariant** on A . This means that if $B <: A$ (where B extends A), then $F[B] <: F[A]$.
- $F[-A]$ makes F **contravariant** on A . If $B <: A$, then $F[A] <: F[B]$.
- $F[A]$ makes F **invariant** on A . If $B <: A$, then there is no relationship between $F[A]$ and $F[B]$.

InvariantDecoder

We are now going to explore this variance concept with an example. Let's start with a simple class hierarchy, as shown in the following code:

```
trait Animal
case class Cat(name: String) extends Animal
case class Dog(name: String) extends Animal
```

With this hierarchy, I can declare a variable of an `Animal` type and assign it to an instance of a `Cat` or `Dog` type. The following code will compile:

```
val animal1: Animal = Cat("Max")
val animal2: Animal = Dog("Dolly")
implicitly[Dog <:< Animal]
```

More generally, the assignment `val a: A = b: B` compiles if `B <:< A`.



You can check that type `B` extends type `A` with the expression `implicitly[B <:< A]`; if it compiles, then `B` is a subtype of `A`.

Then, we define an `InvariantDecoder` trait that has a single `decode` method. There is no + or - sign, and so `InvariantDecoder` is invariant on `A`, as shown in the following code:

```
trait InvariantDecoder[A] {
  def decode(s: String): Option[A]
}
```

After this, we implement `InvariantDecoder` for `Cat`, as shown in the following code:

```
object InvariantCatDecoder extends InvariantDecoder[Cat] {
  val CatRegex = """Cat\((\w+)\)""".r
  def decode(s: String): Option[Cat] = s match {
    case CatRegex(name) => Some(Cat(name))
    case _ => None
  }
}
InvariantCatDecoder.decode("Cat(Max)")
// res0: Option[Cat] = Some(Cat(Max))
```

When we call `decode` with a string that matches the `CatRegex` regular expression, we obtain a `Cat` instance wrapped in an `option` instance.

But what if we declare a variable of the `InvariantDecoder[Animal]` type? Can we assign our `InvariantCatDecoder` to it? Let's try it:

```
| val invariantAnimalDecoder: InvariantDecoder[Animal] = InvariantCatDecoder
```

The preceding code does not compile, but the compiler is very helpful in this case. The following code is the error you will get:

```
error: type mismatch;
  found   : InvariantCatDecoder.type
  required: InvariantDecoder[Animal]
    Note: Cat <: Animal (and InvariantCatDecoder.type <:
    InvariantDecoder[Cat]), but trait InvariantDecoder is invariant in
    type A.
  You may wish to define A as +A instead. (SLS 4.5)
    val invariantAnimalDecoder: InvariantDecoder[Animal] =
      InvariantCatDecoder
                                         ^
                                         ^
```

It tells us that if we want to make this line compile, we have to make `InvariantDecoder` covariant in type `A`. To do so, we have to add a `+` sign in front of the `A` parameter in `InvariantDecoder`.

CovariantDecoder

Let's follow the compiler's advice and create a new `CovariantDecoder[+A]`, along with a `CovariantCatDecoder` instance that extends it, as shown in the following code:

```
trait CovariantDecoder[+A] {
  def decode(s: String): Option[A]
}
object CovariantCatDecoder extends CovariantDecoder[Cat] {
  (...)
```

We do not show the implementation of `decode` in `CovariantCatDecoder`; it is the same as in `InvariantCatDecoder`. With this covariant parameter, the following relationship is verified:

```
| implicitly[CovariantDecoder[Cat] <:< CovariantDecoder[Animal]]
```

This time, we can assign the `CovariantCatDecoder` to an instance of `CovariantDecoder[Animal]`, as shown in the following code:

```
val covariantAnimalDecoder: CovariantDecoder[Animal] = CovariantCatDecoder
covariantAnimalDecoder.decode("Cat(Ulysse)")
// res0: Option[Animal] = Some(Cat(Ulysse))
```

When we call `decode` on it, we get back an `Option[Animal]`.

At first, glance, having `covariantDecoder` seems natural—if my decoder can produce `cat`, and `cat` is an `Animal`, my decoder should also be a decoder of `Animal`. On the other hand, if I have an instance of `Decoder[Animal]`, I would expect it to be able to decode any `Animal`—not only `cat`, but also `Dog` instances—and this is not the case for our earlier `covariantAnimalDecoder`.

There is no right or wrong design here; this is just a matter of taste. In general, I would advise you to use invariant type parameters first, and if you then experience some limitations with it, you can decide to make them covariant or contravariant.

The full covariant implementation for getting `cat` and `Dog` would be the following:

```
| object CovariantCatAndDogDecoder extends CovariantDecoder[Animal] {
```

```
val CatRegex = """Cat\\((\\w+\\))""".r
val DogRegex = """Dog\\((\\w+\\))""".r
def decode(s: String): Option[Animal] = s match {
  case CatRegex(name) => Some(Cat(name))
  case DogRegex(name) => Some(Dog(name))
  case _ => None
}
}

val covariantCatsAndDogsDecoder = CovariantCatAndDogDecoder
covariantCatsAndDogsDecoder.decode("Cat(Garfield)")
// res4: Option[Animal] = Some(Cat(Garfield)))
covariantCatsAndDogsDecoder.decode("Dog(Aiko)")
// res5: Option[Animal] = Some(Dog(Aiko)))
```

Contravariant encoder

Now, we would like to model the opposite of decoding a string to an object—encoding an object to a string! We make our `Encoder` contravariant by adding a `-` sign in front of the `A` type parameter, as shown in the following code:

```
| trait Encoder[-A] {  
|   def encode(a: A): String  
| }
```

The following code is an instance of this `Encoder`:

```
| object AnimalEncoder extends Encoder[Animal] {  
|   def encode(a: Animal): String = a.toString  
| }
```

We have the `Cat <:< Animal` relationship, and `Encoder` is contravariant on its argument. This implies that `Encoder[Animal] <:< Encoder[Cat]`, and I can, therefore, assign `Encoder[Animal]` to a variable of type `Encoder[Cat]`, as shown in the following code:

```
| val catEncoder: Encoder[Cat] = AnimalEncoder  
| catEncoder.encode(Cat("Luna"))  
| // res1: String = Cat(Luna)
```

Similar to the covariant decoder, the contravariance of the encoder seems natural—if I can encode any `Animal`, I can also encode `Cat`.

However, if we wanted to create a single `Codec` trait that can encode and decode, we would be in trouble. The type parameter cannot be covariant and contravariant at the same time.

The only way to make it work would be to make the type parameter invariant, as in the following implementation:

```
| object CatAndDogCodec extends Codec[Animal] {  
|   val CatRegex = """Cat\\((\\w+\\))""".r  
|   val DogRegex = """Dog\\((\\w+\\))""".r  
|  
|   override def encode(a: Animal) = a.toString  
|  
|   override def decode(s: String): Option[Animal] = s match {  
|     case CatRegex(name) => Some(Cat(name))  
|     case DogRegex(name) => Some(Dog(name))  
|   }
```

```
| }   case _ => None  
| }
```

But let's see what happens if we attempt to use a covariance. The compiler would return the following error:

```
| trait Codec[+A] {  
|   def encode(a: A): String  
|   def decode(s: String): Option[A]  
| }  
| Error:(55, 15) covariant type A occurs in contravariant position in  
|   type A of value a  
|   def encode(a: A): String  
|     ^
```

The compiler complains about the `A` type being in a contravariant position. This is because in functions, the parameters are always in a contravariant position, and the result is always in a covariant position. For instance, if you open `scala.Function3`, you will see the following declaration:

```
| trait Function3[-T1, -T2, -T3, +R] extends AnyRef { self =>
```

This implies the following two things:

- If you declare a type parameter as covariant with `+A`, then the `A` type can only appear in the **result** of a method
- If you declare a type parameter as contravariant with `-A`, then the `A` type can only appear in the **parameters** of a method

In our `decode` method, `A` appears in the result and hence is in a covariant position. This is why we could make the decoder covariant on `A` by using `+A` in `CovariantDecoder`.

Conversely, in our `encode` method, `A` appears in the parameters, and hence it is in a contravariant position. This is why we could make the encoder contravariant on `A` by using `-A` in `Encoder`.

Another way to implement our `Codec` would be to use a type class. This is explained in [Chapter 5, Type Classes](#).

Covariance in collections

Most collection types in the SDK are covariant. For instance, if you open the `vector` class, you will see the following:

```
| final class Vector[+A] (...)
```

This allows us to assign `Vector[B]` to a variable of `Vector[A]` if $B <: <: A$ type, as shown in the following code:

```
| val cats: Vector[Cat] = Vector(Cat("Max"))
| val animals: Vector[Animal] = cats
```

Now here is a bit of magic:

```
| val catsAndDogs = cats :+ Dog("Medor")
| // catsAndDogs: Vector[Product with Serializable with Animal] =
| // Vector(Cat(Max), Dog(Medor))
```

Scala not only allows us to add `Dog` to `vector[Cat]`, but it also automatically infers the new collection to be of a `vector[Product with Serializable with Animal]` type.

We saw earlier that the parameters of a function are in contravariant position. Therefore, it should not even be possible to have a `:+(a: A)` method that can add elements to the `vector` class, because `vector` is covariant on `A`! But there is a trick. If you look in the `Vector` source code at the definition of `:+`, the following code is what you will see:

```
| override def :+[B >: A, That](elem: B)(implicit bf: CanBuildFrom[Vector[A], B, That]): T
```

The method takes a `B` type parameter with the constraint that `B` must be a supertype of `A`. In our previous example, the `A` type was `cat` and our `elem` argument was of the `Dog` type. The Scala compiler automatically inferred the `B` type to be the closest common supertype of `Dog` and `cat`, which is `Product with Serializable with Animal`.

If we add `String` to this `vector`, the resulting type will be the next common supertype between `Animal` and `String`, which is `Serializable`, as shown in the following code:

```
| val serializables = catsAndDogs :+ "string"
| // serializables: Vector[Serializable] = Vector(Cat(Max), Dog(Medor),
| // string)
| val anys = serializables :+ 1
| // anys: Vector[Any] = Vector(Cat(Max), Dog(Medor), string, 1)
```

Then, when we add an `Int` to the `vector` class, the next common supertype between `Serializable` and `Int` is `Any`.



If you have a class that has a covariant `myClass[+A]` type parameter, and you need to implement a method with a parameter of an `A` type, then you can define it with a `B >: A` type parameter, written as `def myMethod[B >: A](b: B) = ...`

Currying and partially applied functions

The name *currying* is a reference to the mathematician and logician Haskell Curry. The process of currying consists of transforming a function that takes multiple arguments into a sequence of functions, each with a single argument.

Function value

Before we start currying functions, we need to understand the difference between a function and a function value.

You are already familiar with functions—they begin with the keyword `def`, take one or several parameter lists between `()` symbols, optionally declare a return type after a `:` sign, and have a defined body after the `=` sign, as shown in the following example:

```
| def multiply(x: Int, y: Int): Int = x * y  
| // multiply: multiply[](val x: Int, val y: Int) => Int
```

A function value (also called a **function literal**) is similar to any other value, such as `"hello": String`, `3: Int`, or `true: Boolean`. As with other values, you can pass a function value as an argument to a function, or assign it to a variable using the `val` keyword.

You can declare function values directly, as shown in the following code:

```
| val multiplyVal = (x: Int, y: Int) => x * y  
| // multiplyVal: (Int, Int) => Int = ...
```

Or you can transform a function into a function value by adding a `_` character at the end of the function's name, as shown in the following code:

```
| val multiplyVal2 = multiply _  
| // multiplyVal2: (Int, Int) => Int = ...
```

When it comes to applying arguments to a function, the syntax is the same whether we call a function or a function value, as shown in the following code:

```
multiply(2, 3)  
multiplyVal(2, 3)  
multiplyVal2(2, 3)
```

Currying

A curried function is a function that takes one parameter and returns another function that takes one parameter. You can convert a function value into a curried function value by calling the `.curried` method, as shown in the following code:

```
| val multiplyCurried = multiplyVal.curried  
| // multiplyCurried: Int => (Int => Int) = ...
```

The call to `.curried` transforms the function value's type from `(Int, Int) => Int` to `Int => (Int => Int)`. The `multiplyVal` takes two integers as parameters and returns an integer. The `multiplyCurried` takes one `Int` and returns a function that takes `Int` and returns `Int`. The two function values have exactly the same functionality—the difference lies in the way we call them, as shown in the following code:

```
multiplyVal(2, 3)  
// res3: Int = 6  
multiplyCurried(2)  
// res4: Int => Int = ...  
multiplyCurried(2)(3)  
// res5: Int = 6
```

When we call `multiplyCurried(2)`, we apply only the first argument, and this returns a `Int => Int` function. At this stage, the function is not completely applied—it is a **partially applied** function. If we want to obtain the final result, we have to apply the second argument by calling `multiplyCurried(2)(3)`.

Partially applied functions

In practice, there is no need to call `.curried` to define curried functions. You can declare curried functions directly with multiple parameter lists. Here is a curried function that calculates a discount to an `Item` class:

```
| case class Item(description: String, price: Double)
| def discount(percentage: Double)(item: Item): Item =
|   item.copy(price = item.price * (1 - percentage / 100))
```

We can **fully apply** the function if we provide two argument lists, as follows:

```
| discount(10)(Item("Monitor", 500))
| // res6: Item = Item(Monitor,450.0)
```

But we can also partially apply the function if we just provide the first argument list and add a `_` character to indicate that we want a function value, as shown in the following code:

```
| val discount10 = discount(10) _
| // discount10: Item => Item = ...
| discount10(Item("Monitor", 500))
| // res7: Item = Item(Monitor,450.0)
```

The `discount10` function value is a partially applied function that takes `Item` and returns `Item`. We can then call it with an `Item` instance to fully apply it.

A partially applied function is especially useful when we need to pass anonymous functions to higher order functions (functions that accept functions as arguments), such as `map` or `filter`, as shown in the following code:

```
| val items = Vector(Item("Monitor", 500), Item("Laptop", 700))
| items.map(discount(10))
| // res8: Vector[Item] = Vector(Item(Monitor,450.0), Item(Laptop,630.0))
```

In this example, the `map` function expects an argument of a `Item => B` type. We pass the `discount(10)` argument, which is a partially applied function of a `Item => Item` type. Thanks to partially applied functions, we managed to apply a discount to a collection of items without having to define a new function.

Implicits

As its name indicates, the Scala keyword `implicit` can be used to implicitly add some extra code to the compiler. For instance, an implicit parameter in a function definition allows you to omit this parameter when you call the function. As a result, you do not have to pass this parameter explicitly.

There are different kinds of implicit in Scala that we will cover in this section:

- An implicit parameter is declared in a function definition
- An implicit value is passed as an argument to a function that has an implicit parameter
- An implicit conversion converts one type to another type

This is an extremely powerful feature that can feel a bit like magic sometimes. In this section, we will see how it can help writing more concise code and also how to use it to validate some constraints at compile time. In the next chapter, we will use them to define another powerful concept: type classes.

Implicit parameters

In a function definition, the last parameter list can be marked as `implicit`. Such a function can then be invoked without passing the corresponding arguments. When you omit the implicit arguments, the Scala compiler will try to find implicit values of the same type in the current scope and will use them as arguments to the function.

Here is an illustration of this mechanism that you can type into a Scala worksheet:

```
case class ApplicationContext(message: String)
implicit val myAppCtx: ApplicationContext = ApplicationContext("implicit world")

def greeting(prefix: String)(implicit appCtx: ApplicationContext): String =
  prefix + appCtx.message

greeting("hello ")
// res0: String = hello implicit world
```

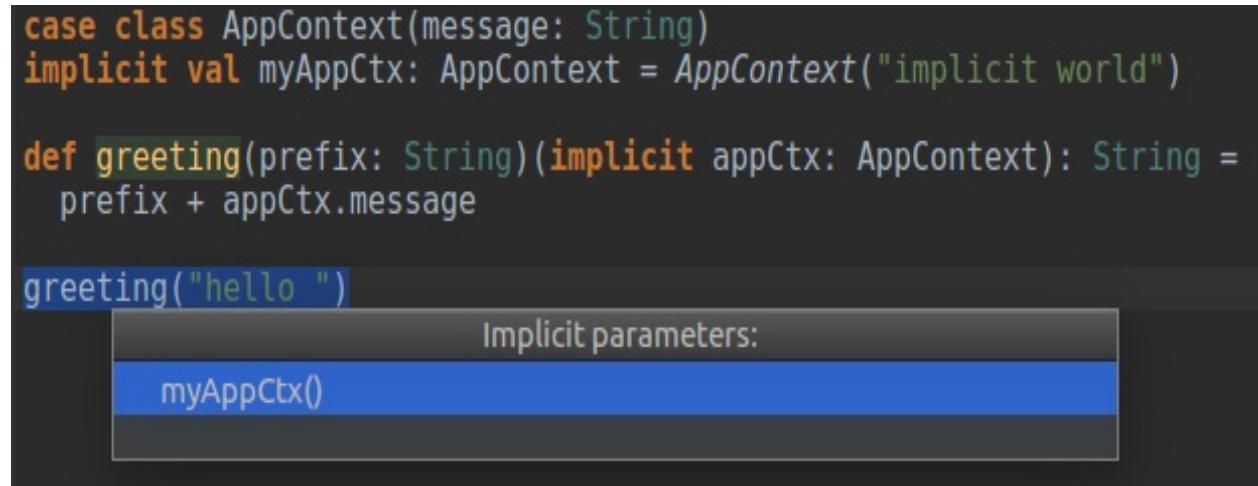
We first declare a new `ApplicationContext` class and assign a new instance of this class to an `implicit val`. The `val myAppCtx` can be used like a normal `val`, but in addition, the `implicit` keyword indicates to the compiler that this `val` is a candidate for **implicit resolution**.

The definition of the `greeting` function has an `implicit` marker on its last parameter list. When we call it without passing the `appctx` argument, the compiler tries to resolve this implicit parameter. This implicit resolution tries to find an implicit value of an `ApplicationContext` type in the current scope. The only implicit value with this type is `myAppCtx`, and hence this is the argument that is used for `appctx`.

Note that the implicit resolution is performed at compile time. If the compiler cannot resolve the implicit parameter, it will raise an error. Also, if there are several implicit values of the same type in the current scope, the compiler will be unable to decide which one to pick, and will raise an error because of ambiguous implicit values.

In a large code base, it can sometimes be difficult to know what implicit value is chosen on a given function call. Fortunately, IntelliJ can show this to you.

Position your cursor on the `greeting("hello ")` line and go to View | Implicit parameters, or press *Ctrl + Shift + P* (Linux/Windows) or *Meta + Shift + P* (macOS). You should see the following tooltip, as shown in the following screenshot:



The screenshot shows a code editor with the following Scala code:

```
case class AppContext(message: String)
implicit val myAppCtx: AppContext = AppContext("implicit world")

def greeting(prefix: String)(implicit appCtx: AppContext): String =
  prefix + appCtx.message

greeting("hello ")
```

A tooltip is displayed over the line `greeting("hello ")`. The tooltip has a title "Implicit parameters:" and a single item "myAppCtx()". The "myAppCtx()" item is highlighted with a blue background.

You can then click on the parameter shown in the tooltip. IntelliJ will jump to the declaration of the implicit value.

The arguments for the implicit parameter list can also be passed explicitly. The following call is equivalent to the previous one:

```
| greeting("hello ")(myAppCtx)
// res0: String = hello implicit world
```

When you pass the implicit arguments explicitly, the implicit resolution mechanism does not kick in.



An implicit parameter should have a type that has very few instances. It would not make sense to have an implicit parameter of a type string; there would be too many candidates for resolving it. This would make the code difficult to understand.

Implicit parameter usage

Implicit parameters are useful when you have to repeatedly pass the same argument to many functions. This happens frequently for configuration parameters.

Passing a timeout

Imagine that you implemented a `PriceService` trait that calls an external website to get the price of a product. You defined its interface as follows:

```
import cats.data.ValidatedNel
case class Timeout(millis: Int)

trait PriceService {
  def getPrice(productName: String)(implicit timeout: Timeout): ValidatedNel[String, Double]
}
```

The external website might not be responsive, and so our service has to wait a certain amount of time before giving up, indicated by the parameter `timeout`. As we have seen in [Chapter 3, Handling Errors](#), the service would return `Invalid[NonEmptyList[String]]` if there were any problem getting the price, or `Valid[Double]` if we can obtain the price.

In a large application, you could have many more services defined like this. Using an implicit parameter allows you to call these functions without having to pass the `timeout` argument each time. Furthermore, if you need to add other configuration parameters, you could add more implicit parameters without having to change all your function calls.

When you want to call the service, you would need to have `implicit val timeout: Timeout` in your scope. This gives you a good amount of flexibility, as you have total control over where you define this `timeout` and how you bring it to the current scope. The following list shows some options for this:

- You could define it only once in a `object AppConfig { implicit val defaultTimeout: Timeout = ??? }` singleton for the whole application. In this object, you could hard code its value or read it from a configuration file. When you need to call a service, all you have to do is `import AppConfig.defaultTimeout` to bring it to the current scope.
- You could use one value for production code and a different one for test code.
- You could have one part of your application using one value, say for fast services, and another part using a different value for slow services.

Passing an application context

If you have many other configuration parameters to pass to your functions, it can be more convenient to put them in an `ApplicationContext` class and declare an implicit parameter of this class in your functions. The added benefit is that this context can not only store configuration parameters, but it can also hold references to commonly used service classes. This mechanism can effectively replace dependency injection frameworks, such as Spring or Guice, that you might have used with Java.

For instance, say that we have an application that implements a `DataService` trait. It has two methods that can load and save `Product` objects from a database, as shown in the following code:

```
case class Product(name: String, price: Double)

trait DataService {
  def getProduct(name: String): ValidatedNel[String, Product]
  def saveProduct(product: Product): ValidatedNel[String, Unit]
}
```

We would typically have two implementations of this trait:

- One in the production code that will interact with a real database.
- One in the test code that will save a `Product` in memory for the duration of the test. This will allow us to run tests more quickly and to keep the tests independent from any external system.

We can then define an `AppContext` class, as follows:

```
class AppContext(implicit val defaultTimeout: Timeout,
                 val priceService: PriceService,
                 val dataService: DataService)
```

This context would have different implementations for the production and the test code. This will let you implement complex functions without having to connect to the database or to an external service when running your tests. For instance, we can implement an `updatePrice` function using an `implicit appContext` parameter:

```
import cats.implicits._  
def updatePrice(productName: String)(implicit appContext: AppContext)  
: ValidatedNel[String, Double] = {  
    import appContext._  
    (dataService.getProduct(productName),  
     priceService.getPrice(productName)).tupled.andThen {  
        case (product, newPrice) =>  
            dataService.saveProduct(product.copy(price = newPrice)).map(_ =>  
                newPrice  
            )  
    }  
}
```

This function loads a product from the database, obtains its new price by calling the `priceService`, and saves the product with the updated price. It will return a `Valid[Double]` containing the new price, or `Invalid[NonEmptyList[String]]` containing the errors if any of the services go wrong. When writing the unit test for this function, we would pass `AppContext`, which holds fake implementations of `PriceService` and `DataService`.

Examples in the SDK

The **Scala Development Kit (SDK)** makes use of implicit parameters in various places. We will explore some common uses that you will encounter as a more experienced Scala developer.

breakOut

The definition of several methods on the Scala collection API, such as `map`, has an implicit parameter of a `canBuildFrom` type. This type is used to build a collection of the same type as the input type.

For instance, when you call `map` on a `vector`, the return type will still be a `vector`, as shown in the following code run in the REPL:

```
| val vec = Vector("hello", "world").map(s => s -> s.length)
| // vec: scala.collection.immutable.Vector[(String, Int)] =
| // Vector((hello,5), (world,5))
```

When you position your cursor in IntelliJ at the `map` method and press cmd + left-click, you will see that `map` is declared in `TraversableLike`, as follows:

```
| def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That = {...}
```

The `TraversableLike` trait is a super trait of many Scala collections, such as `vector`, `List`, `HashSet`, and many more. It implements many methods that are common to all these collections. In many of these methods, the `bf: CanBuildFrom` parameter is used to build a collection of the same type as the original collection. If you jump to the definition of `CanBuildFrom`, you will see that it has three type parameters, as shown in the following code:

```
| trait CanBuildFrom[-From, -Elem, +To]
```

The first parameter, `From`, is the type of the original collection (`vector`, `List`, and many more). The second parameter, `Elem`, is the type of the elements contained in the collection. The third parameter, `To`, is the type of the target collection.

Going back to our example, this means that when we called `.map` on our `vector`, an implicit parameter of the `CanBuildFrom` type was passed. We can see where it is declared by once more positioning our cursor at the `map` method, and going to View | Implicit parameters, or pressing cmd + shift + P. If we then click on the tooltip text, we will jump to this definition in `vector.scala`, as shown in the following code:

```
| object Vector extends IndexedSeqFactory[Vector] {
```

```
| ... implicit def canBuildFrom[A]: CanBuildFrom[Coll, A, Vector[A]] = ...
```

We can see that the `to` target parameter in `canBuildFrom` is of the `Vector[A]` type. This explains why the `vec` variable in our example is of the `Vector[(String, Int)]` type.

This mechanism is quite complicated, but you do not have to understand it in detail unless you want to implement your own collection types. The SDK does a good job of hiding these details when you are just a user of the library.

However, one thing that is useful to remember is that you can pass a different `canBuildFrom` parameter to avoid unnecessary transformations. For instance, imagine that we want to build `Map[String, Int]`, where the key is a string and the value is the length of that string. Going back to our example, the most immediate way would be to call `.toMap`, as shown in the following code:

```
| val vec = Vector("hello", "world").map(s => s -> s.length)
| vec.toMap
| // res0: scala.collection.immutable.Map[String,Int] = Map(hello -> 5, world -> 5)
```

The problem with this approach is that it will iterate twice through the elements of the `Vector` class: once for mapping the elements, and once for building `Map`. On a small collection, this is not an issue, but on large collections, the performance can be affected.

Fortunately, we can build our `Map` in one iteration. If we pass the special `breakOut` object when calling `map`, the target type of `CanBuildFrom` will be the type of the receiving variable, as shown in the following code:

```
| import scala.collection.breakOut
| val map: Map[String, Int] = Vector("hello", "world").map(s => s -> s.length)(breakOut)
| // map: Map[String,Int] = Map(hello -> 5, world -> 5)
```

This simple trick can improve the performance of your application, and does not degrade the readability.



Collection transformation operations, such as `.toMap`, `.toVector`, and many more, can often be removed. Try to pass `breakout` in the previous transformation; it will save one iteration.

executionContext

The `Future` class in the Scala SDK allows you to run computations asynchronously. We will explore this in more detail in [Chapter 6, *Online Shopping – Persistence*](#), but in this section, we will explore how it makes use of implicit parameters to provide an execution context.

Open a Scala console and enter the following code. It should create a `Future` computation that will print the name of the current thread when it gets executed:

```
| scala> import scala.concurrent.Future  
| import scala.concurrent.Future  
| scala> Future(println(Thread.currentThread().getName))
```

Since we are missing an implicit in our scope, you should see the following error:

```
| <console>:13: error: Cannot find an implicit ExecutionContext. You might pass  
|   an (implicit ec: ExecutionContext) parameter to your method  
|   or import scala.concurrent.ExecutionContext.Implicits.global.  
|     Future(println(Thread.currentThread().getName))
```

The compiler tells us that we must have an `implicit ExecutionContext` in scope. An `ExecutionContext` is a class that can execute some computations asynchronously, typically using a thread pool. As suggested by the compiler, we can use the default execution context by importing

`scala.concurrent.ExecutionContext.Implicits.global`, as shown in the following code:

```
| scala> import scala.concurrent.ExecutionContext.Implicits.global  
| import scala.concurrent.ExecutionContext.Implicits.global  
| scala> import scala.concurrent.Future  
| import scala.concurrent.Future  
  
| scala> Future(println(Thread.currentThread().getName))  
scala-execution-context-global-11  
res1: scala.concurrent.Future[Unit] = Future(Success(()))
```

In the previous snippet, the value of `res1` could be as:

 `res1: scala.concurrent.Future[Unit] = Future(<not completed>)`

As this is `Future`, we don't know when it's going to finish; it will depend on your machine.

We can see that the name of the thread used to execute our `println` statement was `scala-execution-context-global-11`. If we want to run our computation using a

different thread pool, we can declare a new `ExecutionContext`. Restart the Scala console and enter the following code:

```
scala> import scala.concurrent.Future
import scala.concurrent.Future

scala> import java.util.concurrent.Executors
import java.util.concurrent.Executors

scala> import scala.concurrent.{ExecutionContext, Future}
import scala.concurrent.{ExecutionContext, Future}

scala> implicit val myThreadPool: ExecutionContext = ExecutionContext.fromExecutor(Execu
myThreadPool: scala.concurrent.ExecutionContext = scala.concurrent.impl.ExecutionContext

scala> Future println(Thread.currentThread().getName())
pool-1-thread-1
res0: scala.concurrent.Future[Unit] = Future<not completed>
```

We can observe that the thread used to run our code is now coming from a different thread pool. Many methods on `Future` have an implicit `ExecutionContext` parameter. By changing the implicit in scope, you can control how the asynchronous computations get executed.

This is especially useful when using some database drivers—you would typically use a separate thread pool to query the database, with one thread per database connection. On the other hand, CPU-bound computations can use the default thread pool, which will be initialized with the number of CPU cores available on your machine.

Implicit conversion

An implicit conversion converts a source type to a target type. This allows you to do the following:

- Use methods of the target type as if they were declared on the source type
- Pass the source type as an argument in functions that accept the target type

For instance, we could treat the `String` type as if it were `LocalDate` using the following code:

```
import java.time.LocalDate  
implicit def stringToLocalDate(s: String): LocalDate = LocalDate.parse(s)
```

Note that IntelliJ highlights a yellow warning Advanced language feature: implicit conversion on the `implicit` keyword.

If you want to get rid of this warning, you can position the cursor on the `implicit` keyword, then press *Alt + Enter*, and choose to Enable implicit conversion.

After this declaration, if we have a `String` object that can be parsed into a `LocalDate`, we can call any of the methods normally available on `LocalDate`, as shown in the following code:

```
"2018-09-01".getDayOfWeek  
// res0: java.time.DayOfWeek = SATURDAY  
  
"2018-09-01".getYear  
// res1: Int = 2018
```

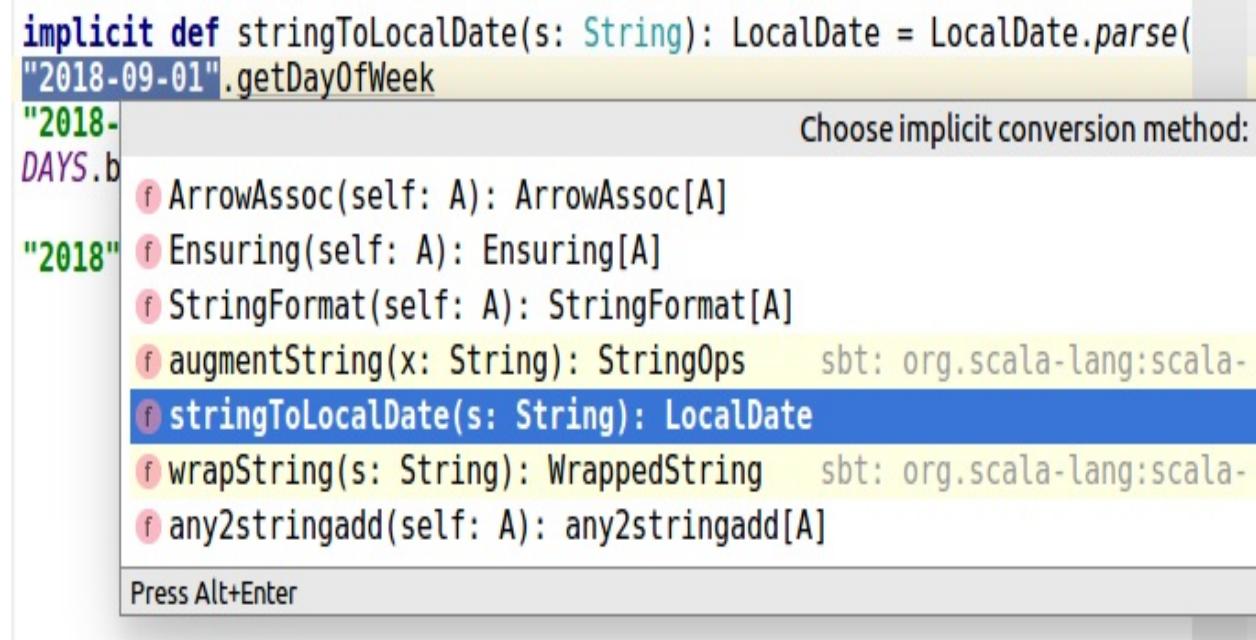
We can also call functions that accept `LocalDate` using normal strings as arguments, as shown in the following code:

```
import java.time.temporal.ChronoUnit.DAYS  
DAYS.between("2018-09-01", "2018-10-10")  
// res2: Long = 39
```

This looks a bit like magic, and it is indeed not easy to spot that there is an implicit conversion happening when reading the code. Fortunately, IntelliJ can help us.

First, you might notice that the `getDayOfWeek` and `getYear` methods are underlined. This is to show that the method is defined on an implicitly converted type.

IntelliJ can also help us find where the implicit conversion is defined. Position your cursor on one of the strings and press `ctrl + Q` on macOS (or click on Navigate | Implicit conversion). You should see the following popup:



The popup highlights the implicit conversion function that is applied. You can then click on it to jump to its declaration. Note that IntelliJ also shows some other possible implicit conversions coming from the SDK.

This conversion to `LocalDate` may appear to be quite nice; however, if we use a string that cannot be parsed, the code will throw exceptions at runtime, as shown in the following code. As we have seen in [Chapter 3, Handling Errors](#), this is best avoided:

```
"2018".getMonth  
// java.time.format.DateTimeParseException: Text '2018' could not be parsed at index 4
```

This example of implicit conversion was for illustrative purposes only.

Since our conversion can throw exceptions, it would make the code unsafe if we were to use it in production code. It can, however, be useful for writing more concise unit tests.



Implicit conversions are very powerful, and with great power comes great responsibility! It is not recommended that you define implicit conversions from common types of the SDK (`string`, `int`, and many more) to other SDK types. It can quickly make your code difficult to read.

Implicit class

Implicit conversions are often used to add additional methods to an existing type. This is called the **pimp my library** pattern. For instance, if we want to add a `square` method on the `Int` type, we can proceed as follows. Type the following code in a Scala console:

```
scala> class IntOps(val i: Int) extends AnyVal {
    def square: Int = i * i
}
scala> implicit def intToIntOps(i: Int): IntOps = new IntOps(i)
intToIntOps: (i: Int)IntOps
scala> 5.square
res0: Int = 25
```

The `5` gets implicitly converted to `IntOps`, which provides the `square` method.

Note that `IntOps` extends `AnyVal`. This extension makes it a **value class**. The benefit of a value class is that the compiler will avoid allocating a new object when we call the `square` method. The produced bytecode will be as efficient as if `square` was defined directly inside the `Int` class. The compile-time type is `IntOps`, but the runtime type will be `Int`.



One of the limitations of value classes is that they must be defined in the top level of a file or inside an object. If you try to run the preceding code in a Scala worksheet, you will get a `value class may not be a member of another class compilation error`. This is a consequence of the way Scala worksheets are evaluated—the code inside a worksheet belongs to a non-static object.

This pimp my library pattern is very useful whenever you want to add new capabilities to classes you cannot change, such as the following:

- Classes that are part of the SDK or that come from a third-party library.
- For your own classes, you can make some methods accessible from a server module, but not from a client module.

Scala offers some syntactic sugar to make this pattern more concise. We can rewrite the preceding code with an implicit class definition as follows:

```
scala> implicit class IntOps(val i: Int) extends AnyVal {
    def square: Int = i * i
```

```
| }  
| scala>5.square  
| res0: Int = 25
```

The compiler transforms an implicit class declaration into a class and implicit conversion. The two forms are equivalent.

This pattern is commonly used in the SDK, especially for "pimping" classes coming from the Java Development Kit.

For instance, `java.lang.String` can be pimped by `scala.collection.immutable.StringOps`, as shown in the following code:

```
"abcd".reverse  
| val abcd: StringOps = Predef.augmentString("abcd")  
| abcd.reverse
```

In the first line, we call the `reverse` method, which is a pimped method coming from `StringOps`. By underlining the `reverse` method, IntelliJ shows you that it is not a method defined on `java.lang.String`. If you move your cursor to the first string, `"abcd"`, and press *Ctrl + Shift + Q*, you should see a pop-up window showing you that `"abcd"` is implicitly converted to `StringOps` using `Predef.augmentString`.

In lines 2 and 3, we are showing you how we could explicitly convert our string to `StringOps` and call the same method. This is for illustrative purposes; in a real project, you would only rely on the implicit conversion.

How are implicits resolved?

So far, we have declared implicit values and implicit conversions in the same scope as where they were used. But we can define them in other files too.

The Scala compiler has a set of rules to find implicit parameters or implicit conversions. The compiler goes through the following steps and adheres to the following rules:

1. Look at the current scope as follows:

- **Implicits defined in the current scope:** These should be in the same function or class or object. This is how we defined it in the previous section.
- **Explicit import:** You can define an implicit value `implValue` in an object `myObj`, and bring it to the current scope with the statement `import myObj.implValue`.
- **Wildcard import:** `import myObj._`.

2. Look at the associated types:

- **Companion object of the source type:** For instance, in the companion object of `Option`, there is an implicit conversion to `Iterable`. This allows you to call any method of `Iterable` on an `Option` instance. Also, if a function expects an `Iterable` parameter, you can pass an `Option` instance.
- **Companion object of a parameter's type:** For instance, if you call `List(1, 2, 3).sorted`, the `sorted` method actually takes an implicit parameter of the `Ordering[Int]` type. This implicit value can be found in the companion object of `Ordering`.
- **Companion object of a parameter's type parameter:** When a function's parameter has an `A` type parameter, for instance, `Ordering[A]`, the companion object of `A` is searched for implicits. Here is an example:

```
case class Person(name: String, age: Int)
object Person {
    implicit val ordering: Ordering[Person] = Ordering.by(_.age)
}
List(Person("Omer", 40), Person("Bart", 10)).sorted
```

In this example, the `sorted` method expects an implicit parameter of an `Ordering[Person]` type, which can be found in the companion object of the `Person` type parameter.

Summary

We have covered quite a lot of material in this chapter. You learned how to improve performance using the `lazy` variable, and learned about covariance and contravariance. You also learned how to partially apply functions using currying techniques, and finally, we went through all the different ways of using `implicit` in Scala. Some concepts, such as currying, are also used in other functional programming languages, such as Haskell.

In the next chapter, we are going to go deeper into type theory by introducing the type classes. Type classes is the concept of grouping the same types sharing a common behavior.

Type Classes

In this chapter, you will learn about concepts that are built on top of Scala. The concepts in this chapter will be abstract, and they will require some concentration to understand; do not feel bad if you do not get it right away. Each individual part is relatively easy to understand, but when you put them all together, things can get complicated.

We will be focusing on type classes, with a definition for each one. They will be followed by an example illustrating how type classes can be useful in typical programs. As these concepts can be difficult, we also suggest some optional exercises that can strengthen your understanding. You do not have to do them to be able to follow the rest of the chapter. The solutions to the exercises are available on GitHub.

The majority of type classes presented here come from a library called Cats, created by Typelevel.

In this chapter, we will cover the following type classes:

- `scala.math.Ordering`
- `org.scalactic.Equality`
- `cats.Semigroup`
- `cats.Monoid`
- `cats.Functor`
- `cats.Apply`
- `cats.Applicative`
- `cats.Monad`

Understanding type classes

A **type class** represents a group of types that share a common behavior. A type class is to a type what a class is to an object. As with traditional classes, a type class can define methods. These methods can be invoked on all types that belong to the type class.

Type classes were introduced in the Haskell programming language. However, thanks to the power of implicits, we can also use them in Scala. In Scala, type classes are not built-in language constructs (like they are in Haskell) and, as a result, we need to write a bit of boilerplate code to define them.

In Scala, we declare a type class by using `trait`, which accepts a type parameter. For instance, let's define a `Combine` type class that allows for combining two objects into one, as follows:

```
trait Combine[A] {  
    def combine(x: A, y: A): A  
}
```

Then, we can define two type class instances for `Combine`, as follows:

- One for `Int`, which will add the two arguments
- One for `String`, which will concatenate them

The code definition is as follows:

```
object Combine {  
    def apply[A](implicit combineA: Combine[A]): Combine[A] = combineA  
  
    implicit val combineInt: Combine[Int] = new Combine[Int] {  
        override def combine(x: Int, y: Int): Int = x + y  
    }  
  
    implicit val combineString: Combine[String] = new Combine[String] {  
        override def combine(x: String, y: String): String = x + y  
    }  
}
```

First, we define an `apply` constructor for our type class, which just returns the `implicit` argument. Then, we declare the type class instances by using `implicit val`. This way, the compiler will be able to automatically discover them by using the

implicit resolution rules that we saw in the previous section.

Now, we can instantiate and use our type class, as follows:

```
| Combine[Int].combine(1, 2)
| // res0: Int = 3
| Combine[String].combine("Hello", " type class")
| // res1: String = Hello type class
```

When we call `Combine[Int]`, we actually call `Combine.apply[Int]`. Since our `apply` function accepts an implicit parameter of the type `Combine[Int]`, the compiler tries to find it. One of the implicit resolution rules is to search in the companion object of the argument's type.

As we declared `combineInt` in the companion object of `Combine`, the compiler uses it as the argument for `Combine.apply`.

Once we have obtained an instance of the `Combine` type class, we can invoke its method, `combine`. When we call it with two `Int` it will sum them, and when we call it with two `String` it will concatenate them.

So far, so good; but, this is a bit cumbersome to use. It would be more practical to call `combine` as if it were a method on `Int` or `String`.

As you saw in the previous section, we can define an implicit class inside of the `Combine` object, as follows:

```
object Combine {
  ...
  implicit class CombineOps[A](val x: A)(implicit combineA: Combine[A]) {
    def combine(y: A): A = combineA.combine(x, y)
  }
}
```

This implicit class allows us to call `combine` on any type `A` that has a type class instance `combine[A]`. Therefore, we can now call `combine` on `Int` or `String`, as follows:

```
2.combine(3)
// res2: Int = 5
"abc" combine "def"
// res3: String = abcdef
```

It might not look impressive; you might say that we merely gave another name to the `+` method. The key benefit of using type classes is that we can make the `combine` method available for any other type, without having to change it.

In traditional object-oriented programming, you would have to change all classes and make them extend a trait, which is not always possible.

Type classes allow us to morph a type into another one on demand (in our example, from `Int` to `Combine`). This is what we call **ad hoc polymorphism**.

Another key benefit is that by using `implicit def`, we can generate type class instances for parameterized types, such as `Option` or `Vector`. All we have to do is add them in the `Combine Companion object`:

```
object Combine {  
  ...  
  implicit def combineOption[A](implicit combineA: Combine[A])  
    : Combine[Option[A]] = new Combine[Option[A]] {  
    override def combine(optX: Option[A], optY: Option[A]): Option[A] =  
      for {  
        x <- optX  
        y <- optY  
      } yield combineA.combine(x, y)  
  }  
}
```

As long as we have an `implicit Combine[A]` for a type parameter `A`, our function, `combineOption`, can generate `Combine[Option[A]]`.

This pattern is extremely powerful; it lets us generate type class instances by using other type class instances! The compiler will automatically find the right generator, based on its return type.

This is so common that Scala provides some syntactic sugar to simplify the definitions of such functions. We can rewrite `combineOption` as follows:

```
implicit def combineOption[A: Combine]: Combine[Option[A]] = new Combine[Option[A]] {  
  override def combine(optX: Option[A], optY: Option[A]): Option[A] =  
    for {  
      x <- optX  
      y <- optY  
    } yield Combine[A].combine(x, y)  
}
```

Having a function that has a type parameter declared with `A: MyTypeClass` is equivalent to having an implicit parameter of the type `MyTypeClass[A]`.

However, when we use this syntax, we do not have a name for that implicit; we just have it in the current scope. Having it in scope is sufficient to call any other function that accepts an implicit parameter of the type `MyTypeClass[A]`.

That is why we can call `combine.apply[A]` in the preceding example. With this `combineOption` definition, we can now call `combine` on `option`, too:

```
| Option(3).combine(Option(4))
| // res4: Option[Int] = Some(7)
| Option(3) combine Option.empty
| // res5: Option[Int] = None
| Option("Hello ") combine Option(" world")
| // res6: Option[String] = Some>Hello world
```

Exercise: Define a type class instance for `Combine[Vector[A]]` that concatenates two vectors.

Exercise: Define a type class instance for `Combine[(A, B)]` that combines the first and second elements of two tuples. For instance, `(1, "Hello ") combine (2, "World")` should return `(3, "Hello World")`.

Type class recipe

To summarize, if we want to create a type class, we have to perform the following steps:

1. Create `trait MyTypeClass[A]`, that accepts a parameterized type `A`. It represents the type class interface.
2. Define an `apply[A]` function in the companion object of `MyTypeClass`, in order to facilitate the instantiation of type class instances.
3. Provide implicit instances of `trait` for all of the desired types (`Int`, `String`, `Option`, and so on).
4. Define an implicit conversion to an `ops` class, so that we can call methods of the type class as if they were declared in the target type (like you saw previously, with `2.combine(3)`).

These definitions can be written manually, like in the previous section.

Alternatively, you can use **simulacrum** to generate some of them for you. This has the double benefit of reducing boilerplate code and ensuring some consistency. You can check it out here: <https://github.com/mpilquist/simulacrum>.

Exercise: Use simulacrum to define the `combine` type class.

Common type classes

In a typical project, you will not create many of your own type classes. Since type classes capture behaviors that are common across several types, it is likely that someone else has already implemented a type class similar to what you need, located in a library. It is usually more productive to reuse type classes defined in the SDK (or in third-party libraries) than to try and define your own.

In general, these libraries define predefined instances of a type class for the SDK types (`String`, `Int`, `Option`, and so on). You would typically reuse these instances to derive instances for your own types.

In this section, we will present the type classes that you are most likely to encounter, and how to use them to solve day-to-day programming challenges.

scala.math.Ordering

`ordering` is an SDK type class that represents a strategy to sort the instances of a type. The most common use case is to sort the elements of a collection, as follows:

```
| Vector(1,3,2).sorted  
| // res0: scala.collection.immutable.Vector[Int] = Vector(1, 2, 3)
```

If you look at the declaration of `sorted`, you will see that it accepts an implicit, `ordering[B]`:

```
| def sorted[B >: A](implicit ord: Ordering[B]): Repr
```

When we called `sorted` on our `vector[Int]`, the compiler found an implicit value of the type `ordering[Int]` to pass to the function. This implicit was found in the companion object of `ordering`, which also defines instances for `String`, `Long`, `Option`, `Tuples`, and so on.

We can define an instance of the type class for `LocalDate`, so that we can compare dates, or sort them more easily:

```
import java.time.LocalDate  
implicit val dateOrdering: Ordering[LocalDate] =  
  Ordering.fromLessThan[LocalDate](_ isBefore _)  
import Ordering.Implicits._  
LocalDate.of(2018, 5, 18) < LocalDate.of(2017, 1, 1)  
// res1: Boolean = false  
Vector(  
  LocalDate.of(2018, 5, 18),  
  LocalDate.of(2018, 6, 1))  
.sorted(dateOrdering.reverse)  
// res2: Vector[LocalDate] = Vector(2018-06-01, 2018-05-18)
```

With `Ordering[LocalDate]` and `Ordering.Implicits` in scope, we can use the `<` operator to compare dates. `Ordering.Implicits` also defines other useful pimped methods, such as `<`, `>`, `<=`, `>=`, `max`, and `min`.

We can also easily sort `vector[LocalDate]` in reverse order by using a reversed `ordering[LocalDate]`. This is more efficient than sorting in ascending order and then reversing the vector.

Exercise: Define an `Ordering[Person]` that can order instances of `case class Person(name: String, age: Int)` from the oldest to the youngest. You will need to use `Ordering.by`.

org.scalactic.Equality

The `Equality` type class is used by ScalaTest unit tests whenever you write assertions, such as the following:

```
| actual shouldBe expected
| actual should === (expected)
```

Most of the time, you will not have to worry about it; the default instance compares types by using the `equals` method. However, whenever you compare double values or classes that contain double attributes, it becomes necessary to provide a different `Equality` instance.

Consider the following unit test:

```
class EqualitySpec extends WordSpec with Matchers with TypeCheckedTripleEquals{
  "Equality" should {
    "allow to compare two Doubles with a tolerance" in {
      1.6 + 1.8 should === (3.4)
    }
  }
}
```

This test should pass, right? Wrong! Try to run it; the assertion fails, as follows:

```
| 3.4000000000000004 did not equal 3.4
```

This is because `Double` is an IEEE754 double-precision floating point number.

As such, some decimal numbers cannot be represented exactly with `Double`. To make our test pass, we need to provide an `Equality[Double]` instance that will return true if the absolute difference between the two numbers is less than a certain tolerance:

```
class EqualitySpec extends WordSpec with Matchers with
TypeCheckedTripleEquals{
  implicit val doubleEquality: Equality[Double] =
  TolerantNumerics.tolerantDoubleEquality(0.0001)
  "Equality" should {...}
```

Run the test again; it will pass.

Exercise: Implement a type class derivation, `Equality[Vector[A]]`, so that we can

compare two `Vector[Double]` instances with a tolerance.

The following is the declaration of the equality instance that you have to implement in the exercise:

```
| class EqualitySpec extends WordSpec with Matchers with TypeCheckedTripleEquals{  
|   implicit val doubleEquality: Equality[Double] =  
|     TolerantNumerics.tolerantDoubleEquality(0.0001)  
|   implicit def vectorEquality[A](implicit eqA: Equality[A]):  
|     Equality[Vector[A]] = ???
```

Once you have done the exercise, it should make the test pass, as follows:

```
| "Equality" should {  
|   (...)  
|   "allow to compare two Vector[Double] with a tolerance" in {  
|     Vector(1.6 + 1.8, 0.0051) should === (Vector(3.4, 0.0052))  
|   }  
| }
```

For more information about floating point number encoding, please refer to <https://www.javaworld.com/article/2077257/learn-java/floating-point-arithmetic.html>.

cats.Semigroup

In the type class introduction, we defined a `combine` type class. It turns out that this type class is already defined in the Cats library. Its name is `Semigroup`; this name comes from the mathematical representation of this algebraic structure.

Open it in IntelliJ to see how it is defined:

```
/**  
 * A semigroup is any set `A` with an associative operation (`combine`).  
 */  
trait Semigroup[@sp(Int, Long, Float, Double) A] extends Any with Serializable {  
    /**  
     * Associative operation taking which combines two values.  
     */  
    def combine(x: A, y: A): A  
    (...)  
}
```

The `@sp` annotation is an optimization to avoid the boxing/unboxing of primitive types. Apart from that, the definition of `Semigroup` is the same as our `combine` type class.

Laws

The documentation mentions a very important point: the instances of the type class must implement an associative combine function.

This means that they must verify the following law:

```
| a combine (b combine c) = (a combine b) combine c
```

Most type classes in Cats have their own specific laws. The library guarantees that the type class instances that it defines verify this law. However, if you implement your own instance of a type class, it is your responsibility to verify that it does not break any law.

A user of a type class instance expects that it verifies all of the laws of the type class; it is part of the type class's contract.



A type class contract is the type class trait plus the laws.

When a type class verifies certain laws, you can reason about generic code more easily, and you can confidently apply some transformations.

For instance, if you know that the associativity law is verified, you can evaluate `a combine b combine c combine d` in parallel: one thread can evaluate `(a combine b)`, while another one evaluates `(c combine d)`.

Cats provide a `laws` module to help check your type class instances. You can check that your type class respects the laws by writing unit tests. This will not be detailed in this book; if you are interested in more information, you can go to <http://typelevel.org/cats/typeclasses/lawtesting.html>.

Usage examples

Cats provide several derivations of the `Semigroup` type class. It also declares a `|+|` operator in `SemigroupOps`, which is an alias for `combine`.

Some examples are as follows:

```
import cats.implicitly._  
1 |+| 2  
// res0: Int = 3  
"Hello " |+| "World !"  
// res1: String = Hello World !  
(1, 2, "Hello ") |+| (2, 4, "World !")  
// res2: (Int, Int, String) = (3,6,Hello World !)
```

Thanks to the built-in derivations brought up with `import cats.implicitly._`, we can combine `Int`, `String`, `Tuple2`, and `Tuple3`.

We can also combine some parameterized types, such as `Option` and `Vector`:

```
Vector(1, 2) |+| Vector(3, 4)  
// res3: Vector[Int] = Vector(1, 2, 3, 4)  
Option(1) |+| Option(2)  
// res4: Option[Int] = Some(3)  
Option(1) |+| None |+| Option(2)  
// res5: Option[Int] = Some(3)
```

For `Option`, notice that an empty `Option` is ignored whenever we combine it with a non-empty `Option`.

In [Chapter 3, Handling Errors](#), you saw that `Option` was one way of handling errors, and you learned that whenever an error message is needed, you can use `Either` instead of `Option`.

What happens, then, if we call `combine` on `Either`? Consider the following code:

```
1.asRight |+| 2.asRight  
// res6: Either[B, Int] = Right(3)  
1.asRight[String] |+| 2.asRight |+| "error".asLeft  
// res7: Either[String, Int] = Left(error)  
"error1".asLeft[Int] |+| "error2".asLeft  
// res8: Either[String, Int] = Left(error1)
```

The `|+|` (or `combine`) function returns the first argument of type `Left`. If all the

combined values are of type Right, their values are combined and put in Right.

In the first line, all the combined values are of type Right, hence the result is a Right(3), because 3 is the result of combine applied to 1 and 2.

In the second line, the first combined value of type Left is Left("error"), hence the result is also Left("error").

In the third line, the first combined value of type Left is Left("error1"), hence the result is Left("error1").

Exercise: Use the `|+|` operator to combine instances of `ValidatedNel[String, Int]`. What happens when you combine several invalid values?

cats.Monoid

A `Monoid` is `Semigroup` with an additional `empty` function, also called an **identity element**.

The following is an extract of this trait's definition in Cats:

```
trait Monoid[@sp(Int, Long, Float, Double) A] extends Any with Semigroup[A] {  
    /**  
     * Return the identity element for this monoid.  
     */  
    def empty: A
```

The `Monoid` trait extends the `Semigroup` trait. As such, it has all of the methods of `Semigroup`, plus this additional `empty` method. We have already seen several examples of the `combine` operation of `Semigroup`'s for different types.

Let's see what happens when we call `empty` for the same types, as follows:

```
import cats.implicitly._  
import cats.kernel.Monoid  
Monoid[Int].empty  
// res0: Int = 0  
Monoid[String].empty  
// res1: String =  
Monoid[Option[Double]].empty  
// res2: Option[Double] = None  
Monoid[Vector[Int]].empty  
// res2: Vector[Int] = Vector()  
Monoid[Either[String, Int]].empty  
// res4: Either[String,Int] = Right(0)
```

For each type, the `empty` element is quite natural: `0` for `Int`, `None` for `option`, and so on.

Laws

We can see why this `empty` function is called the identity element; if you combine any object with the identity element, it returns the same object, as follows:

```
((3 |+| Monoid[Int].empty) == 3  
("Hello identity" |+| Monoid[String].empty) == "Hello identity"  
(Option(3) |+| Monoid[Option[Int]].empty) == Option(3))
```

This property is formally defined by the identity laws, as follows:

- Left identity: For all x of type A , $\text{Monoid}[A].empty \mid+| x == x$
- Right identity: For all x of type A , $x \mid+| \text{Monoid}[A].empty == x$

Usage examples

This is interesting—how can we use `Monoid` in our day-to-day programs? The most compelling use case is to fold data structures. When you have `Monoid[A]`, it is trivial to combine all of the elements of `Vector[A]` to obtain one `A`. For instance, we can get the sum of all of the elements of `Vector[Int]`, as follows:

```
| Vector(1, 2, 3).combineAll  
| // res8: Int = 6
```

This is equivalent to calling `foldLeft` on `Vector`:

```
| Vector(1, 2, 3).foldLeft(0) { case (acc, i) => acc + i }
```

Indeed, `foldLeft` accepts two arguments, as follows:

- A start value, for which we can pass the monoid's empty value
- A function, for which we can pass the monoid's `combine` function

Cats also provide a `foldMap` function, which lets you transform the elements of a collection into `Monoid` before folding them:

```
| Vector("1", "2", "3").foldMap(s => (s, s.toInt))  
| // res10: (String, Int) = (123, 6)
```

Exercise: Implement an instance of `Monoid[Int]` to multiply all of the elements of a `Vector[Int]`.

Exercise: Use `foldMap` to compute the average of `Vector[Double]`.

Hint: The return type of the call to `foldMap` should be `(Int, Double)`.

Higher-kinded types

Before exploring other type classes, it would be useful to be familiar with the concept of **higher-kinded** types and **arities**.

You are already familiar with values and functions. A value is a literal or an object, such as `1`, `false`, or `"hello world"`.

Arity

A function takes one or many values as parameters and returns another value.

The **arity** of a function is the number of parameters it takes.

For instance:

- A **nullary** (arity 0) function does not take any parameter
- A **unary** (arity 1) function takes only one parameter
- A **binary** (arity 2) function takes two parameters

A **type constructor** is a type that accepts parameters. It is called type constructor, because it constructs a concrete type when we pass a concrete type to it. For instance, `option[A]` is a type constructor. When we pass a concrete type `Int` to it, we obtain a concrete type `option[Int]`.

As type constructors can accept 0 to n arguments, the concept of arity also applies here:

- A nullary type does not take any parameter. It is a concrete type—`Int`, `Boolean`, and many more
- A unary type takes one parameter—`option[A]`, `vector[A]`
- A binary type takes two parameters—`Either[L, R]`, `Map[K, V]`
- A ternary type takes three parameters—`Tuple3[A, B, C]`

Higher-order function

The **order** of a function is the nesting depth of function arrows:

- Order 0—values, for instance, `1`, `false` or `"hello"`
- Order 1—functions `A => B`, for instance `def addOne: Int => Int = x => x + 1`
- Order 2—higher-order functions `A => B => C`, for instance:

```
| def map: Vector[Int] => (Int => Int) => Vector[Int] =  
|   xs => f => xs.map(f)
```

- Order 3—higher-order functions `A => B => C => D`

Any function of an order strictly greater than 1 is a higher-order function.

Higher-kinded types

It turns out that similar concepts exist with types and **kinds**:

- The kind of ordinary types such as `Int` or `Boolean` is `*`
- The kind of unary type constructors is `* -> *`, for instance, `Option` or `List`
- The kind of binary type constructors is `(*, *) -> *`, for instance, `Either` or `Map`

Similarly to functions and order, we can order kinds by the number of type arrow `->` they have:

- Order 0 (`*`): Ordinary types such as `Int`, `Boolean`, or `String`
- Order 1 (`* -> *` or `(*, *) -> *`): Type constructors `Option`, `Vector`, `Map`, and many more
- Order 2 (`(* -> *) -> *`): Higher-kinded types `Functor`, `Monad`, and many more

A **higher-kinded** type is a type constructor that has strictly more than one arrow `->`. In the following section, we are going to explore type classes that are defined using higher-kinded types.

Cats.Functor

`Functor` is a **higher-kinded type** of **arity one**. It accepts a unary type parameter `F[_]`. In other words, its type parameter must be a type that has a type parameter itself; for instance, `Option[A]`, `Vector[A]`, `Future[A]`, and so on. It declares a `map` method that can transform the elements inside of `F`. Here is a simplified definition of `cats.Functor`:

```
| trait Functor[F[_]] {  
|   def map[A, B](fa: F[A])(f: A => B): F[B]
```

This should be familiar. We have already seen several classes of the SDK that define a `map` function doing the same thing: `Vector`, `Option`, and so on. Hence, you might wonder why you would ever need to use an instance of `Functor[Option]` or `Functor[Vector]`; they would only define a `map` function that is already available.

One advantage of having this `Functor` abstraction in Cats is that it lets us write more generic functions:

```
| import cats.Functor  
| import cats.implicits._  
| def addOne[F[_] : Functor](fa: F[Int]): F[Int] = fa.map(_ + 1)
```

This function adds `1` for any `F[Int]` that has a `Functor` type class instance. The only thing I know about `F` is that it has a `map` operation. Hence, this function will work for many parameterized types such as `Option` OR `Vector`:

```
| addOne(Vector(1, 2, 3))  
| // res0: Vector[Int] = Vector(2, 3, 4)  
| addOne(Option(1))  
| // res1: Option[Int] = Some(2)  
| addOne(1.asRight)  
| // res2: Either[Nothing, Int] = Right(2)
```

Our function, `addOne`, applies the principle of the least power; given a choice of solutions, it picks the least powerful solution capable of solving your problem.

We use a parameter type that is more generic, and hence, less powerful (it only has one `map` function). This makes our function more reusable, easier to read, and easier to test:

- **More reusable:** The same function can be used with `option` or `vector` or `List`, or anything that has a `Functor` type class instance.
- **Easier to read:** When you read the signature of `addOne`, you know that the only thing it can do is transform the elements inside of the `F`. It cannot, for instance, shuffle the order of the elements, nor can it drop some elements. This is guaranteed by the `Functor` laws. Therefore, you do not have to read its implementation to make sure that it does not get into any mischief.
- **Easier to test:** You can test the function with the simplest type that has a `Functor` instance, which is `cats.Id`. The code coverage will be the same. A simple test would be, for instance, `addOne[cats.Id](1) == 2`.

Laws

In order to be `Functor`, the `Functor` instance's `map` function must satisfy two laws. In the rest of this chapter, we will define our laws in terms of an equality: `left_expression == right_expression`. These equalities must be true for any types and instances specified.

Given a type `F` that has a `Functor[F]` instance, for any type `A` and any instance `fa: F[A]`, the following equalities must be satisfied:

- **Identity preservation:** `fa.map(identity) == fa`. The identity function always returns its argument. Mapping with this function should not change `fa`.
- **Composition preservation:** For any function `f` and `g`, `fa.map(f).map(g) == fa.map(f andThen g)`. Mapping `f` and `g` successively is the same as mapping with a composition of these functions. This law allows us to optimize code. When we find ourselves calling `map` many times on a large vector, we know that we can replace all of the `map` calls with a single one.

Exercise: Write an instance of `Functor[Vector]` that breaks the identity preservation law.

Exercise: Write an instance of `Functor[Vector]` that breaks the composition preservation law.

Usage examples

Whenever you have a function $A \Rightarrow B$, you can lift it to make a function $F[A] \Rightarrow F[B]$, as long as you have a $\text{Functor}[F]$ instance in scope, as follows:

```
| def square(x: Double): Double = x * x
| def squareVector: Vector[Double] => Vector[Double] =
|   Functor[Vector].lift(square)
|   squareVector(Vector(1, 2, 3))
| // res0: Vector[Double] = Vector(1.0, 4.0, 9.0)
| def squareOption: Option[Double] => Option[Double] =
|   Functor[Option].lift(square)
|   squareOption(Some(3))
| // res1: Option[Double] = Some(9.0)
```

Another handy function is `fproduct`, which tuples value with the result of applying a function:

```
| Vector("Functors", "are", "great").fproduct(_.length).toMap
| //res2: Map[String,Int] = Map(Functors -> 8, are -> 3, great -> 5)
```

We created `Vector[(String, Int)]` by using `fproduct`, and then converted it to `Map`. We obtain `Map` (keyed with a word) whose associated value is the number of characters of the word.

cats.Apply

`Apply` is a subclass of `Functor`. It declares an additional `ap` function. Here is a simplified definition of `cats.Apply`:

```
trait Apply[F[_]] extends Functor[F] {
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]
  /** Alias for [[ap]]. */
  @inline final def <*>[A, B](ff: F[A => B])(fa: F[A]): F[B] =
    ap(ff)(fa)
```

This signature means that for a given context `F`, if we have a function `A => B` inside of `F`, we can apply it to `A` inside of another `F` to obtain `F[B]`. We can also use `ap` alias operator, `<*>`. Let's try it out with different `F` contexts, as follows:

```
import cats.implicits._

Option[String => String]("Hello " + _).ap(Some("Apply"))
// res0: Option[String] = Some(Hello Apply)
Option[String => String]("Hello " + _) <*> None
// res1: Option[String] = None
Option.empty[String => String] <*> Some("Apply")
// res2: Option[String] = None
```

For `F = Option`, `ap` returns a nonempty `option` only if both arguments are nonempty.

For `F = Vector`, we get the following:

```
def addOne: Int => Int = _ + 1
def multByTwo: Int => Int = _ * 2
Vector(addOne, multByTwo) <*> Vector(1, 2, 3)
// res3: Vector[Int] = Vector(2, 3, 4, 2, 4, 6)
```

In the case of `vector`, `ap` takes each element for the first `vector` and applies it to each element of the second `vector`. Hence, we obtain all of the combinations of applying every function to every element.

Exercise: Use `ap` with `Future`.

Laws

As with other Cats type classes, an `Apply` instance must obey certain laws.

Given a type `F` that has an `Apply[F]` instance, for all types `A`, given an instance `fa: F[A]`, the following equalities must be verified:

1. **Product associativity:** For all `fb: F[B]` and `fc: F[C]`, the following applies:

```
| (fa product (fb product fc)) ==  
|   ((fa product fb) product fc).map {  
|     case ((a, b), c) => (a, (b, c))  
|   }
```

We can change the parenthesis and hence the evaluation order without changing the result.

2. **ap function composition:** For all types `B` and `C`, given the instances `fab: F[A => B]` and `fbc: F[B => C]`, the following applies:

```
| (fbc <*> (fab <*> fa)) == ((fbc.map(_.compose[A] _)) <*> fab) <*> fa)
```

This is similar to the function composition law that we saw in the `Functor` section: `fa.map(f).map(g) == fa.map(f andThen g)`.

Do not get lost in reading this law; the `<*>` function is applied right to left, and the `andThen` used for a function is `.compose[A]` for `Functor`.

Exercise: Verify the product associativity for `F = Option`. You can use specific values for `fa`, `fb`, and `fc`, for instance `val (fa, fb, fc) = (Option(1), Option(2), Option(3))`.

Exercise: Verify the `ap` function composition for `F = Option`. As before, you can use specific values for `fa`, `fab`, and `fbc`.

Usage examples

This is all good, but in practice, I rarely put functions inside of contexts. I find that the function `map2` in `Apply` is more useful. It is defined in `Apply` by using `product` and `map`. The `product` defines itself, using `ap` and `map`:

```
trait Apply[F[_]] extends Functor[F] ... {
  (...)

  def map2[A, B, Z](fa: F[A], fb: F[B])(f: (A, B) => Z): F[Z] =
    map(product(fa, fb))(f.tupled)
  override def product[A, B](fa: F[A], fb: F[B]): F[(A, B)] =
    ap(map(fa)(a => (b: B) => (a, b)))(fb)
```

The `map2` object allows for applying a function to two values inside of an `F` context. This can be used to combine two values inside `F`, as follows:

```
def parseInt0(s: String): Option[Int] = Either.catchNonFatal(s.toInt).toOption
parseInt0("6").map2(parseInt0("2"))(_ / _)
// res4: Option[Int] = Some(3)
parseInt0("abc").map2(parseInt0("def"))(_ / _)
// res5: Option[Int] = None
```

In the preceding example, for `F = Option`, `map2` lets us call the function `/` if both values are nonempty.

Cats also provide an `Apply` instance for `F = Either[E, ?]`. Therefore, we can change the signature of `parseIntOpt` to return `Either[Throwable, Int]`, and the rest of the code will be the same:

```
def parseIntE(s: String): Either[Throwable, Int] = Either.catchNonFatal(s.toInt)
parseIntE("6").map2(parseIntE("2"))(_ / _)
// res6: Either[Throwable, Int] = Right(3)
parseIntE("abc").map2(parseIntE("3"))(_ / _)
// res7: Either[Throwable, Int] = Left(java.lang.NumberFormatException: For input string:
```

This `map2` function works well for two elements, but what if we have three, four, or N elements? `Apply` does not define a `map3` or `map4` function, but fortunately, Cats defines a `mapN` function on tuples:

```
(parseIntE("1"), parseIntE("2"), parseIntE("3")).mapN( (a,b,c) => a + b + c)
// res8: Either[Throwable, Int] = Right(6)
```

In [Chapter 3, Handling Errors](#), we saw that `Either` is used when we want to stop at the first error. This is what we saw in the previous example: the error mentions

that "abc" cannot be parsed, but it does not mention anything about "def".

Applying what we just learned, if we want to accumulate all errors, we can use `ValidatedNel`:

```
import cats.data.ValidatedNel
def parseIntV(s: String): ValidatedNel[Throwable, Int] = Validated.catchNonFatal(s.toInt)
  (parseIntV("abc"), parseIntV("def"), parseIntV("3")).mapN( (a,b,c) => a + b + c)
// res9: ValidatedNel[Throwable,Int] = Invalid(NonEmptyList(
// java.lang.NumberFormatException: For input string: "abc",
// java.lang.NumberFormatException: For input string: "def")
```

Exercise: Use `mapN` with `Future[Int]`. This allows you to run several computations in parallel, and to process their results when they are complete.

Exercise: Use `mapN` with `Vector[Int]`.

cats.Applicative

`Applicative` is a subclass of `Apply`. It declares an additional function, called `pure`:

```
| @typeclass trait Applicative[F[_]] extends Apply[F] {  
|   def pure[A](x: A): F[A]  
| }
```

The `pure` function puts any value of type `A` into the `F` context. A type `F` that has an instance of `Applicative[F]` and that respects the associated laws is called an **Applicative Functor**.

Let's try this new `pure` function with different `F` contexts, as follows:

```
import cats.Applicative  
import cats.data.{Validated, ValidatedNel}  
import cats.implicits._  
  
Applicative[Option].pure(1)  
// res0: Option[Int] = Some(1)  
3.pure[Option]  
// res1: Option[Int] = Some(3)  
type Result[A] = ValidatedNel[Throwable, A]  
Applicative[Result].pure("hi pure")  
// res2: Result[String] = Valid(hi pure)  
"hi pure".pure[Result]  
// res3: Result[String] = Valid(hi pure)
```

In most cases, `pure` is equivalent to the `apply` constructor. We can call it by using the function declared on the `Applicative` trait, or by calling `.pure[F]` on any type.

Laws

As you would expect, `Applicative` must conform to some laws.

Given a type `F` that has an `Applicative[F]` instance, for all types `A` given an instance `fa: F[A]`, the following equalities must be verified:

1. The Applicative identity is as follows:

$$| ((\text{identity}[A] \ _\cdot) \cdot \text{pure}[F] \ <*> \ fa) \ == \ fa$$

When we put the `identity` function in a `F` context using `pure` and call `<*>` on `fa`, it does not change `fa`. It is similar to the identity law in `Functor`.

2. The Applicative composition, given the instances `fab: F[A => B]` and `fbc: F[B => C]`, is as follows:

$$| (fbc \ <*> \ (fab \ <*> \ fa)) \ == \ ((fbc \cdot \text{map}(_\cdot \text{compose}[A] \ _\cdot) \ <*> \ fab) \ <*> \ fa)$$

It is similar to the composition preservation in `Functor`. By using `compose`, we can change the parenthesis around the `<*>` expressions without changing the result.

3. The Applicative homomorphism is as follows:

$$| \text{Applicative}[F] \cdot \text{pure}(f) \ <*> \ \text{Applicative}[F] \cdot \text{pure}(a) \ == \ \text{Applicative}[F] \cdot \text{pure}(f(a))$$

When we call `pure(f)` and then `<*>`, it is the same as applying `f` and then calling `pure`.

- The Applicative interchange, given the instance `fab: F[A => B]`, is as follows:

$$| fab \ <*> \ \text{Applicative}[F] \cdot \text{pure}(a) \ == \\ | \quad \quad \quad \text{Applicative}[F] \cdot \text{pure}((f: A \ => B) \Rightarrow f(a)) \ <*> \ fab$$

We can flip the `fab` argument of `<*>`, provided that we wrap `a` on the left side of the equality, or `f(a)` on the right side of the equality.

As an exercise, I encourage you to open the `cats.laws.ApplicativeLaws` class from the Cats source code. There are a few other laws to discover, as well as the implementations of all of the tests.

Usage examples

In the section *cats.Apply* about `Apply`, you saw that we can combine many values in an `F` context by using `mapN`. However, what if the values that we want to combine are in a collection instead of a tuple?

In that case, we can use the `Traverse` type class. Cats provide instances of this type class for many collection types, such as `List`, `Vector`, and `SortedMap`.

The following is a simplified definition of `Traverse`:

```
@typeclass trait Traverse[F[_]] extends Functor[F] with Foldable[F] with UnorderedTraver
  def traverse[G[_]: Applicative, A, B](fa: F[A])(f: A => G[B]): G[F[B]]
  ...
}
```

This signature means that I can call it with a collection, `fa: F[A]` (for instance, `vector[String]`), and a function that takes `A` and returns `G[B]`, `G` being `Applicative Functor` (for instance, `option[Int]`). It will run the `f` function on all of the values inside the `F`, and will return `F[B]` in a `G` context.

Let's see it in action with concrete examples, as follows:

```
import cats.implicitly._
def parseInt0(s: String): Option[Int] =
  Either.catchNonFatal(s.toInt).toOption
Vector("1", "2", "3").traverse(parseInt0)
// res5: Option[Vector[Int]] = Some(Vector(1, 2, 3))
Vector("1", "boom", "3").traverse(parseInt0)
// res6: Option[Vector[Int]] = None
```

We can safely parse `Vector[String]` to return `Option[Vector[Int]]`. The result will be `None` if any value cannot be parsed. In this example, we called `traverse` with `F = Vector`, `G = Option`, `A = String`, and `B = Int`.

If we want to keep some details about the parsing error, we can use `G = validatedNel`, as follows:

```
import cats.data.{Validated, ValidatedNel}
def parseIntV(s: String): ValidatedNel[Throwable, Int] = Validated.catchNonFatal(s.toInt)
Vector("1", "2", "3").traverse(parseIntV)
```

```

// res7: ValidatedNel[Throwable, Vector[Int]] = Valid(Vector(1, 2, 3))
Vector("1", "boom", "crash").traverse(parseIntV)
// res8: ValidatedNel[Throwable, Vector[Int]] =
// Invalid(NonEmptyList(
//   NumberFormatException: For input string: "boom",
//   NumberFormatException: For input string: "crash"))

```

Exercise: Use `Traverse` with `F = Future`. This will let you run a function in parallel for each element of the collection.

Another common use case is to flip a structure `F[G[A]]` into an `F[G[A]]`, using `sequence`.

The following is the definition of `sequence` in the `Cats.Traverse` trait:

```

def sequence[G[_]: Applicative, A](fga: F[G[A]]): G[F[A]] =
  traverse(fga)(ga => ga)

```

We can see that `sequence` is actually implemented using `traverse`.

The following is an example, with `F = Vector` and `G = Option`:

```

val vecOpt: Vector[Option[Int]] = Vector(Option(1), Option(2), Option(3))
val optVec: Option[Vector[Int]] = vecOpt.sequence
// optVec: Option[Vector[Int]] = Some(Vector(1, 2, 3))

```

And here is another example, with `F = List` and `G = Future`:

```

import scala.concurrent._
import ExecutionContext.Implicits.global
import duration.Duration

val vecFut: Vector[Future[Int]] = Vector(Future(1), Future(2), Future(3))
val futVec: Future[Vector[Int]] = vecFut.sequence

Await.result(futVec, Duration.Inf)
// res9: Vector[Int] = Vector(1, 2, 3)

```

The call to `sequence` returns `Future` which will complete only when the three futures inside of `vecFut` complete.

cats.Monad

`Monad` is a subclass of `Applicative`. It declares an additional function, `flatMap`, as follows:

```
| @typeclass trait Monad[F[_]] extends FlatMap[F] with Applicative[F]@typeclass trait Flat
|   def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
| }
```

This signature tells us that in order to produce `F[B]`, `flatMap` will somehow have to extract `A` inside `fa: F[A]`, and then call the function, `f`.

Previously, you saw that `Applicative` and `mapN` allow us to process several `F[A]` values in parallel, and combine them into a single `F[B]`. What `Monad` adds is the capability of processing `F[]` values in sequence: `flatMap` must process the `F` effect first, then call the `f` function.

Similar to `Functor` and `map`, many classes of the SDK already have a `flatMap` method, such as `option`, `vector`, `Future`, and so on. One advantage of having this `Monad` abstraction is that we can write functions that accept `Monad` so that it can be reused with different types.

Laws

Given a type F that has a `Monad[F]` instance, for all types A given an instance $fa: F[A]$, the following equalities must be verified:

- **All of the laws from the super traits:** See `Applicative`, `Apply`, and `Functor`.
- **FlatMap associativity:** Given two types B and C , and two functions $f: A \Rightarrow F[B]$ and $g: B \Rightarrow F[C]$, the following applies:

| $((fa \text{ flatMap } f) \text{ flatMap } g) == (fa \text{ flatMap}(f(_)) \text{ flatMap } g)$

- **Left identity:** Given a type B , a value $a: A$, and a function $f: A \Rightarrow F[B]$, the following applies:

| $\text{Monad}[F].pure(a).flatMap(f) == f(a)$

Bringing a value in the F context and calling `flatMap f` should provide the same result as calling the function f directly.

- **Right identity:**

| $fa.flatMap(\text{Monad}[F].pure) == fa$

The fa object should not change when we call `flatMap` and `pure`.

Usage examples

Suppose that you are building a program to manage a shop's inventory. The following is a simplified API to manage the items:

```
import cats.{Id, Monad}
import cats.implicits._

case class Item(id: Int, label: String, price: Double, category: String)

trait ItemApi[F[_]] {
  def findAllItems: F[Vector[Item]]
  def saveItem(item: Item): F[Unit]
}
```

The API is parametrized with an `F` context. This allows you to have different implementations of your API, as follows:

- In your unit tests, you would use `class TestItemApi extends ItemApi[cats.Id]`. If you look for the definition of `Id`, you will find `type Id[A] = A`. This means that this `TestItemApi` can directly return `Vector[Item]` in `findAllItems`, and `Unit` in `saveItem`.
- In your production code, you will need access to a database, or to call a remote REST service. These actions take time and can fail; hence, you will need to use something like `F = Future`, OR maybe `F = cats.effects.IO`. You will, for instance, define `class DbItemApi extends ItemApi[Future]`.



The `Future` class in the SDK has some issues and breaks some laws. I encourage you to use better abstractions, such as `cats.effects.IO` (<https://typelevel.org/cats-effect/datatypes/io.html>) or `monix.eval.Task` (<https://monix.io/docs/2x/eval/task.html>).

Equipped with this API, we can implement some business logic. The following is the implementation of a function that applies a discount to all items:

```
def startSalesSeason[F[_] : Monad](api: ItemApi[F]): F[Unit] = {
  for {
    items <- api.findAllItems
    _ <- items.traverse { item =>
      val discount = if (item.category == "shoes") 0.80 else 0.70
      val discountedItem = item.copy(price = item.price * discount)
      api.saveItem(discountedItem)
    }
  } yield ()
}
```

The `F[_]: Monad` type parameter constraint implies that we can call `startSalesSeason` with any `ItemApi[F]`, as long as `F` has a `Monad` instance. The presence of this implicit in scope allows us to call `map` and `flatMap` on an instance `F[A]`. Since a `for` comprehension is transformed by the compiler into a combination of `map/flatMap`, we can use a `for` comprehension to make our function more readable. In the section *cats.Applicative*, about `Applicative`, you saw that we can call `traverse` on `Vector`, as long as the function returns `F` which has an `Applicative[F]` instance. Since `Monad` extends `Applicative`, we could use `traverse` to iterate through the items and save each of them.

If you squint at this code, it looks very similar to what an imperative implementation would look like. We managed to write a purely functional function while keeping readability. The advantage of this technique is that we can easily unit test the logic of `startSalesSeason` with `F = Id`, without having to deal with `Futures`. In the production code, the same code can use `F = Future`, or even `F = Future[Either[Exception, ?]]`, and gracefully handle the process in multiple threads.

Exercise: Implement `TestItemApi`, which extends `ItemApi[Id]`. You can use a mutable `Map` to store the items. After that, write a unit test for `startSalesSeason`. Then, implement a production version of the API that extends `ItemApi[Future]`.

This approach is known as **tagless final encoding**. You can find more information about this pattern at <https://www.beyondthelines.net/programming/introduction-to-tagless-final/>.

Summary

We covered some challenging concepts in this chapter. Type classes are also used in other functional programming languages, such as Haskell.

For convenience, the following table summarizes the type classes that we enumerated in this chapter:

Name	Method	Law(s)	Example(s)
Semigroup	def combine(x: A, y: A) : A	Associativity	<pre> Option(1) + None + Option(2) // res5: Option[Int] = Some(3)</pre>
Monoid	def empty: A	Identity	<pre> Vector(1, 2, 3).combineAll // res8: Int = 6 Vector("1", "2", "3").foldMap(s => // res10: (String, Int) = (123, 6)</pre>
Functor	def map[A, B](fa: F[A])(f: A => B): F[B]	Identity, Composability	<pre> def square(x: Double): Double = x * def squareVector: Vector[Double] => Vector[Double] = Functor[Vector].lift(square) squareVector(Vector(1, 2, 3)) // res0: Vector[Double] = Vector(1.0, 4.0, 9.0) Vector("Functors", "are", "great") .fproduct(_.length) .toMap // res2: Map[String, Int] = Map(Functor, 3, Vector)</pre>
Apply	def ap[A, B](ff: F[A => B])(fa: F[A]): F[B] alias <*>	Associativity, Composability	<pre> Option[String => String] ("Hello " + _).ap(Some("Apply")) // res0: Option[String] = Some(Hello Apply) Option[String => String]("Hello " + _) // res1: Option[String] = None def addOne: Int => Int = _ + 1 def multByTwo: Int => Int = _ * 2 Vector(addOne, multByTwo) <*> Vector("1", "2", "3") // res3: Vector[Int] = Vector(2, 3, 6)</pre>
	def	Identity,	<pre> import cats.data.{Validated, Validation} def parseIntV(s: String): Validated[Int] = Vector("1", "2", "3").traverse(parseIntV)</pre>

Applicative	<code>pure[A] (x: A): F[A]</code>	Composability, Homomorphism, Interchange	<pre>// res7: ValidatedNel[Throwable, Vec[Vector("1", "boom", "crash")]] .traverse(parseIntV) // res8: ValidatedNel[Throwable, Vec[NonEmptyList[Int]]] // Invalid(NonEmptyList(// NumberFormatException: For input string: "boom" // NumberFormatException: For input string: "crash")])</pre>
Monad	<code>def flatMap[A, B] (fa: F[A]) (f: A => F[B]): F[B]</code>	Identity, Associativity, Composability, Homomorphism, Interchange	<pre>import cats.{Id, Monad} import cats.implicits._ case class Item(id: Int, label: String, price: Double, category: String) trait ItemApi[F[_]] { def findAllItems: F[Vector[Item]] def saveItem(item: Item): F[Unit] } def startSalesSeason[F[_] : Monad](api: ItemApi[F]): F[Unit] = { for { items <- api.findAllItems _ <- items.traverse { item => val discount = if (item.category == "shoes") 0.80 else 0.70 val discountedItem = item.copy(item.price * discount) api.saveItem(discountedItem) } } yield () }</pre>

If you would like to explore type classes in more detail, I encourage you to view the Cats documentation at <https://typelevel.org/cats>. I have also found it very helpful to read the source code and tests in the SDK, or in libraries such as Cats.

Cats is the main library of the Typelevel initiative, but there are many more fascinating projects hosted under this umbrella, as shown at <https://typelevel.org/projects/>.

In the next chapter, we will implement a cart for a shopping website, using frameworks that are popular in the Scala community.

Online Shopping - Persistence

In the next four chapters, we will write a project using the most common libraries and framework from the Scala ecosystem.

We are going to implement the cart management of an online shopping website, from the front-end to the database.

Starting from the database, we are going to implement a persistence layer. The responsibility of this layer is to persist in a relational database the content of the cart, for that purpose we are going to use a relational persistence framework named Slick.

Then, we are going to spend times to define an API to access the database, this API will use RESTfull web services architecture and JSON as message protocol. The API will be fully documented and testable from a generated website.

Finally, we are going to implement the user interface layer. With this interface, the user can add products into its cart, remove products and update the quantity of a particular product in the cart. Scala.js is used to implement this interface.

In this chapter, we will explain how to persist data in a relational database. The data will be the contents of a cart for a shopping website.

If we want to build a robust website accepting lots of simultaneous connection, special care is needed to have all of the layers of the solution to scale with the demand.

At the level of the persistence layer, a key point to scale would be to not overuse the system resources, more precisely the threads, each time data is written into the database. Indeed, if each request to the database is blocking a thread, the limit of concurrent connections will be reached quickly.

For that purpose, we will use an asynchronous framework called Slick to perform database actions.

In order to use the Slick framework, an introduction to Scala `Future` will be necessary. `Future` is one of the basic tools to handle asynchronous code in Scala.

As we rarely host a website at home, we are going to deploy this layer, and later the whole website, to the internet by using a cloud service provider named Heroku. This means that the shopping cart will be accessible from all over the world.

In this chapter, we will cover the following topics:

- Creating the project
- Persistence
- Deploying the application
- Heroku configuration

Creating the project

To facilitate the project creation, we are providing a template which generates the skeleton of the project. For that purpose, [Gitter8](#) will help us to generate the complete project based on a template hosted in Git.

We are not going to directly use the command line of Gitter8. We are instead going to use the integration with sbt to generate the project.

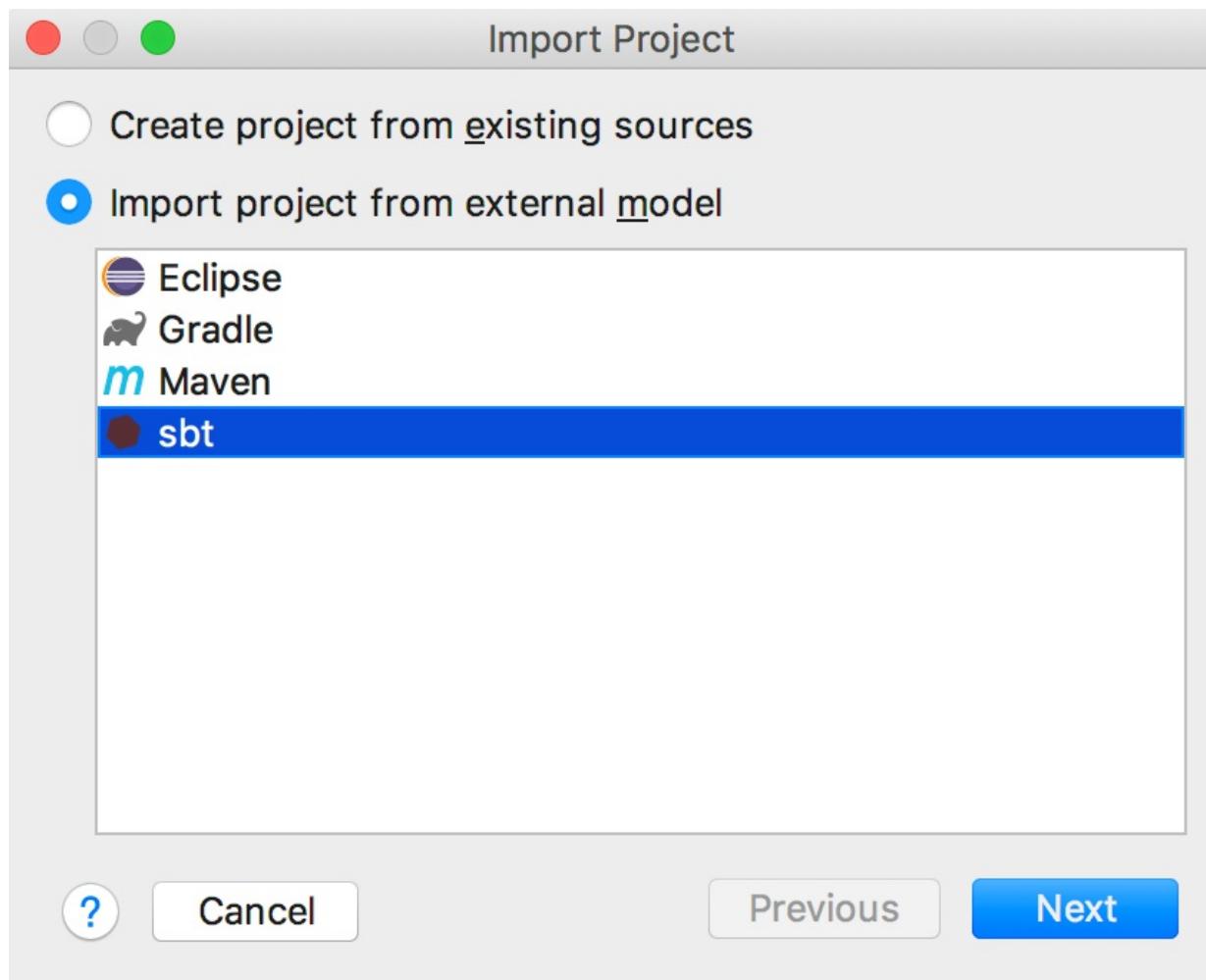
The template can be found on Github at [GitHub - scala-fundamentals/scala-play.g8: Template used for the online shopping](#). This template has been forked from <https://github.com/vmunier/play-scalajs.g8>. We essentially changed the framework test from Specs2 to ScalaTest and added all the dependencies needed for our shopping project.

To create the project, enter the following in your console:

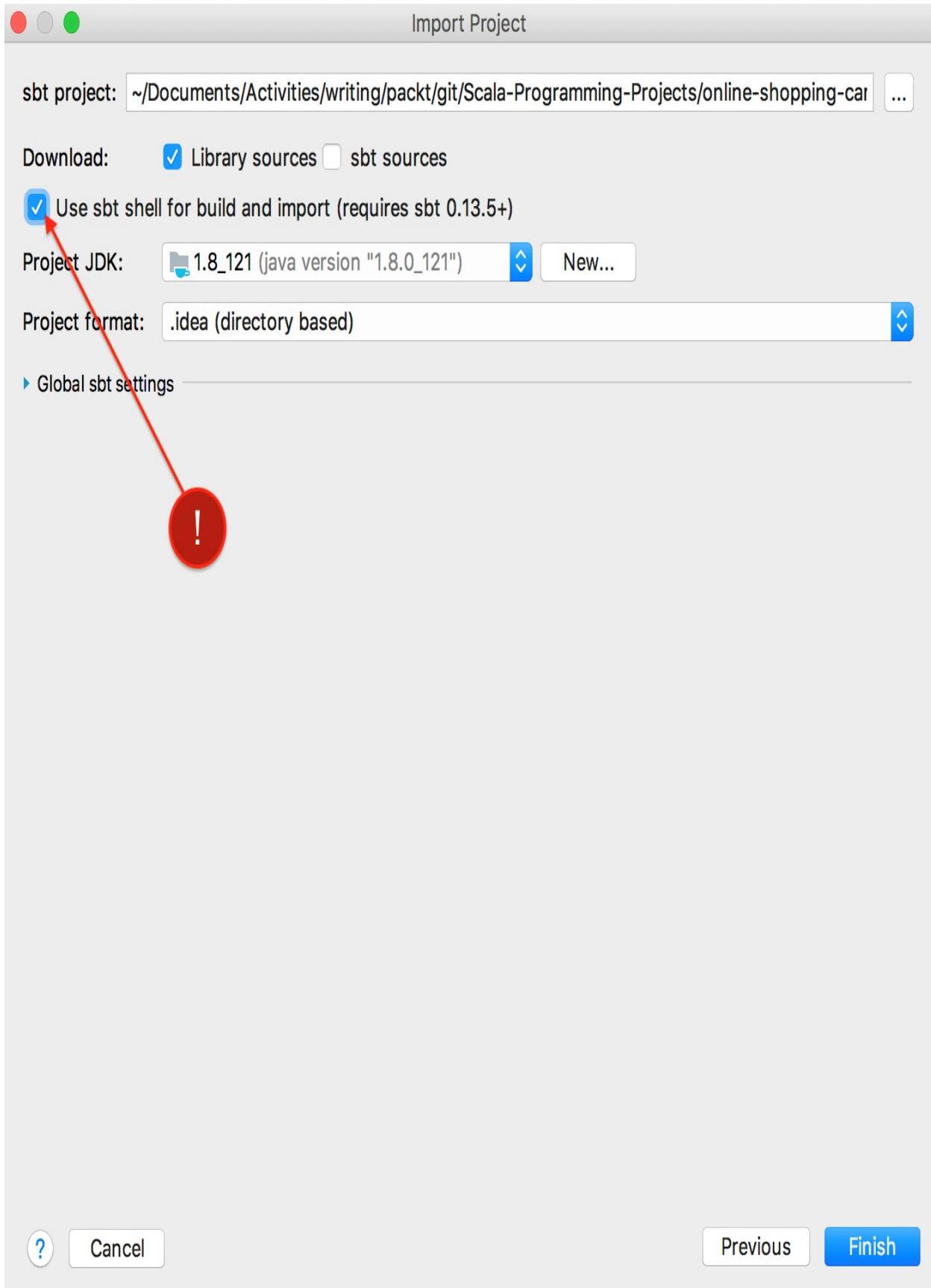
```
| sbt new scala-fundamentals/scala-play.g8 --name=shopping --organization=io.fscala
```

This will create a new folder with all the files and folders needed for our project.

You can now import this project in IntelliJ, click on Import Project select sbt on the first dialogue box:



Click on Next, on the next dialogue box, please check the option Use sbt shell for build and import (requires sbt 0.13.5+) as shown in the following:



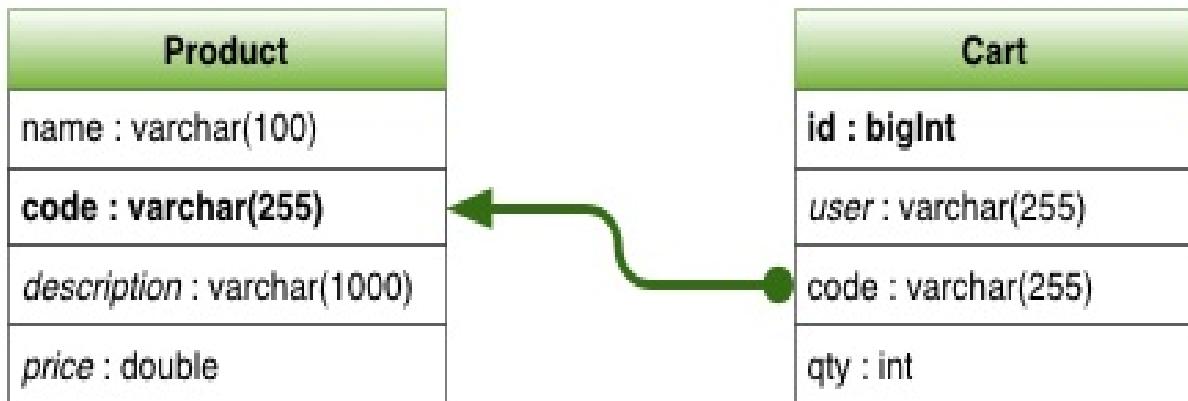
That's it, we are ready for the implementation.

Persistence

In the context of the online shopping project, we are going to create a simple data model with only two tables—the cart and product tables. The product represents what we would like to sell. It has a name, a code, a description, and a price.

The cart what a customer is about to buy. It has an ID, which is autoincremented for any new cart created, and a user, representing the user identification. For the purpose of this book, we are going to use the username sent during the login process. The cart also has a quantity and a code, representing a link to the product table.

The following diagram is a representation of our model:



For the purposes of this book, we will choose a database that requires no administration, is fast, with a small footprint, and can be deployed quickly and smoothly. The H2 database fulfills all of these requirements.

To access our data, we would like to take advantage of the Scala language to statically check our code at compile time. The Slick library is perfect for this task.

Slick can generate SQL for multiple databases, and it supports the following RDBMS (with the respective JDBC driver versions):

Database	JDBC Driver
SQLServer 2008, 2012, 2014	jTDS - SQL Server and Sybase JDBC driver (https://sourceforge.net/projects/jtds/) and Microsoft JDBC Driver 6.0 for SQL Server (https://www.microsoft.com/en-gb/download/details.aspx?id=11774)
Oracle 11g	http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
DB2 10.5	http://www-01.ibm.com/support/docview.wss?uid=swg21363866
MySQL	mysql-connector-java:5.1.23 (https://dev.mysql.com/downloads/connector/j/)
PostgreSQL	PostgreSQL JDBC Driver: 9.1-901.jdbc4 (https://jdbc.postgresql.org)
SQLite	sqlite-jdbc:3.8.7 (https://bitbucket.org/xerial/sqlite-jdbc/downloads/)
Derby/JavaDB	derby:10.9.1.0 (https://db.apache.org/derby/derby_downloads.html)
HSQLDB/HyperSQL	hsqldb:2.2.8 (https://sourceforge.net/projects/hsqldb/)

H2

com.h2database.h2:1.4.187 (<http://h2database.com/html/download.html>)

Setting up Slick

What do we need to set up Slick? In the `build.sbt` file from the project generated on the *Developing a full project* chapter, the requested libraries are set in the server section. Slick is well integrated with Play, and the full list of dependencies is as follows:

```
libraryDependencies ++= Seq(  
    "com.typesafe.play" %% "play-slick" % "3.0.0",  
    "com.typesafe.play" %% "play-slick-evolutions" % "3.0.0",  
    "com.h2database" % "h2" % "1.4.196"  
)
```

We added Slick dependencies, as well as the Evolution module. Evolution is a module that simplifies schema management; we will come back to that later.

We had to add the JDBC driver as well; this is due to the fact that Slick does not come bundled with all of the drivers.

Setting up the database

The database setup is done in the `conf/application.conf` file. When a database is needed, it must be configured in this file. Slick provides a default configuration, named `default`. For a new database, replace this name with the name of your database.

We will enable `evolution` and tell it to automatically run the scripts for database creation and deletion.

In our case, the entry looks as follows:

```
# Default database configuration
slick.dbs.default.profile="slick.jdbc.H2Profile$"
slick.dbs.default.db.driver="org.h2.Driver"
slick.dbs.default.db.url="jdbc:h2:mem:shopping"

play.evolutions.enabled=true
play.evolutions.db.default.autoApply=true
```

The complete configuration option can be found in the Play Framework documentation (<https://www.playframework.com/documentation/2.6.x/PlaySlick>).

Database test

Before we get started, we should check that the database created by `evolution` is correct, and we should define the expected behaviors.

Product test

We should also verify that three product rows are inserted when the application is started.

Create a test class, named `ProductDaoSpec`, which extends `PlaySpec`. Now, `PlaySpec` is the integration of `ScalaTest` in Play. The `ProductDaoSpec` class also needs to extend the `GuiceOneAppPerSuite` trait. This trait adds a new instance of the `Application` object to the `ScalaTest` suite:

```
class ProductDaoSpec extends PlaySpec with ScalaFutures with GuiceOneAppPerSuite {  
    "ProductDao" should {  
        "Have default rows on database creation" in {  
            val app2dao = Application.instanceCache[ProductDao]  
            val dao: ProductDao = app2dao(app)  
  
            val expected = Set(  
                Product("PEPPER", "ALD2", "PEPPER is a robot moving with wheels  
                and with a screen as human interaction", 7000),  
                Product("NAO", "ALD1", "NAO is an humanoid robot.", 3500),  
                Product("BEOBOT", "BE01", "Beobot is a multipurpose robot.",  
                159.0)  
            )  
            dao.all().futureValue should contain theSameElementsAs (expected)  
        }  
    }  
}
```

Play provides a helper method to create an instance in the cache. As you can see, `app2dao` can create an instance of `ProductDao`, which was the type parameter passed to `instanceCache`.



The matcher on the set is not strict, and it does not take into account the order of the rows received. If you would like to be stricter, ScalaTest provides the `theSameElementsInOrderAs` matcher, which checks the order of the elements in the collection.

As the `dao.all()` function returns `Future`, `ScalaTest` provides the `.futureValue` helper to wait on `Future` to finish before testing the value.

Cart test

What about the cart? We would like to make sure that the cart is empty when the application is run so that we can add items to it.

Just like we did for the product, we will create a class named `CartDaoSpec`. The test looks as follows:

```
class CartDaoSpec extends PlaySpec with ScalaFutures with GuiceOneAppPerSuite {

    "CartDao" should {
        val app2dao = Application.instanceCache[CartDao]

        "be empty on database creation" in {
            val dao: CartDao = app2dao(app)
            dao.all().futureValue shouldBe empty
        }

        "accept to add new cart" in {
            val dao: CartDao = app2dao(app)
            val user = "userAdd"

            val expected = Set(
                Cart(user, "ALD1", 1),
                Cart(user, "BE01", 5)
            )
            val noise = Set(
                Cart("userNoise", "ALD2", 10)
            )
            val allCarts = expected ++ noise

            val insertFutures = allCarts.map(dao.insert)

            whenReady(Future.sequence(insertFutures)) { _ =>
                dao.cart4(user).futureValue should contain theSameElementsAs expected
                dao.all().futureValue.size should equal(allCarts.size)
            }
        }
    }
}
```

The `be empty on database creation` test makes sure that no carts exist upon creation of the application, and `accept to add new cart` makes sure that we can insert products into a specific cart; when the cart is read back, only the product of that cart is returned. This is tested by the fact that we are adding a new product to the cart of `user2`, instead of `user1`.

For consistency, we would like to have a constraint on the database where we only have a unique tuple `user` and `productCode`. In case we do not have a unique

pair, we should expect an error to be thrown from the database saying that the cart already exists:

```
"error thrown when adding a cart with same user and productCode" in {
    val dao: CartDao = app2dao(app)
    val user = "userAdd"
    val expected = Set(
        Cart(user, "ALD1", 1),
        Cart(user, "BE01", 5)
    )
    val noise = Set(
        Cart(user, "ALD1", 10)
    )
    val allCarts = expected ++ noise
    val insertFutures = allCarts.map(dao.insert)
    recoverToSucceededIf[org.h2.jdbc.JdbcSQLException]{
        Future.sequence(insertFutures)
    }
}
```

In `expected.map(dao.insert(_)) ++ noise.map(dao.insert(_))`, we are creating a set of Future by adding the Futures from the expected cart insertion and the noise cart insertion.

To test if an error is thrown, ScalaTest provides the `recoverToSucceededIf[T]` function that tests if the `Future` passed as a parameter throws the type `[T]` error.

We would also like to test whether we can remove an item from the cart.

The following code will perform this test:

```
"accept to remove a product from a cart" in {
    val dao: CartDao = app2dao(app)
    val user = "userRmv"
    val initial = Vector(
        Cart(user, "ALD1", 1),
        Cart(user, "BE01", 5)
    )
    val expected = Vector(Cart(user, "ALD1", 1))

    whenReady(Future.sequence(initial.map(dao.insert(_)))) { _ =>
        dao.remove(ProductInCart(user, "BE01")).futureValue
        dao.cart4(user).futureValue should contain theSameElementsAs
            (expected)
    }
}
```

First, we add an initial cart with two products, and then, we remove one product from the cart. Notice that we introduced a new class, named `ProductInCart`, which represents a product in a cart.

To be complete, our `cartDao` should accept updating the product quantity in a cart; this is represented by the following code:

```
"accept to update quantities of an item in a cart" in {
    val dao: CartDao = app2dao(app)
    val user = "userUpd"
    val initial = Vector(Cart(user, "ALD1", 1))
    val expected = Vector(Cart(user, "ALD1", 5))

    whenReady(Future.sequence(initial.map(dao.insert(_)))) { _ =>
        dao.update(Cart(user, "ALD1", 5)).futureValue
        dao.cart4(user).futureValue should contain theSameElementsAs
            (expected)
    }
}
```

In this test, we first set the cart for `userUpd` with a quantity of 1 unit of `ALD1`, and then update it with a quantity of 5 units of `ALD1`.

Of course, as there is no implementation, the test does not even compile; it is time to create the database and implement the **Data Access Objects (DAO)**. Before going further, notice the piece of code with `.futureValue`. This is the perfect time to explain what Futures are all about.

Future

As you can see in the test code, `wsClient.url(testURL).get()` returns `Future`; more precisely, it returns `Future of Response (Future[Response])`.

`Future` represents a piece of code executed asynchronously. The code starts its execution upon the creation of `Future`, without knowing when it will finish its execution.

So far, so good; but how can we get the result?

Before we answer this question, there are some important points to understand. What is the purpose of writing asynchronous code?

We write it to improve performance. Indeed, if the code is run in parallel, we can take advantage of the multiple cores that are available on modern CPUs. This is all fine, but in my program, I cannot parallelize every piece of code. Some pieces are dependent on values coming from others.

Wouldn't it be nice if I could compose my code in such a way that as soon as a value is finished being evaluated, the program goes ahead and uses that variable? That is the exact purpose of `Future`. You can compose pieces of asynchronous code together; the result of the composition is another `Future`, which can be composed with another `Future`, and so on.

Getting a concrete value

OK, we can compose Futures to have new Futures but, at some point, we will need to have a concrete value instead of `Future`.

When we asked to get the response of a REST call, we receive `Future` from the function. The particularity of `Future` is that we do not know when it is going to finish so in our test we need to wait until we get the concrete value of `Future`.

To get a concrete value, you can either wait for `Future` to complete or provide a callback. Let us go into the details of both cases.

Waiting on a Future

The `Await.result` method is waiting for the result to be available. We have the possibility to give a timeout to the method so that it does not block forever.

The signature is as follows:

```
| Await.result(awaitable: Awaitable[T], atMost: Duration)
```

The first parameter waits for `Awaitable` (`Future` extends `Awaitable`), and the second is `Duration`. `Duration` is the time to wait before throwing `TimeoutException`.

This is a pretty convenient method to get the value in our test.



*If you add `import scala.concurrent.duration._` in the import section, you can use a **Domain Specific Language (DSL)** to express the duration in plain English, as follows:*

*1 second
2 minutes*

Callback

Another way to get the result is to use a callback function. In this case, we stay asynchronous to get the value. The syntax is as follows:

```
Import scala.concurrent.{Await, Future}
import scala.util.{Failure, Success}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._

val f: Future[String] = Future {
    Thread.sleep(1000)
    "Finished"
}
f.onComplete {
    case Success(value) => println(value)
    case Failure(e) => e.printStackTrace()
}
```

First, `Future` is created and assigned to `f`; then, we handle the success case, and then, the failure case.

Composing Futures

You have learned how to get a concrete value out of `Future`; you will now learn how to compose multiple `Futures`.

Using for-comprehension

`Future` has a `map` and a `flatMap` method. Therefore, as we saw in [Chapter 3, Handling Errors](#), for `Either`, we can use `Future` in a `for`-comprehension. For instance, we can define three `Future`s, as follows:

```
| val f1 = Future {1}  
| val f2 = Future {2}  
| val f3 = Future {3}
```

The `Futures` are simply successfully returning an integer. If we would like to sum up all of the integers we could write the following:

```
| val res = for {  
|   v1 <- f1  
|   v2 <- f2  
|   v3 <- f3  
| } yield (v1 + v2 + v3)
```

The `res` variable will be `Future[Int]`; hence, we can call `Await` to get the final value:

```
| val response = Await.result(res, 1 second)
```

The response would be `6`, in our case.

You just learned that inside the `for`-comprehension, the value from `Future` can be used and can be composed with other values. But there is more; you can also add an `if` condition inside, acting as a filter. Suppose that we would like to check that the three numbers from the previous example, when added together, are greater than five. If this is the case, then it should return a tuple with the three numbers; otherwise, it should return a failure. We can first define a function that takes an undefined number of parameters and returns their `sum`, as follows:

 `| def sum(v: Int*) = v.sum`
*You can define multiple parameters by adding * after the type in the parameter definition; this is called a **variadic** or `varargs` function.*

The `for`-comprehension, with our filter, will look as follows:

```
| val minExpected = 5  
|  
| val res = for {  
|   v1 <- f1
```

```
|   v2 <- f2
|   v3 <- f3
|   if (sum(v1, v2, v3) > minExpected)
| } yield (v1, v2, v3)
```

We can employ what we learned in the previous section, and use a callback to get the value, as follows:

```
| res.onComplete {
|   case Success(result) => println(s"The result is $result")
|   case Failure(e) => println("The sum is not big enough")
| }
```

The following code will be printed in the console:

```
| The result is (1,2,3)
```

However, if you set `minExpected` to 7, you should obtain the following:

```
| The sum is not big enough
```

In fact, `Future` is a failure; its representation is as follows:

```
| Future(Failure(java.util.NoSuchElementException: Future.filter predicate is not satisfied))
```

One last thing—I am sure that you noticed the following in the first piece of code that we imported:

```
| import scala.concurrent.ExecutionContext.Implicits.global
```

What is this strange import? It is the execution context, which will be covered in the next section.

Execution context

When we create a `Future`, the code is executed asynchronously on the JVM, but what is used to execute that code? In fact, the only way to execute code in parallel is to use threads. A naive approach would be to suggest: I should just create a new thread each time I would like to execute a new piece of code. However, that is a really bad idea. First, the number of threads is limited by the operating system; you cannot spawn as many threads as you want. Secondly, you could face thread starvation (<https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html>); this happens when your CPU spends all of its time switching contexts between threads, instead of executing real code.

OK, but how can you manage the thread creation? That is the purpose of the execution context. You can set the policy that you wish to manage your threads. Scala provides a default policy that creates and manages a pool of threads; the number of threads is automatically defined by the number of processors available to the JVM.

So, by importing `scala.concurrent.ExecutionContext.Implicits.global`, you are just saying that you would like to use the default policy to manage your thread, which should be fine for the majority of your code. You might need to define your own `ExecutionContext`, if, for example, you are creating `Future` that gets data from a legacy software blocking the call.

Rounding up Futures

The basic operations on `Future` are `map` and `flatMap`. In fact, when we use the `for`-comprehension, under the hood, the compiler transforms our loop with `map` and `flatMap`.

Futures are very important in Scala; we have only learned the basics of Futures; just enough to understand the code in this book. Let's stop here and go back to our shopping cart.

Database creation

Before discussing Futures, we were writing tests to check `cart` and the `Product` behavior. But, as the classes were not defined, the code was not even compiling. Let's start to create the database and then implement the DAO.

We can use Evolution to automatically create the database each time the server starts for the first time.

To do so, we need to add a script named `1.sql` in `conf/evolutions/default/`, where `default` is the database name used in the configuration file. This file is an SQL file, with a couple of tags to handle the creation and destruction of the database upon application start and stop, respectively.

We will start with the creation of the product table; the script is as follows:

```
# --- !Ups
CREATE TABLE IF NOT EXISTS PUBLIC.PRODUCTS (
    name VARCHAR(100) NOT NULL,
    code VARCHAR(255) NOT NULL,
    description VARCHAR(1000) NOT NULL,
    price INT NOT NULL,
    PRIMARY KEY(code)
);
```

In this script, we can add some default data upon database creation; this data will be used in our tests defined earlier:

```
| INSERT INTO PUBLIC.PRODUCTS (name,code, description, price) VALUES ('NAO','ALD1','NAO is
| INSERT INTO PUBLIC.PRODUCTS (name,code, description, price) VALUES ('PEPER','ALD2','PEPF
| INSERT INTO PUBLIC.PRODUCTS (name,code, description, price) VALUES ('BEOBOT','BE01','Bec
```

The next step is to add the cart table, as follows:

```
CREATE TABLE IF NOT EXISTS PUBLIC.CART (
    id BIGINT AUTO_INCREMENT,
    user VARCHAR(255) NOT NULL,
    code VARCHAR(255) NOT NULL,
    qty INT NOT NULL,
    PRIMARY KEY(id),
    CONSTRAINT UC_CART UNIQUE (user,code)
);
```

We just created the table. Notice that we added a constraint to the cart table; we

would like to have a unique row with the same `user` and `productCode`.

We explained how table creation and data insertion is done, and we can now concentrate on when and how this script is executed.

The following line, with a special meaning, can be seen in the script:

```
| # --- !Ups
```

This instruction tells Play Evolution how to create the database upon application start.

We can also tell Play how to clean up the database when the application stops. The instruction is as follows:

```
| # --- !Downs
```

In our case, when the application quits, we just delete the tables. In fact, as our database is in the memory, we don't really have to delete the table; this is just to illustrate the point:

```
| # --- !Downs
| DROP TABLE PRODUCTS;
| DROP TABLE CART;
```

You can now start the Play application. As soon as you browse `index.html` (`http://localhost:9000/index.html`), you'll notice Play asking for permission to execute the script.

Click on Apply this script now!



If you make a mistake in the script, Play Evolution will notify you of an error and will provide a Mark it resolved button after the error is fixed.

The database is now ready to be exploited. Let's create the Slick schemas and the data access layer.

Data Access Object creation

To access our database, we need to define the schema for Slick to perform queries against the database and wrap everything in a data access class.

The `ProductsDao` class is as follows:

```
class ProductsDao @Inject()(protected val dbConfigProvider: DatabaseConfigProvider)(impl
    import profile.api._

    def all(): Future[Seq[Product]] = db.run(products.result)

    def insert(product: Product): Future[Unit] = db.run(products
        insertOrUpdate product).map { _ => () }

    private class ProductsTable(tag: Tag) extends Table[Product](tag,
        "PRODUCTS") {
        def name = column[String]("NAME")

        def code = column[String]("CODE")

        def description = column[String]("DESCRIPTION")

        def price = column[Double]("PRICE")

        override def * = (name, code, description, price) <>
            (Product.tupled, Product.unapply)
    }

    private val products = TableQuery[ProductsTable]
}
```

The schema of our database is expressed with a private class, named `ProductsTable`. This class is a type definition of our table; each column (`name`, `code`, and `description`) is defined by using the `column` parameterized method. The name of the column in the database is defined by the parameter.



H2 is case sensitive by default, and it converts all of the column names to uppercase. If you change the case of the column name in the `ProductTable` definition, you will get an error saying that the column does not exist. You can change the case of a selected text in IntelliJ by hitting the keys cmd + Shift + U.

The link between this schema and our object model is established with the parameterized type of the extended `Table`. In our case, the `Product` class is as follows:

```
| case class Product(name: String,
```

```
|   code : String,  
|   description : String,  
|   price: Double)
```

This case class is defined in the `Models.scala` file, located in the `models` package.

Another interesting value is `TableQuery[ProductsTable]`, assigned to `products`. This is an object used to create queries against this table; for example, to create the query to add a product into the table, the syntax is `products += product` (with `product` being a new `product` instance).

To execute a query against the database, you will need the following two things:

- First, you will need the query; this is built by `products` (a query object generated by the `TableQuery` macro). You can build a query such as `products.result` to get all of the rows of the table, or `products.filter(_.price > 10.0)` to get all of the products with a price higher than `10.0`.
- Secondly, once you have built your query, you will need to execute it to get a materialized value. This is done by using the `db` variable defined in the `HasDatabaseConfigProvider` class. For example, to get all of the rows of the table, you can use `db.run(products.result)`.

For `products`, we only have the possibility to query all of the products and add a new `Product` to the table. This is represented by the `all()` and `insert(product: Product)` methods. In the `insert` method, after executing the query, we map the result by using `.map { _ => () }`; this is just to return `Unit` for the execution of the side effect.

You will have noticed that the return type of all of the methods is `Future`; this means that the code is executed by Slick completely asynchronously.

For the cart, the code should be more involved; indeed, we need to create a cart, add a product to it, remove a product, and even update the quantity for a product, as follows:

```
| class CartsDao @Inject()(protected val dbConfigProvider: DatabaseConfigProvider)(implicit  
|   import profile.api._  
  
|   def cart4(usr : String): Future[Seq[Cart]] =  
|     db.run(carts.filter(_.user === usr).result)  
  
|   def insert(cart: Cart): Future[Unit] = db.run(carts += cart)
```

```

def remove(cart: ProductInCart): Future[Int] =
  db.run(carts.filter(c => matchKey(c, cart)).delete)

def update(cart: Cart): Future[Int] = {
  val q = for {
    c <- carts if matchKey(c, cart)
  } yield c.quantity
  db.run(q.update(cart.quantity))
}

private def matchKey(c: CartTable, cart: CartKey): Rep[Boolean] = {
  c.user === cart.user && c.productCode === cart.productCode
}

def all(): Future[Seq[Cart]] = db.run(carts.result)

private class CartsTable(tag: Tag) extends Table[Cart](tag, "CART") {
  def user = column[String]("USER")
  def productCode = column[String]("CODE")
  def quantity = column[Int]("QTY")

  override def * =
    (user, productCode, quantity) <=> (Cart.tupled, Cart.unapply)
}

private val carts = TableQuery[CartsTable]
}

```

The model of our cart is defined by the following class:

```

abstract class CartKey {
  def user: String
  def productCode: String
}

case class ProductInCart(user:String, productCode: String) extends CartKey
case class Cart(user:String, productCode: String, quantity: Int) extends CartKey

```

In the queries, the operators are the same as the ones used in Scala, except that, for the equivalence, you will need to use the `==` operator.

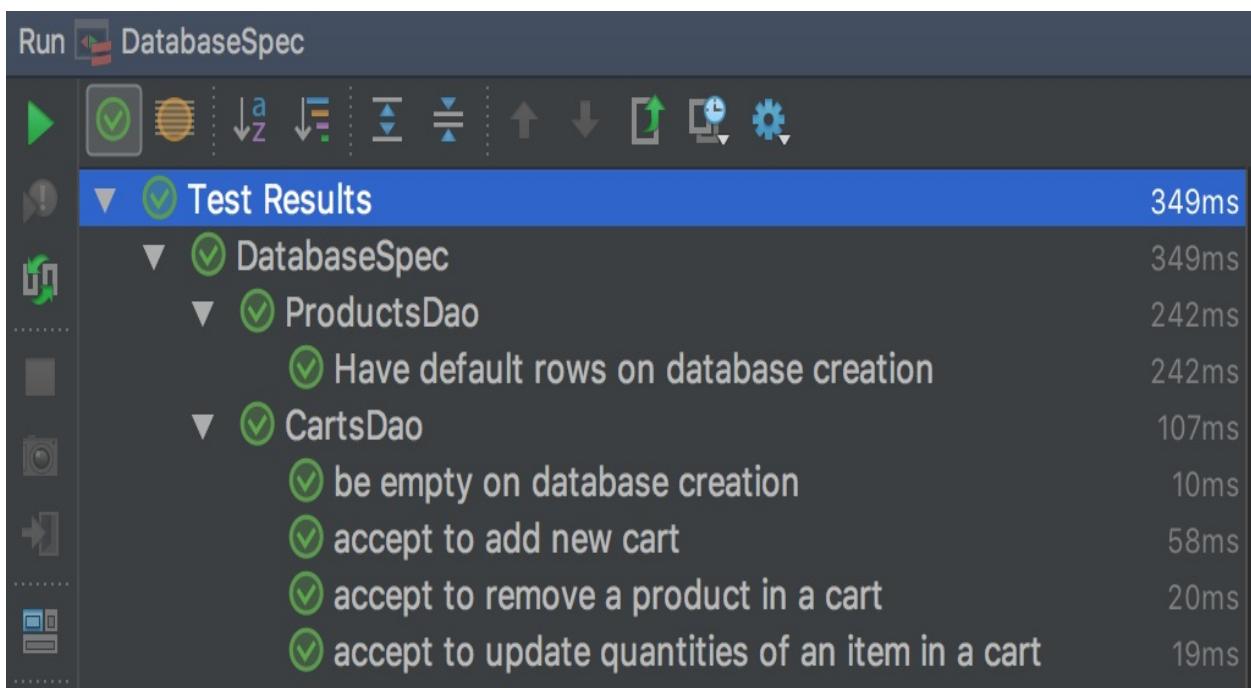
We can spend a little bit of time on the `update` method. As explained in the *Future* section, you can use a `for`-comprehension to build your query. In our case, we would like to update the quantity of a product in the cart for a specific user. We use `for`-comprehension to select the cart based on the user, and then update the quantity of the product using the quantity of the cart received as a parameter.

Running the test

We have now created the database using Evolution, configured Slick, and implemented the data access objects to access the tables.

Let's execute the tests we wrote at the beginning; they should compile and should all be successful.

Run `DatabaseSpec` to be sure, as follows:



Our persistence layer is now ready to be used. We can deploy this layer on the cloud to make sure the deployment is working smoothly.

Deploying the application

The server is running fine on our localhost, as this application is not doing much. This is the perfect time to perform all of the steps to deploy it. With each new feature, we are going to deploy it and get used to this process. This way, we can tackle the deployment issues little by little, instead of solving everything in one shot, usually under stress.

We have decided to deploy this application on Heroku.

Heroku is a **Platform as a Service (PaaS)** that supports multiple languages, including Scala. Thanks to its simplicity and versatility, the deployment process is simple and smooth.

Setting up an account

The first thing to do is create an account on the Heroku platform. Heroku provides a free account, which is perfect for our usage.

Go to the website (<https://www.heroku.com/>) and sign up for a free account. You will receive an email from Heroku to validate your account and set up a password. As soon as you set up your password, you will land on a page to create a new application; this is exactly what we want to do.

Click on the Create New App button and pick an application name. This name needs to be unique, as it will be used in the URL to reach the application on the internet. I am using `shopping-fs`; feel free to pick the name you wish. The name should be unique. If it is not, an error will tell you to change the name.

Select the region closest to your location, and click on the Create App button. The creation of the application is instantaneous, and you will be redirected directly to your Dashboard, under the Deploy tab.

Deploying your application

On the Heroku website, under the Deploy tab, at the bottom, you will see the instructions to deploy your application. The first thing to do is install the Heroku CLI; follow this link (<https://devcenter.heroku.com/articles/heroku-cli>) and pick your operating system to install the Heroku CLI.

Once the CLI has been installed, go to IntelliJ and click on the Terminal tab located at the bottom of the window. IntelliJ will set the current path of the Terminal to the root path of the current project.

From inside the Terminal, log in to Heroku by using the following command:

```
| heroku login
```

Type in your email address and password to log in.



If you are on macOS and are using Keychain Access to generate and save the password, for some reason, the password generated upon sign up is not saved on Keychain. If this is the case, just log out from the Heroku dashboard, and, on the login form, click on Forgot Password. You'll receive an email to change your password. On that page, you can use the password generation, and Keychain will remember it!

Once you have logged in, initialize Git with the following command:

```
| git init
```

Then, you will need to add the Heroku reference to Git, as follows:

```
| heroku git:remote -a shopping-fs
```

Replace `shopping-fs` with the application name that you picked previously.

You should see the following printed in your console:

```
| set git remote heroku to https://git.heroku.com/shopping-fs.git
```

Let's add the file and commit it locally in Git, as follows:

```
| git add . git commit -am 'Initial commit'
```

The final step is to deploy it with the following command:

```
| git push heroku master
```

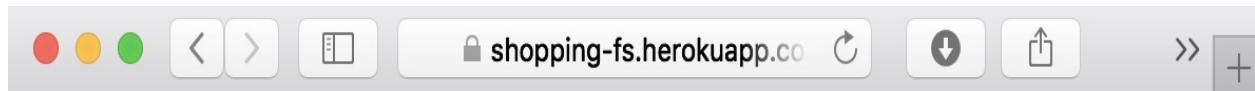
The deployment is executed on the Heroku server, and the log of the server is printed to your local console.

The process takes a little bit of time. Finally, you should see the following in the log:

```
| .....  
| remote: -----> Launching... remote: Released v4 remote:https://shopping-fs.herokuapp.com
```

That's it; your application has been compiled, packaged, and executed in the server.

Let's browse <https://shopping-fs.herokuapp.com/> to confirm it. The following page should appear:



Play and Scala.js share a same message

- Play shouts out: *It works!*
- Scala.js shouts out: *It works!*

Congratulations! You have deployed your application on the internet. Notice that your application can be reached for free on a secure HTTP protocol, with valid certificates.

Heroku configuration

How does Heroku know how to execute the application?

Heroku knows how to execute the application by reading the instructions from a file named `Procfile`, located at the root of the project; the content is as follows:

```
| web: server/target/universal/stage/bin/server -Dhttp.port=$PORT -Dconfig.file=server/cor
```

The first indication in the line is the type of application. This can be anything. The `web` value is a special type, telling Heroku that this process only receives requests from HTTP.

The second part is the path to the executable. In fact, the SBT project creates this executable for us during the build process.

The last part is a list of properties for the Play application, namely

- `-Dhttp.port`, which sets the port to listen to by using the Heroku variable, `$PORT`
- `-Dconfig.file`, which set the configuration file path to be used by the application.

Summary

In this chapter, we tackled the persistence layer. We created a simple model based on two tables, Cart and Product. We used an in-memory database named H2. We configured a framework named Slick to asynchronously access data from H2 and added a script to create the table and insert data on it. We went through the mechanism used by Play Evolution to create the database.

Tests have been written to define the behaviors of the cart and the product objects. As the data queries are done asynchronously, we spent time understanding how to deal with `Future`. Finally, we deployed this layer in the cloud using a cloud application service named Heroku.

In the next chapter, we are going to define a RESTful API to expose the data persisted in this chapter.

Online Shopping - REST API

In this chapter, we will explain how to develop a REST API using Play Framework. **API** is an acronym for **Application Programming Interface**. The **REST** acronym stands for **Representational State Transfer**. Basically, we will provide an interface for our application, so that other programs can interact with it. REST is an architectural pattern that will guide us in designing our API.

Typically, a program calling our API will be a user interface running in a browser, which we will implement in the next chapter. It could also be another backend application, which could be from another program, and so on.

In this chapter, we will cover the following topics:

- REST principles
- Implementing the API with persistence
- Swagger
- Deploying on Heroku

The REST API

The objective of the REST API is to interact with the shopping cart from the user interface in the browser. The main interactions are as follows:

- Creating the cart
- Adding, removing, and updating products in the cart

We will design our API by following REST architecture principles which was defined in 2000, by Roy Fielding.



A formal description of the REST API can be found in Fielding Dissertation, CHAPTER 5, *Representational State Transfer (REST)*, at http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

The main principles of the architecture are as follows:

- It is based on client-server architecture, which means that a server can serve multiple clients.
- It should be stateless—the server should not keep any context between client calls. The context should be kept by the client. All of the information required for the processing on the server should be part of the message sent.
- As no context is kept on the server, it should be possible to cache responses at the server level.
- Each resource on the system should be identified uniquely. Since our resources are web resources, we use a **Uniform Resource Identifier (URI)** for that purpose.

Keeping all of the preceding in mind, let's design our API. In [Chapter 6, Online Shopping – Persistence](#), two tables were defined:

- The cart
- The product

Intuitively, in this chapter, we will keep these two objects to design our API.

For the cart, we would like to perform the following actions:

- Add a product
- Delete a product
- Update the quantity of a product
- Get a list of products in the cart

For the product, we would like to perform the following actions:

- List the products
- Add a product

Writing the tests

First, let's write the tests. We will create one test per API call, to cover all the cases. By doing so, we are going to define the URI for each case. All of the test implementations will be grouped in a test class; let's call it `APISpec`.

We can create the class and define the URLs of our API; at this point, the `APISpec` class should be as follows:

```
class APISpec extends PlaySpec with ScalaFutures with GuiceOneServerPerSuite {  
    val baseURL = s"localhost:$port/v1"  
    val productsURL = s"http://$baseURL/products"  
    val addProductsURL = s"http://$baseURL/products/add"  
    val productsInCartURL = s"http://$baseURL/cart/products"  
    def deleteProductInCartURL(productId: String) =  
        s"http://$baseURL/cart/products/$productId"  
    def actionProductInCartURL(productId: String, quantity: Int) =  
        s"http://$baseURL/cart/products/$productId/quantity/$quantity"  
  
    "The API" should {  
        val wsClient = app.injector.instanceOf[WSClient]  
    }  
}
```

As in `DatabaseSpec`, we extend the `ScalaTest` integration class, `PlaySpec`, as well as a Play server, `GuiceOneServerPerSuite`, and define all of the URLs required. We defined the `wsClient` value, a helper from Play, to define a web service client.

We will start with a test of the product API, and, more precisely, with the list of products. The test is as follows:

```
"list all the products" in {  
    val response = Await.result(  
        wsClient.url(productsURL).get(),  
        1 seconds)  
    response.status mustBe OK  
}
```

`wsClient` is a convenient class to perform a REST call; we just need to set the URL and call the HTTP method.

Let's define the `add a product` case, as follows:

```
"add a product" in {  
    val newProduct =
```

```

"""
{
    "name" : "NewOne",
    "code" : "New",
    "description" : "The brand new product",
    "price" : 100.0
}
"""

val posted = wsClient.url(addProductsURL).
    post(newProduct).futureValue
posted.status mustBe OK
}

```

First, we define the new product to insert. Notice that we are using a string representation of the new product, using the JSON format. We could have defined it as an object, but that will be covered later in this chapter. To add something to the database, we are using the `HTTP POST` method.

We are now done with the product section. Now, we need to add new tests for listing all of the products in a cart, adding a product to the cart, deleting a product from the cart, and updating the quantity of a product in the cart. The corresponding unit tests are as follows:

```

"add a product in the cart" in {
    val productID = "ALD1"
    val quantity = 1
    val posted = wsClient.url(actionProductInCartURL(productID,
        quantity)).post("").futureValue
    posted.status mustBe OK
}
"delete a product from the cart" in {
    val productID = "ALD1"
    val quantity = 1
    val posted = wsClient.url(deleteProductInCartURL(productID))
        .delete().futureValue
    posted.status mustBe OK
}
"update a product quantity in the cart" in {
    val productID = "ALD1"
    val quantity = 1
    val posted = wsClient.url(actionProductInCartURL(productID,
        quantity))
        .post("").futureValue
    posted.status mustBe OK

    val newQuantity = 99
    val update = wsClient.url(actionProductInCartURL(productID,
        newQuantity)).put("").futureValue
    update.status mustBe OK
}

```

We have defined basic tests for all of the functions. When we run `Apispec`, all of the tests will fail with the error `404 was not equal to 200`. This is expected, as no

route has been defined in Play.

Defining the routes

We define all of the URLs of our API in the `config/routes` file. In this file, we define the mapping between the URL and the code. In our example, the file looks as follows:

```
# Product API
GET  /v1/products           W.listProduct
POST /v1/products/add       W.addProduct

# Cart API
GET   /v1/cart/products     W.listCartProducts()
DELETE /v1/cart/products/:id W.deleteCartProduct(id)
POST  /v1/cart/products/:id/quantity/:qty W.addCartProduct(id,qty)
PUT   /v1/cart/products/:id/quantity/:qty W.updateCartProduct(id,qty)
```

For more clarity, we squeezed the `controllers.WebServices` package to `w`, to fit the page width.

For each line, if it starts with `#`, the line is a comment; otherwise, the first column defines the HTTP action to perform, followed by the context URL. Finally, the last column is the method to call with the parameters, if any.

At the URL level, you can use the wildcard, `:`, to define a variable in the path; this variable can be used in the method call. For example, the `id` variable is defined in the `cart/products/:id` path, and then used in the `controllers.Cart.deleteProduct(id)` method call.

We have now defined the route that Play is going to create; the next step is to define the methods defined in this routing file.

To do so, create a new file, named `WebServices`, in the `controllers` folder. In this file, the implementation is as follows:

```
@Singleton
class WebServices @Inject()(cc: ControllerComponents, productDao: ProductsDao) extends
  cc.FakeApplication {
  // ***** CART Controller *****
  def listCartProducts() = play.mvc.Results.TODO
  def deleteCartProduct(id: String) = play.mvc.Results.TODO
  def addCartProduct(id: String, quantity: String) =
    play.mvc.Results.TODO
}
```

```
def updateCartProduct(id: String, quantity: String) =  
  play.mvc.Results.TODO  
  
// ***** Product Controller ***** //  
def listProduct() = play.mvc.Results.TODO  
  
def addProduct() = play.mvc.Results.TODO  
  
}
```

We have defined all of the methods, but instead of coding the details of all of the implementations, we will set it to `play.mvc.Results.TODO`. At that point, we can try to run the test, to make sure that we do not have any compilation errors.

Running the test

When running the test, `APISpec`, you should not get 404 errors anymore. However, the tests should now fail with the error 501 was not equal to 200.

This is expected. The server can now find the URL mapping for our REST calls, but in our code, all methods are implemented with `play.mvc.Results.TODO`. This special return value makes the server return the HTTP status error code 501.

What have we achieved? Well, Play is serving all of the URLs of our API. For each of them, it calls the associated method, and returns an error code instead of the real implementation!

Checking the API

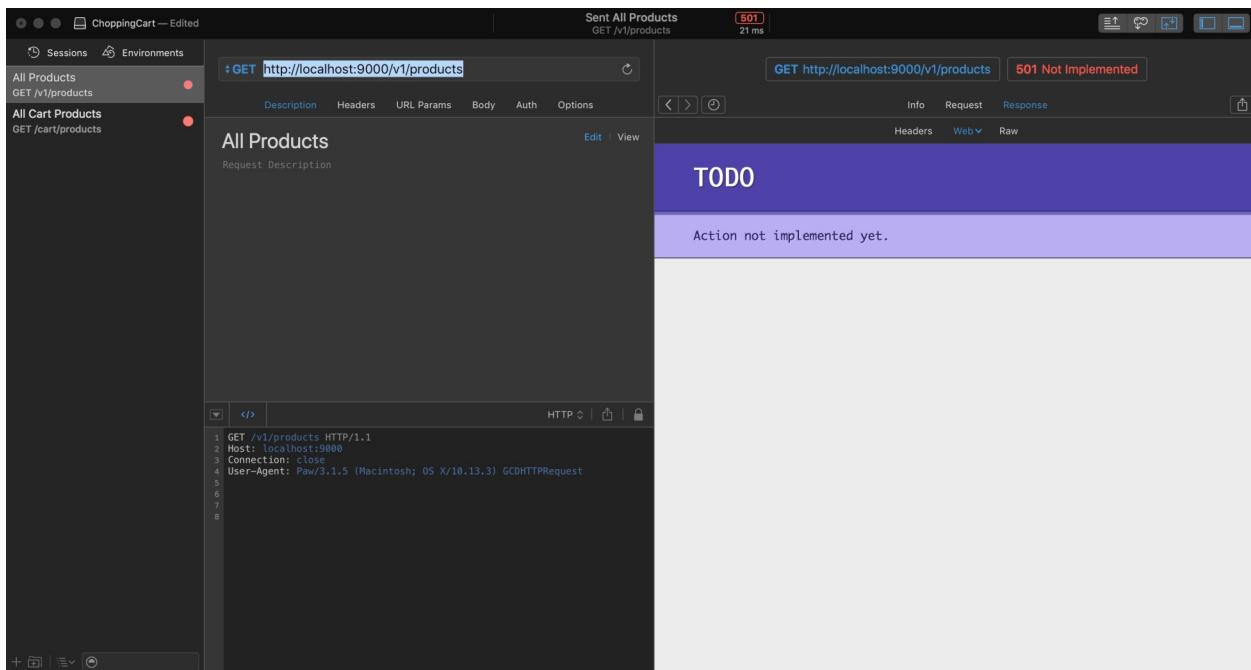
At this point, it might be interesting to introduce external tools to check the API.

Indeed, these tools are widely used and can make your life easier, especially when you want to explain it to someone or perform multiple calls on a different server.

These tools are as follows:

- **Paw:** This is a paid tool and only works on macOS. You can look upon it at <https://paw.cloud/>.
- **Postman:** This is a free and multiplatform application. Postman is a Google Chrome extension. You can look upon it at <https://chrome.google.com/webstore/detail/postman/fhbjgjbgflinjbddggehcbnccccdomop?hl=en>.

Once one of these tools installed, start the project in IntelliJ, and then browse to, for example, <http://localhost:9000/cart/products>. You should receive the error 501 Not Implemented:



Observe the error in the TODO section

All of the defined URLs will have the same behavior.

The advantage of this tool is that you can see all of the details of the request and the response. It is very useful to understand the HTTP protocol with all of the headers, URL parameters, and so on.

Implementing the API with persistence

In this chapter, we created the route for the API without an implementation. In the previous chapter, [Chapter 6, *Online Shopping – Persistence*](#), we created the database to persist the carts. It is now time to bind the API with the persistence.

Completing the product tests

We want to check not only the HTTP status but the content received back; for example, for the list of products, we would like to make sure that we are receiving all of the default products:

```
"list all the products" in {
    val response = wsClient.url(productsURL).get().futureValue
    println(response.body)
    response.status mustBe OK
    response.body must include("PEPER")
    response.body must include("NAO")
    response.body must include("BEOBOT")
}
```

In this test, we look at the body of the response and make sure that the three default products are present.

Similarly, to check the `add a product` function, we first add the product, and then call the list of products:

```
"add a product" in {
    val newProduct =
        """
        {
            "name" : "NewOne",
            "code" : "New",
            "description" : "The brand new product",
            "price" : 100.0
        }
        """

    val posted = wsClient.url(addProductsURL).post(newProduct).
        futureValue
    posted.status mustBe OK

    val response = wsClient.url(productsURL).get().futureValue
    println(response.body)
    response.body must include("NewOne")
}
```

Implementing the product API

All of the tests are ready, so let's implement the API for all the methods.

Product list

The first method that we are going to implement is the list of products. As you will remember from [Chapter 6, Online Shopping – Persistence](#), we created a list of default products upon the creation of the application. Hence, our first task will be to get these products and send them back as JSON objects.

As we have already written a data access layer, the extraction from the database is pretty simple. Indeed, we just need to call the `all()` method from the product DAO instance, as follows:

```
def listProduct() = Action.async { request =>
  val futureProducts = productDao.all()
  for (
    products <- futureProducts
  ) yield (Ok(products.mkString(",")))}
```

The `productDao.all()` method returns all of the products from the database as `Future[Seq[Product]]`. We can reuse what we learned in the previous chapter and use the `for` comprehension to extract, in the `products` variable, the sequence of products from `Future`.

From the `products` variable, thanks to `.mkString(",")`, we return a string with all of the products, separated by commas.

But we are not quite done yet. We mentioned that we would like to return a JSON representation of this sequence of products. Therefore, we need some mechanism to transform our case class instances into JSON.

We could improve the code and create it manually, but there are many good libraries that can help us to do it. It would be better to use one of them and reduce the boilerplate code.

Encoding JSON with Circe

There are many Scala libraries to manipulate JSON, so why did we choose Circe?

Circe is a very clean JSON framework with good performance, but the real reason we have chosen it is that Circe provides a complete documentation, with an explanation of all of the principles used to play with JSON. Circe uses Cats underneath, which we used in [chapter 3, Handling Errors](#). Cats is a library from Typelevel. Typelevel is a community that is extremely kind to newcomers in functional programming. It provides lots of great documentation; you can check it out at <https://typelevel.org/>. In fact, if you wish to dig deeper into functional programming, this is the place to start! The downside of Circe is the number of transitive dependencies; hence, it is fine to use it in a server application, but it might be a little heavy if you want a smaller footprint.

In order to integrate Circe with Play, we can use the integration done by Jilen at <https://github.com/jilen>. We have already added the dependency to our template, but for reference, the following needs to be added to `libraryDependencies`, in `build.sbt`:

```
| libraryDependencies += "com.dripower" %% "play-circe" % "2609.0"
```

Then, we need to add the `circe` trait to our controller, as follows:

```
| class WebServices @Inject()(cc: ControllerComponents, productDao: ProductsDao) extends A
```

We will import the required classes, as follows:

```
| import play.api.libs.circe.Circe
| import io.circe.generic.auto._
| import io.circe.syntax._
```

We are almost there; we need to replace `.mkString(",")` with `.asJSON`. That's it!

The final code is as follows:

```
| def listProduct() = Action.async { request =>
|   val futureProducts = productDao.all()
|   for(
```

```
|     products <- futureProducts  
| }     ) yield (Ok(products.asJson))
```

Now we can run `APISpec`; we should have your first working test for the API!

Action definition

In the previous code, the `for` comprehension retrieves the products from the database and converts them to JSON. We are already familiar with this kind of syntax, but what about `Action.async` and `ok()`?

In Play, all of the `Action` elements are asynchronous. The expected return of an `Action` is a status, which represents the HTTP status code (`OK() = 200 OK`, `Created() = 201 CREATED`, and so on).

As you may have noticed, the return type of the `for` comprehension is a status wrapped in `Future`. The `Action.async` helper function allows us to create an asynchronous `Action` from this `Future`.

Adding a product

The next method to implement is the ability to add a product to the database.



If you are annoyed by the errors shown by IntelliJ when you start a new implementation, you can add a dummy return status.

First, we will return a dummy status to avoid IntelliJ errors:

```
| def addProduct() = Action.async { request =>
|   Future.successful(Ok)
| }
```

Then, we will complete the implementation, as follows:

```
def addProduct() = Action.async { request =>
  val productOrNot = decode[Product]
  (request.body.asText.getOrElse(""))
  productOrNot match {
    case Right(product) => {
      val futureInsert = productDao.insert(product).recover {
        case e => {
          Logger.error("Error while writing in the database", e)
          InternalServerError("Cannot write in the database")
        }
      }
      futureInsert.map(_ => Ok)
    }
    case Left(error) => {
      Logger.error("Error while adding a product", error)
      Future.successful(BadRequest)
    }
  }
}
```

We expect to receive the new product details as a JSON payload in the body. Therefore, in the first line, we get the body as text; if it is not defined, we replace it with an empty string.

Circe provides a method, named `decode`, which takes a string as a parameter and transforms it into an object. The type parameter (`[Product]`, in our case) defines the class of the target object.

This `decode` method returns an `Either` instance. If there is an error, it will be `Left`, and if the decoding works, it will be `Right`. We can pattern match on this `Either` value to return `ok`; or, in the case of an error, to return a `BadRequest` status.

When the decoding works, we call `productDao.insert` to store the new product in the database. If anything goes wrong during the insert, the `.recover` block will return an internal server error.

Completing the cart test

The first thing that we would like to do is test the list of products in the customer's cart. But how do we make sure that the customer can only see their cart and not somebody else's?

Logging in

To solve this problem, we are going to add a `login` endpoint on the API. This endpoint will create a session cookie and will keep the session ID in it. This way, the session ID will be passed every time a request is sent to the server. The server will then be able to associate a session ID with a specific customer.

Unit test

When the client calls the login URL, the server responds with a `Set-Cookie` header. In this header, the encrypted session data can be obtained by using the `PLAY_SESSION` key.

The following is the unit test for the new `login` endpoint:

```
"return a cookie when a user logins" in {
    val cookieFuture = wsClient.url(login).post("myID").map {
        response =>
        response.headers.get("Set-Cookie").map(
            header => header.head.split(";")
                .filter(_.startsWith("PLAY_SESSION")).head
        )
    }
    val loginCookies = Await.result(cookieFuture, 1 seconds)
    val play_session_Key = loginCookies.get.split("=").head
    play_session_Key must equal("PLAY_SESSION")
}
```

The test sends a `POST` request to the `login` endpoint. For now, we send a dummy payload, `myID`, which represents a user identifier. Once posted, we map the returned response to get the `Set-Cookie` header. This `header` contains multiple values, separated by semicolons. We are not interested in the other values; hence, we need to process this `header` to get only the `PLAY_SESSION` cookie. To do so, we split on the semicolon and keep only the element starting with the `PLAY_SESSION` key.

We assign this transformed response to a value: `cookieFuture`. We then wait for `Future` to complete; then, the response is split on `=`, and only the key is kept and checked.

You can run the test now; it should fail, with a 404 Not Found error.

Implementation

First, we need to add the new endpoint to the `routes` file, as follows:

```
| # Login
| POST    /v1/login    controllers.WebServices.login
```

With this new entry, Play reacts to the `POST` action and calls the `login` method of the `Webservices` class.

The implementation of this `login` method is as follows:

```
| def login() = Action { request =>
|   request.body.asText match {
|     case None => BadRequest
|     case Some(user) => Ok.withSession("user" -> user)
|   }
| }
```

If a body with the username is present, the `OK` status is returned, with a new session cookie. The username is added in the cookie, with the `user` key, and can be retrieved on a subsequent request.

Run `APISpec` again; the login test should now be green.

Passing the cookie

From now on, every time we interact with the cart, we will have to pass the cookie with the session ID to bind our user with the cart. The test to get the list of products in the cart is as follows:

```
"list all the products in a cart" in {
    val loginCookies =
        Await.result(wsClient.url(login).post("me").map(p =>
            p.headers.get("Set-Cookie").map(_.head.split(";").head)), 1
        seconds)
    val play_session = loginCookies.get.split("=").tail.mkString("")

    val response = (wsClient.url(productsInCartURL).
        addCookies(DefaultWS.Cookie("PLAY_SESSION",
            play_session)).get().futureValue
        println(response)
        response.status mustBe OK

    val listOfProduct = decode[Seq[Cart]](response.body)
    listOfProduct.right.get mustBe empty
}
```

First, we call the `login` endpoint to build the session cookie; then, the cookie is passed in the second call. In order to check the number of products in the cart, we use Circe to transform the JSON response to a sequence of a cart.

Since the session cookie has to be used by all of the subsequent tests, we can move the code to get the cookie from a `lazy val`, as follows:

```
lazy val defaultCookie = {
    val loginCookies = Await.result(wsClient.url(login).post("me")
        .map(p => p.headers.get("Set-Cookie").map(
            _.head.split(";").head)), 1 seconds)
    val play_session = loginCookies.get.split("=").tail.mkString("")

    DefaultWS.Cookie("PLAY_SESSION", play_session)
}
```



The `lazy` keyword indicates that the code will be evaluated as late as possible, upon the first usage of the value.

Then, we can refactor our test to use it, as follows:

```
"list all the products in a cart" in {
    val response = wsClient.url(productsInCartURL)
        .addCookies(defaultCookie).get().futureValue
    response.status mustBe OK
    val listOfProduct = decode[Seq[Cart]](response.body)
    listOfProduct.right.get mustBe empty
}
```

```
| }
```

Check the addition of a product to the cart, as follows:

```
"add a product in the cart" in {
    val productID = "ALD1"
    val quantity = 1
    val posted = wsClient.url(actionProductInCartURL(productID,
        quantity)).addCookies(defaultCookie).post("").futureValue
    posted.status mustBe OK

    val response = wsClient.url(productsInCartURL)
        .addCookies(defaultCookie).get().futureValue
    println(response)
    response.status mustBe OK
    response.body must include("ALD1")
}
```

We must be able to remove a product from the cart, as follows:

```
"delete a product from the cart" in {
    val productID = "ALD1"
    val posted = wsClient.url(deleteProductInCartURL(productID))
        .addCookies(defaultCookie).delete().futureValue
    posted.status mustBe OK

    val response = wsClient.url(productsInCartURL)
        .addCookies(defaultCookie).get().futureValue
    println(response)
    response.status mustBe OK
    response.body mustNot include("ALD1")
}
```

The last test is to update the quantity of a product in the cart, as follows:

```
"update a product quantity in the cart" in {
    val productID = "ALD1"
    val quantity = 1
    val posted = wsClient.url(actionProductInCartURL(productID,
        quantity)).addCookies(defaultCookie).post("").futureValue
    posted.status mustBe OK

    val newQuantity = 99
    val update = wsClient.url(actionProductInCartURL(productID,
        newQuantity)).addCookies(defaultCookie).put("").futureValue
    update.status mustBe OK

    val response = wsClient.url(productsInCartURL)
        .addCookies(defaultCookie).get().futureValue
    println(response)
    response.status mustBe OK
    response.body must include(productID)
    response.body must include(newQuantity.toString)
}
```

As the test is now defined, let's implement the endpoints.

Listing products in cart

Something is missing from the implementation; indeed, we do not have a `cartDao` instance in our `WebService` class.

To add it, just add it as a new parameter; as all of the parameters are injected, Play will automatically do it for you. The `WebService` class definition becomes the following code:

```
| class WebServices @Inject()(cc: ControllerComponents, productDao: ProductsDao, cartsDao:
```

The implementation to get all of the products is as follows:

```
def listCartProducts() = Action.async { request =>
  val userOption = request.session.get("user")
  userOption match {
    case Some(user) => {
      Logger.info(s"User '$user' is asking for the list of product in
                  the cart")
      val futureInsert = cartsDao.all(user)
      futureInsert.map(products => Ok(products.asJson)).recover {
        case e => {
          Logger.error("Error while writing in the database", e)
          InternalServerError("Cannot write in the database")
        }
      }
    }
    case None => Future.successful(Unauthorized)
  }
}
```

The `addCartProduct` implementation is as follows:

```
def addCartProduct(id: String, quantity: String) =
  Action.async { request =>
  val user = request.session.get("user")
  user match {
    case Some(user) => {
      val futureInsert = cartsDao.insert(Cart(user, id,
                                             quantity.toInt))
      futureInsert.map(_ => Ok).recover {
        case e => {
          Logger.error("Error while writing in the database", e)
          InternalServerError("Cannot write in the database")
        }
      }
    }
    case None => Future.successful(Unauthorized)
  }
}
```

It appears that the code is the same in the `.recover` partial function of `addCartProduct` and `listCartProducts`; in order to avoid code duplication, we could extract it as follows:

```
val recoverError: PartialFunction[Throwable, Result] = {
  case e: Throwable => {
    Logger.error("Error while writing in the database", e)
    InternalServerError("Cannot write in the database")
  }
}
```

We can refactor the list of products and add a product to use the new variable. The delete product action is as follows:

```
def deleteCartProduct(id: String) = Action.async { request =>
  val userOption = request.session.get("user")
  userOption match {
    case Some(user) => {
      Logger.info(s"User '$user' is asking to delete the product
                  '$id' from the cart")
      val futureInsert = cartsDao.remove(ProductInCart(user, id))
      futureInsert.map(_ => Ok).recover(recoverError)
    }
    case None => Future.successful(Unauthorized)
  }
}
```

Finally, the update product action is as follows:

```
def updateCartProduct(id: String, quantity: String) = Action.async {
  request =>
  val userOption = request.session.get("user")
  userOption match {
    case Some(user) => {
      Logger.info(s"User '$user' is updating the product'$id' in it
                  is cart with a quantity of '$quantity'")
      val futureInsert = cartsDao.update(Cart(user, id,
                                              quantity.toInt))
      futureInsert.map(_ => Ok).recover(recoverError)
    }
    case None => Future.successful(Unauthorized)
  }
}
```

Congratulations; all of the tests are now passing!

Swagger

An API needs to be documented to be usable. Indeed, when you want to use an API, you will not want to read a complete manual beforehand. It is better to have a self-explanatory and intuitive API.

To help with the documentation and testing part, there is a useful framework: Swagger.

Swagger not only helps to write the documentation; it also allows you to test the API directly while reading the documentation. In order to visualize the documentation with the Swagger UI, you must first declare a specification file, in JSON or YAML format. This specification file defines all of the URLs and data models that constitute your API.

There are multiple ways to use Swagger, as follows:

- You can write the specification of your API using the Swagger Editor, and Swagger will generate a skeleton of the code for you
- You can add the Swagger specification directly in the `route.conf` file
- You can add annotations in your code to generate the Swagger `specification` file

For our project, we are going to generate the Swagger `specification` file by using annotations in our code.

The advantage of doing it this way is that all of the associated documentation will be in one place. It makes it easier to keep the documentation in sync, especially when code gets refactored. Many Swagger options are configurable, simply by adding annotations.

Installing Swagger

The installation is already done, thanks to the Gitter8 template we used to generate this project, so the following details are just for reference.

The integration that we use is based on the [swagger-api/swagger-play](#) GitHub repository; please refer to it for any updates. We have to add a reference to the library in `build.sbt`. The `libraryDependencies` variable must contain the following code:

```
| "io.swagger" %% "swagger-play2" % "1.6.0"
```

Then, the module must be enabled by adding the following to `application.conf`:

```
| play.modules.enabled += "play.modules.swagger.SwaggerModule"
```

From here, we can publish the JSON definition by adding the following route in the `routes` file:

```
| GET /swagger.json controllers.ApiHelpController.getResources
```

We would like to serve the API documentation directly from our server. In order to do that, we need to add the `swagger-ui` dependency to `libraryDependencies`, in `build.sbt`:

```
| "org.webjars" % "swagger-ui" % "3.10.0",
```

To get the `swagger-ui` exposed in Play, the `routes` file needs to be updated with the following:

```
| GET /docs/swagger-ui/*file controllers.Assets.at(path:String="/public/lib/swagger-ui",
```

The Swagger UI uses JavaScript with inline code. By default, the security policy of Play forbids inline code. Also, we would like to allow requests from the localhost and from Heroku where it will be deployed. Hence, the following code needs to be added to `application.conf`:

```
| play.filters.hosts {  
|   # Allow requests from heroku and the temporary domain and localhost:9000.  
|   allowed = ["shopping-fs.herokuapp.com", "localhost:9000"]  
| }
```

```
| play.filters.headers.contentSecurityPolicy = "default-src * 'self' 'unsafe-inline' data:
```

OK, the plumbing is done. It is time to add the definition of our project.

Declaring endpoints

Now that Swagger is installed in our project, we need to provide some information about our API. First, we need to add the following to `application.conf`:

```
api.version = "1.0.0"
swagger.api.info = {
  description : "API for the online shopping example",
  title : "Online Shopping"
}
```

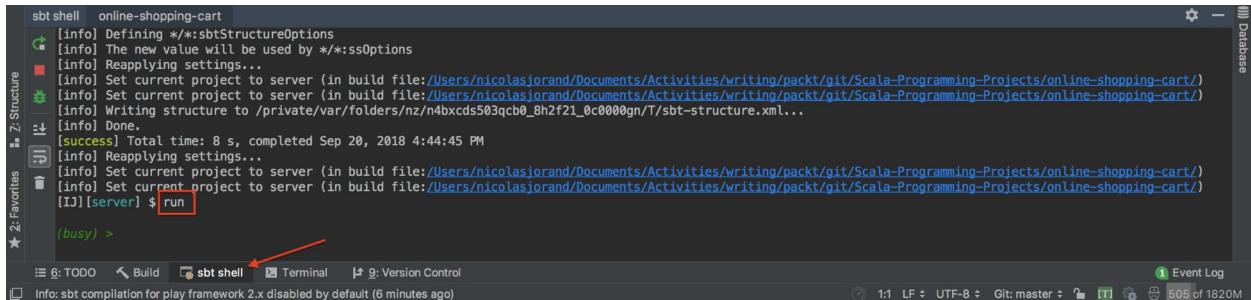
Then, we need to declare our controller. For this, we will add the `@Api` annotation to the controller named `Application`:

```
@Singleton
@Api(value = "Product and Cart API")
class WebServices @Inject()(cc: ControllerComponents, productDao: ProductsDao, cartsDao:
```

Running the application

At this point, we can run Play to see the result.

In the sbt shell tab of IntelliJ enter `run` followed by *Enter* as in the following:



The screenshot shows the IntelliJ IDEA interface with the sbt shell tab active. The terminal window displays the output of the sbt command, which includes setting up the project structure, applying settings, and starting the server. The command `sbt shell` is highlighted with a red arrow. The status bar at the bottom indicates the compilation was disabled by default 6 minutes ago.

```
sbt shell online-shopping-cart
[info] Defining */*:sbtStructureOptions
[info] The new value will be used by */*:ssOptions
[info] Reapplying settings...
[info] Set current project to server (in build file:/Users/nicolasjorand/Documents/Activities/writing/packt/git/Scala-Programming-Projects/online-shopping-cart/)
[info] Set current project to server (in build file:/Users/nicolasjorand/Documents/Activities/writing/packt/git/Scala-Programming-Projects/online-shopping-cart/)
[info] Writing structure to /private/var/folders/nz/n4bxcds503qcb0_8h2f21_0c0000gn/T/sbt-structure.xml...
[info] Done.
[success] Total time: 8 s, completed Sep 20, 2018 4:44:45 PM
[info] Reapplying settings...
[info] Set current project to server (in build file:/Users/nicolasjorand/Documents/Activities/writing/packt/git/Scala-Programming-Projects/online-shopping-cart/)
[info] Set current project to server (in build file:/Users/nicolasjorand/Documents/Activities/writing/packt/git/Scala-Programming-Projects/online-shopping-cart/)
[IJ] [server] $ run
★ (busy) >
```

The console should print:

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

You can now safely browse `http://localhost:9000/docs/swagger-ui/index.html?url=/v1/swagger.json`, Play is going to compile all the files and after a while, the Swagger UI will appear, as follows:

The screenshot shows the Swagger UI interface for an API named "Online Shopping" at version 1.0.0. The top navigation bar includes the Swagger logo, a search bar containing "/v1/swagger.json", and a "Explore" button. Below the header, the title "Online Shopping" is displayed with a "1.0.0" badge. A note indicates the base URL is "localhost:9000/" and provides a link to "v1/swagger.json". A brief description states "API for the online shopping example" and links to "License". The main content area features a large "Product and Cart API" section with a right-pointing arrow, and a "Models" section below it, also with a right-pointing arrow. In the bottom right corner of the content area, there is a red box containing the word "ERROR" and a small icon.

swagger

/v1/swagger.json

Explore

Online Shopping 1.0.0

[Base URL: localhost:9000/]

[/v1/swagger.json](#)

API for the online shopping example

[License](#)

Product and Cart API >

Models >

ERROR {…}

Click on the Product and Cart API link. All of the endpoints are shown, as follows:

Product and Cart API



POST /v1/login

GET /v1/products

POST /v1/products/add

POST /v1/cart/products/{id}/quantity/{quantity}

PUT /v1/cart/products/{id}/quantity/{quantity}

GET /v1/cart/products

DELETE /v1/cart/products/{id}

This looks good, but we have to add some information about our API. For example, if you click on the `login` endpoint, there is no way to test it with a body containing the username.

Login

To test the `login` endpoint, go to the `WebService` class and add the following annotations, just before the `login` definition:

```
@ApiOperation(value = "Login to the service", consumes = "text/plain")
@ApiImplicitParams(Array(
    new ApiImplicitParam(
        value = "Create a session for this user",
        required = true,
        dataType = "java.lang.String", // complete path
        paramType = "body"
    )
))
@ApiResponses(Array(new ApiResponse(code = 200, message = "login           success"), ne
    "Invalid user name supplied")))
def login() = Action { request =>
```

The `ApiOperation` object adds a description of the operation and defines the content type of the body sent. In this case, we are not sending JSON, but just a plain string. The `ApiImplicitParams` object defines the body sent to the server. The `ApiResponses` object informs the user about the possible statuses that can be returned in the response. The `login` documentation is now as follows:

POST

/v1/login Login to the service

Parameters

[Try it out](#)

Name	Description
------	-------------

body * required
(body) Create a session for this user

Example Value | Model

"string"

Parameter content type

text/plain

Responses

Response content type

application/json

v

Code	Description
------	-------------

200 *login success*

400 *Invalid user name supplied*

If you click on Try it out, you can type in a name and submit it by clicking on Execute:

POST

/v1/login Login to the service

Parameters

Cancel

Name	Description
------	-------------

body * required
(*body*) Create a session for this user

Example Value | Model

Chuck

Cancel

Parameter content type

text/plain



Execute

The call should return a successful status code 200, as follows:

Responses

Response content type

application/json

Curl

```
curl -X POST "http://localhost:9000/v1/login" -H "accept: application/json" -H "Content-Type: text/plain" -d "Chuck"
```

Request URL

```
http://localhost:9000/v1/login
```

Server response

Code

Details

200

Response headers

```
content-length: 0  
date: Sun, 25 Feb 2018 18:44:00 GMT
```

Responses

Code

Description

200

login success

400

Invalid user name supplied

List of products

Now, we can add some annotations to the list of products, as follows:

```
@ApiOperation(value = "List all the products")
@ApiResponses(Array(new ApiResponse(code = 200, message = "The list
of all the product")))
def listProduct() = Action.async { request =>
```

When we use Swagger to call `GET /v1/products`, the result of the execution is a JSON representation of all of the products, as follows:

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "name": "NAO", "code": "ALD1", "description": "NAO is an humanoid robot.", "price": 3599 }, { "name": "PEPER", "code": "ALD2", "description": "PEPPER is a robot moving with wheels and with a screen as human interaction", "price": 7000 }, { "name": "BEOBOT", "code": "BEO1", "description": "Beobot is a multipurpose robot.", "price": 159 }]</pre> <p>Response headers</p> <pre>content-length: 319 content-type: application/json date: Sun, 25 Feb 2018 18:48:16 GMT</pre>

For `addProduct`, the following are the annotations that we need to add:

```
@ApiOperation(value = "Add a product", consumes = "text/plain")
@ApiImplicitParams(Array(
  new ApiImplicitParam(
    value = "The product to add",
    required = true,
    dataType = "models.Product", // complete path
    paramType = "body"
  )
))
@ApiResponses(Array(new ApiResponse(code = 200, message = "Product
added"),
  new ApiResponse(code = 400, message = "Invalid
body supplied"),
```

```
    new ApiResponse(code = 500, message = "Internal  
server error, database error")))  
def addProduct() = Action.async { request =>
```

Thanks to the `dataType = "models.Product"` declaration, the Model section in Swagger shows a JSON model that corresponds to the `Product` case class:

Name	Description
body * required (<i>body</i>)	The product to add Example Value Model <pre>Product <v> { name* string code* string description* string price* number(\$double) }</pre>

Responses Response content type **application/json** ▾



Cart endpoints

Now, let's document the cart section with the list of products in the cart:

```
@ApiOperation(value = "List the product in the cart", consumes =
    "text/plain")
@ApiResponses(Array(new ApiResponse(code = 200, message = "Product
    added"),
    new ApiResponse(code = 401, message = "unauthorized, please login
    before to proceed"),
    new ApiResponse(code = 500, message = "Internal server error,
    database error")))
def listCartProducts() = Action.async { request =>
```

If we call `listCartProducts`, we receive an empty array. To test it with some products, complete the declaration of `addCartProduct` with the following:

```
@ApiOperation(value = "Add a product in the cart", consumes =
    "text/plain")
@ApiResponses(Array(new ApiResponse(code = 200, message = "Product
    added in the cart"),
    new ApiResponse(code = 400, message = "Cannot insert duplicates in
    the database"),
    new ApiResponse(code = 401, message = "unauthorized, please login
    before to proceed"),
    new ApiResponse(code = 500, message = "Internal server error,
    database error")))
def addCartProduct(
    @ApiParam(name = "id", value = "The product code", required =
        true) id: String,
    @ApiParam(name = "quantity", value= "The quantity to add",
        required = true) quantity: String) = Action.async { request
    =>
```

In Swagger, we can now add a new product to the cart, as follows:

POST /v1/cart/products/{id}/quantity/{quantity} Add a product in the cart

Parameters

Name **Description**

Id * required string (path)	The product code <input type="text" value="ALD1"/>
quantity * required string (path)	The quantity to add <input type="text" value="22"/>

Execute **Clear**

Then, the list of products will return the following:

Server response

Code	Details
200	Response body <pre>[{ "user": "toto", "productCode": "ALD1", "quantity": 22 }]</pre> Response headers <pre>content-length: 52 content-type: application/json date: Sun, 25 Feb 2018 20:54:46 GMT</pre>

After that, we can try to update a product. Add the following annotations to `updateCartProduct`:

```

@ApiOperation(value = "Update a product quantity in the cart",
  consumes = "text/plain")
@ApiResponses(Array(new ApiResponse(code = 200, message = "Product
  added in the cart"),
  new ApiResponse(code = 401, message = "unauthorized, please login
  before to proceed"),
  new ApiResponse(code = 403, message = "not enough money"))

```

```

    new ApiResponse(code = 500, message = "Internal server error,
    database error")))
def updateCartProduct(@ApiParam(name = "id", value = "The product
    code", required = true, example = "ALD1") id: String,
    @ApiParam(name = "quantity", value= "The quantity to update",
    required = true) quantity: String) = Action.async { request =>

```

Then, use Swagger to update the quantity, as follows:

PUT /v1/cart/products/{id}/quantity/{quantity} Update a product quantity in the cart

Parameters

Name	Description
Id * required string (path)	The product code <input type="text" value="ALD1"/>
quantity * required string (path)	The quantity to update <input type="text" value="42"/>

Execute **Clear**

After the update, the list of products returns, as follows:

Server response

Code	Details
200	Response body <pre>[{ "user": "toto", "productCode": "ALD1", "quantity": 22 }]</pre> Response headers <pre>content-length: 52 content-type: application/json date: Sun, 25 Feb 2018 20:54:46 GMT</pre>

Perfect; the last operation to document is `deleteCartProduct`:

```
@ApiOperation(value = "Delete a product from the cart", consumes =
    "text/plain")
@ApiResponses(Array(new ApiResponse(code = 200, message = "Product
    delete from the cart"),
    new ApiResponse(code = 401, message = "unauthorized, please login
    before to proceed"),
    new ApiResponse(code = 500, message = "Internal server error,
    database error")))
def deleteCartProduct(@ApiParam(name = "id", value = "The product
    code", required = true) id: String) = Action.async { request =>
```

We now have complete Swagger documentation for our API, and users can test it directly from their browsers.

Deploying on Heroku

The API is now finished. We can deploy it to Heroku, to make it available on the internet. As we already set up Heroku in the previous chapter, there is just one command to achieve our deployment. From the command line in the root of your project, enter the following:

```
| git push heroku master
```

Once the deployment has finished, you can browse to <https://shopping-fs.herokuapp.com/docs/swagger-ui/index.html?url=/v1/swagger.json>.

Congratulations! You can now test the API online.

Summary

In this chapter, you learned how to design and implement a REST API, while respecting the REST architectural principles.

We created tests to check the API from a client's perspective. We implemented all of the methods by using the DAO that we wrote in the last chapter. All of the calls are asynchronous because we used `Future`, which guarantees that our server can handle a lot of concurrent requests.

You also learned how to use the excellent Circe library to encode and decode JSON from Scala. Finally, we added a web interface to document and test the API, using Swagger.

In the next chapter, we are going to use this API to create a web interface based on Scala.js.

Online Shopping - User Interface

In this chapter, we are going to use Scala.js to build the user interface. In this interface, you can select a product to add to your cart, update the number of products that you wish to buy, and remove them from the cart if needed.

Scala.js is a project initiated by Sébastien Doeraene back in 2013. This project is mature and provides a clean way to build frontend applications. Indeed, you can code with a strongly-typed system to avoid stupid mistakes, but this is not only for strong typing; the code—written in Scala—is compiled into a highly efficient JavaScript. It can interoperate with all of the JavaScript frameworks. Moreover, the code can be shared between the front-end and the back-end developers. This feature simplifies communication between developers, as they are using the same concepts and classes.

Thanks to its interoperability, there are multiple ways to use Scala.js. You can use an HTML template and adapt it to interoperate with Scala.js. For example, you can buy the excellent SmartAdmin (<https://wrapbootstrap.com/theme/smrtadmin-responsive-webapp-WB0573SK0>) template (HTML5 version) as a base for the layout and all of the components/widgets, and then use Scala.js to implement the specific behaviors.

Another way is to start from scratch and build the HTML layout, CSS, components, and behaviors using the Scala.js ecosystem. This is the option that we will choose in this book. To generate the HTML and CSS, ScalaTags (<http://www.lihaoyi.com/scalatags/>) from Li Haoyi (<http://www.lihaoyi.com/>) will be used.

This chapter will explain how to develop a dynamic web UI using Scala.js.

We will cover the following topics:

- Defining the layout
- Creating the layout
- Building the layout
- Main layout
- Product list panel

- Cart panel
- Introducing the UI manager

Learning objectives

The objectives of this chapter are to introduce a user interface into our project and interact with the server to get data from it.

More precisely, we will learn the following skills:

- How to develop a simple web UI
- How to apply styles
- How to interact with the server using a web service call
- How to debug Scala code on the client side

Setting up

Note that this setup has already been completed when you start using the template. The following steps are only for reference:

1. To enable Scala.js with Play, you first need to add the following code to `project/plugins.sbt`:

```
| addSbtPlugin("org.scala-js" % "sbt-scalajs" % "0.6.24")
| addSbtPlugin("com.vmunier" % "sbt-web-scalajs" % "1.0.8-0.6")
```

2. In the `build.sbt`, you need to add the plugins by adding the following code in the `client` variable:

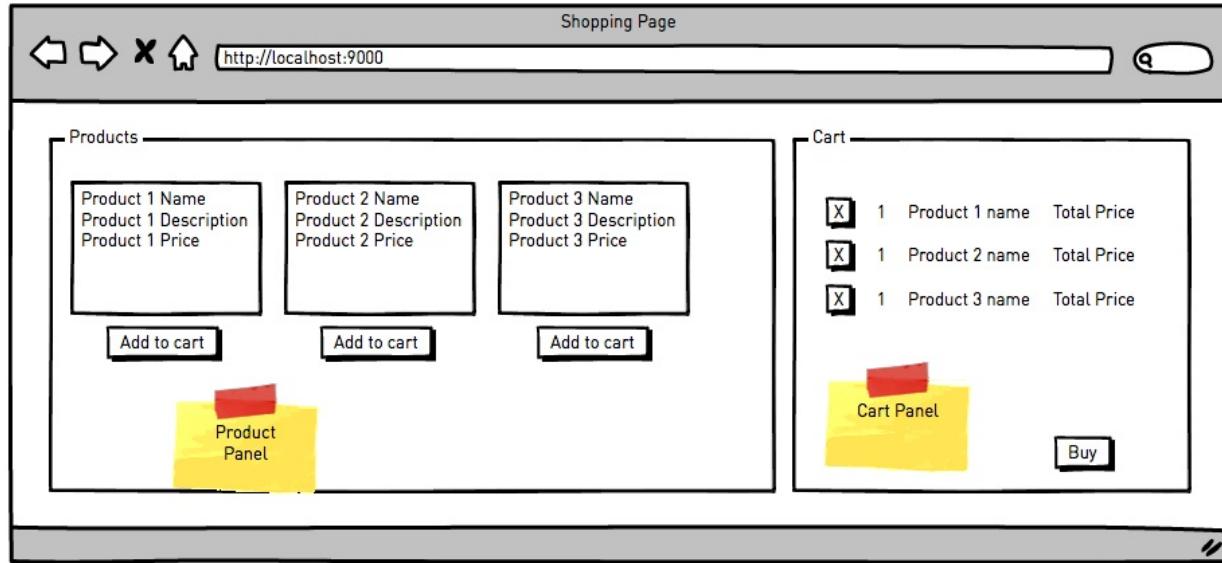
```
| .enablePlugins(ScalaJSPlugin, ScalaJSWeb)
```

3. You need to add the ScalaTags dependency by adding the following in `libraryDependencies` of the client configuration:

```
| "com.lihaoyi" %% "scalatags" % "0.6.7"
```

Defining the layout

For the purpose of this book, the shopping cart is designed as shown in the following screenshot:



On the left-hand side, a panel lists all of the products with all of their details. Underneath, a button adds the product to the cart. On the right-hand panel, there is a list of all of the products that have been added to the cart. On the **Cart Panel**, the number of products can be changed by clicking on the number and entering the right number. Each line has a button to delete the product from the list.

Creating the layout

Multiple technologies are used to create the layout—more specifically, the top layout—using the `<html>` and `<body>` tags, and the two `div` containers for the products and the cart panels respectively are going to be built with a Play template engine, named Twirl. Using this template, the inner HTML of the product and cart instances of `div` are going to be filled with ScalaTags.

Let's first create the main entry point. We named it `index.html`, and it is implemented by creating a file called `index.scala.html` in the `view` package of the server.

The content is as follows:

```
@(title: String)
<!DOCTYPE html>
<html>
  <head>
    <title>@title</title>
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div id="productPanel" class="col-8">
          <div id="products" class="row">
            </div>
        </div>
        <div id="cartPanel" class="col-4">
          </div>
      </div>
    </div>

    @scalajs.html.scripts("client",
      routes.Assets.versioned(_).toString,
      name => getClass.getResource(s"/public/$name") != null)
  </body>
</html>
```

This file looks like a standard HTML file. In fact, this file is a template that is processed by the server.

The first line starts with an `@` character. It defines the input parameter that is passed by the caller, but who is calling this template? It is the `index()` function in the `Application` controller that calls the template, and effectively, the template is called by using the title of the page.

On the line starting with `@scalajs.html.scripts`, we are using a helper method provided by the `sbt-web-scalajs` plugin. This method retrieves all of the scripts generated by the `client` Scala.js project.

The two instances of `div` are going to be set from the code; we will look at this in more detail in the next chapter.

Building the page

We will have two main sections on the main page: the product list and the cart.

To create the layout, we could use basic HTML tags, such as `table` and `div`, but this is quite laborious for our task. Instead, let's introduce a framework named Bootstrap (<https://getbootstrap.com/>).

This open source framework is widely used and very mature. It allows us to build a responsive website based on a grid, with a lot of components such as notifications, menus, badges, and tooltips. Bootstrap needs a CSS and some JavaScript libraries to work.

For now, we just need to add the Bootstrap CSS by adding the link in the HTML header, as follows:

```
<head>
  <title>@title</title>
  <!-- Bootstrap CSS -->
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0
    /css/bootstrap.min.css" integrity="sha384-
    Gn5384xqqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJ1SAwiGgFAW
    /dAis6JXm" crossorigin="anonymous">

</head>
```

Main layout

In Bootstrap, a grid is a container composed of rows, each with 12 columns. A `class` attribute is used to define the type of `div`.

So, in our case, we would like to have the product list occupy two-thirds of the screen, with the cart getting the rest of `row`.

The structure of the body in our `index.scala.html` is as follows:

```
<body>
  <div class="container">
    <div class="row">
      <div id="productPanel" class="col-8">
        <div id="products" class="row"></div>
      </div>
      <div id="cartPanel" class="col-4"></div>
    </div>
  </div>
</body>
```

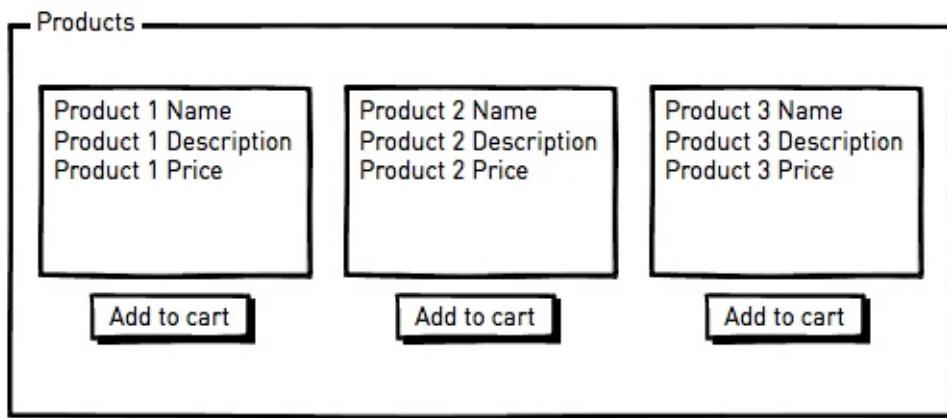
Product list panel

To structure our HTML page, we are going to create a panel called `productPanel`. This panel is a container for all product details.

A product is defined with a name, description, and button to add it to the cart, as shown in the following screenshot:



As we have multiple products, we would like to add each one in `productPanel` and fit the total width of `productPanel`, as shown in the following screenshot:



To reach this goal, we can recreate a row inside `productPanel`, with `products` forming a column of the row, as shown in the following code:

```
<div id="productPanel" class="col-8">
  <div id="products" class="row">
    <!-- Added programatically -->
  </div>
</div>
```

There we go. The main layout is done. Now we have to create the HTML product's representation as `div`, with its name, description, price, and a button to add it to the cart.

This looks almost the same as what we did when we modeled the product for the database. Wouldn't it be great if we could reuse the model created on the server side on the client? As we are using Scala.js, this is possible; indeed, we are using the same language. This is what we call an **isomorphic application**.

All we have to do is move the model code from the server project to the shared project. Using IntelliJ, just drag the `Models.scala` file from `server/app/models` to `shared/src/main/scala/io/fscala/shopping/shared`. By doing so, we can use the same model to create our product representation.

Create a new class named `ProductDiv` under `client/src/main/scala/io/fscala/shopping`. This class represents the HTML content of a product with a button to add itself to the cart.

The `ProductDiv` class contains the `Product` model, and looks like the following:

```
case class ProductDiv(product: Product) {
    def content: Div = div(`class` := "col")
        (getProductDescription, getButton).render
    private def getProductDescription =
        div(
            p(product.name),
            p(product.description),
            p(product.price))

    private def getButton = button(
        `type` := "button",
        onclick := addToCart)("Add to Cart")
    private def addToCart = () => ???
}
```

The main method is the `content` method. It creates the product description and the button.

The `getProductDescription` method creates an HTML `div` with a paragraph for each of its properties.

The `getButton()` method creates an HTML button and uses the `addToCart` function to handle the `onclick` event. For now, we are not going to look at the implementation

details of the `onclick` event.

Cart panel

The cart panel is a representation of the cart. It has a line for every product that is added, displaying the number of items, the name of the item type, the total price, and a button to remove it from the cart, as shown in the following screenshot:

X	1	Product 1 name	Total Price
---	---	----------------	-------------

We would like to add a line each time a new product is added to the cart, as shown in the following screenshot:

X	1	Product 1 name	Total Price
X	1	Product 2 name	Total Price
X	1	Product 3 name	Total Price

In this case, we do not need to modify the main layout as we are going to represent each line as a row with columns. The model of a line in the cart is shown in the following code:

```
| case class CartLine(qty: Int, product: Product)
```

The HTML content of the cart is shown in the following code:

```
| def content: Div = div(`class` := "row", id := s"cart-${product.code}-row")(
|   div(`class` := "col-1")(getDeleteButton),
|   div(`class` := "col-2")(getQuantityInput),
|   div(`class` := "col-6")(getProductLabel),
|   div(`class` := "col")(getPriceLabel)
| ).render
```

The previous code is for a row with four columns. The code for the button to delete it from the cart (`getDeleteButton`) is as follows:

```
| private def getDeleteButton = button(
|   `type` := "button",
|   onclick := removeFromCart)("X").render
```

Note how you can add a listener to an event emitted by the component just by adding the method name on the `onclick` event. For now, we are not going to implement the action, and will leave it unimplemented (???), as shown in the following code:

```
| private def removeFromCart = () => ???
```

The `input` text field representing the quantity in the cart (`quantityInput`) is written as follows:

```
| private def getQuantityInput = input(
|   id := s"cart-${product.code}-qty",
|   onchange := changeQty,
|   value := qty.toString,
|   `type` := "text",
|   style := "width: 100%;").render
```

Again, on the `onchange` event, we call the `changeQty` function, defined as follows:

```
| private def changeQty = () => ???
```

The product name (`getProductLabel`) is written as follows:

```
| private def getProductLabel = label(product.name).render
```

Finally, the total price is written as `getPriceLabel`, as follows:

```
| private def getPriceLabel = label(product.price * qty).render
```

As we will have the definition of a line in the cart, we can define the cart's `div`.

This `div` should provide an HTML representation of all of the lines and allow you to add a cart `line`. The implementation is as follows:

```
| case class CartDiv(lines: Set[CartLine]) {
|   def content = lines.foldLeft(div.render) { (a, b) =>
|     a.appendChild(b.content).render
|     a
|   }
|   def addProduct(line: CartLine): CartDiv = {
|     new CartDiv(this.lines + line)
|   }
| }
```

Upon its creation, `CartDiv` receives the list of lines represented by the `lines` value.

To get the HTML representation, the `content` function is called. In this function, we create an empty `div` and then append each `cartLine` into that `div`.

This is implemented using `foldLeft`. The empty `div` is created as the initial value, and then for each `cartLine` an anonymous function is called with `(a, b)` as parameters. The `a` parameter is the previous value (the empty `div` on the first iteration) and the `b` parameter is the next `cartLine` of the collection. The body of the method just appends the content of `cartDiv` to `div` and returns `div` for the next iteration.

We added a function to add a product to `div` (`addProduct()`). We could have implemented this method by creating a mutable variable that holds the list of `cartLine` and updates it each time we want to add `cartLine`, but this would not be in the spirit of functional programming.

Instead, a new `cartDiv` with new `cartLine` that we added is returned from the function call.

Now that we have defined the product `div` and the cart `div`, it is time to build the interaction between these instances of `div`.

Introducing the UI manager

At some point, we will need to have a class that is responsible for the workflow of the user experience. For example, when a user clicks on the Add to cart button, the product has to be added to the cart at the server level and the user interface has to be updated.

The UI manager takes responsibility for managing the workflow of the user experience, handling all communication with the server when needed, and is used as an entry point to start the Scala.js code. This is the main entry point of our client application when the application is executed in the browser.

For communicating with the server, we are going to use jQuery. This JavaScript library is widely used and is a reference in the JavaScript world.

This is one of the advantages of Scala.js. We can use existing JavaScript libraries, such as jQuery, from Scala. To use existing JavaScript libraries, we only need to define an interface, which is called a **facade** in Scala.js. The facade can be seen as an interface to redefine the JavaScript type and the JavaScript function signature. It means we need a facade for all of the JavaScript libraries we want to use. Fortunately, there are already a number of facades that already exist for the most important JavaScript frameworks. A list of available facades can be found at the Scala website (<https://www.scala-js.org/libraries/facades.html>).

Adding jQuery to our project

To add jQuery and its facade into our project, we need to add the Scala facade and the JavaScript library.

For the facade, add the following dependency to the `libraryDependencies` variable in the `build.sbt` file :

```
| "org.querki" %%% "jquery-facade" % "1.2"
```

To add the JavaScript library, add the following dependency to the `jsDependencies` variable:

```
| jsDependencies +=  
|   "org.webjars" % "jquery" % "2.2.1" / "jquery.js"  
|   minified "jquery.min.js"
```



*This is the first library we are using in which we are adding a WebJar as a JavaScript library.
This WebJar is a repository of JavaScript libraries packaged as a JAR file.*

Calling our API

The first call we have to perform is the login to the server. For the purposes of this book, we have not designed a proper login page. Besides, the login itself is not really a login as it accepts any user!

We are going to log in with a random user every time we browse the index of the website.

By the way, what is the entry point of our client application?

Setting the main method

By default, Scala.js only creates a JavaScript library with all your dependencies. To make it an application, you have to add the client configuration of the `build.sbt` file, as shown in the following code:

```
| scalaJSUseMainModuleInitializer := true
```

Once defined, Scala.js looks for an object containing the `main` method, as in a normal JVM application. We can create that object in the `client/src/main/scala/io/fscala/shopping/client` folder. Create a Scala file named `UIManager.scala`.

In the `main` function, we would like to log in to the API and initialize the interface with `ProductDiv` and `CartDiv`, which we defined earlier, as shown in the following code:

```
object UIManager {  
    val origin: UndefOr[String] = dom.document.location.origin  
    val cart: CartDiv = CartDiv(Set.empty[CartLine])  
    val webSocket: WebSocket = getWebSocket  
    val dummyUserName = s"user-${Random.nextInt(1000)}"  
    def main(args: Array[String]): Unit = {  
        val settings = JQueryAjaxSettings  
            .url(s"$origin/v1/login")  
            .data(dummyUserName)  
            .contentType("text/plain")  
        $.post(settings._result).done((_: String) => {  
            initUI(origin)  
        })  
    }  
}
```

We define three properties on the `UIManager` object:

- The first is the `origin` property. This property uses the `dom` utility object from Scala.js; we will get `document.location.origin` from it. This represents the server location, with the protocol, hostname, and port. In development mode, this looks like `http://localhost:9000`.
- The second property is `cart`, representing `CartDiv`. This is to keep a reference to it on the manager for later use. In the `main` function, we perform the login with a hardcoded user and, once successful, we initialize the user interface.

- The last property is `dummyUserName`, representing a randomly generated username. This will simplify the code as we are not going to implement a real login process.

Note how we can use jQuery from Scala. This is the beauty of the facade—we can use almost the same syntax as in JavaScript but with the advantage of strong Scala typing.

For example, to create the settings of the `post` call, we can use a method on the `JQueryAjaxSettings` object instead of creating a `Map` with a string as a key and anything as a value. This way, it is less error-prone, and we can take advantage of the IDE to autocomplete all of the possible properties.

The signature of the `done` jQuery function is `Function(PlainObject data, String textStatus, jqXHR)`. You can find out more about these types on the jQuery website:

- <http://api.jquery.com/Types/#Function>
- <http://api.jquery.com/Types/#PlainObject>
- <http://api.jquery.com/Types/#String>
- <http://api.jquery.com/Types/#jqXHR>

This function takes three parameters, but as we are just interested in the first one, the `data` response, we can ignore the others. This is a particularity of JavaScript. The implementation of the corresponding facade is as follows:

```
| def done(doneCallbacks: js.Function*): JQueryDeferred = js.native
```

The function uses variadic parameters, noted with the asterisk character after the type. This is a perfect match with JavaScript, where the parameters are not mandatory.

It is now time to look at the creation of the user interface based on the data coming from the server.

Initializing the user interface

To initialize the user interface, we need to get all of the products from the database—including the cart for the user, if there are any—through the web service API and add it to the layout. The code for this looks like the following:

```
private def initUI(origin: UndefOr[String]) = {
    $.get(url = s"$origin/v1/products", dataType = "text")
        .done((answers: String) => {
            val products = decode[Seq[Product]](answers)
            products.right.map { seq =>
                seq.foreach(p =>
                    $("#products").append(ProductDiv(p).content)
                )
                initCartUI(origin, seq)
            }
        })
        .fail((xhr: JQueryXHR, textStatus: String, textError: String) =>
            println(s"call failed: $textStatus with status code:
            ${xhr.status} $textError")
    )
}
```

It will come as no surprise that we use jQuery to perform the `GET` method on the API. The `dataType` asks for a text response, so that we can use Circe to parse the response and transform it into a sequence of `Product`.

But, is `decode[Seq[Product]]` the same code that we used in [Chapter 2, Developing a Retirement Calculator](#), the REST API, when we received JSON and converted it into a class?

Yes, we are using the same code with the same framework (Circe) to decode JSON and encode the class to JSON! The code running on the server, compiled as a JVM bytecode, is the same as the one running on the client, compiled as JavaScript.

Once we get the list of products, for each of them, we add `ProductDiv` in the `products` container. Again, jQuery is used to get an HTML element using its `id` attribute. At this point, knowledge of jQuery is more important than the Scala syntax.

The product panel is set up. Now it is the cart's turn.

The `initCartUI()` function is responsible for creating the HTML code representing the cart. The cart for the user is taken from the server. We cast it to a `cart` sequence and, for each of them, we get the corresponding product in order to have the name and price. Finally we append the line in `cartDiv`, as shown in the following code:

```
private def initCartUI(origin: UndefOr[String], products: Seq[Product]) = {
  $.get(url = s"$origin/v1/cart/products", dataType = "text")
    .done((answers: String) => {
      val carts = decode[Seq[Cart]](answers)
      carts.right.map { cartLines =>
        cartLines.foreach { cartDao =>
          val product = products.find(
            _.code == cartDao.productCode)
          product match {
            case Some(p) =>
              val cartLine = CartLine(cartDao.quantity, p.name,
                cartDao.productCode, p.price)
              val cartContent = UIManager.cart.addProduct(cartLine)
                .content
              $("#cartPanel").append(cartContent)
            case None =>
              println(
                s"product code ${cartDao.productCode} doesn't
                  exists in the catalog")
          }
        }
      })
    .fail((xhr: JQueryXHR, textStatus: String, textError: String) =>
      println(
        s"call failed: $textStatus with status code:
          ${xhr.status} $textError")
  )
}
```

In case of failure, we just print the error in the browser console.

With all this code, our user interface is now initialized. We can now implement the action on the user interface that we left unimplemented.

Implementing UI actions

When the application is started, the user interface is the representation of the database in terms of the product and the user cart.

In this chapter, we will implement actions such as adding a product to the cart, updating the quantity to buy, and removing a product from the cart.

Adding a product to the cart

To add a product to the cart, we have to click on the Add to cart button of the product panel. We need to edit `ProductDiv` again and implement the `addToCart` method.

As we said in the *Introducing the UI manager* section of this chapter, we would like to delegate the user interface manipulation to the `UIManager` class, so the `addToCart` method implementation is as follows:

```
| private def addToCart() = () => UIManager.addOneProduct(product)
```

Indeed, we are asking `UIManager` to add the product to the cart. The `UIManager` builds a `div` representing the product in the cart, and if it is already in the cart, nothing happens.

The implementation is as follows:

```
def addOneProduct(product: Product): JQueryDeferred = {
  val quantity = 1
  def onDone = () => {
    val cartContent = cart.addProduct(CartLine(quantity, product))
      .content
    $("#cartPanel").append(cartContent)
    println(s"Product $product added in the cart")
  }
  postInCart(product.code, quantity, onDone)
}
```

The `postInCart` method is called with the product code and the initial quantity of one to create a new entry in the `cart` table. Once created, the `onDone()` method is called. This method adds the HTML elements that are needed to visualize the cart line in the user interface.

The `postInCart` method receives `productCode`, the quantity, and the method to call once the web service call is a success, as shown in the following code:

```
private def postInCart(productCode: String, quantity: Int, onDone: () => Unit) = {
  val url = s"${UIManager.origin}/v1/cart/products/$productCode
  /quantity/$quantity"
  $.post(JQueryAjaxSettings.url(url)._result)
    .done(onDone)
    .fail(() => println("cannot add a product twice"))
}
```

If the web service call is a failure, we just print the error in the browser console and nothing is added to the user interface.

Removing a product from the cart

The action to remove a product from the cart is triggered when the X button related to a cart entry is clicked. This is implemented in the `removeFromCart()` method, located in the `cartLine` class. This is similar to the method we used in the previous section. The code is as follows:

```
| private def removeFromCart() =  
|   () => UIManager.deleteProduct(product)
```

We delegate the action to `UIManager`, and the implementation is as follows:

```
def deleteProduct(product: Product): JQueryDeferred = {  
  def onDone = () => {  
    val cartContent = $(s"#cart-${product.code}-row")  
    cartContent.remove()  
    println(s"Product ${product.code} removed from the cart")  
  }  
  
  deletefromCart(product.code, onDone)  
}
```

This time, we call the `deleteFromCart` method and remove the row with the related ID.

The implementation of the web service call is as follows:

```
private def deletefromCart(  
  productCode: String,  
  onDone: () => Unit) = {  
  val url = s"${UIManager.origin}/v1/cart/products/$productCode"  
  $.ajax(JQueryAjaxSettings.url(url).method("DELETE")._result)  
  .done(onDone)  
}
```

As jQuery does not have a `delete()` method, we have to use the `ajax()` method and set the HTTP method.

Updating the quantity

To update the quantity of a product in the cart, an HTML input text is used. As soon as the value is changed, we update the database with the new value. The `onchange()` event of the input text is used for this purpose.

It should come as no surprise that, in `cartDiv`, as we did previously, we delegate the call to `UIManager`, as shown in the following code:

```
| private def changeQty() =  
|   () => UIManager.updateProduct(product)
```

The implementation of `updateProduct` is as follows:

```
| def updateProduct(productCode: String): JQueryDeferred = {  
|   putInCart(product.code, quantity(product.code))  
| }
```

We call the web service using the `quantity` set in `inputText`. The method to get the quantity is as follows:

```
| private def quantity(productCode: String) = Try {  
|   val inputText = $(s"#cart-$productCode-qty")  
|   if (inputText.length != 0)  
|     Integer.parseInt(inputText.`val`().asInstanceOf[String])  
|   else 1  
| }.getOrElse(1)
```

We get the quantity from the HTML input text element. If it exists, we parse it as an integer. If the field does not exist or we are having a parsing error (a letter is inputted), we return the quantity of 1.

The web service call to update the product quantity is as follows:

```
| private def putInCart(productCode: String, updatedQuantity: Int) = {  
|   val url =  
|     s"${UIManager.origin}/v1/cart/products/  
|     $productCode/quantity/$updatedQuantity"  
|     $.ajax(JQueryAjaxSettings.url(url).method("PUT")._result)  
|     .done()  
| }
```

There we are. We have finished the implementation of the user interface for the shopping cart.

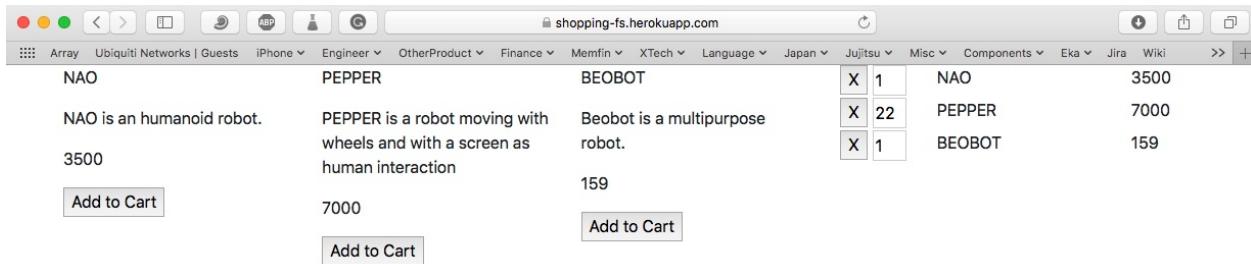
It is time to deploy it and check whether it works.

Deploying the user interface

To deploy from the command line in the root of your project, enter the following code:

```
| git push heroku master
```

Once the deployment is a success, you can browse <https://shopping-fs.herokuapp.com/>. The interface will be displayed as shown in the following screenshot:



The screenshot shows a web browser window with the title "shopping-fs.herokuapp.com". The page displays a shopping cart interface. The cart contains three items:

Product	Description	Quantity	Price		
NAO	PEPPER	BEOBOT	X 1	NAO	3500
NAO is an humanoid robot.	PEPPER is a robot moving with wheels and with a screen as human interaction	Beobot is a multipurpose robot.	X 22	PEPPER	7000
3500			X 1	BEOBOT	159

Below the cart, there are two "Add to Cart" buttons. The total price of 159 is also visible.

You can now play with the interface.

Debugging the interface

During development, we will not write the right code on the first draft. As humans, we make mistakes and do not perfectly remember all of the framework that we use.

In this chapter, we would like to give an entry point into debugging the code. The most obvious debugging system is to print in the console of the browser. This is done by directly using `println()` in Scala, and then looking at the log that is displayed in the console.

To have a look at the console and other debugging tools, you have to enable the developer tools on your browser. I am using Safari on Macintosh, but if you do not want to use it, I would recommend Google Chrome; the features are almost the same.

In Safari, enable the developer tools by clicking on the Show Develop menu in menu bar checkbox from the Advanced Preferences.

Once done, a new Develop menu will appear. Open this menu and select Show JavaScript Console. A new section will appear in the Safari window with the console. If you click on the button to delete a cart row, a log is printed in the console, as shown in the following screenshot:

Product	Description	Quantity	Price	Add to Cart	
NAO	PEPPER	BEOBOT	X 1	NAO	3500
NAO is an humanoid robot.	PEPPER is a robot moving with wheels and with a screen as human interaction	Beobot is a multipurpose robot.	X 22	PEPPER	154000
3500					

```
kd -- adcce59c9c0fdbdd0e722e82c0b18b70-client-opt.js:548:275
>
```

You can interact with JavaScript by typing anything in the last line of the console.

For example, if you input `$("#productPanel")`, then the product `div` is selected, and you can inspect it, as shown in the following screenshot:

Product	Description	Quantity	Price	Add to Cart	
NAO	PEPPER	BEOBOT	X 1	NAO	3500
NAO is an humanoid robot.	PEPPER is a robot moving with wheels and with a screen as human interaction	Beobot is a multipurpose robot.	X 22	PEPPER	154000
3500					

```
> $("#productPanel")
< $n (1) = $1
0 <div id="productPanel" class="col-8">
  <div id="products" class="row">
    <div class="col">
      <div>
        <p>NAO</p>
        <p>NAO is an humanoid robot.</p>
        <p>3500</p>
        <button type="button">Add to Cart</button>
      </div>
    <div class="col">
      <div>
        <p>PEPPER</p>
        <p>PEPPER is a robot moving with wheels and with a screen as human interaction</p>
        <p>7000</p>
        <button type="button">Add to Cart</button>
      </div>
    <div class="col">
      <div>
        <p>BEOBOT</p>
        <p>Beobot is a multipurpose robot.</p>
        <p>159</p>
        <button type="button">Add to Cart</button>
      </div>
    </div>
  </div>
</div>
```

The inspect element code of the web page

You can even run tests. If you input `$("#productPanel").remove()`, the `div` will be removed from `dom` and your page will look like the following screenshot:

The inspect element code of the web page

```
> $("#productPanel").remove()
< ▾ n {1} = $1
  0 > <div id="productPanel" class="col-8">...</div>
  ▾ n Prototype
```

The inspect element code for the test

Refresh the page to get back to the product list. You can even debug the Scala code from inside the browser.



You need to have the project in development mode to have the necessary files generated for debugging (the source maps files).

Click on the Debugger tab and look for `UIManager.scala` on the left-hand panel, under `Sources/client-fast-opt.js`, as shown in the following screenshot:

```
UIManager.scala
  package io.lioc.scala.shopping.client
  ...
  private def initCartUI(origin: UriFor[String]): Unit = {
    ...
    private def initCartUI(origin: UriFor[String]): Unit = {
      ...
      private def initCartUI(origin: UriFor[String]): Unit = {
        ...
        private def initCartUI(origin: UriFor[String]): Unit = {
          ...
          private def initCartUI(origin: UriFor[String]): Unit = {
            ...
            private def initCartUI(origin: UriFor[String]): Unit = {
              ...
              private def initCartUI(origin: UriFor[String]): Unit = {
                ...
                private def initCartUI(origin: UriFor[String]): Unit = {
                  ...
                  private def initCartUI(origin: UriFor[String]): Unit = {
                    ...
                    private def initCartUI(origin: UriFor[String]): Unit = {
                      ...
                      private def initCartUI(origin: UriFor[String]): Unit = {
                        ...
                        private def initCartUI(origin: UriFor[String]): Unit = {
                          ...
                          private def initCartUI(origin: UriFor[String]): Unit = {
                            ...
                            private def initCartUI(origin: UriFor[String]): Unit = {
                              ...
                              private def initCartUI(origin: UriFor[String]): Unit = {
                                ...
                                private def initCartUI(origin: UriFor[String]): Unit = {
                                  ...
                                  private def initCartUI(origin: UriFor[String]): Unit = {
                                    ...
                                    private def initCartUI(origin: UriFor[String]): Unit = {
                                      ...
                                      private def initCartUI(origin: UriFor[String]): Unit = {
                                        ...
                                        private def initCartUI(origin: UriFor[String]): Unit = {
                                          ...
                                          private def initCartUI(origin: UriFor[String]): Unit = {
                                            ...
                                            private def initCartUI(origin: UriFor[String]): Unit = {
                                              ...
                                              private def initCartUI(origin: UriFor[String]): Unit = {
                                                ...
                                                private def initCartUI(origin: UriFor[String]): Unit = {
                                                  ...
                                                  private def initCartUI(origin: UriFor[String]): Unit = {
                                                    ...
                                                    private def initCartUI(origin: UriFor[String]): Unit = {
                                                      ...
                                                      private def initCartUI(origin: UriFor[String]): Unit = {
                                                        ...
                                                        private def initCartUI(origin: UriFor[String]): Unit = {
                                                          ...
                                                          private def initCartUI(origin: UriFor[String]): Unit = {
                                                            ...
                                                            private def initCartUI(origin: UriFor[String]): Unit = {
                                                              ...
                                                              private def initCartUI(origin: UriFor[String]): Unit = {
                                                                ...
                                                                private def initCartUI(origin: UriFor[String]): Unit = {
                                                                  ...
                                                                  private def initCartUI(origin: UriFor[String]): Unit = {
                                                                    ...
                                                                    private def initCartUI(origin: UriFor[String]): Unit = {
                                                                      ...
                                                                      private def initCartUI(origin: UriFor[String]): Unit = {
                                                                        ...
                                                                        private def initCartUI(origin: UriFor[String]): Unit = {
                                                                          ...
                                                                          private def initCartUI(origin: UriFor[String]): Unit = {
                                                                            ...
                                                                            private def initCartUI(origin: UriFor[String]): Unit = {
                                                                              ...
                                                                              private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                ...
                                                                                private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                  ...
                                                                                  private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                    ...
                                                                                    private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                      ...
                                                                                      private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                        ...
                                                                                        private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                          ...
                                                                                          private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                            ...
                                                                                            private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                              ...
                                                                                              private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                                ...
                                                                                                private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                                  ...
                                                                                                  private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                                    ...
                                                                                                    private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                                      ...
                                                                                                      private def initCartUI(origin: UriFor[String]): Unit = {
                                                                                                        ...
................................................................
```

The UIManager.scala source code

Once `UIManager.scala` is selected, you can see the Scala source on the middle panel. Click on the gutter in line **30**. A breakpoint will be set when the UI is initialized and product `div` instances are appended.

If you refresh the page, the engine will stop at that point and, on the right-hand panel, you will have all of the variables, even the local variables, such as `p`, representing the product to add at this point.

Click on the Continue script execution button, as shown in the following screenshot:



The script will continue until the next element in the collection, and the `p` variable on the right-hand panel will be updated with the next element.

I've just scratched the surface of all of the Develop menu possibilities. You can have metrics that govern the time spent to load elements in the page and to process, inspect, and change any HTML elements in the page.

For more information, read the official documentation for Safari (<https://support.apple.com/en-in/guide/safari-developer/welcome/mac>) and for Google Chrome (<https://developers.google.com/web/tools/chrome-devtools/>).

Summary

In this chapter, we learned how to build a user interface from scratch by first creating a mockup of the interface.

Then, we implemented the main layout, representing the skeleton of the application with all of the files that need to be linked, such as the CSS files and scripts. Once the layout was ready, we modeled the different HTML parts of the user interface in Scala, such as the product panel and the cart panel. The last step was to create the navigation system and user interaction. For that purpose, we created a UI manager, responsible for all the interaction.

As a side note, our user interface is pretty simple, without much interaction. This is why we chose to write the UI manager by hand. If the interface becomes more complex, then it might be useful to use a framework to manage it. At the time of writing, React.js and Angular are two of the most popular frameworks. Be aware, however, that frameworks come with a learning curve, and can quickly become obsolete.

Another solution would be to use Akka.js and more specifically the FSM actor to manage your user interface. After all, this is a state machine, reacting and acting based on events. This will be developed in the next chapter on the automatic price updater.

We also looked at the debugging facilities provided by the browser. By now, you should be aware of the advantages of using the same principles and code when writing the back-end and front-end of a complete solution.

We are going to go a step further in the next chapter. We will enable our application to get data from external sources and asynchronously update the user interface using Akka/Akka.js.

Interactive Browser

In this chapter, we will introduce the actor model by extending our shopping project. The extension will consist of a notification, provided to anyone connected to the website, about who is adding/removing a product to/from the cart.

Indeed, each time someone acts on the cart, a message will be broadcast to all of the connected browsers, and it will include the name of the user, the action (add or remove), and the product name.

The workflow will be as follows. When someone is connected to the website, a web socket will be opened between the browser and the server; a reference to that web socket will be kept at the server level, inside of an Actor.

As soon as an action is performed on the cart, a message with the username, action, and product name will be sent to the server through the web socket; the server will receive this message, transform it into an alarm message, and broadcast it to all of the connected browsers. Each browser will then show the alarm as a notification. As soon as the browser disconnects (or the web socket timeout is reached), the web socket reference will be removed from the server.

As you may have noticed, the term Actor was used in the preceding workflow. The theory of the actor model originated in 1973 (<https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask>), and multiple implementations have been created since, in many languages.

In this book, we are going to use the Akka framework. It was written by Jonas Bonér in 2009 and is based on the Scala Actor implementation created by Philipp Haller. We will only scratch the surface of the framework by providing an introduction to it. This framework would require an entire book to explain all of its features and possible patterns.

In this chapter, we will explain how to communicate between the client and the server using a web socket.

We will cover the following topics in this chapter:

- Actor models
- Implementing the server side
- Implementing the client side

The objectives of this chapter are to establish asynchronous communication between the browser and the server using a web socket and to use Actors to handle the communication at the server level. You will learn the following:

- How to create asynchronous communication between the client and the server
- How to use Actors

Actors

How can we define the term Actor? In our first attempt, we considered explaining it technically, using a threading model, concurrency, call stacks, a mailbox, and so on. Then, we realized that a technical description doesn't reflect the essence of a solution based on Actors.

In fact, every time we have to design a solution based on Actors, we can see the Actor as a human working in a company; this person has a name, and maybe an email address (**Actor reference**). The first important fact is that he is not alone; he is going to interact with others (**messaging**), receiving messages from his hierarchy and transmitting other messages to colleagues or subordinates (**supervisors**).

The imaginary company is structured using a hierarchy; a supervisor (**user guardian**) is looking at the health of its subordinate, and, when a problem is raised, if the supervisor can handle it, they will perform actions to fix it. If the error is unmanageable, the supervisor will escalate it to their own superior (**supervisor strategy**), and so on, until it reaches the director (**root guardian**).

Another similarity to communication between humans is that when you ask a colleague to do something and they do not answer. After a certain amount of time (a **timeout**), you might decide to ask again. If you still do not receive an answer, you might think that they are too busy and ask someone else. All of this protocol is performed asynchronously and is based on how long you are prepared to wait (**latency**).

Now, with these concepts in mind, technically, we could define an Actor as a lightweight unit of a process running on only one thread and handling messages one after the other; the Actor receives messages, processes them, and perhaps changes their internal states, based on the message. It then sends another message to the initial sender or any other Actor.

To perform all of this workflow, the Actor needs the following:

- A reference, to be reached from within the same room (JVM) or remotely

- A mailbox, to queue incoming messages
- A state, to save its private state
- A behavior, to act based on the received messages and current state
- A child Actor, as each Actor could potentially be a supervisor

The purpose of this book is not to make you an expert on the Akka framework; instead, this book should provide you with the basic knowledge to feel comfortable with the fundamental concepts. The concepts that you will learn will allow you to build an application, and, if you wish, to go deeper into other components of the framework.

As a reference, the complete Akka documentation can be found directly on the website of the Akka project
at <https://doc.akka.io/docs/akka/current/general/index.html>.

Let's get straight to work and look at how this can be implemented in the real world.

Setting up

To set up our project, we will need the Akka library, in order to create Actors on the server, and Notify.js. Notify.js is a JavaScript library used to pop up notifications on the browser; we picked this library because it does not have any dependencies on other frameworks.

To add this JavaScript library, just add the following to `build.sbt`, under `jsDependencies` in the `client` variable:

```
| jsDependencies ++= Seq(  
|   ...  
|   "org.webjars" % "notifyjs" % "0.4.2" / "notify.js")
```

The preceding code is for the configuration of the project.

Implementing the server side

At the server level, we need to open a communication channel between the server and the browser; once the communication is open, we need to implement the message reception and broadcast it to all of the connected browsers using Actors.

Creating the web socket route

To create the route, the `conf/routes` file needs to be modified by adding the following:

```
| GET /v1/cart/events controllers.WebSockets.cartEventWS
```

Notice that the route is defined in the same way as a regular web service call; so, the `GET` call on `/v1/cart/events` is routed to the `cartEventWS` method of the `controllers.WebSockets` instance.

Next, we need to create the `WebSockets` class in the `controllers` package of the server module, and add the `cartEventWS` method, as follows:

```
@Singleton
class WebSockets @Inject()(  
    implicit actorSystem: ActorSystem,  
    materializer: Materializer,  
    cc: ControllerComponents) extends AbstractController(cc) {  
  
    def cartEventWS = WebSocket.accept[String, String] {  
        implicit request  
        =>  
            ActorFlow.actorRef { out =>  
                // handle upstream  
            }  
    }  
}
```

There are not many lines of code, but a lot is happening in this snippet.

On the class constructor, Google Guice (the dependency injection used in Play) is going to inject `ActorSystem`. `ActorSystem` is the root guardian of the system; this is the top level in the hierarchy of Actors and is unique for every JVM.

Play uses Akka-stream underneath; a materializer is needed. First, let us explain these new terms. Akka-stream is an Akka component used to nicely handle streams, and it is exactly what we need to take care of our stream between the server and the browser. Akka-stream is well engineered; there is a clear separation between the definitions in the stream, such as where the data should be taken, how to process it and where to move it, and the stream runtime. To define the stream, a **Domain-Specific Language (DSL)** is available, and the

materializer is the runtime of the stream. That is why we need to provide `Materializer` in our code.

The upstream is created with `ActorFlow.actorRef { out => }`, where `out` is an Actor representing the browser. This function should return an Actor handling messages from the browser. We will come back to the implementation later.

To summarize, at this point, our server opens a new route on `/v1/cart/events`. At that entry point, a web socket connection is expected, and for each new connection, a new communication stream is started.

Well, it is time to code the communication handling; but what do we want to do?

Implementing BrowserManager

Each time a new connection is accepted (representing a new browser), we would like to keep a reference to that connection, so that events can be sent to it later. This connection container is handled by an Actor. This Actor will need an internal state with the list of connected browsers; we should add a new browser, and remove it once it is disconnected.

To create an Actor, we use the `Props` class from Akka, as follows:

```
| val managerActor = actorSystem.actorOf(  
|   BrowserManagerActor.props(),  
|   "manager-actor")
```

This Actor is created from the guardian root Actor; in our case, it is named `actorSystem`. From the system Actor, the `actorOf` method is called; this method expects `Props` as the first parameter, representing our Actor factory, and the name of the Actor as the second parameter. `BrowserManagerActor` is composed of a class and its companion object. The companion object is used to create an instance of the Actor, and it is a good practice to also define the messages related to that Actor, as follows:

```
| object BrowserManagerActor {  
|   def props() = Props(new BrowserManagerActor())  
|  
|   case class AddBrowser(browser: ActorRef)  
| }
```

We define the `props()` method used to create the Actor instance. There is nothing special here; the factory is defined on the companion object and is the best pattern to create an Actor. In this class, we also define the specific messages of this Actor; in this case, we only have one, named `AddBrowser`.

The `BrowserManagerActor` class implementation is as follows:

```
| private class BrowserManagerActor() extends Actor with ActorLogging {  
|  
|   val browsers: ListBuffer[ActorRef] = ListBuffer.empty[ActorRef]  
|  
|   def receive: Receive = {  
|  
|     case AddBrowser(b) =>  
|       context.watch(b)
```

```

    browsers += b
    log.info("websocket {} added", b.path)

    case CartEvent(user, product, action) =>
      val messageText = s"The user '$user' ${action.toString}
      ${product.name}"
      log.info("Sending alarm to all the browser with '{}' action: {}",
              messageText,
              action)
      browsers.foreach(_ ! Alarm(messageText, action).asJson.noSpaces)

    case Terminated(b) =>
      browsers -= b
      log.info("websocket {} removed", b.path)
  }
}

```

To become an Actor, the class needs to extend the `Actor` class; we also extend `ActorLogging`. This will provide a `log` object in our class, which can be used to log interesting information.

As noted earlier, we would like to keep the list of browsers connected to the server. For that purpose, we use the `browsers` variable, with `ListBuffer[ActorRef]` as the type.

Notice that we are using a mutable collection to define this list; this is completely fine in this context, as it is only accessible by this Actor and is guaranteed to be thread-safe.

It is possible to avoid this mutable variable by using another component from the Akka framework. This component is named **Final State Machine (FSM)**. The full details of the FSM implementation are out of the scope of this book. If you are interested, the link to the full documentation can be found at <https://doc.akka.io/docs/akka/current/fsm.html>.

Earlier, we mentioned that an Actor receives messages; this is the purpose of the `receive` method. It is a partial function, with `Any -> Unit` as a signature. To implement this function, we define the cases that we would like to handle; putting it in a different way, we define the messages that the Actor is handling.

Our manager Actor handles three messages, as follows:

- `case AddBrowser(b):` Here, a new connection is created, and `b` represents the browser Actor. First, by performing `context.watch(b)`, we ask the Akka framework to watch the `b` Actor and inform us when it dies by sending

a terminated message.

- `case CartEvent(user, product, action)`: Here, a message comes from a browser, namely, `cartEvent`. We would like to inform all of the connected browsers about this event. This is done by sending an alarm message to all of the browsers in our browser list. Notice that we convert the message to a JSON format using Circe.
- `case Terminate(b)`: This message is received because we are supervising the `b` Actor. The `b` Actor dies, and the only thing to do is remove it from our list of browsers.

We are almost done. With this Actor, we keep track of the connected browsers and send an alarm when any of them emits an event.

But wait; something looks suspicious. Indeed, we never sent the `AddBrowser` and `cartEvent` messages to our manager. Who should send them? The answer is in the next section.

Handling WebSocket

Going back to the `Websockets` class, and, more specifically, to the `cartEventWS` method, we can finish the implementation, as follows:

```
def cartEventWS = WebSocket.accept[String, String] { implicit request =>
    ActorFlow.actorRef{out =>
        Logger.info(s"Got a new websocket connection from
        ${request.host}")
        managerActor ! BrowserManagerActor.AddBrowser(out)
        BrowserActor.props(managerActor)
    }
}
```

After logging, we send the `AddBrowser` message to the manager by using the `!` command (pronounced bang); this is syntactic sugar, and we could also use the `.tell()` method.

`ActorFlow.actorRef` needs `ActorRef` to handle the upstream of the web socket; for that purpose, we create `BrowserActor` by using the `props` function of the `BrowserActor` companion object, as follows:

```
object BrowserActor {
    def props(browserManager :ActorRef) =
        Props(new BrowserActor(browserManager))
}
```

`BrowserActor` references the manager; indeed, the manager has the responsibility of sending the message to all of the browsers. The `BrowserActor` class implementation is as follows:

```
class BrowserActor(browserManager: ActorRef) extends Actor with ActorLogging {
    def receive = {
        case msg: String =>
            log.info("Received JSON message: {}", msg)
            decode[CartEvent](msg) match {
                case Right(cartEvent) =>
                    log.info("Got {} message", cartEvent)
                    browserManager forward cartEvent
                case Left(error) => log.info("Unhandled message : {}", error)
            }
    }
}
```

This implementation gets all of the messages coming from the socket in string format, converts them into `CartEvent` using Circe, and forwards them to the

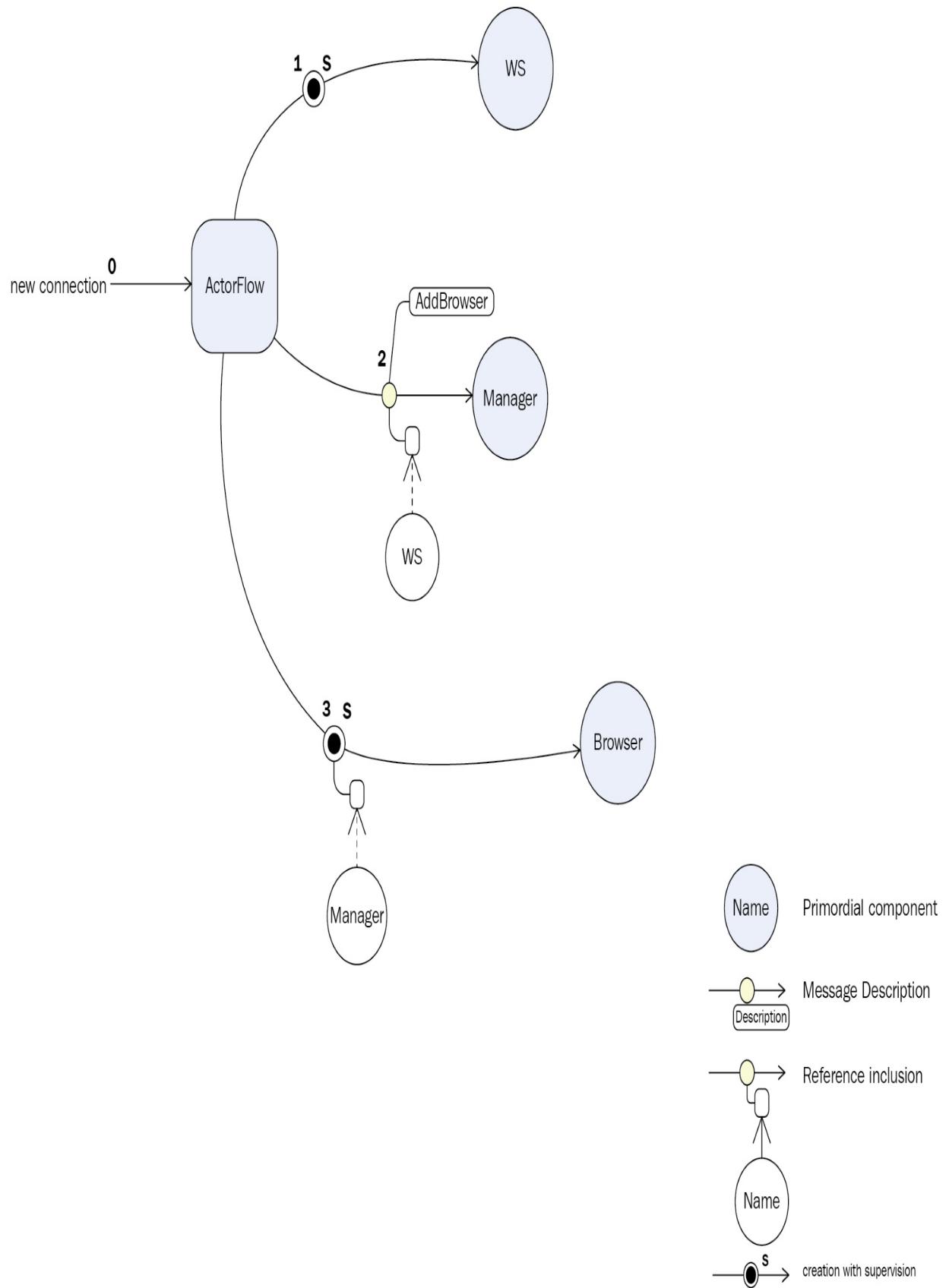
browser manager.

Keep in mind that the message flow can become more complicated; that is why it is a good idea to create a diagram of the system of Actors.

A diagram of Actors

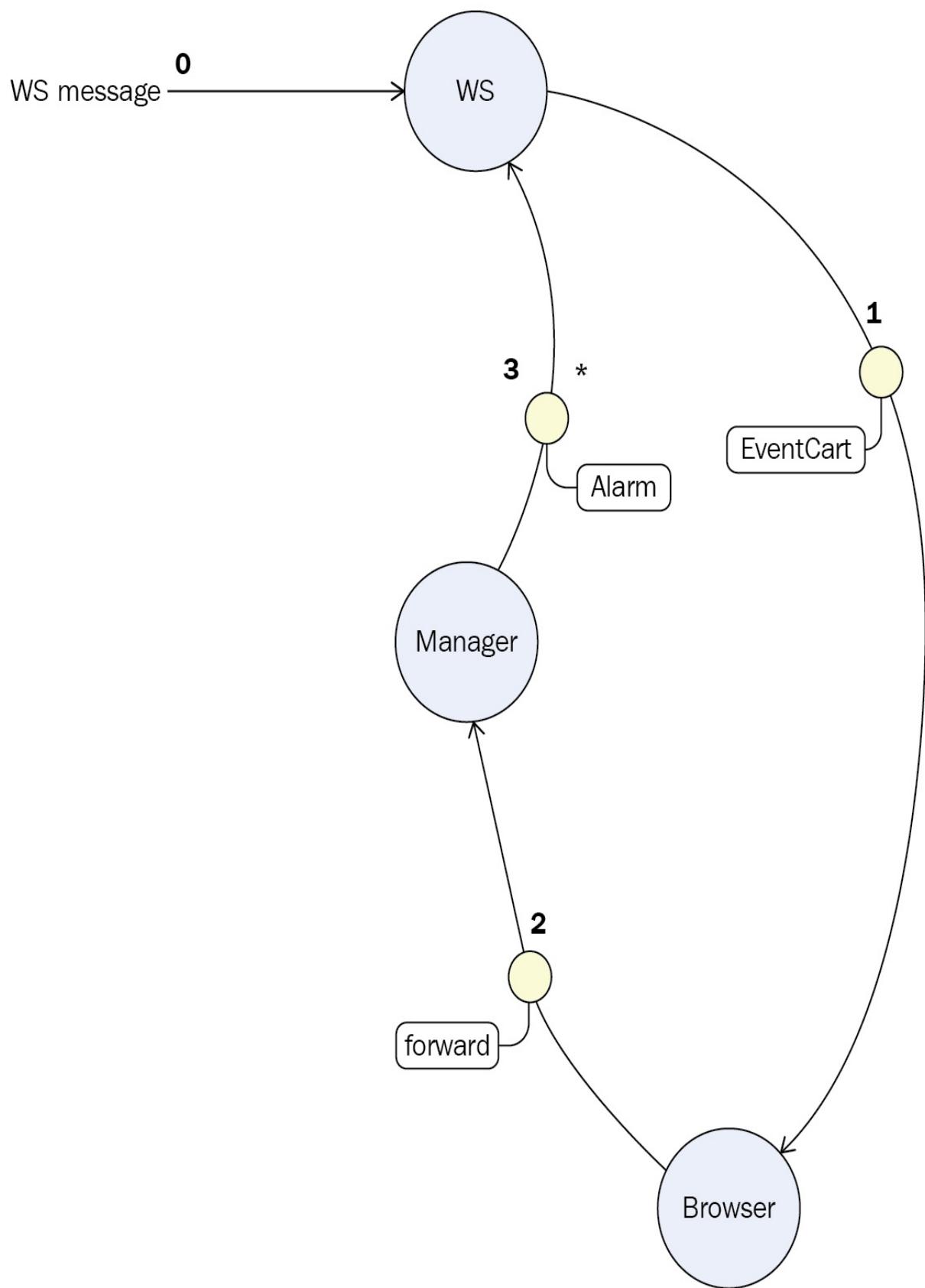
It is sometimes necessary to represent the Actor flow using a diagram. The more Actors there are in your system, the more difficult it is to picture the entire workflow, especially if you do not work on the code for a while and then come back to it.

The following is a diagram of our project, illustrating the workflow when a new browser is connected:



With this kind of diagram, you can clearly understand who is creating the Actors, and can also understand the sequences of messages between them.

The following diagram illustrates a message being sent from the browser:



Notice that the start of the third message indicates that the alarm is sent to multiple instances of the web socket.



For OmniGraffle users, you can find a stencil to create these diagrams at Diagramming Reactive Systems | Graffletopia (<https://www.graffletopia.com/stencils/1540>).

We have now finished looking at Actors; we are missing many features, but the aim was to provide you with enough knowledge to understand the basics of this beautiful framework.

The server is now completely implemented, and we can safely move to the client side.

Implementing the client side

In the following sections, we will look at the client side. On the client side, we have to initiate the web socket connection with the server, send a `CartEvent` when a product is added or removed from the cart, and show alerts when other browsers make changes to the cart.

First, let's create the connection with the server using a web socket.

Adding the web socket

To add the web socket to the client, we are going to use the `UIManager` object, which is the entry point of the client. In `Scala.js`, `WebSocket` is part of the framework; edit the `UIManager` and add the web socket property to it, as follows:

```
| val webSocket: WebSocket = getWebSocket
```

Some configuration is needed to create the `WebSocket`. We encapsulate all of the initializations into a function named `getWebSocket`, as follows:

```
private def getWebSocket: WebSocket = {
  val ws = new WebSocket(getwebsocketUri(dom.document,
    "v1/cart/events"))

  ws.onopen = { (event: Event) =>
    println(s"webSocket.onOpen `${event.type}`")
    event.preventDefault()
  }

  ws.onerror = { (event: Event) =>
    System.err.println(s"webSocket.onError `${event.getClass}`")
  }

  ws.onmessage = { (event: MessageEvent) =>
    println(s"[webSocket.onMessage] `${event.data.toString}`...")
    val msg = decode[Alarm](event.data.toString)
    msg match {
      case Right(alarm) =>
        println(s"[webSocket.onMessage] Got alarm event : $alarm")
        notify(alarm)
      case Left(e) =>
        println(s"[webSocket.onMessage] Got a unknown event : $msg")
    }
  }

  ws.onclose = { (event: CloseEvent) =>
    println(s"webSocket.onClose `${event.type}`")
  }
  ws
}
```

To create a `WebSocket`, we first need to give the URL of the server and then handle all of the events happening in the socket. To get the URL of the server, we use a utility function named `getwebsocketUri`:

```
private def getwebsocketUri(document: Document, context: String): String = {
  val wsProtocol =
    if (dom.document.location.protocol == "https:")
      "wss"
    else
```

```
    "ws"
  s"$wsProtocol://${{
    dom.document.location.host
}}/$context"
}
```

This function just checks the protocol and defines the `WebSocket` protocol as `wss` if encrypted or `ws` if not encrypted. Then, the full URL is built by using string interpolation. In production, we usually use SSL, but when developing, we do not use encryption.

Once the URL is defined, we define all of the socket event handlers as follows:

- `onopen`: When a new connection is created, we just log it and mark the event as canceled, so it will not be taken into consideration if another handler receives it.
- `onerror`: Just log the error in the error pipe.
- `onmessage`: When a message is received, we use Circe to decode it and check whether it is an alarm message. If that is the case, we call `notify(alarm)`, otherwise, we just log the fact that we received an unknown message. The `notify(alarm)` will be explained later.
- `onclose`: Again, we just log this event.

Now, we have defined the socket and it is ready to be used; if this code is run, a connection to the server will be created as soon as the page is browsed. But before that, we need to define the notification system.

Notifying the user

To notify the user, we picked a JavaScript library called Notify.js. Notify.js is a jQuery plugin without any dependencies, and it has a simple interface. We will implement only one method: `$.notify(string, options)`.

As Notify.js is a jQuery plugin, we need to extend jQuery with this function.

Extending jQuery

Extending jQuery using Scala.js is done by extending the famous jQuery \$ symbol. We can create a file named `Notify.scala` in the `io.fscala.shopping.client` package, in the client project.

In this file, we can first define the extension with the following piece of code:

```
@js.native
@JSGlobal("$")
object NotifyJS extends js.Object {
  def notify(msg: String, option: Options): String = js.native
}
```

An object named `NotifyJS` is defined, extending the Scala.js object named `js.Object`. It is necessary to inform the compiler that we are creating a facade of an existing JavaScript library.

The first annotation is `@js.native`; this annotation tells the compiler that the implementation is completely done in JavaScript. The second annotation is `@JSGlobal("$")` ; this is to express the fact that the API we are extending is a JavaScript class, and this class is named \$. The last thing is to define the signature of the function that we would like to call and to use `js.native` as implementation; again, the compiler is going to make the bridge between our code and the JavaScript implementation.

The parameters of the function are `msg` (for the first one) and `options` (for the second). `options` need to be defined, as this is part of the facade.

By reading the Notify.js documentation (<https://notifyjs.jpillora.com/>), you can see that there are a lot of options available, such as the positioning of the notification and the animation of the notification.

From the Notify.js documentation, we can get the definitions of all of the options, as follows:

```
{
  // whether to hide the notification on click
  clickToHide: true,
  // whether to auto-hide the notification
```

```

autoHide: true,
// if autoHide, hide after milliseconds
autoHideDelay: 5000,
// show the arrow pointing at the element
arrowShow: true,
// arrow size in pixels
arrowSize: 5,
// position defines the notification position though uses the
// defaults below
position: '....',
// default positions
elementPosition: 'bottom left',
globalPosition: 'top right',
// default style
style: 'bootstrap',
// default class (string or [string])
className: 'error',
// show animation
showAnimation: 'slideDown',
// show animation duration
showDuration: 400,
// hide animation
hideAnimation: 'slideUp',
// hide animation duration
hideDuration: 200,
// padding between element and notification
gap: 2
}

```

We can create an options class in Scala, as follows:

```

@ScalaJSDefined
trait Options extends js.Object {
  // whether to hide the notification on click
  var clickToHide: js.UndefOr[Boolean] = js.undefined
  // whether to auto-hide the notification
  var autoHide: js.UndefOr[Boolean] = js.undefined
  // if autoHide, hide after milliseconds
  var autoHideDelay: js.UndefOr[Int] = js.undefined
  // show the arrow pointing at the element
  var arrowShow: js.UndefOr[Boolean] = js.undefined
  // arrow size in pixels
  var arrowSize: js.UndefOr[Int] = js.undefined
  // position defines the notification position
  // though uses the defaults below
  var position: js.UndefOr[String] = js.undefined
  // default positions
  var elementPosition: js.UndefOr[String] = js.undefined
  var globalPosition: js.UndefOr[String] = js.undefined
  // default style
  var style: js.UndefOr[String] = js.undefined
  // default class (string or [string])
  var className: js.UndefOr[String] = js.undefined
  // show animation
  var showAnimation: js.UndefOr[String] = js.undefined
  // show animation duration
  var showDuration: js.UndefOr[Int] = js.undefined
  // hide animation
  var hideAnimation: js.UndefOr[String] = js.undefined
  // hide animation duration
  var hideDuration: js.UndefOr[Int] = js.undefined
  // padding between element and notification
}

```

```
| } var gap: js.UndefOr[Int] = js.undefined
```

The `@ScalaJSDefined` annotation tells the compiler that this is a type defined in Scala, and not in JavaScript.

Then, for each property, we check the type candidate in the documentation and define it using `js.UndefOr[Int]`; this type acts like a bridge between the `undefinedJavaScript` and `options` types in Scala.

Everything is now defined for our facade; we can use this facade and implement the missing `notify(alarm)` function of the `UIManager` class, as follows:

```
private def notify(alarm: Alarm): Unit = {
  val notifyClass = if (alarm.action == Add) "info" else "warn"
  NotifyJS.notify(alarm.message, new Options {
    className = notifyClass
    globalPosition = "right bottom"
  })
}
```

First, we check the type of action to set the class name of the notification, and then we use the `notify` native call by passing the message and the notification's options.

We are all done. Now, if the server is running, each time you add or remove a product in the cart, a notification will be sent to all of the connected browsers, as shown in the following screenshot:

The screenshot shows a web browser window titled "Shopping Page" at the URL "localhost:9000". The page displays three products:

Product	Description	Quantity	Unit Price	Total Price
NAO	NAO is an humanoid robot.	X 1	3500	3500
PEPPER	PEPPER is a robot moving with wheels and with a screen as human interaction	X 1	7000	7000
BEOBOT	Beobot is a multipurpose robot.	X 1	159	159

Each product row contains an "Add to Cart" button. Below the table, two notifications are shown:

- A yellow notification: "⚠ The user 'user-749' Remove PEPPER"
- A blue notification: "● The user 'user-968' Add NAO"

The browser's address bar shows "localhost:9000". The top right corner of the browser window has a menu icon.

The shopping page with the notification for cart updates

Summary

In this chapter, you learned how to create a WebSocket communication between the server and the browser. At the server level, we kept a reference to all the browsers connected, so that events could be dispatched to all the browsers. An important piece of the system is the actor model, defined at the server level. We learned that the actor model programming paradigm is adequate as soon as we have an interaction between asynchronous systems.

You learned that a diagram of the interactions between Actors can be helpful as your system is growing. It is particularly useful when someone needs to go back to the code after being away for a while. As we are not calling methods but sending messages to `ActorRef`, the navigation in the IDE is not easy, so it is difficult to understand the flow just by reading the code.

Once the first steps in this framework are made, development is natural and close to real-world interaction.

We also introduced Akka. Akka is a complete framework, separated into different modules. We strongly encourage you to take a spin in the Akka website at <https://akka.io/>.

On the client side, thanks to Scala.js, the integration of a framework is completed with only a few lines of code; once defined in Scala, we can use all of the knowledge learned from a backend system and apply it to the front. This is especially true when we are sharing the code between the backend and frontend.

This ends the chapter. At this point, you should have the necessary information to build your own client-server program. In the next chapters, we will introduce you to how to use Scala to process and analyze large amounts of data, using Apache Spark, Kafka and Zeppelin.

Fetching and Persisting Bitcoin Market Data

In this chapter, we will develop a data pipeline to fetch, store, and, later on, analyze bitcoin transaction data.

After an introduction to Apache Spark, we will see how to call a REST API to fetch transactions from a cryptocurrency exchange. A cryptocurrency exchange allows customers to trade digital currencies, such as bitcoin, for fiat currencies, such as the US dollar. The transaction data will allow us to track the price and quantity exchanged at a certain point in time.

We will then introduce the Parquet format. This is a columnar data format that is widely used for big data analytics. After that, we will build a standalone application that will produce a history of bitcoin/USD transactions and save it in Parquet. In the following chapter, we will use Apache Zeppelin to query and analyze the data interactively.

The volume of data that we will deal with is not very large, but the tools and techniques used will be the same if the data were to grow or if we were to store the data for more currencies or from different exchanges. The benefit of using Apache Spark is that it can scale horizontally and that you can just add more machines to your cluster to speed up your processing, without having to change your code.

Another advantage of using Spark is that it makes it easy to manipulate table-like data structures and to load and save them from/to different formats. This advantage remains even when the volume of data is small.

In this chapter, we will cover the following topics:

- Apache Spark
- Calling the REST API of a cryptocurrency exchange
- Parquet format and partitioning

After we are done with this chapter, we have learned a few things such as the following:

- How to store large volumes of data
- How to use the Spark Dataset API
- How to use the `IO` `Monad` to control side effects

Setting up the project

Create a new SBT project. In IntelliJ, go to **File | New | Project | Scala | sbt**.

Then edit `build.sbt` and paste the following:

```
name := "bitcoin-analyser"

version := "0.1"

scalaVersion := "2.11.11"
val sparkVersion = "2.3.1"

libraryDependencies ++= Seq(
  "org.lz4" % "lz4-java" % "1.4.0",
  "org.apache.spark" %% "spark-core" % sparkVersion % Provided,
  "org.apache.spark" %% "spark-core" % sparkVersion % Test classifier
  "tests",
  "org.apache.spark" %% "spark-sql" % sparkVersion % Provided,
  "org.apache.spark" %% "spark-sql" % sparkVersion % Test classifier "tests",
  "org.apache.spark" %% "spark-catalyst" % sparkVersion % Test classifier "tests",
  "com.typesafe.scala-logging" %% "scala-logging" % "3.9.0",
  "org.scalatest" %% "scalatest" % "3.0.4" % "test",
  "org.typelevel" %% "cats-core" % "1.1.0",
  "org.typelevel" %% "cats-effect" % "1.0.0-RC2",
  "org.apache.spark" %% "spark-streaming" % sparkVersion % Provided,
  "org.apache.spark" %% "spark-sql-kafka-0-10" % sparkVersion %
    Provided exclude ("net.jpountz.lz4", "lz4"),
  "com.pusher" % "pusher-java-client" % "1.8.0")

  scalacOptions += "-Ypartial-unification"

// Avoids SI-3623
target := file("/tmp/sbt/bitcoin-analyser")
```

We use Scala 2.11 because, at the time of writing, Spark does not provide its libraries for Scala 2.12. We are going to use the following:

- `spark-core` and `spark-sql` for reading the transactions and saving them to Parquet. The `Provided` configuration will make SBT exclude these libraries when we package the application in an assembly JAR file.
- `ScalaTest` for testing our code.
- `scala-logging`, a convenient and fast logging library that wraps SLF4J.
- `cats-core` and `cats-effects` for managing our side effects with the `IO` `Monad`.
- `spark-streaming`, `spark-sql-kafka`, and `pusher` for the next chapter.

The `-Ypartial-unification` compiler option is required by `cats`.

The last line makes SBT write the classes to the `/tmp` folder, in order to avoid a *file name too long* bug with the Linux encrypted home folders. You might not need it on your platform.

If you do not wish to retype the code examples, you can check out the complete project code from GitHub at <https://github.com/PacktPublishing/Scala-Programming-Projects>.

Understanding Apache Spark

Spark is an open source framework built to perform analytics on large datasets. Unlike other tools such as R, Python, and MathLab that are using in-memory processing, Spark gives you the possibility to scale out. And thanks to its expressiveness and interactivity, it also improves developer productivity.

There are entire books dedicated to Spark. It has a vast number of components and lots of areas to explore. In this book, we aim to get you started with the fundamentals. You should then be more comfortable exploring the documentation if you want to.

The purpose of Spark is to perform analytics on a collection. This collection could be in-memory and you could run your analytics using multiple threads, but if your collection is becoming too large, you are going to reach the memory limit of your system.

Spark solved this issue by creating an object to hold all of this data. Instead of keeping everything in the local computer's memory, Spark chunks the data into multiple collections and distributes it on multiple computers. This object is called an **RDD** (short for **Resilient Distributed Dataset**). RDD keeps references to all of the distributed chunks.

RDD, DataFrame, and Dataset

The core concept of Spark is the RDD. From the user's point of view, for a given type `A`, `RDD[A]` looks similar to a standard Scala collection, such as `Vector[A]`: they are both **immutable** and share many well-known methods, such as `map`, `reduce`, `filter`, and `flatMap`.

However, the RDD has some unique characteristics. They are as follows:

- **Lazy:** When you call a **transformation** function, such as `map` or `filter`, nothing happens immediately. The function call is just added to a computation graph that is stored in the RDD class. This computation graph is executed when you subsequently call an **action** function, such as `collect` or `take`.
- **Distributed:** The data in the RDD is split in several partitions that are scattered across different **executors** in a cluster. A **task** represents a chunk of data and the transformation that must be applied to it.
- **Resilient:** If one of the executors dies when you execute a job, Spark automatically resends the tasks that were lost to another executor.

Given two types, `A` and `B`, `RDD[A]` and `RDD[B]` can be joined together to obtain `RDD[(A, B)]`. For instance, consider `case class Household(id: Int, address: String)` and `case class ElectricityConsumption(houseHoldId: Int, kwh: Double)`. If you want to count the number of households consuming more than 2 kWh, you could do either of the following:

- Join `RDD[Household]` with `RDD[ElectricityConsumption]` and then apply `filter` on the result
- Apply `filter` to `RDD[ElectricityConsumption]` first, and then join it with `RDD[Household]`

The result will be the same but the performances will be different; the second algorithm will be faster. Wouldn't it be nice if Spark could perform this kind of optimization for us?

Spark SQL

The answer is yes and the module is called Spark SQL. Spark SQL sits on top of Spark Core and allows the manipulation of structured data. Unlike with the basic RDD API, the `DataFrame` API provides more information to the Spark engine. Using this information, it can change the execution plan and optimize it.

You can also use the module to execute SQL queries as you would with a relational database. It makes it easy for people comfortable with SQL to run queries on heterogeneous sources of data. You can, for instance, join a data table coming from a CSV file with another one stored in Parquet in a Hadoop filesystem and with yet another one coming from a relational database.

Dataframe

Spark SQL is composed of three main APIs:

- The SQL literal syntax
- The `DataFrame` API
- `DataSet`

`DataFrame` is conceptually the same as a table in the relational database. The data is distributed in the same way as in an RDD. `DataFrame` has a schema but is untyped. You can create `DataFrame` from an RDD or manually build it. Once created, `DataFrame` will contain a schema that maintains the name and type for each column (field).

If you then want to use `DataFrame` in an SQL query, all you need to do is create a named view (equivalent to the table name in the relational database) using the `Dataframe.createTempView(viewName: String)` method. In the SQL query, the fields that are available in the `SELECT` statement will come from the schema of `DataFrame` and the name of the table used in the `FROM` statement will come from `viewName`.

Dataset

As Scala developers, we are used to working with types and with a friendly compiler that infers types and tells us our mistakes. The problem with `DataFrame` API and Spark SQL is that you can write a query such as `Select lastname From people`, but in your `DataFrame`, you might not have a `lastname` column, but `surname` one. In this case, you are only going to discover that mistake at runtime with a nasty exception!

Wouldn't it be nice to have a compilation error instead?

This is why Spark introduced `Dataset` in version 1.6. `Dataset` attempts to unify the RDD and the `DataFrame` APIs. `Dataset` has a type parameter, and you can use anonymous functions to manipulate the data as you would with an RDD or a vector.

Actually, `DataFrame` is, in fact, a type alias for `DataSet[Row]`. This means you can seamlessly mix the two APIs and use in the same query a filter using a Lambda expression followed by another filter using a `DataFrame` operator.

In the next sections, we are only going to use `Dataset`, which is a good compromise between code quality and performance.

Exploring the Spark API with the Scala console

If you are not already familiar with Spark, it can be a bit intimidating to write Spark jobs straight away. To make it easier, we are first going to explore the API using a Scala console. Start a new Scala console (*Ctrl + Shift + D* in IntelliJ), and type the following code:

```
| import org.apache.spark.sql.SparkSession  
| val spark = SparkSession.builder().master("local[*]").getOrCreate()  
| import spark.implicits._
```

This will initialize a new Spark session and bring some handy implicit in scope. The master "local[*]" URL means that we will use all of the cores available on the localhost when running jobs.

The Spark session is available to accept new jobs. Let's use it to create `Dataset` containing a single string:

```
| import org.apache.spark.sql.Dataset  
|  
| val dsString: Dataset[String] = Seq("1", "2", "3").toDS()  
| // dsString: org.apache.spark.sql.Dataset[String] = [value: string]
```

The implicit that we imported earlier let us use the `.toDS()` function on `Seq` to produce `Dataset`. We can observe that the `.toString` method that was called by the Scala console was output by the schema of `Dataset`—it has a single column `value` of the `String` type.

However, we could not see the content of `Dataset`. This is because `Dataset` is a lazy data structure; it just stores a computation graph, which is not evaluated until we call one of the action methods. Still, for debugging, it is very handy to be able to evaluate `Dataset` and print its content. For that we need to call `show`:

```
| dsString.show()
```

You should see the following output:

```
| +---+  
| |
```

value
1
2
3

`show()` is an **action**; it will submit a job to the Spark cluster, collect the results in the driver, and print them. By default, `show` limits the number of rows to 20 and truncates the columns. You can call it with extra parameters if you want more information.

Now we would like to convert each string to `int`, in order to obtain `Dataset[Int]`. We have two ways of doing that.

Transforming rows using map

Type the following in the Scala console:

```
| val dsInt = dsString.map(_.toInt)
| // dsInt: org.apache.spark.sql.Dataset[Int] = [value: int]
| dsInt.explain()
```

You should see something similar to this:

```
== Physical Plan ==
*(1) SerializeFromObject [input[0, int, false] AS value#96]
+- *(1) MapElements <function1>, obj#95: int
  +- *(1) DeserializeToObject value#91.toString, obj#94:
    java.lang.String
      +- LocalTableScan [value#91]
```

The `explain()` method shows the execution plan that will be run if we call an action method, such as `show()` or `collect()`.

From this plan, we can deduce that calling `map` is not very efficient. Indeed, Spark stores the rows of `Dataset` off-heap in binary format. Whenever you call `map`, it has to deserialize this format, apply your function, and serialize the result in binary format.

Transforming rows using select

A more efficient way to transform rows is to use `select(cols: Column*)`:

```
import org.apache.spark.sql.DataFrame
import org.apache.spark.sql.types.IntegerType

val df = ds.select($"value".cast(IntegerType))
// df: org.apache.spark.sql.DataFrame = [value: int]
val dsInt = df.as[Int]
// dsInt: org.apache.spark.sql.Dataset[Int] = [value: int]
```

The implicits that we imported earlier let us use the `$"columnName"` notation to produce a `Column` object from `String`. The string after the `$` sign must refer to a column that exists in the `DataFrame` source; otherwise, you would get an exception.

We then call the `.cast` method to transform each `String` into `Int`. But, at this stage, the resulting `df` object is not of the `Dataset[Int]` type; it is `DataFrame`. `DataFrame` is actually a type alias for `Dataset[Row]`, and `Row` is akin to a list of key-value pairs. `DataFrame` is a representation of the distributed data in an untyped way. The compiler does not know the type or names of each column; they are only known at runtime.

In order to obtain `Dataset[Int]`, we need to cast the type of the elements using `.as[Int]`. This would fail at runtime if the elements of `DataFrame` cannot be cast to the target type.



Force a specific type for the elements of your `Dataset`; this will make your programs safer. You should only expose `DataFrame` in a function if it genuinely does not know what the types of the columns will be at runtime; for instance, if you are reading or writing arbitrary files.

Let's see what our `explain` plan looks like now:

```
| dsInt.explain()
```

You should see this:

```
| == Physical Plan ==
| LocalTableScan [value#122]
```

This time we can see that there is no extra serialization/deserialization step. The

evaluation of this `dataset` will be faster than when we used `map`.



Exercise: Filter the elements of `dataset[Int]` to keep only the elements that are greater than 2. First use `filter(func: Int => Boolean)`, and then use `filter(condition: Column)`. Compare the execution plans for both implementations.

The conclusion of this is that you should prefer functions that use `column` arguments whenever possible. They can fail at runtime, as opposed to the type-safe alternatives because they can refer to column names that do not exist in your `dataset`. However, they are more efficient.

Fortunately, there is an open source library called **Frameless** that can let you use these efficient methods in a type-safe way. If you are writing large programs that use `Dataset`, I recommend that you check it out here: <https://github.com/typelevel/frameless>.

Execution model

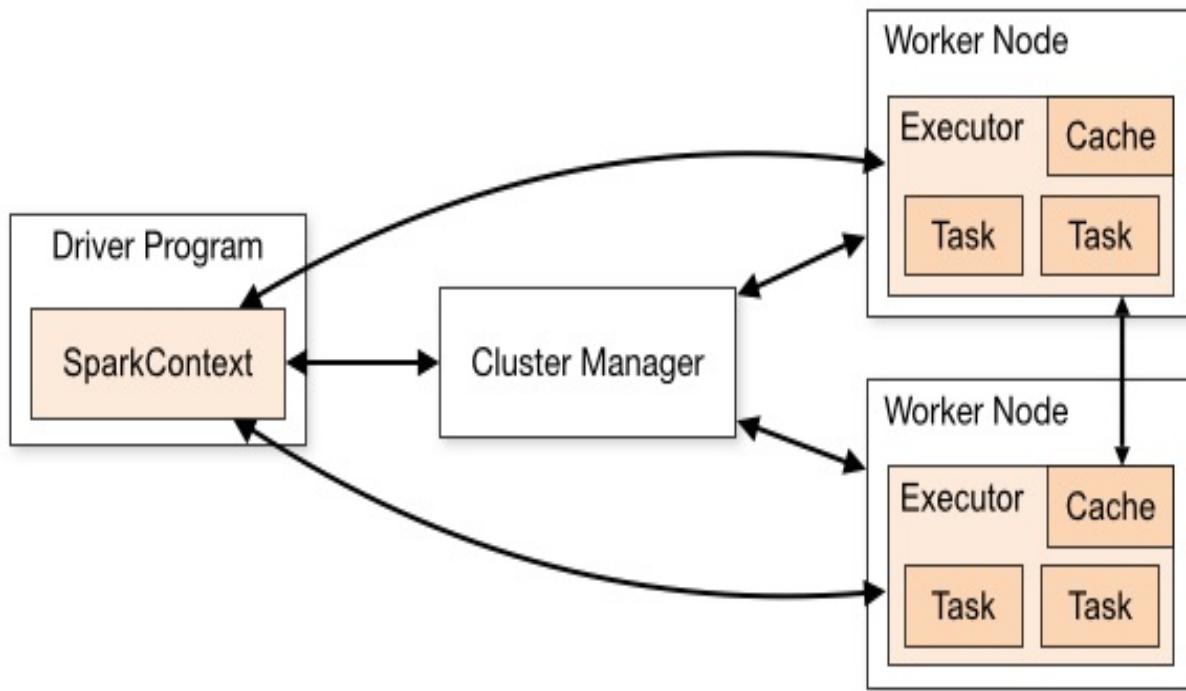
The methods available in `Dataset` and the RDD are of two kinds:

- **Transformations:** They return a new `Dataset` API that will apply the transformation later when an action method is called. For instance, `map`, `filter`, `join`, and `flatMap`.
- **Actions:** They trigger the execution of a Spark job, for instance, `collect`, `take`, and `count`.

When a **job** is triggered by an action method, it is divided into several **stages**. A stage is a part of a job that can be run without having to **shuffle** the data across different nodes of the cluster. It can encompass several transformations, such as `map` and `filter`. But as soon as one transformation, such as `join`, requires the data be moved (shuffling), another stage must be introduced.

The data contained in `Dataset` is split into several **partitions**. The combination of a stage (code to execute) with a partition (data used by the stage) is a **task**. The ideal parallelism would be achieved when you have *nb of tasks = nb of cores* in the cluster. When you need to optimize a job, it can be beneficial to repartition your data to better match the number of cores at your disposal.

When Spark starts executing a job, the **Driver** program distributes the tasks to all the executors of the cluster:



The preceding diagram is described as follows:

- The driver is on the same JVM as the code that called the action method
- An executor runs on its own JVM on a remote node of the cluster. It can use many cores to execute several tasks in parallel

The Spark **master** coordinates several **worker** nodes in a cluster. When you start a Spark application, you have to specify the URL of the master. It will then ask its worker to spawn executor processes that will be dedicated to the jobs of your application. At a given point in time, a worker can manage several executors running completely different applications.

When you want to quickly run or test an application without using a cluster, you can use a **local** master. In this special mode, only one JVM is used for the master, driver, and executor.

Implementing the transaction batch producer

In this section, we will first discuss how to call a REST API to fetch BTC/USD transactions. Then we will see how to use Spark to deserialize the JSON payload into a well-typed distributed dataset.

After that, we will introduce the parquet format and see how Spark makes it easy to save our transactions in this format.

With all of these building blocks, we will then implement our program in a purely functional way using the **Test-Driven-Development (TDD)** technique.

Calling the Bitstamp REST API

Bitstamp is a cryptocurrency exchange that people use to trade a cryptocurrency, such as bitcoin, for a conventional currency, such as US dollar or euro. One of the good things about Bitstamp is that it provides a REST API, which can be used to get information about the latest trades, and can also be used to send orders if you have an account.

You can find out more here: <https://www.bitstamp.net/api/>.

For this project, the only endpoint we are interested in is the one that gets the latest transactions that happened on the exchange. It will give us an indication of the price and of the quantity of currency exchanged in a given period of time. This endpoint can be called with the following URL: <https://www.bitstamp.net/api/v2/transactions/btcusd/?time=hour>.

If you paste this URL in your favorite browser, you should see a JSON array containing all of the BTC (Bitcoin)/USD (US dollar) transactions that happened during the last hour. It should look similar to this:

```
[  
  {  
    "date": "1534582650",  
    "tid": "72519377",  
    "price": "6488.27",  
    "type": "1",  
    "amount": "0.05000000"  
  },  
  {  
    "date": "1534582645",  
    "tid": "72519375",  
    "price": "6488.27",  
    "type": "1",  
    "amount": "0.01263316"  
  },  
  ...  
]
```

In the previous result, if we inspect the first transaction, we can see that 0.05 bitcoins were sold ("type": "1" means sell) at a price of 6488.27 USD for 1 BTC.

There are many Java and Scala libraries to call a REST endpoint, but to keep things simple, we are just going to use the Scala and Java SDK to call the

endpoint. Start a new Scala console, and run this code:

```
import java.net.URL  
import scala.io.Source  
  
val transactions = Source.fromURL(new URL("https://www.bitstamp.net/api/v2/transactions/"))
```

With the help of the `scala.io.Source` class, we can get the HTTP response in a string. This is the first building block of our program. The next thing we need to do is parse the JSON objects into a collection of Scala objects to make them easier to manipulate.



As we read the whole HTTP response in a string in one go, we need enough memory on the driver process to keep that string in the heap. You might think that it would be better to read it with `InputStream`, but unfortunately, it is not possible with Spark Core to split a stream of data. You would have to use Spark Streaming to do this.

Parsing the JSON response

We have observed that when we call the Bitstamp endpoint, we get a string containing a JSON array, and each element of the array is a JSON object that represents a transaction. But it would be nicer to have the information in a Spark `Dataset`. This way, we will be able to use all of the powerful Spark functions to store, filter, or aggregate the data.

Unit testing jsonToHttpTransaction

First, we can start by defining a case class that represents the same data as in our JSON payload. Create a new package, `coinyser`, and then a class, `coinyser.HttpTransaction`, in `src/main/scala`:

```
package coinyser

case class HttpTransaction(date: String,
                           tid: String,
                           price: String,
                           `type`: String,
                           amount: String)
```



In Scala, if you want to use a variable name already defined as a Scala keyword, you can enclose the variable with backticks such as the `type` variable name in this example: ``type``: `String`.

This class has the same attribute names with the same types (all string) as the JSON objects. The first step is to implement a function that transforms a JSON string into `Dataset[HttpTransaction]`. For this purpose, let's create a new test class, `coinyser.BatchProducerSpec`, in `src/test/scala`:

```
package coinyser

import org.apache.spark.sql._
import org.apache.spark.sql.test.SharedSparkSession
import org.scalatest.{Matchers, WordSpec}

class BatchProducerSpec extends WordSpec with Matchers with SharedSparkSession {
    val httpTransaction1 =
        HttpTransaction("1532365695", "70683282", "7740.00", "0",
                        "0.10041719")
    val httpTransaction2 =
        HttpTransaction("1532365693", "70683281", "7739.99", "0",
                        "0.00148564")

    "BatchProducer.jsonToHttpTransaction" should {
        "create a Dataset[HttpTransaction] from a Json string" in {
            val json =
                """[{"date": "1532365695", "tid": "70683282", "price": "7740.00", "type": "0", "amount": "0.10041719"}, {"date": "1532365693", "tid": "70683281", "price": "7739.99", "type": "0", "amount": "0.00148564"}]""".stripMargin

            val ds: Dataset[HttpTransaction] =
                BatchProducer.jsonToHttpTransactions(json)
            ds.collect() should contain theSameElementsAs
                Seq(httpTransaction1, httpTransaction2)
        }
    }
}
```

Our test extends `SharedSparkSession`. This trait provides an implicit `SparkSession` that can be shared across several tests.

First, we defined a string containing a JSON array with two transactions that we extracted from Bitstamp's endpoint. We defined two instances of `HttpTransaction` that we expect to have in our `Dataset` outside of the test because we will reuse them in another test later on.

After the call to `jsonToHttpTransaction` that we are going to implement, we obtain `Dataset[HttpTransaction]`. However, Spark's `Dataset` is lazy—at this stage, nothing has been processed yet. In order to *materialize* `Dataset`, we need to force its evaluation by calling `collect()`. The return type of `collect()` here is `Array[HttpTransaction]`, and we can therefore use ScalaTest's assertion, `toContain theSameElementsAs`.

Implementing jsonToHttpTransaction

Create the `coinyser.BatchProducer` class and type the following code:

```
package coinyser

import java.time.Instant
import java.util.concurrent.TimeUnit

import cats.Monad
import cats.effect.{IO, Timer}
import cats.implicits._
import org.apache.spark.sql.functions.{explode, from_json, lit}
import org.apache.spark.sql.types._
import org.apache.spark.sql.{Dataset, SaveMode, SparkSession}

import scala.concurrent.duration._

object BatchProducer {

    def jsonToHttpTransactions(json: String)(implicit spark: SparkSession): Dataset[HttpTransaction] = {
        import spark.implicits._

        val ds: Dataset[String] = Seq(json).toDS()
        val txSchema: StructType = Seq.empty[HttpTransaction].schema
        val schema = ArrayType(txSchema)
        val arrayColumn = from_json($"value", schema)
        ds.select(explode(arrayColumn).alias("v"))
            .select("v.*")
            .as[HttpTransaction]
    }
}
```

Some of the imports will be used later. Do not worry if they show up as being unused in IntelliJ. Let's explain, step by step, what is happening here. I would encourage you to run each step in a Scala console and call `.show()` after each `Dataset` transformation:

```
| def jsonToHttpTransactions(json: String)(implicit spark: SparkSession)
| : Dataset[HttpTransaction] =
```

As specified in the unit test, the signature of our function takes `String` containing a JSON array of transactions and returns `Dataset[HttpTransaction]`. As we need to produce `Dataset`, we also need to pass a `SparkSession` object. It is a good practice to pass it as an implicit parameter, as there is only one instance of this class in a typical application:

```
| import spark.implicits.
| val ds: Dataset[String] = Seq(json).toDS()
```

The first step is to produce `Dataset[String]` from our JSON string. This `Dataset` will have a single row containing the whole JSON array of transactions. For this, we use the `.toDS()` method that was made available when we called `import spark.implicits:`

```
| val txSchema: StructType = Seq.empty[HttpTransaction].toDS().schema
| val schema = ArrayType(txSchema)
| val arrayColumn = from_json($"value".cast(StringType), schema)
```

We have seen in the previous section that it is more efficient to use Spark functions that take `Column` as an argument. In order to parse the JSON, we use the `from_json` function, which is in the `org.apache.spark.sql.functions` package. We use this particular signature: `def from_json(e: Column, schema: StructType): Column`:

- The first parameter is the column we want to parse. We pass the "value" column, which is the default column name for our single-column `Dataset`.
- The second argument is the target schema. `StructType` represents the structure of `Dataset`—the names, types, and order of its columns. The schema we pass to the function must match the names and types of the JSON string. You can create a schema by hand but, to make things easier, we first create `txSchema` using an empty `Dataset[HttpTransaction]`. `txSchema` is the schema for a single transaction, but as our JSON string contains an array of transactions, we must wrap `txSchema` in `ArrayType`:

```
| ds.select(explode(arrayColumn).alias("v"))
|   .select("v.*")
|   .as[HttpTransaction]
```

If we just select `arrayColumn`, we would obtain `Dataset[Seq[HttpTransaction]]`—one row containing a collection. But what we want is `Dataset[HttpTransaction]`—one row per element of the array.

For this purpose, we use the `explode` function, which is similar to `flatten` for vectors. After `explode`, we obtain several rows, but at this stage, each row has a single `StructType` column that contains the desired columns—`date`, `tid`, and `price`. Our transaction data is actually wrapped in an object. In order to unwrap it, we first rename this `StructType` column "v", and then call `select("v.*")`. We obtain `DataFrame` with the `date`, `tid`, and `price` columns, so that we can safely cast to `HttpTransaction`.

You can run the unit test; it should pass now.

Unit testing httpToDomainTransactions

We now have all of the pieces to fetch transactions and put them in `Dataset[HttpTransaction]`. But it would not be wise to store these objects as they are and then run some analytics with them, because of the following:

- The API could change in the future, but we would want to keep the same storage format regardless of these changes
- As we will see in the next chapter, the Bitstamp WebSocket API for receiving live transactions uses a different format
- All of the attributes of `HttpTransaction` are of the `String` type. It would be easier to run analytics if the attributes were properly typed

For these reasons, it would better to have a different class that represents a transaction. Let's create a new class called `coinyser.Transaction`:

```
package coinyser

import java.sql.{Date, Timestamp}
import java.time.ZoneOffset

case class Transaction(timestamp: Timestamp,
                      date: Date,
                      tid: Int,
                      price: Double,
                      sell: Boolean,
                      amount: Double)
```

It has the same attributes as `HttpTransaction`, but with better types. We have to use `java.sql.Timestamp` and `java.sql.Date`, because they are the types exposed externally by Spark for timestamps and dates. We also added a `date` attribute that will contain the date of the transaction. The information is already contained in `timestamp`, but this denormalization will be useful later on when we want to filter transactions for a specific date range.

In order to avoid having to pass the date, we can create a new `apply` method in the companion object:

```

object Transaction {
  def apply(timestamp: Timestamp,
           tid: Int,
           price: Double,
           sell: Boolean,
           amount: Double) =
    new Transaction(
      timestamp = timestamp,
      date = Date.valueOf(
        timestamp.toInstant.atOffset(ZoneOffset.UTC).toLocalDate),
      tid = tid,
      price = price,
      sell = sell,
      amount = amount)
}

```

Now we can write a unit test for a new function, `httpToDomainTransactions`, that you need to create inside the existing `BatchProducerSpec`:

```

"BatchProducer.httpToDomainTransactions" should {
  "transform a Dataset[HttpTransaction] into a Dataset[Transaction]"
  in {
    import testImplicits._
    val source: Dataset[HttpTransaction] = Seq(httpTransaction1,
                                                httpTransaction2).toDS()
    val target: Dataset[Transaction] =
      BatchProducer.httpToDomainTransactions(source)
    val transaction1 = Transaction(timestamp = new
      Timestamp(1532365695000L), tid = 70683282, price = 7740.00,
      sell = false, amount = 0.10041719)
    val transaction2 = Transaction(timestamp = new
      Timestamp(1532365693000L), tid = 70683281, price = 7739.99,
      sell = false, amount = 0.00148564)

    target.collect() should contain theSameElementsAs
      Seq(transaction1, transaction2)
  }
}

```

The test is quite straightforward. We build `Dataset[HttpTransaction]`, call the `httpToDomainTransactions` function, and make sure that the result contains the expected `Transaction` objects.

Implementing httpToDomainTransactions

This implementation uses `select` to avoid an extra serialization/deserialization.
Add the following function in `BatchProducer`:

```
def httpToDomainTransactions(ds: Dataset[HttpTransaction]):  
    Dataset[Transaction] = {  
        import ds.sparkSession.implicits._  
        ds.select(  
            $"date".cast(LongType).cast(TimestampType).as("timestamp"),  
            $"date".cast(LongType).cast(TimestampType).  
            cast(DateType).as("date"),  
            $"tid".cast(IntegerType),  
            $"price".cast(DoubleType),  
            $"type".cast(BooleanType).as("sell"),  
            $"amount".cast(DoubleType))  
            .as[Transaction]  
    }
```

We use `cast` to convert the string columns into the appropriate types. For converting `in` to `TimestampType`, we have to first convert `in` to `LongType`, and for converting `in` to `DateType`, we have to first convert `in` to `TimestampType`. Since all the types match the target `Transaction` object, we can call `.as[Transaction]` at the end to obtain `Dataset[Transaction]`.

You can now run `BatchProducersSpec` and make sure the two tests pass.

Saving transactions

We now have all of the functions required to fetch the last 24 hours of transactions from the Bitstamp API, and produce well-typed transaction objects inside `dataset`. What we need after that is to persist this data on disk. This way, once we have run our program for many days, we will be able to retrieve transactions that happened in the past.

Introducing the Parquet format

Spark supports many different formats for persisting datasets: CSV, Parquet, ORC, JSON, and many others, such as Avro, with the appropriate library.

With a row format, such as CSV, JSON, or Avro, the data is saved row by row. With a columnar format, such as Parquet or ORC, the data in the file is stored by columns.

For instance, we might have the following dataset of transactions:

timestamp	tid	price	sell	amount
2018-08-02 07:22:34	0	7657.58	true	0.1
2018-08-02 07:22:47	1	7663.85	false	0.2
2018-08-02 07:23:09	2	7663.85	false	0.3

If we write using a row format, the file would look like this:

```
| 2018-08-02 07:22:34|0|7657.58|1|0.1;2018-08-02 07:22:47|1|7663.85|0|0.2;2018-08-02 07:23:09|2|7663.85|0|0.3
```

In contrast, if we write using a columnar format, the file would look like this:

```
| 2018-08-02 07:22:34|2018-08-02 07:22:47|2018-08-02 07:23:09;0|1|2;7657.58|7663.85|7663.85
```

Using a columnar data format offers several performance benefits when reading the data, including the following:

- **Projection push-down:** When you need to select a few columns, you do not need to read the whole row. In the preceding example, if I am only interested in the evolution of the price of transactions, I can select only the timestamp and price, and the rest of the data will not be read from the disk.
- **Predicate push-down:** When you want to retrieve only the rows where a column has a specific value, you can quickly find these rows by scanning the column data. In the preceding example, if I want to retrieve the transactions that happened between 07:22:00 and 07:22:30, the columnar storage will allow me to find these rows by only reading the timestamp

column on the disk.

- Better compression: Row formats can be compressed before being stored on the disk, but the columnar format has a better compression ratio. The data is indeed more homogeneous, as consecutive column values differ less than consecutive rows.
- Splittable: When a Spark cluster runs a job that reads or writes to Parquet, the job's tasks are distributed across many executors. Each executor will read/write chunks of rows from/to its own set of files in parallel.

All these benefits make a columnar format particularly well suited for running analytics queries. This is why, for our project, we are going to use Parquet to store transactions. The choice is a bit arbitrary; ORC would work equally well.



*In a typical Spark cluster production setting, you have to store files in a **distributed filesystem**. Every node of the cluster must indeed have access to any chunk of the data. If one of your Spark nodes were to die, you would still want to have access to the data that was saved by it. You would not be able to do so if the files were stored on the local filesystem. Generally, people use the **Hadoop** filesystem or **Amazon S3** to store their parquet files. They both offer a distributed, reliable way of storing files, and they have good parallelism characteristics.*

In a production project, it can be beneficial to benchmark the performance of different formats. Depending on the shape of your data and the type of queries, one format might be better suited than the others.

Writing transactions in Parquet

Add the following `import` and function declaration in `BatchProducer`:

```
| import java.net.URI  
| def unsafeSave(transactions: Dataset[Transaction], path: URI): Unit = ???
```

Let's have a look in more detail:

- Writing to a file is a side effect; this is why we prefixed our function with `unsafe`. As functional programmers, we strive to control side effects, and it is a good practice to name any side effecting function explicitly. We will see in the next section how to use the `IO` `Monad` to push this side effect to the boundaries of our application.
- We use `java.net.URI` to pass the path to the directory where our files will be written. This makes sure that the path we pass to the function is really a path. As usual, we try to avoid using strings for our parameters to make our code more robust.

The corresponding test will actually write to the filesystem; hence, it is rather more an integration test than a unit test. We are therefore going to create a new test with the `IT` suffix for the integration test.

Create a new test called `coinyser.BatchProducerIT` in `src/test/scala`:

```
package coinyser  
  
import java.sql.Timestamp  
  
import cats.effect.{IO, Timer}  
import org.apache.spark.sql.test.SharedSparkSession  
import org.scalatest.{Matchers, WordSpec}  
  
class BatchProducerIT extends WordSpec with Matchers with SharedSparkSession {  
  
    import testImplicits._  
  
    "BatchProducer.unsafeSave" should {  
        "save a Dataset[Transaction] to parquet" in withTempDir { tmpDir =>  
            val transaction1 = Transaction(timestamp = new  
                Timestamp(1532365695000L), tid = 70683282, price = 7740.00,  
                sell = false, amount = 0.10041719)  
            val transaction2 = Transaction(timestamp = new  
                Timestamp(1532365693000L), tid = 70683281, price = 7739.99,
```

```
    sell = false, amount = 0.00148564)
val sourceDS = Seq(transaction1, transaction2).toDS()

    val uri = tmpDir.toURI
    BatchProducer.unsafeSave(sourceDS, uri)
    tmpDir.list() should contain("date=2018-07-23")
    val readDS = spark.read.parquet(uri.toString).as[Transaction]
        sourceDS.collect() should contain theSameElementsAs
        readDS.collect()
    }
}
}
```

We use the handy `withTempDir` function from `sharedSparkSession`. It creates a temporary directory and deletes it after the test is finished. Then, we create a sample `Dataset[Transaction]`, and call the function we want to test.

After having written the dataset, we assert that the target path contains a directory named `date=2018-07-23`. We indeed want to organize our storage with a `date` partition to make it faster to retrieve a specific date range.

Finally, when we read back the file, we should get the same elements as in the original dataset. Run the test and make sure it fails as expected.

Now that we have a failing test, we can implement `BatchProducer.unsafeSave`:

```
def unsafeSave(transactions: Dataset[Transaction], path: URI): Unit =  
  transactions  
    .write  
    .mode(SaveMode.Append)  
    .partitionBy("date")  
    .parquet(path.toString)
```

First, `transactions.write` creates `DataFrameWriter`. This is an interface that lets us configure some options before calling a final action method, such as `parquet(path: String): Unit`.

We configure the `DataFrameWriter` with the following options:

- `mode(SaveMode.Append)`: With this option, if there is already some data saved in the path, the content of `Dataset` will be appended to it. This will be useful when we call `unsafeSave` at regular intervals to get new transactions.
 - `partitionBy("date")`: In the context of storage, a partition is an intermediate directory that will be created under the path. It will have a name, such as `date=2018-08-16`. Partitioning is a good technique for optimizing the storage

layout. This will allow us to speed up all the queries that only need the data for a specific date range.



Do not confuse a storage partition (an intermediate folder in the filesystem) with a Spark partition (a chunk of the data, stored on a node in the cluster).

You can now run the integration test; it should pass.

An interesting property of storage partitioning is that it further reduces file sizes. You might be worried that by storing both the timestamp and date, we would waste some storage space to store the date. It turns out that when the date is a storage partition, it is not stored in the Parquet files at all.

To convince yourself, add the following line in the unit test, just after the call to `unsafeSave`:

```
| spark.read.parquet(uri + "/date=2018-07-23").show()
```

Then run the unit test again. You should see this in the console:

```
+-----+-----+-----+-----+
|      timestamp|      tid|  price| sell|   amount|
+-----+-----+-----+-----+
| 2018-07-23 18:08:15|70683282| 7740.0|false|0.10041719|
| 2018-07-23 18:08:13|70683281|7739.99|false|0.00148564|
+-----+-----+-----+-----+
```

The date column is missing! This means that the `date` column is not stored at all in the Parquet files. In the unit test, when we were reading from the URI, Spark detected that there was a partition, `date=2018-07-23`, under that directory and added a column, `date`, containing the value `2018-07-23` for all values.



If you want to add a new column that has the same value for all rows, the easiest way is to create an intermediate directory, `myColumn=value`.

Using the IO Monad

We mentioned earlier that our function, `unsafeSave`, has a side effect, which is to write to a file. But as functional programmers, we try to only write pure functions that have no side effects. However, at the end of the program, you still want this side effect to happen; otherwise, there would be no point in running it!

A common way of solving this dilemma is to use a parametrized type that encapsulates the side effect to run it asynchronously. A good candidate for that is the `cats.effect.IO` class in the `cats.effect` library (see <https://typelevel.org/cats-effect/datatypes/io.html>).

Here is an example that you can try in a Scala Console:

```
import cats.effect.IO
val io = IO{ println("Side effect!"); 1 }
// io: cats.effect.IO[Int] = ...
io.unsafeRunSync()
// Side effect!
// res1: Int = 1
```

We can observe that nothing happened when we declared the `io` variable. At this point, the block passed to the `IO` constructor is only registered and will be executed later. The actual execution only happens when we call `unsafeRunSync()`. Our `io` variable is a pure, immutable value, and hence preserves referential transparency.

`IO` is `Monad`, and as such we can use `map`, `flatMap` and `for` comprehensions to compose side effects:

```
val program = for {
  a <- io
  b <- io
} yield a+b
// program: cats.effect.IO[Int]
program.unsafeRunSync()
// IO is run!
// IO is run!
// res2: Int = 2
```

We can reuse the `io` variable many times; the side effect that it encapsulates will be run as many times as necessary *at the end of the world* when we call

`unsafeRunSync()`.



If we had used `scala.concurrent.Future` instead of `cats.effect.IO`, the side effect would have been only run once. This is because `Future` memorizes the result. The behavior of `Future` may be desirable in some cases, but in some other cases, you really want your effects to be performed as many times as you define them in your code. The approach of `IO` also avoids shared state and memory leaks.

`IO` values can also be run in parallel. They can effectively replace

`scala.concurrent.Future`:

```
import cats.effect.IO
import cats.implicits._
import scala.concurrent.ExecutionContext.Implicits.global

val io = IO{ Thread.sleep(100); Thread.currentThread().getName }
val program = (io, io, io).parMapN((a, b, c) => s"$a\n$b\n$c")
program.unsafeRunSync()
// res2: String =
// ForkJoinPool-1-worker-5
// ForkJoinPool-1-worker-3
// ForkJoinPool-1-worker-1
```

The `IO` block returns the current thread's name as a string. We create a program of the `IO[String]` type using `parMapN` to indicate that we want to execute the `IO` values in the tuple in parallel. The output of `unsafeRunSync` shows that the program was executed in three different threads.

Going back to our transaction saving, all we have to do to make our `unsafeSave` function safe is to wrap it in `IO`:

```
| def save(transactions: Dataset[Transaction], path: URI): IO[Unit] =
|   IO(unsafeSave(transactions, path))
```

Alternatively, you can inline `unsafeSave` and change the integration test to call `save` instead:

```
|     BatchProducer.save(sourceDS, uri).unsafeRunSync()
```

We can now save transactions while controlling side effects and keeping our functions pure.

Putting it all together

At this point, we can read transactions from the REST API, transform the JSON payload in `dataset[Transaction]`, and save it to parquet. It is time to put all these pieces together.

The Bitstamp API allows us to get the transactions that happened in the last 24 hours, in the last hour, or in the last minute. At the end of the day, we would like to build an application that regularly fetches and saves new transactions for long-term analysis. This application is our *batch* layer, and it is not meant to get real-time transactions. Therefore, it will be enough to get the transactions for the last hour. In the next chapter, we will build a *speed* layer to process the live transactions.

Our `BatchProducer` application will work as follows:

1. On startup, fetch the last 24 hours of transactions. Set `start` = current day at midnight UTC and `end` = last transaction's timestamp.
2. Filter the transactions to only keep those between `start` and `end` and save them to Parquet.
3. Wait 59 minutes.
4. Fetch the last one hour of transactions. We have a one minute overlap to make sure that we do not miss any transaction. Set the `start` = `end` and `end` = last transaction timestamps.
5. Go to step 2.

To implement this algorithm, we are going to write a `processOneBatch` function that encompasses steps 2 to 4, and after that, we will implement step 1 and the infinite loop.

Testing processOneBatch

Our function will need a few configuration parameters and implicit values. To keep our signature tidy, we are going to put them in a class. Create a new class, `coinyser.AppContext`:

```
| class AppContext(val transactionStorePath: URI)
|   (implicit val spark: SparkSession,
|    implicit val timer: Timer[IO])
```

`AppContext` contains the target location for the Parquet files, the `sparkSession` object, and a `Timer[IO]` object that is required by `cats.effect` when we need to call `IO.sleep`.

Then declare the `processOneBatch` function in `BatchProducer`:

```
| def processOneBatch(fetchNextTransactions: IO[Dataset[Transaction]],
|   transactions: Dataset[Transaction],
|   saveStart: Instant,
|   saveEnd: Instant)(implicit appCtx: AppContext)
| : IO[(Dataset[Transaction], Instant, Instant)] = ???
```

The function accepts these parameters:

- `fetchNextTransactions` is an `IO` operation that will return the transactions of the past hour when run. We pass it as a parameter so that we can simulate the call to the Bitstamp API in a unit test.
- `transactions` is `Dataset` containing the last transactions that were read (steps 1 or 4 in our algorithm).
- `saveStart` and `saveEnd` is the time interval used to filter `transactions` before saving them.
- `appCtx` is as described previously.

Our function will have to perform side effects; hence, it returns `IO`. This `IO` will contain a tuple with the following:

- `Dataset[Transaction]` that will be obtained by running `fetchNextTransactions`
- The next `saveStart` and the next `saveEnd`

Now that we have a good declaration of our function, we can write an integration test for it. The test is quite long; hence, we are going to describe it bit

by bit. Create a new integration test in `BatchProducerIT`:

```
"BatchProducer.processOneBatch" should {
    "filter and save a batch of transaction, wait 59 mn, fetch the next
    batch" in withTempDir { tmpDir =>
    implicit object FakeTimer extends Timer[IO] {
        private var clockRealTimeInMillis: Long =
            Instant.parse("2018-08-02T01:00:00Z").toEpochMilli

        def clockRealTime(unit: TimeUnit): IO[Long] =
            IO(unit.convert(clockRealTimeInMillis, TimeUnit.MILLISECONDS))

        def sleep(duration: FiniteDuration): IO[Unit] = IO {
            clockRealTimeInMillis = clockRealTimeInMillis +
                duration.toMillis
        }

        def shift: IO[Unit] = ???
        def clockMonotonic(unit: TimeUnit): IO[Long] = ???
    }
    implicit val appContext: ApplicationContext = new
        ApplicationContext(transactionStorePath = tmpDir.toURI)
```

We first define `FakeTimer` that implements the `Timer[IO]` interface. This timer lets us simulate a clock that starts at `2018-08-02T01:00:00Z`. This way, we will not have to wait 59 minutes to run our test. The implementation uses `var clockRealTimeInMillis` that keeps the current time of our fake clock and updates it when `sleep` is called.

Then, we create `ApplicationContext` using the temporary directory, and the implicits that are in scope: `FakeTimer` and `SparkSession`.

The next portion of the test defines some transactions:

```
implicit def toTimestamp(str: String): Timestamp =
    Timestamp.from(Instant.parse(str))
val tx1 = Transaction("2018-08-01T23:00:00Z", 1, 7657.58, true,
    0.021762)
val tx2 = Transaction("2018-08-02T01:00:00Z", 2, 7663.85, false,
    0.01385517)
val tx3 = Transaction("2018-08-02T01:58:30Z", 3, 7663.85, false,
    0.03782426)
val tx4 = Transaction("2018-08-02T01:58:59Z", 4, 7663.86, false,
    0.15750809)
val tx5 = Transaction("2018-08-02T02:30:00Z", 5, 7661.49, true, 0.1)

val txs0 = Seq(tx1)
val txs1 = Seq(tx2, tx3)
val txs2 = Seq(tx3, tx4, tx5)
val txs3 = Seq.empty[Transaction]
```

The `implicit` conversion `toTimestamp` lets us declare our transaction objects with `String` instead of `Timestamp`. This makes the test easier to read. We use it to declare five `Transaction` objects with timestamps ranging around the initial clock

of `FakeTimer`.

Then, we declare batches of transactions that simulate what would have been read from the Bitstamp API. We cannot indeed call the real Bitstamp API from our integration test; the data would be random and our integration test could fail if the API is not available:

- `txs0` is `Seq[Transaction]`, which simulates an initial batch of transactions that we read at 01:00. If you remember the `BatchProducer` algorithm, this initial batch would contain the last 24 hours of transactions. In our example, this batch only contains `tx1`, even though `tx2`'s timestamp is 01:00. This is because, with the real API, we would not get a transaction that happened exactly at the same time. There is always a bit of lag.
- `txs1` is the batch of transactions that we read 59 minutes after, at 01:59. In this batch, we consider that the API lag makes us miss `tx4`, which happens at 01:58:59.
- `txs2` is the batch that we read 59 minutes after `txs1`, at 02:58.
- `txs3` is the batch that we read 59 minutes after `txs2`, at 03:57.

The following portion actually calls the function under test, `processOneBatch`, three times:

```
val start0 = Instant.parse("2018-08-02T00:00:00Z")
val end0 = Instant.parse("2018-08-02T00:59:55Z")
val threeBatchesIO =
  for {
    tuple1 <- BatchProducer.processOneBatch(IO(txs1.toDS()),
      txs0.toDS(), start0, end0)
    (ds1, start1, end1) = tuple1

    tuple2 <- BatchProducer.processOneBatch(IO(txs2.toDS()), ds1,
      start1, end1)
    (ds2, start2, end2) = tuple2

    _ <- BatchProducer.processOneBatch(IO(txs3.toDS()), ds2, start2,
      end2)
  } yield (ds1, start1, end1, ds2, start2, end2)
val (ds1, start1, end1, ds2, start2, end2) =
  threeBatchesIO.unsafeRunSync()
```

For the first call, we pass the following:

- `txs0.toDS()` represents the initial batch of transactions. This would cover the last 24 hours of transactions.
- `start0 = 00:00`. In our algorithm, we choose to cut the first batch to start at midnight. This way, we won't save partial data for the previous day.

- `end0` = 00:59:55. Our clock starts at 01:00, but the API has always some lag for making a transaction visible. We estimate that lag to not exceed five seconds.
- `IO(txs1.toDS())` represents the next batch of transactions to be fetched. It will be fetched 59 minutes after the initial one.

The subsequent calls pass the results of the previous calls, as well as the `IO` value to fetch the following batch.

We then run the three calls with `unsafeRunSync()`, and obtain the results of the two first calls in `ds1`, `start1`, `end1`, `ds2`, `start2`, and `end2`. This allows us to verify the results with the following assertions:

```
ds1.collect() should contain theSameElementsAs txs1
start1 should ===(end0)
end1 should ===(Instant.parse("2018-08-02T01:58:55Z"))

ds2.collect() should contain theSameElementsAs txs2
start2 should ===(end1)
end2 should ===(Instant.parse("2018-08-02T02:57:55Z"))

val lastClock = Instant.ofEpochMilli(
  FakeTimer.clockRealTime(TimeUnit.MILLISECONDS).unsafeRunSync())
lastClock should ===(Instant.parse("2018-08-02T03:57:00Z"))
```

Lets have a look in detail at the preceding code:

- `ds1` is the batch that was obtained by running the `IO(txs1.toDS())`. It must, therefore, be the same as `txs1`
- `start1` must be equal to `end0`—we need to shift the time period without any gap
- `end1` must be equal to the initial clock (`01:00`) + 59 mn (*wait time*) - 5 seconds (API lag)
- `ds2`, `start2`, and `end2` follow the same logic
- `lastClock` must be equal to the initial clock + 3 * 59 mn

Finally, we can assert that the right transactions were saved to disk:

```
val savedTransactions = spark.read.parquet(tmpDir.toString).as[Transaction].collect()
val expectedTxs = Seq(tx2, tx3, tx4, tx5)
savedTransactions should contain theSameElementsAs expectedTxs
```

The assertion excludes `tx1`, as it happened in the previous day. It also verifies that even though our batches, `txs1` and `txs2`, had some overlap, there is no duplicate transaction in our Parquet file.

You can compile and run the integration test. It should fail with `NotImplementedError` as expected.

Implementing processOneBatch

Here is the implementation of `BatchProducer.processOneBatch`. As is often the case, the implementation is much shorter than the test:

```
val WaitTime: FiniteDuration = 59.minute
val ApiLag: FiniteDuration = 5.seconds

def processOneBatch(fetchNextTransactions: IO[Dataset[Transaction]],
                    transactions: Dataset[Transaction],
                    saveStart: Instant,
                    saveEnd: Instant)(implicit appCtx: ApplicationContext)
: IO[(Dataset[Transaction], Instant, Instant)] = {
  import appCtx._
  val transactionsToSave = filterTxs(transactions, saveStart, saveEnd)
  for {
    _ <- BatchProducer.save(transactionsToSave,
                            appCtx.transactionStorePath)
    _ <- IO.sleep(WaitTime)
    beforeRead <- currentInstant
    end = beforeRead.minusSeconds(ApiLag.toSeconds)
    nextTransactions <- fetchNextTransactions
  } yield (nextTransactions, saveEnd, end)
}
```

We first filter the transactions using a `filterTxs` function that we will define shortly. Then, using a `for` comprehension, we chain several `IO` values:

1. Save the filtered transactions, using the `save` function that we implemented earlier
2. Wait 59 minutes, using the implicit `Timer` that was brought in scope with
`import appCtx._`
3. Get the current time, using a `currentInstant` function that we will define shortly
4. Fetch the next transactions using the first argument

Here is the implementation of the helper function, `filterTxs`:

```
def filterTxs(transactions: Dataset[Transaction],
              fromInstant: Instant, untilInstant: Instant): Dataset[Transaction] = {
  import transactions.sparkSession.implicits._
  transactions.filter(
    $"timestamp" >=
      lit(fromInstant.getEpochSecond).cast(TimestampType)) &&
   ($"timestamp" <
      lit(untilInstant.getEpochSecond).cast(TimestampType)))
}
```

We did not need to pass an implicit `sparkSession`, as it is already available in the transaction `dataset`. We only keep transactions for the interval `(fromInstant, untilInstant)`. The end instant is excluded so that we do not have any overlap when we loop over `processOneBatch`.

Here is the definition of `currentInstant`:

```
| def currentInstant(implicit timer: Timer[IO]): IO[Instant] =  
|   timer.clockRealTime(TimeUnit.SECONDS) map Instant.ofEpochSecond
```

We use the `Timer` class to get the current time. As we saw while writing the integration test, this allowed us to use a fake timer to simulate a clock.

Implementing processRepeatedly

We are now ready to implement the algorithm of our `BatchProducer` application, which will loop repeatedly over `processOneBatch`. We are not going to write an integration test for it, as it merely assembles other parts that have been tested. Ideally, in a production system, you should write an end-to-end test that would start the application and connect to a fake REST server.

Here is the implementation of `processRepeatedly`:

```
def processRepeatedly(initialJsonTxs: IO[Dataset[Transaction]],
                      jsonTxs: IO[Dataset[Transaction]])
                      (implicit appContext: ApplicationContext): IO[Unit] = {
    import appContext._

    for {
        beforeRead <- currentInstant
        firstEnd = beforeRead.minusSeconds(ApiLag.toSeconds)
        firstTxs <- initialJsonTxs
        firstStart = truncateInstant(firstEnd, 1.day)
        _ <- Monad[IO].tailRecM((firstTxs, firstStart, firstEnd)) {
            case (txs, start, instant) =>
                processOneBatch(jsonTxs, txs, start, instant).map(_.asLeft)
        }
    } yield ()
}
```

In the function's signature, we have the following:

- A parameter `initialJsonTxs`, which is `IO` that will fetch the last 24 hours of transactions in `Dataset`
- A second parameter, `jsonTxs`, which fetches the last hour of transactions
- A return type, `IO[Unit]`, which will run infinitely when we call `unsafeRunSync` in the main application

The functions' body is a `for` comprehension that chains the `IO` values as follows:

1. We first calculate `firstEnd = current time - 5 seconds`. By using an `ApiLag` of 5 seconds, when we then fetch transactions using `initialJsonTxs`, we are certain that we will get all the transactions until `firstEnd`.
2. `firstStart` is set to midnight on the current day. For the initial batch, we want to filter out transactions from the previous day.

3. We fetch the last 24 hours of transactions in `firstTxs`, of the `Dataset[Transaction]` type.
4. We call `tailRecM` from `Monad`. It calls the anonymous function in the block until it returns `Monad[Right[Unit]]`. But since our function always returns `Left`, it will loop infinitely.

Implementing BatchProducerApp

Finally, all we need to do is create an application that will call `processRepeatedly` with the right parameters.

Create a new class, `coinyser.BatchProducerApp`, in `src/main/scala` and type the following:

```
package coinyser

import java.io.{BufferedReader, InputStreamReader}
import java.net.{URI, URL}

import cats.effect.{ExitCode, IO, IOApp}
import coinyser.BatchProducer.{httpToDomainTransactions, jsonToHttpTransactions}
import com.typesafe.scalalogging.StrictLogging
import org.apache.spark.sql.{Dataset, SparkSession}

import scala.io.Source

class BatchProducerApp extends IOApp with StrictLogging {

    implicit val spark: SparkSession =
        SparkSession.builder.master("local[*]").getOrCreate()
    implicit val appContext: AppContext = new AppContext(new
        URI("./data/transactions"))

    def bitstampUrl(timeParam: String): URL =
        new URL("https://www.bitstamp.net/api/v2/transactions/btcusd?time="
            + timeParam)

    def transactionsIO(timeParam: String): IO[Dataset[Transaction]] = {
        val url = bitstampUrl(timeParam)
        val jsonIO = IO {
            logger.info(s"calling $url")
            Source.fromURL(url).mkString
        }
        jsonIO.map(json =>
            httpToDomainTransactions(jsonToHttpTransactions(json)))
    }

    val initialJsonTxs: IO[Dataset[Transaction]] = transactionsIO("day")
    val nextJsonTxs: IO[Dataset[Transaction]] = transactionsIO("hour")

    def run(args: List[String]): IO[ExitCode] =
        BatchProducer.processRepeatedly(initialJsonTxs, nextJsonTxs).map(_
            => ExitCode.Success)
    }

    object BatchProducerAppSpark extends BatchProducerApp
```

The class extends `cats.effect.IOApp`. It is a helper trait that will call `unsafeRunSync` on `IO` returned by the `run` method. It also extends `strictLogging`. This trait brings an

attribute `logger` in scope that we will use to log messages. The body of our object defines the following members:

- `spark` is the `sparkSession`, required for manipulating datasets. The master is set to `local[*]`, which means that Spark will use all of the cores available on the localhost to execute our jobs. But, as we will see in the next section, this can be overridden when using a Spark cluster.
- `appContext` requires the path for saving our transaction. Here, we use a relative directory on the local filesystem. In a production environment, you would typically use an S3 or HDFS location. `AppContext` also requires two implicits: `SparkSession` and `Timer[IO]`. We already defined the former, and the latter is provided by `IOApp`.
- `bitstampurl` is a function that returns the URL that is used to retrieve the transactions that happened in the last day or in the last hour
- `transactionsIO` fetches the transactions by calling the Bitstamp URL. As seen at the beginning of this chapter, we use `scala.io.Source` to create a string from the HTTP response. We then transform it into `Dataset[Transaction]` using the two functions, `jsonToHttpTransactions` and `httpToDomainTransactions`, that we implemented earlier.
- `initialJsonTxs` and `nextJsonTxs` are the IO values that, respectively, retrieve the last 24 hours of transactions and the last hour of transactions.
- `run` implements the only abstract method of `IOApp`. It produces `IO[ExitCode]` to be run as an application. Here we just call `processRepeatedly` with `vals` that were defined previously. Then, we have to map to change the unit result to `ExitCode.Success` in order to type check. Actually, this exit code will never be returned, because `processRepeatedly` loops infinitely.

If you now try to run `BatchProducerAppSpark`, you will get `ClassNotFoundException` about a Spark class. This is because in `build.sbt`, we declared some libraries as `% Provided`. As we shall see, this configuration is useful for packaging the application, but right now it prevents us from testing our program easily.

The trick to avoid that is to create another object, `coinyser.BatchProducerAppIntelliJ`, in the `src/test/scala` directory, that also extends the class `BatchProducerApp`. IntelliJ indeed brings all of the provided dependencies to the test runtime classpath:

```
| package coinyser
| object BatchProducerAppIntelliJ extends BatchProducerApp
```

This is why we defined a class and an object in `BatchProducerApp.scala`. We can have one implementation that will be used with `spark-submit`, and one that we can run from IntelliJ.

Now run the `BatchProducerAppIntelliJ` application. After a couple of seconds, you should see something similar to this in the console:

```
(...)
18/09/02 22:29:08 INFO BatchProducerAppIntelliJ$: calling https://www.bitstamp.net/api/v1
18/09/02 22:29:15 WARN TaskSetManager: Stage 0 contains a task of very large size (1225)
18/09/02 22:29:15 INFO CodecPool: Got brand-new compressor [.snappy]
18/09/02 22:29:16 INFO FileOutputCommitter: Saved output of task 'attempt_20180902222915'
```

After this point, you should have a Parquet file in the `data/transactions/date=<current date>` directory, containing all of the transactions that happened from midnight on the current day. If you wait one more hour, you will get another Parquet file containing the transactions of the last hour.

If you do not want to wait one hour to see it happening, you can fetch transactions every minute instead:

- Change `BatchProducerApp.nextJsonTxs` to `jsonIO("?time=minute")`
- Change `BatchProducer.WaitTime` to `45.seconds` to have a 15 seconds overlap



The `WARN` message in the console tells us that "Stage 0 contains a task of very large size". This is because `string` that contains the HTTP response is sent as a whole to one Spark task. If we had a much larger payload (several hundreds of MB), it would be less memory intensive to split it and write it to a file.

In the next chapter, we will see how to use Zeppelin to query these Parquet files and plot some charts. But we can already check them using the Scala Console. Start a new Scala Console and type the following:

```
import org.apache.spark.sql.SparkSession
implicit val spark = SparkSession.builder.master("local[*]").getOrCreate()
val ds = spark.read.parquet("./data/transactions")
ds.show()
```

Feel free to play around with the Dataset API. You can try to count the transactions, filter them for a specific period, and find the maximum price or quantity.

Running the application with spark-submit

We have run our application in a standalone way, but when you want to process large datasets you would need to use a Spark cluster. It is out of the scope of this book to explain how to set up a Spark cluster. If you want to set up one, you can refer to the Spark documentation or use an off-the-shelf cluster from a cloud computing vendor.

Nonetheless, the submission process is the same whether we run Spark in local mode or in cluster mode.

Installing Apache Spark

We are going to install Spark to run it in local mode. This mode only uses the CPU cores of the localhost to run jobs. For this, download Spark 2.3.1 from this page: <https://spark.apache.org/downloads.html>.

Then extract it to some folder for instance, `~/` for your `home` folder on Linux or macOS:

```
| tar xfvz spark-2.3.1-bin-hadoop2.7.tgz ~/
```

You can try running `spark shell` to verify that the installation is correct:

```
| cd ~/spark-2.3.1-bin-hadoop2.7/bin  
| ./spark-shell
```

After a couple of seconds, you should see a welcome message followed by the same `scala>` prompt that we had in the Scala console:

```
(...)  
Spark context Web UI available at http://192.168.0.11:4040  
Spark context available as 'sc' (master = local[*], app id = local-1536218093431).  
Spark session available as 'spark'.  
Welcome to  
  
version 2.3.1  
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_112)  
Type in expressions to have them evaluated.  
Type :help for more information.  
scala>
```

A Spark shell is actually a Scala Console connected to a Spark cluster. In our case, it is a Spark local cluster, as you can see with `master = local[*]`. Spark-shell provides a variable, `spark: SparkSession`, that you can use to manipulate datasets. It can be a handy tool, but I generally prefer using IntelliJ's console and create `SparkSession` by hand. IntelliJ's console has the benefit of having syntax highlighting and better code completion.

Packaging the assembly JAR

In order to run our application with a Spark distribution, we need to package our compiled classes and their dependencies in a JAR file. This is what we call an assembly JAR or fat JAR: its size can be quite large if you have many dependencies. To do that, we have to modify our SBT build files.

First, we need to enable the assembly plugin. Add a new file, `assembly.sbt`, in the `bitcoin-analyser/project` folder and type the following:

```
| addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.7"
```

We also need to exclude all the Spark dependencies from our assembly JAR. There is no point in having them as they are already present in the Spark distribution. Excluding them will save space and build time. For this, we had already scoped these dependencies to `% Provided in build.sbt`. This will make sure that the dependencies are present for compiling the project and running the tests, but are excluded when building the assembly JAR.

Then we have to add a few configuration options at the end of `build.sbt`:

```
| assemblyOption in assembly := (assemblyOption in
|   assembly).value.copy(includeScala = false)
| test in assembly := {}
| mainClass in assembly := Some("coinyser.BatchProducerAppSpark")
```

Here is a short explanation talking about what line performs what action:

- The first line excludes all Scala runtime JARs.
- The second line tells SBT to skip the tests when running the assembly task
- The last line declares what is our main class. This declaration will end up in the `MANIFEST.MF` file, and will be used by Spark to bootstrap our program.

Our build files are ready; we can run the assembly task. Open the SBT shell in IntelliJ (*Ctrl + Shift + S*), and type `assembly` after the `sbt>` prompt. This should compile the project and package the assembly JAR. The output of your SBT shell should look like this:

```
| [IJ]sbt:bitcoin-analyser> assembly
```

```
| [info] Strategy 'discard' was applied to 3 files (Run the task at debug level to see det  
| ...  
| [info] Done packaging.
```

The assembly JAR is ready; we can submit it to Spark.

Running spark-submit

Using a console, go to the Spark distribution's `bin` folder, and `run spark-submit` with the path of the assembly JAR:

```
| cd ~/spark-2.3.1-bin-hadoop2.7/bin  
| ./spark-submit /tmp/sbt/bitcoin-analyser/scal.../assembly-0.1.jar
```

`spark-submit` has lots of options that let you change the Spark master, the number of executors, their memory requirements, and so on. You can find out more by running `spark-submit -h`. After having submitted our JAR, you should see something like this in your console (we only show the most important parts):

```
(...)  
2018-09-07 07:55:27 INFO  SparkContext:54 - Running Spark version 2.3.1  
2018-09-07 07:55:27 INFO  SparkContext:54 - Submitted application: coinyser.BatchProducer  
(...)  
2018-09-07 07:55:28 INFO  Utils:54 - Successfully started service 'SparkUI' on port 4040  
(...)  
2018-09-07 07:55:28 INFO  SparkContext:54 - Added JAR file:/tmp/sbt/bitcoin-analyser/scal.../assembly-0.1.jar  
2018-09-07 07:55:28 INFO  Executor:54 - Starting executor ID driver on host localhost  
(...)  
2018-09-07 07:55:28 INFO  NettyBlockTransferService:54 - Server created on 192.168.0.11:  
(...)  
2018-09-07 07:55:29 INFO  BatchProducerApp$:23 - calling https://www.bitstamp.net/api/v2  
(...)  
2018-09-07 07:55:37 INFO  SparkContext:54 - Starting job: parquet at BatchProducer.scala  
(...)  
2018-09-07 07:55:39 INFO  DAGScheduler:54 - Job 0 finished: parquet at BatchProducer.scala
```

You would see a similar output if you were using a remote cluster:

- The line `Submitted application:` Tells us what `main` class we submitted. This corresponds to the `mainClass` setting that we put in our SBT file. This can be overridden with the `--class` option in `spark-submit`.
- A few lines after, we can see that Spark started a **SparkUI** web server on port `4040`. With your web browser, go to the URL `http://localhost:4040` to explore this UI. It allows you to see the progress of running jobs, their execution plan, how many executors they use, the logs of the executors, and so on. SparkUI is a precious tool when you need to optimize your jobs.
- `Added JAR file:` Before Spark can run our application, it must distribute the assembly JAR to all the cluster nodes. For doing this, we can see that it starts a server on port `37370`. The executors would then connect to that server

to download the JAR file.

- Starting Executor ID driver: The driver process coordinates the execution of jobs with the executors. The following line shows that it listens on port 37370 to receive updates from the executors.
- Calling `https://`: This corresponds to what we logged in our code, `logger.info(s"calling $url")`.
- Starting job: Our application started a Spark job. Line 115 in `BatchProducer` corresponds to the `.parquet(path.toString)` instruction in `BatchProducer.save`. This `parquet` method is indeed an action and as such triggers the evaluation of `dataset`.
- Job 0 finished: The job finishes after a few seconds.

After this point, you should have a `parquet` file saved with the last transactions. If you let the application continue for 1 hour, you will see that it starts another job to get the last hour of transactions.

Summary

By now, you should be more comfortable using Spark's `Dataset` API. Our little program is focused on fetching BTC/USD transactions, but it could be interesting to enhance it. For instance, you could fetch and save other currency pairs, such as ETH/EUR or XRP/USD. Use a different cryptocurrency exchange. This would allow you to compare prices in different exchanges, and possibly work out an arbitrage strategy. Arbitrage is a simultaneous purchase and sale of an asset in different marketplaces to profit from an imbalance in the price. You could get data for traditional currency pairs, such as EUR/USD, or use Frameless to refactor the `Dataset` manipulations and make them more type-safe. See the website for further clarification <https://github.com/typelevel/frameless>.

In the next chapter, we are going to exploit saved transaction data to perform some analytics queries.

Batch and Streaming Analytics

In the previous chapter, we introduced Spark and obtained BTC/USD transaction data from www.bitstamp.net. Using that data, we can now perform some analysis on it.

First, we are going to query this data using a notebook tool named Apache Zeppelin. After that, we will write a program that receives the live transactions from <https://www.bitstamp.net/> and sends them to a Kafka topic as they arrive.

Finally, we will use Zeppelin again to run some streaming analytics queries on the data coming to the Kafka topic.

In this chapter, we will cover the following topics:

- Introduction to Zeppelin
- Analyzing transactions with Zeppelin
- Introducing Apache Kafka
- Streaming transactions to Kafka
- Introducing Spark Streaming
- Analyzing Streaming transactions with Zeppelin

Introduction to Zeppelin

Apache Zeppelin is an open source software offering a web interface to create notebooks.

In a notebook, you can inject some data, execute snippets of code to perform analysis on the data, and then visualize it.

Zeppelin is a collaborative tool; several users can use it simultaneously. You can share notebooks and define the role of each user. You would typically define two different roles:

- The writer, usually a developer, can edit all the paragraphs and create forms.
- The end user does not know much about the technical implementation details. He will just want to change some values in a form and then look at the effect on the results. The results can be a table or a graph, and can be exported to CSV.

Installing Zeppelin

Before installing Zeppelin, you need to have Java and Spark installed on your machine.

Follow these steps to install Zeppelin:

1. Download the binary from: <http://zeppelin.apache.org/download.html>.
2. Explode the .tgz file using your favorite program

That's it. Zeppelin is installed and configured with default settings. The next step is to start it.

Starting Zeppelin

To start the Zeppelin daemon process, run the following command in the Terminal:

- Linux and macOS:

```
| <downloadPath>/zeppelin-0.8.0-bin-all/bin/zeppelin-daemon.sh start
```

- Windows:

```
| <downloadPath>/zeppelin-0.8.0-bin-all/bin/zeppelin.cmd start
```

Zeppelin is now running and ready to accept requests on <http://localhost:8080>.

Open the URL in your favorite browser. You should see this page:

localhost:8080/#/

Zeppelin Notebook Job Search anonymous

Welcome to Zeppelin!

Zeppelin is web-based notebook that enables interactive data analytics.
You can make beautiful data-driven, interactive, collaborative document with SQL, code and even more!

Notebook

- [Import note](#)
- [Create new note](#)

Filter

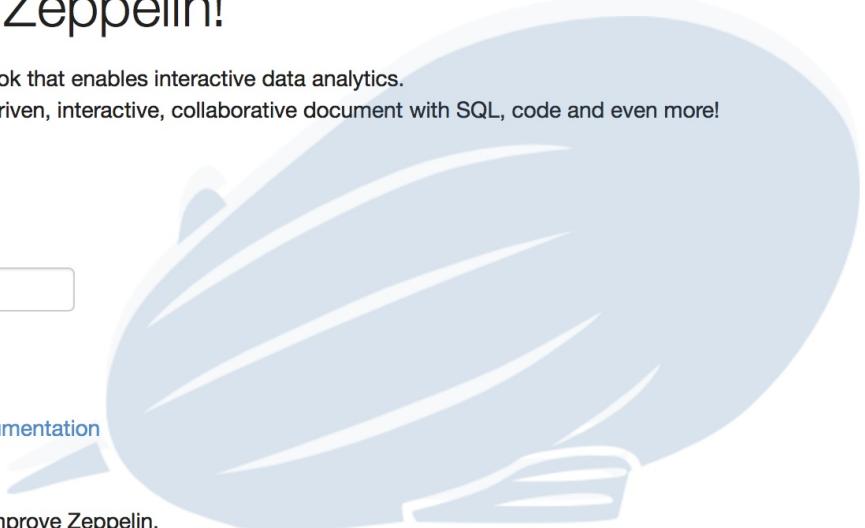
[Zeppelin Tutorial](#)

Help

Get started with [Zeppelin documentation](#)

Community

Please feel free to help us to improve Zeppelin,
Any contribution are welcome!



- [Mailing list](#)
- [Issues tracking](#)
- [Github](#)

Testing Zeppelin

Let's create a new notebook to test if our installation is working correctly.

From the <http://localhost:8080> home page, click on Create New Note and set `Demo` as a name in the popup:

The screenshot shows a modal dialog titled "Create New Note". It has a blue header bar with the title and a close button (an "X"). The main body of the dialog is white. It contains a "Note Name" field with the value "Demo" entered. Below it is a "Default Interpreter" dropdown menu set to "spark". A note below the dropdown says "Use '/' to create folders. Example: /NoteDirA/Note1". At the bottom right is a blue "Create" button.

Create New Note

Note Name

Demo

Default Interpreter

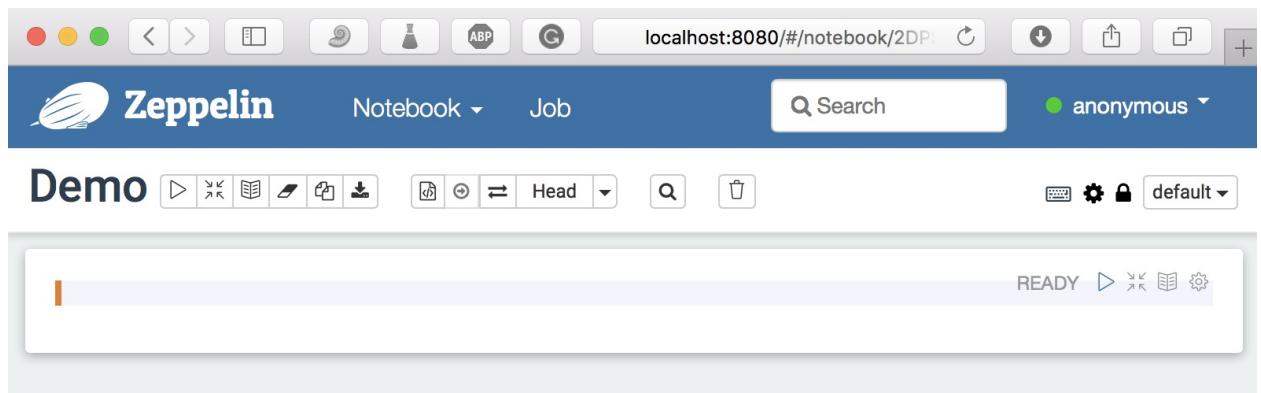
spark

Use '/' to create folders. Example: /NoteDirA/Note1

Create

As mentioned in the window, the default interpreter will be Spark, and this is what we want. Click on Create now.

You just created your first notebook. You should see the following in your browser:



Structure of a notebook

A notebook is a document in which you can add **paragraphs**. Each paragraph can use a different **interpreter**, which interacts with a specific framework or language.

In each paragraph, there are two sections:

- The top one is an editor, into which you can type some source code and run it
- The bottom one displays the results

If, for example, you choose to use a Spark interpreter, all the lines of code you write in the paragraph will be interpreted (like the REPL seen in [Chapter 1, Writing Your First Program](#)). All variables defined will be kept in memory and shared with all the other paragraphs of the notebook. Similarly to the Scala REPL, when you execute it, the output of the execution along with the types of the variables defined will be printed in the result section.

Zeppelin is installed by default with interpreters like Spark, Python, Cassandra, Angular, HDFS, Groovy, and JDBC.

Writing a paragraph

Well, our first notebook is completely empty! We can reuse the example used in [Chapter 10](#), *Fetching and Persisting Bitcoin Market Data*, in the section *Exploring Spark's API with the Scala console*. If you remember, we created `dataset` from a sequence of strings. Enter the following in the notebook:

```
| val dsString = Seq("1", "2", "3").toDS()  
| dsString.show()
```

Then hit *Shift + Enter* (or click on the play triangle of the UI).

The interpreter runs and you should see the following:

The screenshot shows the Zeppelin web interface. At the top, there's a toolbar with standard OS-style buttons (red, yellow, green) and a URL bar showing "localhost:8080/#/notebook/2DP". Below the toolbar is a header bar with the Zeppelin logo, "Notebook" dropdown, "Job" dropdown, a search bar, and a user status "anonymous". The main area has a title "Demo" and a toolbar with various icons. The first cell is titled "FINISHED" and contains the following Scala code:

```
val dsString = Seq("1", "2", "3").toDS()
dsString.show()
```

The output of this code is:

```
dsString: org.apache.spark.sql.Dataset[String] = [value: string]
+----+
|value|
+----+
|  1|
|  2|
|  3|
+----+
```

The second cell is titled "READY" and contains a single orange vertical bar.

The notebook created the dataset from the sequence and printed in the result section of the paragraph.

Notice that in the previous chapter, when we executed this code from the Scala console, we had to create `sparkSession` and had to add some imports. When we use the Spark interpreter in Zeppelin, all the implicits and imports are done automatically and `sparkSession` is created for us.

`SparkSession` is exposed with the variable name `spark`. You can, for instance, get the Spark version with the following code in a paragraph:

```
| spark.version
```

After having run the paragraph, you should see the version printed in the result section:

The screenshot shows the Zeppelin web interface. At the top, there's a toolbar with various icons for file operations like back, forward, and save. Below the toolbar, the header includes the Zeppelin logo, the word "Zeppelin", a "Notebook" dropdown, a "Job" button, a search bar, and a user status indicating "anonymous". The main area is titled "Demo". It contains two paragraphs of Scala code. The first paragraph has been executed and shows the output "2.2.0". The second paragraph is currently being typed, indicated by a cursor at the beginning of the line.

```
val dsString = Seq("1", "2", "3").toDS()
dsString.show()

spark.version

dsString: org.apache.spark.sql.Dataset[String] = [value: string]
+---+
|value|
+---+
|  1|
|  2|
|  3|
+---+

res14: String = 2.2.0
```

READY

At that point, we tested the installation and it is working properly.

Drawing charts

In this section, we are going to create a basic `dataset` and draw a chart of its data.

In a new paragraph, add the following:

```
| case class Demo(id: String, data: Int)
| val data = List(
|   Demo("a",1),
|   Demo("a",2),
|   Demo("b",8),
|   Demo("c",4))
| val dataDS = data.toDS()
| dataDS.createOrReplaceTempView("demoView")
```

We define a `Demo` class with `id` and a `data` property, then we create a list of different `Demo` objects and convert it into `dataset`.

From the `dataDS` dataset ,we call the `.createOrReplaceTempView("demoView")` method. This function is going to register the dataset as a temporary view. With this view defined, we can use SQL to query this dataset. We can try it by adding in a new paragraph, as follows:

```
| %sql
| select * from demoView
```

The newly created paragraph starts with `%sql`. This defines the interpreter that we use. In our case, this is the Spark SQL interpreter.

The query selects all the columns from `demoView`. After you hit *Shift + Enter*, the following table will be shown:

The screenshot shows the Zeppelin web interface running on localhost:8080. At the top, there's a toolbar with various icons. Below it is a header bar with the Zeppelin logo, a search bar, and a user status indicating 'anonymous'. The main area is divided into two sections. The top section, titled 'Demo', contains Scala code:case class Demo(id:String, data: Int)
val data = List(Demo("a",1),Demo("a",2),Demo("b",8),Demo("c",4))
val dataDS = data.toDS()
dataDS.createOrReplaceTempView("demoView")

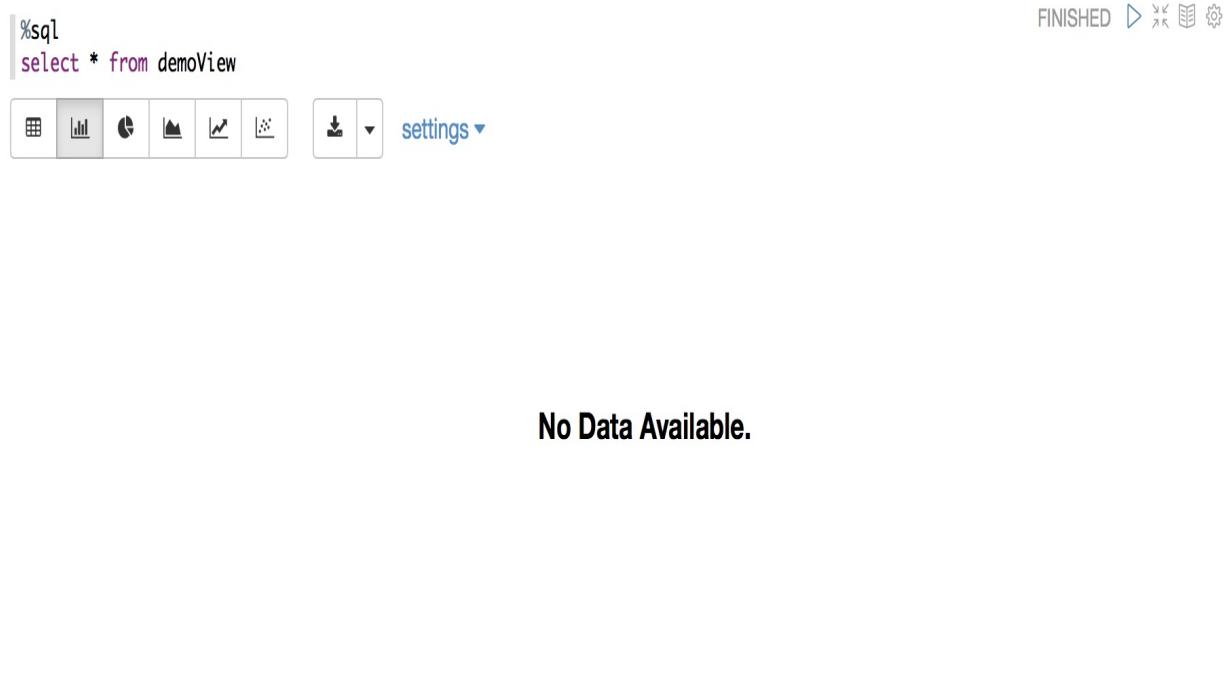
defined class Demo
data: List[Demo] = List(Demo(a,1), Demo(a,2), Demo(b,8), Demo(c,4))
dataDS: org.apache.spark.sql.Dataset[Demo] = [id: string, data: int]The status of this cell is 'FINISHED' with a gear icon. The bottom section contains an SQL query:%sql
select * from demoViewThe status of this cell is also 'FINISHED' with a gear icon. Below the query is a table visualization showing the data from the temporary view:

id	data
a	1
a	2
b	8
c	4

As you can see, the SQL interpreter shows the result of the query by displaying a table in the result section of the paragraph.

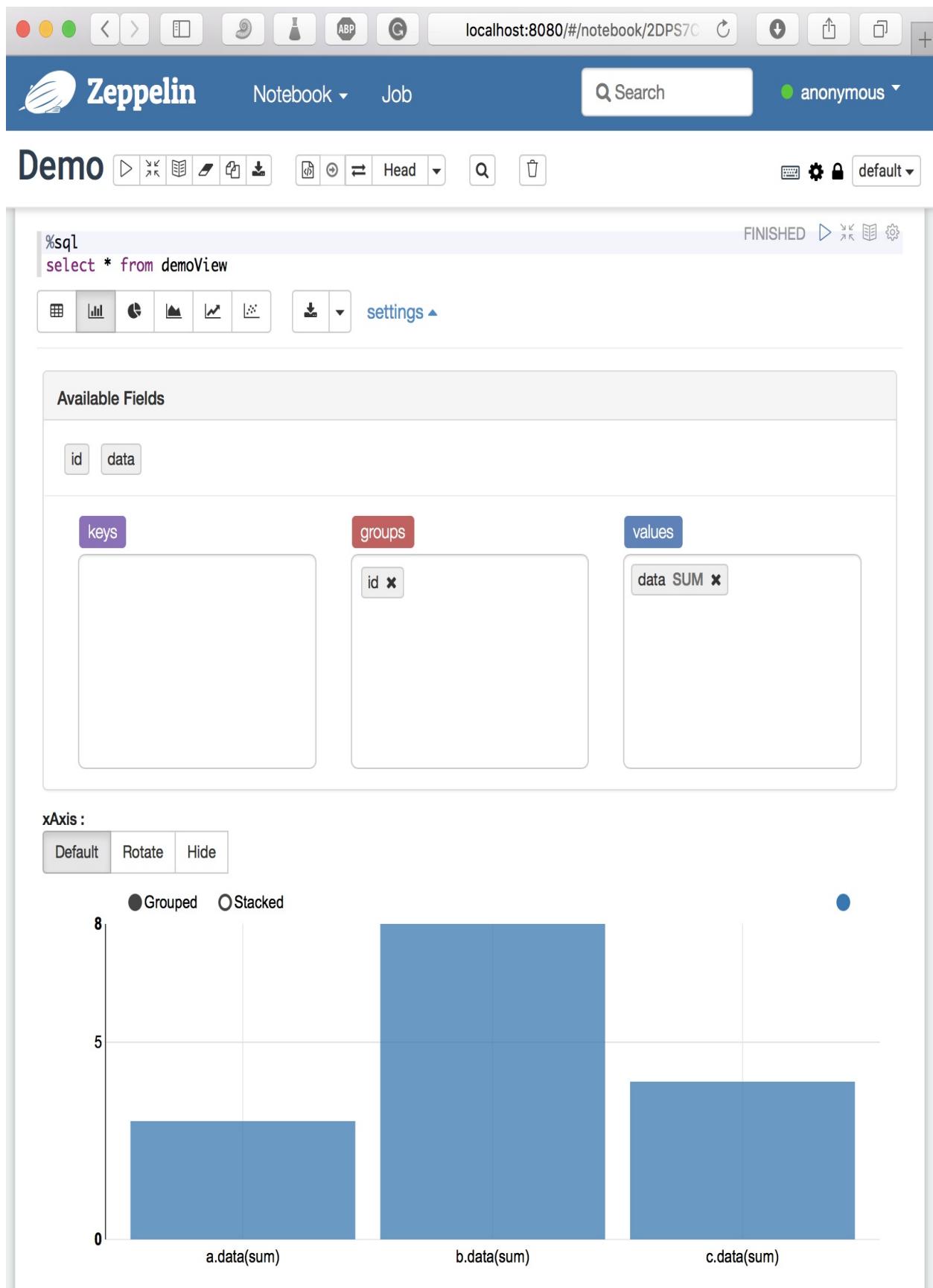
Notice the menu at the top of the table. There are multiple ways to represent the data—bar chart, pie chart, area chart, line chart, and scatter chart.

Click on the bar chart. The notebook now looks like this:



It might seem odd that we don't have data available. Actually, we need to configure the chart to get it working. Click on settings, then define which column is the value and on which one you want to aggregate the data.

Drag the `id` tag to the groups box and `data` to the values box. As we choose to group the IDs, `SUM` of `data` is performed. The configuration and the updated chart should look like this:



The chart is now correct. The sum of all the `id`'s is 3, 8 for the `b` and 4 for the `c`.

We are now familiar enough with Zeppelin to perform analytics on the Bitcoin transactions data that we produced in the previous chapter.

Analyzing transactions with Zeppelin

In the previous chapter, we wrote a program that saves BTC/USD transactions to Parquet files. In this section, we are going to use Zeppelin and Spark to read those files and draw some charts.

If you came directly to this chapter, you first need to set up the `bitcoin-analyser` project, as explained in [Chapter 10, Fetching and Persisting Bitcoin Market Data](#).

Then you can either:

- Run `BatchProducerApp` in `IntelliJ`. This will save the last 24 hours of transactions in the `data` folder of the project directory, then save new transactions every hour.
- Use the sample transaction data that is committed in GitHub. You will have to check out this project: <https://github.com/PacktPublishing/Scala-Programming-Projects>.

Drawing our first chart

With these Parquet files ready, create a new notebook in Zeppelin and name it `Batch analytics`. Then, in the first cell, type the following:

```
| val transactions = spark.read.parquet("<rootProjectPath>/Scala-Programming-Projects/bitc  
z.show(transactions.sort($"timestamp"))
```

The first line creates `DataFrame` from the transactions files. You need to replace the absolute path in the `parquet` function with the path to your Parquet files. The second line uses the special `z` variable to show the content of `DataFrame` in a table. This `z` variable is automatically provided in all notebooks. Its type is `ZeppelinContext`, and it allows you to interact with the Zeppelin renderer and interpreter.

Execute the cell with *Shift + Enter*. You should see the following:

Batch analytics ▶ X 📄 🖌️ 🗑️ Head 🔍 🗑️

```
val transactions = spark.read.parquet("/home/mikael/projects/Scala-Programming-Projects/bitcoin-analyser/data/transactions")
z.show(transactions.sort($"timestamp"))

transactions: org.apache.spark.sql.DataFrame = [timestamp: timestamp, tid: int ... 4 more fields]
```

timestamp	tid	price	sell	amount	date
2018-09-09 01:00:06.0	73661754	6178.31	false	0.005117	2018-09-09
2018-09-09 01:03:21.0	73661805	6178.3	true	0.04047539	2018-09-09
2018-09-09 01:03:21.0	73661806	6178.29	true	0.5	2018-09-09
2018-09-09 01:03:22.0	73661807	6178.29	true	0.5	2018-09-09
2018-09-09 01:03:22.0	73661808	6178	true	0.45952461	2018-09-09
2018-09-09 01:03:31.0	73661814	6178	true	0.00798	2018-09-09
2018-09-09 01:03:39.0	73661816	6178	true	0.59464368	2018-09-09
2018-09-09 01:03:41.0	73661817	6178	true	0.39853434	2018-09-09

Output is truncated to 102400 bytes. Learn more about ZEPPELIN_INTERPRETER_OUTPUT_LIMIT

Took 26 sec. Last updated by anonymous at September 17 2018, 9:02:14 PM. (outdated)

There is a warning message saying that the output is truncated. This is because we retrieved too much data. If we then attempt to draw a chart, some data will be missing.

The solution to this issue could be to change Zeppelin's settings and increase the limit. But if we were to do that, the browser would have to keep a lot of data in memory, and the notebook's file when saved to disk would be large as well.

A better solution is to aggregate the data. We cannot show all the transactions that happened in a day on a chart, but if we can aggregate them with a window of 20 minutes, that will reduce the number of data points to display. Create a new cell with the following code:

```
val group = transactions.groupBy(window($"timestamp", "20 minutes"))

val tmpAgg = group.agg(
    count("tid").as("count"),
    avg("price").as("avgPrice"),
    stddev("price").as("stddevPrice"),
    last("price").as("lastPrice"),
    sum("amount").as("sumAmount"))

val aggregate = tmpAgg.select("window.start", "count", "avgPrice", "lastPrice", "stddevP
z.show(aggregate)
```

Run this new cell. You should see the following output:

```
group: org.apache.spark.sql.RelationalGroupedDataset = RelationalGroupedDataset: [groupi
tmpAgg: org.apache.spark.sql.DataFrame = [window: struct<start: timestamp, end: timestamp
aggregate: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [start: timestamp, c
```

Here is a further description for the preceding code:

- First, we group our transactions with a window of 20 minutes. In the output, you can see that the type of `group` is `RelationalGroupedDataset`. This is an intermediate type on which we must call an aggregation method to produce another `DataFrame`.
- Then we call the `agg` method on `group` to compute several aggregations in one go. The `agg` method takes several `column` arguments as `vararg`. The `column` objects we pass are obtained by calling various aggregation functions from the `org.apache.spark.sql.functions` object. Each `column` is renamed so that we can refer to it easily later on.
- The resulting variable `tmpAgg` is `DataFrame` which has a column `window` of type `struct`. `struct` nests several columns. In our case, it has a

column `start` and a column `end` of type `timestamp`. `tmpAgg` also has all the columns containing the aggregations—`count`, `avgPrice`, and `sumAmount`.

- After that, we select only the columns that we are interested in and then assign the resulting `DataFrame` in a variable `aggregate`. Notice that we can refer to the nested columns of the window column with the “`.`” notation. Here we select all rows apart from `window.end`. We then `sort` the `DataFrame` by ascending `start` time so that we can use `start` as the `x` axis of our future chart. Finally, we `cache` the `DataFrame` so that Spark will not have to reprocess it when we create other cells with different charts.

Underneath the output, Zeppelin displays a table, but this time it does not give us any warning. It is able to load all the data without truncation. We can, therefore, draw a chart:

- Click on the line chart button
- Drag and drop the `start` column in the section
- Drag and drop `avgPrice` and `lastPrice` in the values section

You should see something like this:

Average price & Last price

SPARK JOBS FINISHED ▶ X E ⌂

```
val group = transactions.groupBy(window($"timestamp", "20 minutes"))
val tmpAgg = group.agg(
    count("tid").as("count"),
    avg("price").as("avgPrice"),
    stddev("price").as("stddevPrice"),
    last("price").as("lastPrice"),
    sum("amount").as("sumAmount"))
val aggregate = tmpAgg.select("window.start", "count", "avgPrice", "lastPrice", "stddevPrice", "sumAmount").sort("start").cache()
z.show(aggregate)

group: org.apache.spark.sql.RelationalGroupedDataset = RelationalGroupedDataset: [grouping expressions: [window: struct<start: timestamp, end: timestamp>], value: [time stamp: timestamp, tid: int ... 4 more fields], type: GroupBy]
tmpAgg: org.apache.spark.sql.DataFrame = [window: struct<start: timestamp, end: timestamp>, count: bigint ... 4 more fields]
aggregate: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [start: timestamp, count: bigint ... 4 more fields]
```



Available Fields

`start` `count` `avgPrice` `lastPrice` `stddevPrice` `sumAmount`

keys

`start` X

groups

values

`avgPrice` `SUM` X

`lastPrice` `SUM` X

force Y to 0

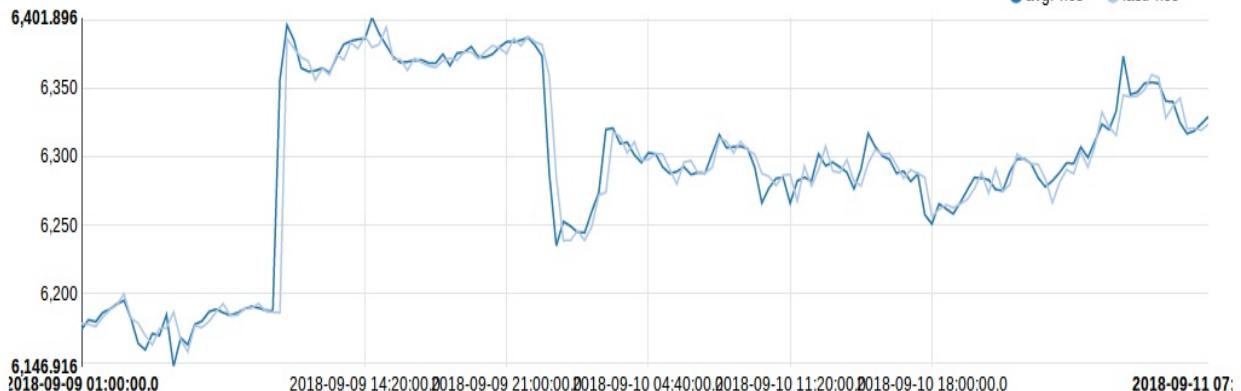
zoom

Date format

XAxis :

`Default` `Rotate` `Hide`

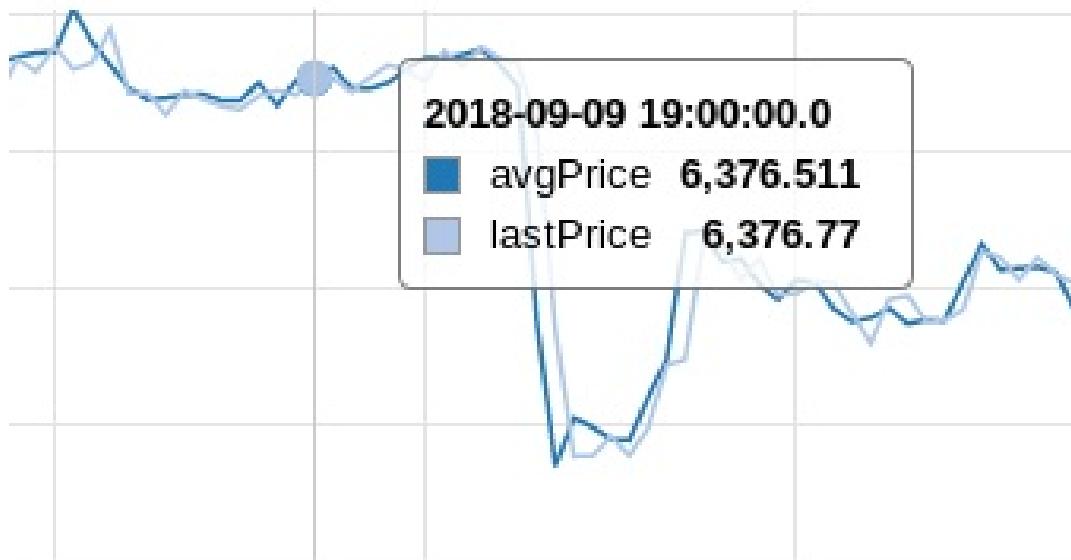
● `avgPrice` ● `lastPrice`



We can see the evolution of the average price and of the last price in 20-minute

increments.

If you hover the mouse on the chart, Zeppelin displays the information of the corresponding data point:



Feel free to try different types of charts, with different *values* for the y axis.

Drawing more charts

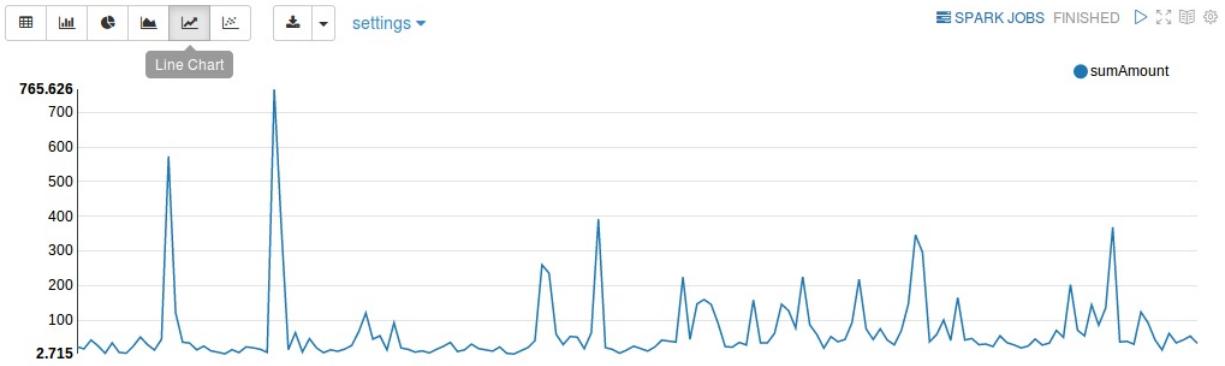
Since our `aggregate DataFrame` is available in scope, we can create new cells to plot different charts. Create two new cells with the following code:

```
|%spark  
|z.show(aggregate)
```

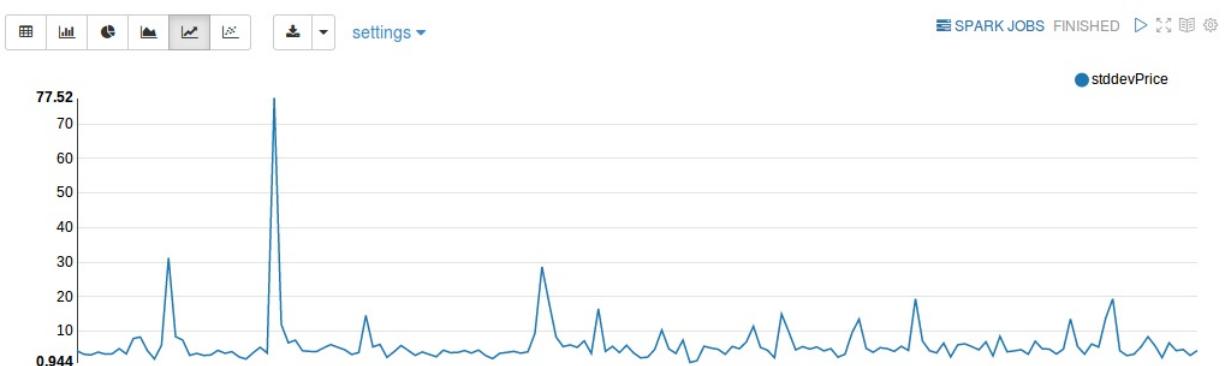
Then, on each new cell, click on the line chart button and drag the `start` column to the keys section. After that:

1. In the first chart, drag and drop `sumAmount` in the values section. The chart shows the evolution of the volume exchanged.
2. In the second chart, drag and drop `stddevPrice` in the values section. The chart shows the evolution of the standard deviation.
3. Click on settings on both cells to hide the settings.
4. Click on Hide editor on both cells.

You should obtain something like this:



Took 1 sec. Last updated by anonymous at September 17 2018, 9:35:17 PM. (outdated)



Took 1 sec. Last updated by anonymous at September 17 2018, 9:35:14 PM. (outdated)

We can observe spikes at around the same time—when there is a spike in volume exchanged, the standard deviation spikes as well. This is because many large transactions move the price significantly, which increases the standard deviation.

You now have the basic building blocks to run your own analysis. With the power of the `Dataset API`, you could drill down to a specific period using `filter` and then plot the evolution of a moving average over different time spans.

The only problem with our notebook is that we would only get new transactions every hour. The `BatchProducerApp` that we wrote in the previous chapter does not produce transactions more frequently, and if you try to call the REST API every few seconds, you will get blacklisted by the Bitstamp server. The preferred way of getting live transactions is to use a `WebSocket API`.

For solving this, in the next section, we are going to build an application called `streamInProducerApp` that will push live transactions to a Kafka topic.

Introducing Apache Kafka

In the previous section, *Introducing Lambda Architecture*, we mentioned that Kafka is used for stream processing. Apache Kafka is a high throughput distributed messaging system. It allows decoupling the data coming in with the data going out.

It means that multiple systems (**producers**) can send messages to Kafka. Kafka will then deliver these messages out to the **consumers** registered.

Kafka is distributed, resilient, and fault-tolerant, and has a very low latency. Kafka can scale horizontally by adding more machines to the system. It is written in Scala and Java.

Kafka is broadly used; Airbnb, Netflix, Uber, and LinkedIn use this technology.

The purpose of this chapter is not to become an expert in Kafka, but rather to familiarize you with the fundamentals of this technology. By the end of the chapter, you will be able to understand the use case developed in this chapter—streaming bitcoin transactions in a Lambda architecture.

Topics, partitions, and offsets

In order to process the messages exchanged between the producers and the consumer, Kafka defines three main components—topics, partitions, and offsets.

A **topic** groups messages of the same type for streaming. It has a name and a number of partitions. You can have as many topics as you want. As Kafka can be distributed on multiple nodes, it needs a way to materialize the stream of messages on these different nodes. This is why the message stream (topic) is split into multiple **partitions**. Each partition contains a portion of the messages sent to a topic.

Each node of the Kafka cluster manages several partitions. A given partition is assigned to several nodes. This avoids losing data if a node is lost, and allows a higher throughput. By default, Kafka uses the hash code of the message to assign it to a partition. You can define a key for a message to control this behavior.

In a partition, the order of the message is guaranteed, and once a message is written on it, it cannot be changed. The messages are **immutable**.

A topic can be consumed by zero to many **consumer** processes. A key feature of Kafka is that each consumer can consume the stream at his own pace: a producer can be sending message 120, while one consumer is processing message 40, and another one is processing message 100.

This asymmetry is made possible by storing the messages on disk. Kafka keeps the messages for a limited amount of time; the default setting is one week.

Internally, Kafka uses IDs to keep track of the messages and a sequence number to generate these IDs. It maintains one unique sequence per partition. This sequence number is called an **offset**. An offset only has a meaning for a specific partition.

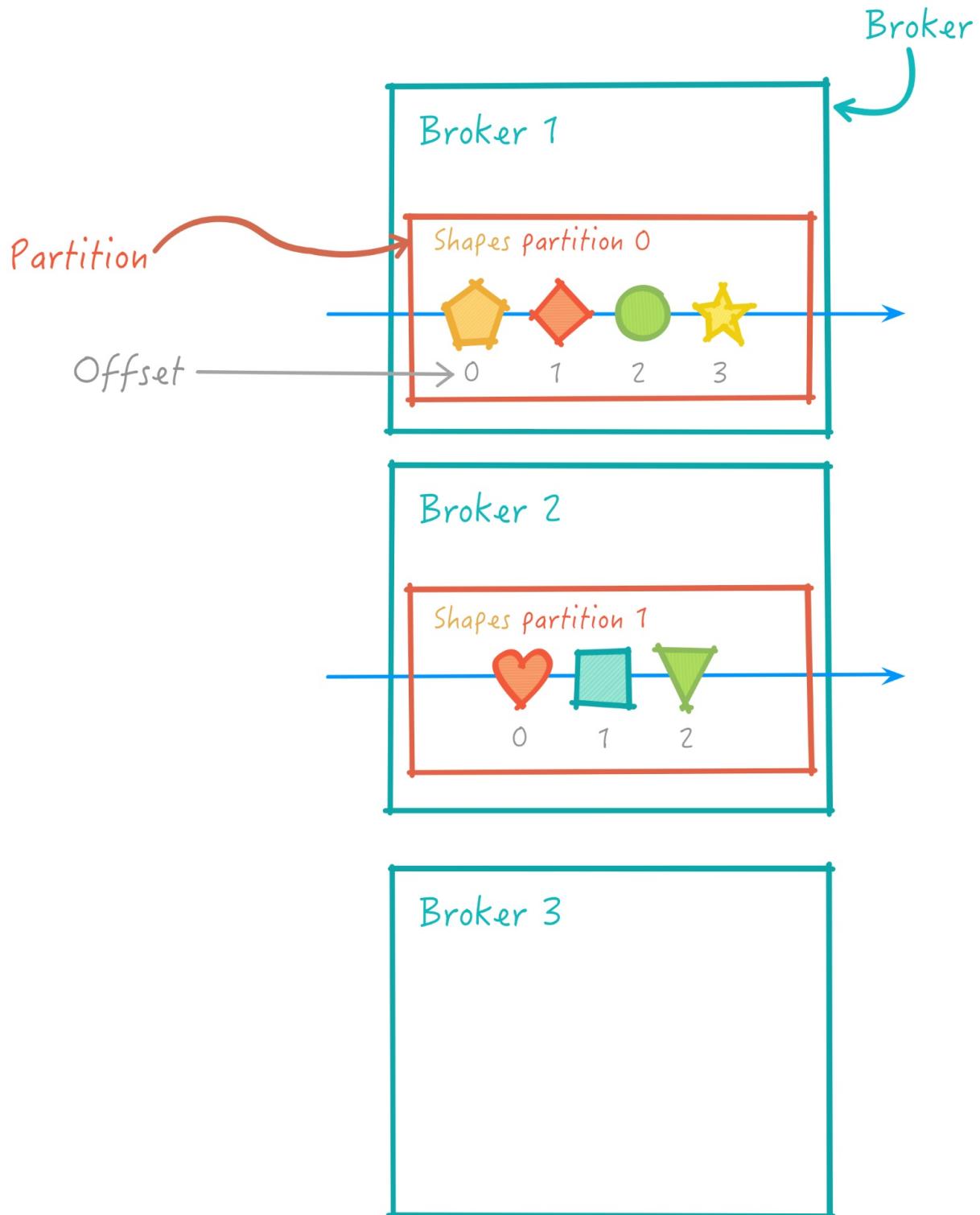
Each node of the Kafka cluster runs a process called a **broker**. Each broker manages several topics with one or more partitions.

Let's summarize everything with an example. We can define a topic named `shapes` with a number of partitions equal to two. This topic receives messages, as shown in the following diagram:

Name: `Shapes` Nb Partition: 2



Let's say we have three nodes in the cluster. The representation of the brokers, partitions, offsets, and messages would be the following:



Notice that as we defined only two partitions and we have three machines, one of the machines is not going to be used.

Another option available when you define a partition is the number of replicas. To be resilient, Kafka replicates the data in multiple brokers, so if one broker is failing, the data can be retrieved from another one.

You should now be more familiar with the fundamentals of the Kafka architecture. We will now spend a little bit of time on two other components: the producer and the consumer.

Producing data into Kafka

The component to send messages to a topic is called a **producer** in Kafka. The responsibility of a producer is to automatically select a partition through a broker to write messages. In case of failure, the producer should automatically recover.

The partition selection is based on a key. The producer will take care of sending all messages with the same key to the same partition. If there is no key provided with the message, the producer load balances the messages using a round-robin algorithm.

You can configure the producer with the level of acknowledgment you want to receive. There are three levels:

- `acks=0`: The producer sends the data and forgets it; no acknowledgment is done. There is no guarantee, and messages could be lost.
- `acks=1`: The producer waits for an acknowledgment of the first replicas. You can be sure that no data will be lost as long as the broker that acknowledged does not crash.
- `acks=all`: The producer waits for an acknowledgment of all the replicas. You can be sure that no data will be lost even if one broker crashes.

Of course, if you want an acknowledgment of all the replicas, you might expect longer latencies. Acknowledgment of the first replica (`ack=1`) is a good compromise between safety and latency.

Consuming data from Kafka

To read messages from a topic, you need to run a **consumer**. As with the producer, the consumer will automatically select the broker to read from and will recover in case of failure.

The consumer reads the messages from all partitions. Within a partition, it is guaranteed to receive messages in the same order as they were produced.

Consumer group

You can have multiple consumers for the same topic in a **consumer group**. If you have the same number of consumers and partitions in a group, each consumer will read only one partition. This allows parallelizing the consumption. If you have more consumers than partitions, the first consumer takes the partition and the rest of the consumers are in waiting mode. They will consume only if a consumer reading on a partition is failing.

Until now, we only have one group of consumers reading from partitions. In a typical system, you would have many consumer groups. For example, in the case of a Bitcoin transaction, we could have a consumer group reading the messages to perform analytics on it, and another group for a user interface that shows a feed of all the transactions. The latency between the two cases are not the same, and we don't want to have a dependency between each use case. For that purpose, Kafka uses the notion of groups.

Offset management

Another important concept is that when a consumer reads a message, it will automatically inform Kafka of the offset that was read. This way, if a consumer dies, Kafka knows the offset of the last message read. When the consumer restarts, it can send the next message to it. As for the producer, we can decide when to commit offsets. There are three options :

- At most once; as soon as the message is received, the offset is committed.
- At least once; the offset is committed after the message has been processed.
- Exactly once; the offset is committed after the message has been processed, and there are additional constraints on the producer—it must not resend messages in case of network failures. The producer must have idempotent and transactional capabilities, which were introduced in Kafka 0.11

The most commonly used is the *at least once* option. If the processing of the message is failing, you can reprocess it, but you might occasionally receive the same message multiple times. In the case of *at most once*, if anything goes wrong during the process, the message will be lost.

OK, enough theor. We have learned about topics, partitions, offsets, consumers, and producers. The last piece of missing knowledge is a simple question—how do I connect my producer or consumer to Kafka?

Connecting to Kafka

We introduced in the section *Topics, partitions, and offsets* the notion of brokers. Brokers are deployed on multiple machines, and all the brokers form what we call a Kafka cluster.

If you want to connect to a Kafka cluster, all you need to know is the address of one of the brokers. All the brokers know about all the metadata of the cluster—brokers, partitions, and topics. Internally, Kafka uses a product named **Zookeeper**. This allows the sharing of all this metadata between brokers.

Streaming transactions to Kafka

In this section, we are going to write a program that produces a stream of the BTC/USD transactions that happen in real time. Our program will:

1. Subscribe to the WebSocket API of Bitstamp to get a stream of transactions in JSON format.
2. For each transaction coming in the stream, it will:
 - Deserialize it
 - Convert it to the same `Transaction` case class that we used in `BatchProducer` in the previous chapter
 - Serialize it
 - Send it to a Kafka topic

In the next section, we will use Zeppelin again with Spark Streaming to query the data streamed to the Kafka topic.

Subscribing with Pusher

Go to Bitstamp's WebSocket API for live transactions: <https://www.bitstamp.net/websocket/>.

You will see that this API uses a tool called Pusher channels for real-time WebSocket streaming. The API documentation provides a Pusher Key that we need to use to receive live transactions.

Pusher channels is a hosted solution for delivering a stream of messages using a publish/subscribe pattern. You can find out more on their website: <https://pusher.com/features>.

Let's try to use Pusher to receive some live BTC/USD transactions. Open the project `bitcoin-analyser`, start a new Scala console, and type the following:

```
import com.pusher.client.Pusher
import com.pusher.client.channel.SubscriptionEventListener

val pusher = new Pusher("de504dc5763aeef9ff52")
pusher.connect()
val channel = pusher.subscribe("live_trades")

channel.bind("trade", new SubscriptionEventListener() {
    override def onEvent(channel: String, event: String, data: String):
        Unit = {
        println(s"Received event: $event with data: $data")
    }
})
```

Let's have a look in detail:

1. In the first line, we create the Pusher client with the key that was specified in Bitstamp's documentation.
2. Then we connect to the remote Pusher server and subscribe to the "live_trades" channel. We obtain an object of type `channel`.
3. Finally, we use the `channel` to register (`bind`) a callback function that will be called every time the `channel` receives a new event with the name `trade`.

After a few seconds, you should see some trades being printed:

```
| Received event: trade with data: {"amount": 0.001, "buy_order_id": 2165113017, "sell_order_id": 2165113018}
| Received event: trade with data: {"amount": 0.008946000000000008, "buy_order_id": 2165113018, "sell_order_id": 2165113019}
| (...)
```

The data is in JSON format, with a schema that conforms to what was defined in Bitstamp's WebSocket documentation.

With these few lines, we can write the first building block of our application. Create a new object, `StreamingProducerApp`, in the package `coinyser` in `src/main/scala` with the following content:

```
package coinyser

import java.sql.Timestamp
import java.text.SimpleDateFormat
import java.util.TimeZone

import cats.effect.IO
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.pusher.client.Client
import com.pusher.client.channel.SubscriptionEventListener
import com.typesafe.scalalogging.StrictLogging

object StreamingProducer extends StrictLogging {

    def subscribe(pusher: Client)(onTradeReceived: String => Unit): IO[Unit] =
        for {
            _ <- IO(pusher.connect())
            channel <- IO(pusher.subscribe("live_trades"))

            _ <- IO(channel.bind("trade", new SubscriptionEventListener() {
                override def onEvent(channel: String, event: String, data: String): Unit = {
                    logger.info(s"Received event: $event with data: $data")
                    onTradeReceived(data)
                }
            }))
        } yield ()
}
```

Our function `subscribe` takes a `Pusher` instance (of type `client`) and a callback function, `onTradeReceived`, and returns `IO[Unit]`. When `IO` is run, it will call `onTradeReceived` each time a new trade is received. The implementation is similar to the few lines that we typed into the console. It basically wraps every side-effecting function in an `IO`.



For the sake of conciseness and readability, we have not exposed the details of this function's unit test. You can check it out in the GitHub repository.

For writing the test, we had to create a `FakePusher` class that implements a few methods of the `client` interface.

Deserializing live transactions

The JSON payload that we receive when we subscribe to live transactions is slightly different from the one we had in the REST endpoint for fetching batch transactions.

We are going to need to deserialize it to a case class before we can transform it to the same `Transaction` Case class that we used in the previous chapter. For this, first create a new case class, `coinyser.WebsocketTransaction` in `src/main/scala`:

```
package coinyser

case class WebsocketTransaction(amount: Double,
                                buy_order_id: Long,
                                sell_order_id: Long,
                                amount_str: String,
                                price_str: String,
                                timestamp: String,
                                price: Double,
                                `type`: Int,
                                id: Int)
```

The names and types of the attributes correspond to the JSON attributes.

After that, we can write a unit test for a new function, `deserializeWebsocketTransaction`. Create a new class, `coinyser.StreamingProducerSpec`, in `src/test/scala`:

```
package coinyser

import java.sql.Timestamp
import coinyser.StreamingProducerSpec._
import org.scalactic.TypeCheckedTripleEquals
import org.scalatest.{Matchers, WordSpec}

class StreamingProducerSpec extends WordSpec with Matchers with TypeCheckedTripleEquals
  "StreamingProducer.deserializeWebsocketTransaction" should {
    "deserialize a valid String to a WebsocketTransaction" in {
      val str =
        """{"amount": 0.045318270000000001, "buy_order_id": 1969499130,
          |"sell_order_id": 1969495276, "amount_str": "0.04531827",
          |"price_str": "6339.73", "timestamp": "1533797395",
          |"price": 6339.729999999996, "type": 0, "id":
          |71826763}""".stripMargin
      StreamingProducer.deserializeWebsocketTransaction(str) should
        ===(SampleWebsocketTransaction)
    }
}
```

```

object StreamingProducerSpec {
    val SampleWebsocketTransaction = WebsocketTransaction(
        amount = 0.04531827, buy_order_id = 1969499130, sell_order_id =
        1969495276, amount_str = "0.04531827", price_str = "6339.73",
        timestamp = "1533797395", price = 6339.73, `type` = 0, id =
        71826763)
}

```

The test is straightforward—we define a sample JSON string, call the function under `test`, and make sure the deserialized `object SampleWebsocketTransaction` contains the same values.

Now we need to implement the function. Add a new `val mapper: ObjectMapper` and a new function `deserializeWebsocketTransaction` to the `StreamingProducer` object:

```

package coinyser

import java.sql.Timestamp
import java.text.SimpleDateFormat
import java.util.TimeZone

import cats.effect.IO
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.pusher.client.Client
import com.pusher.client.channel.SubscriptionEventListener
import com.typesafe.scalalogging.StrictLogging

object StreamingProducer extends StrictLogging {

    def subscribe(pusher: Client)(onTradeReceived: String => Unit): IO[Unit] =
        ...

    val mapper: ObjectMapper = {
        val m = new ObjectMapper()
        m.registerModule(DefaultScalaModule)
        val sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
        sdf.setTimeZone(TimeZone.getTimeZone("UTC"))
        m.setDateFormat(sdf)
    }

    def deserializeWebsocketTransaction(s: String): WebsocketTransaction
        = {
            mapper.readValue(s, classOf[WebsocketTransaction])
        }
}

```

For this part of the project, we use the **Jackson** Java library to deserialize/serialize JSON objects. It is the library that is used under the hood by Spark when it reads/writes dataframe from/to JSON. Hence, it is available without adding any more dependencies.

We define a constant, `mapper: ObjectMapper`, which is the entry point of Jackson for

serializing/deserializing classes. We configure it to write timestamps in a format that is compatible with what Spark can parse. This will be necessary later on when we read the Kafka topic using Spark. Then the function's implementation calls `readValue` to deserialize the JSON into `WebSocketTransaction`.

Converting to transaction and serializing

We are able to listen to live transactions and deserialize them to objects of type `WebsocketTransaction`. The next steps are:

- Convert these `WebsocketTransaction` objects into the same case class, `Transaction`, that we defined in the previous chapter.
- Send these `Transaction` objects to a Kafka topic. But for this, they need to be serialized first. The simplest way is to serialize to JSON.

As usual, we start by writing tests. Add the following tests to `StreamingProducerSpec`:

```
class StreamingProducerSpec extends WordSpec with Matchers with TypeCheckedTripleEquals

"StreamingProducer.deserializeWebsocketTransaction" should {...}

"StreamingProducer.convertTransaction" should {
  "convert a WebSocketTransaction to a Transaction" in {
    StreamingProducer.convertWsTransaction
    (SampleWebsocketTransaction) should
      ===(SampleTransaction)
  }
}

"StreamingProducer.serializeTransaction" should {
  "serialize a Transaction to a String" in {
    StreamingProducer.serializeTransaction(SampleTransaction) should
      ===(SampleJsonTransaction)
  }
}

object StreamingProducerSpec {
  val SampleWebsocketTransaction = WebsocketTransaction(...)

  val SampleTransaction = Transaction(
    timestamp = new Timestamp(1533797395000L), tid = 71826763,
    price = 6339.73, sell = false, amount = 0.04531827)

  val SampleJsonTransaction =
    """{"timestamp": "2018-08-09 06:49:55",
      |"date": "2018-08-09", "tid": 71826763, "price": 6339.73, "sell": false,
      |"amount": 0.04531827}""".stripMargin
}
```

The test for `convertWsTransaction` checks that, once

converted, `SampleWebsocketTransaction` is the same as `SampleTransaction`.

The test for `serializeTransaction` checks that, once serialized, `SampleTransaction` is the same as `SampleJsonTransaction`.

The implementation of these two functions is straightforward. Add the following definitions in `StreamingProducer`:

```
object StreamingProducer extends StrictLogging {

    def subscribe(pusher: Client)(onTradeReceived: String => Unit): IO[Unit] = ...

    val mapper: ObjectMapper = {...}

    def deserializeWebsocketTransaction(s: String): WebsocketTransaction
        = {...}

    def convertWsTransaction(wsTx: WebsocketTransaction): Transaction =
        Transaction(
            timestamp = new Timestamp(wsTx.timestamp.toLong * 1000), tid =
            wsTx.id, price = wsTx.price, sell = wsTx.`type` == 1, amount =
            wsTx.amount)

    def serializeTransaction(tx: Transaction): String =
        mapper.writeValueAsString(tx)
}
```

In `convertWsTransaction`, we have to multiply the timestamp by 1,000 to get the time in milliseconds. The other attributes are just copied.

In `serializeTransaction`, we reuse the `mapper` object to serialize a `Transaction` object to JSON.

Putting it all together

We now have all the building blocks to create our application. Create a new object, `coinyser.StreamingProducerApp`, and type the following code:

```
package coinyser

import cats.effect.{ExitCode, IO, IOApp}
import com.pusher.client.Pusher
import StreamingProducer._
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}
import scala.collection.JavaConversions._

object StreamingProducerApp extends IOApp {
    val topic = "transactions"

    val pusher = new Pusher("de504dc5763aeeef9ff52")

    val props = Map(
        "bootstrap.servers" -> "localhost:9092",
        "key.serializer" ->
            "org.apache.kafka.common.serialization.IntegerSerializer",
        "value.serializer" ->
            "org.apache.kafka.common.serialization.StringSerializer")

    def run(args: List[String]): IO[ExitCode] = {
        val kafkaProducer = new KafkaProducer[Int, String](props)

        subscribe(pusher) { wsTx =>
            val tx = convertWsTransaction(deserializeWebsocket
                Transaction(wsTx))
            val jsonTx = serializeTransaction(tx)
            kafkaProducer.send(new ProducerRecord(topic, tx.tid, jsonTx))
        }.flatMap(_ => IO.never)
    }
}
```

Our object extends `cats.IOApp`, and as such we have to implement a `run` function that returns `IO[ExitCode]`.

In the `run` function, we first create `KafkaProducer[Int, String]`. This Java class from the Kafka client library will allow us to send messages to a topic. The first type parameter is the type of the message's key. The key of our message will be the `tid` attribute in the `Transaction` case class, which is of type `Int`. The second type parameter is the type of the message itself. In our case, we use `String`, because we are going to serialize our messages to JSON. If storage space was a concern, we could have used `Array[Byte]` and a binary serialization format such as Avro.

The `props Map` passed to construct `KafkaProducer` contains various configuration options for interacting with the Kafka cluster. In our program, we pass the minimum set of properties and leave the others with default values, but there are many more fine-tuning options. You can find out more here: <http://kafka.apache.org/documentation.html#producerconfigs>.

Then we call the `StreamingProducer.subscribe` function that we implemented earlier, and pass a callback function that will be called each time we receive a new transaction. This anonymous function will:

1. Deserialize the JSON into `websocketTransaction`.
2. Convert `websocketTransaction` into `Transaction`.
3. Serialize `Transaction` to JSON.
4. Send the JSON transaction to the Kafka topic using `kafkaProducer.send`. For this, we have to create `ProducerRecord`, which contains the `topic` name, the key, and the content of the message.

The `subscribe` function returns `IO[Unit]`. This will start a background thread and complete immediately when we run it. But we do not want to stop the main thread immediately; we need to keep our program running forever. This is why we `flatMap` it and return `IO.never`, which will keep the main thread running until we kill the process.

Running StreamingProducerApp

Before running our application, we need to start a Kafka cluster. For the sake of simplicity, we are just going to start a single broker on your workstation. If you wish to set up a multinode cluster, please refer to the Kafka documentation:

1. Download Kafka 0.10.2.2 from this URL: https://www.apache.org/dyn/closer.cgi?path=/kafka/1.1.1/kafka_2.11-1.1.1.tgz. We use the version 1.1.1 because we have tested it with the `spark-sql-kafka-0.10` library at the time of writing. You could use a later version of Kafka, but it is not guaranteed that an old Kafka client can always communicate with a more recent broker.
2. Open a console and then decompress the package in your favorite directory:
`tar xfvz kafka*.tgz`.
3. Go to the installation directory and start Zookeeper:

```
| cd kafka_2.11-1.1.1  
| bin/zookeeper-server-start.sh config/zookeeper.properties
```

You should see a lot of log output. The last line should contain `INFO` `binding to port 0.0.0.0/0.0.0.0:2181`.

4. Open a new console, and start the Kafka server:

```
| cd kafka_2.11-1.1.1  
| bin/kafka-server-start.sh config/server.properties
```

You should see some logs present in the last line.

5. Once Kafka has started, open a new console and run the following commands:

```
| cd kafka_2.11-1.1.1  
| bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic transact
```

This command starts a console consumer that listens to the `transactions` topic. Any message sent to this topic will be printed on the console. This will allow us to test that our program works as expected.

6. Now run `streamingProducerApp` in IntelliJ. After a few seconds, you should get an output similar to this in IntelliJ:

```
| 18/09/22 17:30:41 INFO StreamingProducer$: Received event: trade with data: {"am
```

Our application received some messages from the WebSocket API and sent them to the Kafka topic `transactions`.

7. If you then go back to the console on which you started the console consumer, you should see new `Transaction` serialized objects being printed in real time:

```
| {"timestamp":"2018-09-22 16:30:40", "date":"2018-09-22", "tid":74611373, "price":66
```

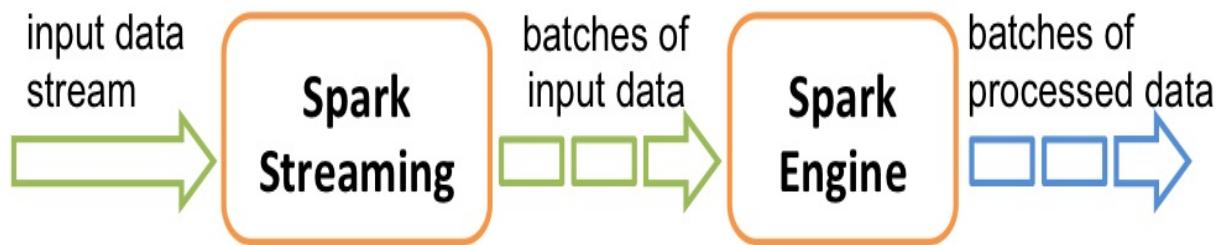
This indicates that our streaming application works as expected. It listens to the Pusher channel to receive BTC/USD transactions and sends everything it receives to the Kafka topic `transactions`. Now that we have some data going to a Kafka topic, we can use Spark Streaming to run some analytics queries.

Introducing Spark Streaming

In [Chapter 10](#), *Fetching and Persisting Bitcoin Market Data*, we used Spark to save transactions in a batch mode. The batch mode is fine when you have to perform an analysis on a bunch of data all at once.

But in some cases, you might need to process data as it is entering into the system. For example, in a trading system, you might want to analyze all the transactions done by the broker to detect fraudulent transactions. You could perform this analysis in batch mode after the market is closed; but in this case, you can only act after the fact.

Spark Streaming allows you to consume a streaming source (file, socket, and Kafka topic) by dividing the input data into many micro-batches. Each micro-batch is an RDD that can then be processed by the **Spark Engine**. Spark divides the input data using a time window. So if you define a time window of 10 seconds, then Spark Streaming will create and process a new RDD every 10 seconds:



Going back to our fraud detection system, by using Spark Streaming we could detect a pattern of a fraudulent transaction as it arises and immediately act on the broker to limit the damage.

In the previous chapter, you learned that Spark offers two APIs for processing batch data—`RDD` and `dataset`. `RDD` is the original and core API, and `dataset` is the more recent one that allows it to perform SQL queries and optimize the execution plan automatically. Similarly, Spark Streaming offers two APIs to process streams of data:

- **Discretized Stream (DStream)** is basically a continuous series of RDDs. Each RDD in a DStream contains data from a certain time interval. You can get more information about it here: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>.
- **Structured streaming** is more recent. It allows you to use the same methods as the Dataset API. The difference is that in `Dataset` you manipulate an unbound table that grows as new input data arrives. You can get more information about it here: <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>.

In the next section, we are going to use Zeppelin to run a Spark-structured streaming query on the BTC/USD transaction data that we previously produced in a Kafka topic.

Analyzing streaming transactions with Zeppelin

At this point, if you have not done so yet, you should start the following processes on your machine. Please refer to the previous sections if you are not sure how to start them:

- Zookeeper
- Kafka broker
- The `StreamingProducerApp` consuming live BTC/USD transactions and pushing them to a Kafka topic named `transactions`.
- Apache Zeppelin

Reading transactions from Kafka

Using your browser, create a new Zeppelin Notebook named `streaming`, and then type the following code in the first cell:

```
case class Transaction(timestamp: java.sql.Timestamp,  
                      date: String,  
                      tid: Int,  
                      price: Double,  
                      sell: Boolean,  
                      amount: Double)  
  
val schema = Seq.empty[Transaction].toDS().schema
```

Execute the cell to define the `Transaction` class and the variable `schema`. We have to redefine the `Transaction` case class in Zeppelin because Zeppelin does not have access to the classes of our IntelliJ project, `bitcoin-analyser`. We could have instead packaged a `.jar` file and added it into Zeppelin's dependency settings, but as this is the only class we need, we found it easier to redefine it.

The `schema` variable is of type `DataType`. We will use it in the following paragraph to deserialize the JSON transactions that we will consume from the Kafka topic.

Then create a new cell underneath with the following code:

```
val dfStream = {  
    spark.readStream.format("kafka")  
    .option("kafka.bootstrap.servers", "localhost:9092")  
    .option("startingoffsets", "latest")  
    .option("subscribe", "transactions")  
    .load()  
    .select(  
        from_json(col("value").cast("string"), schema)  
            .alias("v")).select("v.*").as[Transaction]  
}
```

This creates a new variable `dfStream` of type `DataFrame`. For creating this, we called:

- The method `readStream` on the `spark: SparkSession` object. It returns an object of the `DataStreamReader` type that we can configure further.
- The methods `format("kafka")` and `option("kafka.bootstrap.servers", "localhost:9092")` specify that we want to read data from Kafka and point to the broker `localhost` on port `9092`.
- `option("startingoffsets", "latest")` indicates that we only want to consume the data from the latest offset. Other options are `"earliest"`, or a JSON string

specifying a starting offset for each `TopicPartition`.

- `option("subscribe", "transaction")` specifies that we want to listen to the topic `transactions`. This is the topic name that we used in our `StreamingProducerApp`.
- The call to `load()` returns `DataFrame`. But at this stage, it only contains one `value` column with the raw JSON.
- We then deserialize the JSON using the `from_json` function and the `schema` object that we created in the previous paragraph. This returns a single column of type `struct`. In order to have all the columns at the root level, we rename the column using `alias("v")`, and select all columns inside it using `select("v.*")`.

When you run the paragraph, you will get the following output:

```
| dfStream: org.apache.spark.sql.DataFrame = [timestamp: timestamp, date: string ... 4 more columns]
```

At this point, we have `DataFrame` with all the columns that we need to run further analysis on right? Let's try to display it. Create a new paragraph with this code and run it:

```
| z.show(dfStream)
```

You should see the following error:

```
| java.lang.RuntimeException: java.lang.reflect.InvocationTargetException at org.apache.spark.sql.Dataset.show(Dataset.scala:113)
```

The problem here is that there is a missing writing step. Even though the type `DataFrame` is exactly the same as the one we obtained in the `Batch` notebook, we cannot use it in exactly the same way. We have created a streaming `DataFrame` that can consume and transform messages from Kafka, but what is it supposed to do with these messages?

In the Spark-structured streaming world, the action methods such as `show`, `collect`, or `take` cannot be used. You have to tell Spark where to write the data it consumes.

Writing to an in-memory sink

A Spark structured streaming process has three types of components:

- The **input source** is specified with the `format(source: String)` method on `DataStreamReader`. This source can be a file, a Kafka topic, a network socket, or a constant rate. Once configured with `option`, the call to `load()` returns a `DataFrame`.
- **Operations** are the classic `DataFrame/Dataset` transformations, such as `map`, `filter`, `flatMap`, and `reduce`. They take `Dataset` as input and return another transformed `Dataset` with the recorded transformation.
- The **output sink** writes the transformed data. For specifying the sink, we must first obtain `DataStreamWriter` by calling the `writeStream` method on `Dataset`, and then configure it. For this, we have to call the `format(source: String)` method on `DataStreamWriter`. The output sink can be a file, a Kafka topic, or `foreach` that takes a callback. For debugging purposes, there is also a console sink and a memory sink.

Going back to our streaming transactions, the error that we obtained after calling `z.show(dfStream)` indicated that we were missing a sink for our `DataFrame`. To remediate this, add a new paragraph with the following code:

```
val query = {  
    dfStream  
        .writeStream  
        .format("memory")  
        .queryName("transactionsStream")  
        .outputMode("append")  
        .start()  
}
```

This code configures a memory sink for our `DataFrame`. This sink creates an in-memory Spark table whose name is given by `queryName("transactionsStream")`. A table named `transactionsStream` will be updated every time a new micro-batch of transactions is processed.

There are several strategies for writing to a streaming sink, specified by `outputMode(outputMode: String)`:

- "append" means that only the new rows will be written to the sink.

- "complete" will write all the rows every time there is an update.
- "update" will write all the rows that were updated. This is useful when you perform some aggregation; for instance, counting the number of messages. You would want the new count to be updated every time there is a new row coming in.

Once our `DataStreamWriter` is configured, we call the `start()` method. This will start the whole workflow in the background. It is only from this point that the data starts to be consumed from the Kafka topic and gets written to the in-memory table. All the previous operations were lazy and were just configuring the workflow.

Run the paragraph now, and you will see an output which looks like this:

```
| query: org.apache.spark.sql.streaming.StreamingQuery = org.apache.spark.sql.execution.st
```

This `StreamingQuery` object is a handle to the streaming query running in the background. You can use it to monitor the progress of the streaming workflow, get some information about its execution plan, or `stop` it altogether. Feel free to explore the API of `StreamingQuery` and try to call its methods.

Drawing a scatter chart

Since we have a running `streamingQuery` that writes to an in-memory table called `transactionsStream`, we can display the data contained in this table with the following paragraph:

```
| z.show(spark.table("transactionsStream").sort("timestamp"))
```

Run this paragraph, and if you have some transactions coming into your Kafka topic, you should see a table which looks like this:

The screenshot shows a Jupyter Notebook cell with the following content:

```
%spark
z.show(spark.table("transactionsStream").sort("timestamp"))
```

The cell output displays a table of transaction data. The table has the following structure:

timestamp	date	tid	price	sell	amount
2018-09-23 11:23:37.0	2018-09-23	74649764	6757.22	false	0.001
2018-09-23 11:23:49.0	2018-09-23	74649765	6753.9	true	0.5812
2018-09-23 11:24:14.0	2018-09-23	74649786	6757.22	false	0.01051763
2018-09-23 11:24:37.0	2018-09-23	74649805	6753.9	true	1.04394506
2018-09-23 11:24:39.0	2018-09-23	74649810	6749.32	true	1.51821
2018-09-23 11:24:39.0	2018-09-23	74649811	6749.16	false	0.28184494
2018-09-23 11:24:39.0	2018-09-23	74649809	6749.33	false	1.5
2018-09-23 11:25:48.0	2018-09-23	74649833	6754.14	true	0.01392627

You can then click on the scatter chart button. If you want to draw the evolution of the price, you can drag and drop the `timestamp` in the `x` axis and the `price` in the `y` axis. You should see something like this:



If you want to refresh the chart with the latest transactions, you can just re-run the paragraph.

This is pretty good, but if you let the query run for some time, you might end up with quite a lot of transactions. Ultimately, you will reach the limit of what Zeppelin can handle and you will get the same error, `OUTPUT IS TRUNCATED`, that we had in the section *Drawing our first chart*.

To remediate this, we have to group the transactions using a time window, in the same way that we did for the batch transactions that we read from `parquet`.

Aggregating streaming transactions

Add a new paragraph with the following code:

```
val aggDfStream = {  
    dfStream  
        .withWatermark("timestamp", "1 second")  
        .groupBy(window($"timestamp", "10 seconds").as("window"))  
        .agg(  
            count($"tid").as("count"),  
            avg("price").as("avgPrice"),  
            stddev("price").as("stddevPrice"),  
            last("price").as("lastPrice"),  
            sum("amount").as("sumAmount")  
        )  
        .select("window.start", "count", "avgPrice", "lastPrice",  
        "stddevPrice", "sumAmount")  
}
```

This code is very similar to the one we wrote in the section *Drawing our first chart*. The only difference is the call to `withWatermark`, but the rest of the code is the same. This is one of the main benefits of using Spark-structured streaming—we can reuse the same code for transforming batch datasets and streaming datasets.

Watermarking is mandatory when we aggregate streaming datasets. In a few words, we need to tell Spark how long it will wait to receive late data before discarding it, and what timestamp column it should use to measure the time between two rows.

As Spark is distributed, it can potentially consume the Kafka topic using several consumers, each of them consuming a different partition of the topic. This means that Spark Streaming could potentially process transactions out of order. In our case, we have just one Kafka broker and do not expect a high volume of transactions; hence, we use a low watermark of one second. Watermarking is a quite complex subject. You can find out more on the Spark website: <https://spark.apache.org/docs/2.3.1/structured-streaming-programming-guide.html#window-operations-on-every-time>.

Once you have run this new paragraph, you will have a new `Dataframe`, `aggDfStream`, which will aggregate transactions in 10-second windows. But before we can draw a chart of this aggregated data, we need to create a query to connect an in-

memory sink. Create a new paragraph with this code:

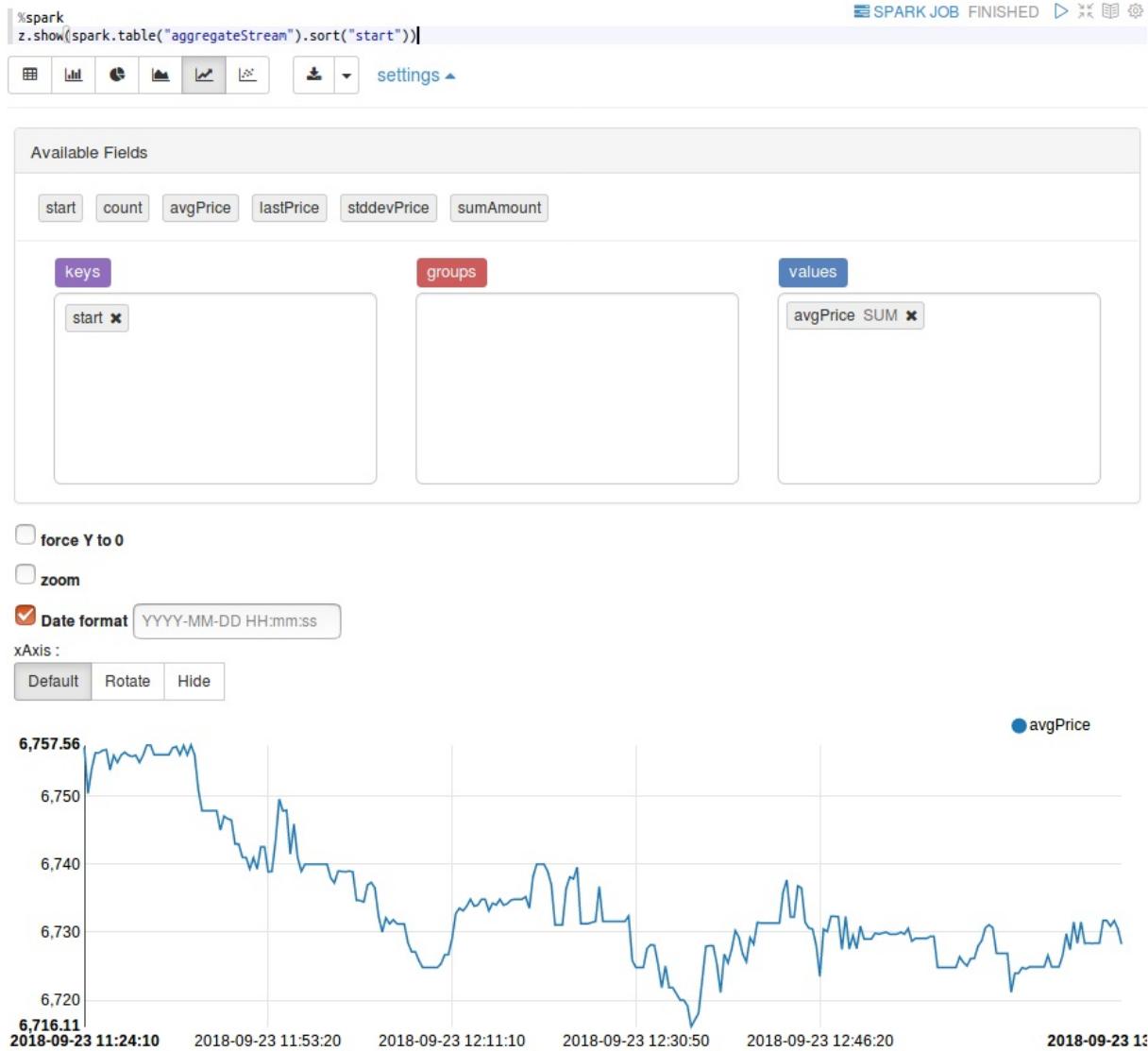
```
| val aggQuery = {  
|   aggDfStream  
|     .writeStream  
|     .format("memory")  
|     .queryName("aggregateStream")  
|     .outputMode("append")  
|     .start()  
| }
```

It is nearly the same as the one we wrote in the section *Drawing a scatter chart*. We just used `aggDfStream` instead of `df`, and the output table name is now called `aggregateStream`.

Finally, add a new paragraph to display the data contained in the `aggregateStream` table:

```
| z.show(spark.table("aggregateStream").sort("start"))
```

You will need to wait at least 30 seconds after having run `aggQuery` to get some aggregated transaction data. Wait a bit, then run the paragraph. After that, click on the line chart button, then drag and drop the `start` column in the keys section and `avgPrice` in the values section. You should see a chart that looks like this:



If you re-run the paragraph after 10 seconds or more, you should see it being updated with new transactions.

It turns out that this `aggregateStream DataFrame` has exactly the same columns as the `aggregate DataFrame` that we created in the section *Drawing our first chart*. The difference between them is that `aggregate` is built using the historical batch data coming from Parquet files, and `aggregateStream` is built using the live data coming from Kafka. They are actually complementary—`aggregate` has got all the transactions from the last hours or days, while `aggregateStream` has got all the transactions from the point in time in which we started the `aggQuery`. If you want to draw a chart containing all the data up to the latest live transaction, you can simply use `union` of both dataframes, adequately filtered so that time windows are

not duplicated.

Summary

In this chapter, you have learned how to use Zeppelin to query parquet files and display some charts. Then, you developed a small program to stream transaction data from a WebSocket to a Kafka topic. Finally, you used Spark Streaming inside Zeppelin to query the data arriving in the Kafka topic in real time.

With all these building blocks in place, you have all the tools to analyze the Bitcoin transaction data in more detail. You could let the `BatchProducerApp` run for several days or weeks to get some historical data. With the help of Zeppelin and Spark, you could then try to detect patterns and come up with a trading strategy. Finally, you could then use a Spark Streaming flow to detect in real time when some trading signal arises and perform a transaction automatically.

We have produced streaming data on only one topic, but it would be quite straightforward to add other topics covering other currency pairs, such as BTC/EUR or BTC/ETH. You could also create another program that fetches data from another cryptocurrency exchange. This would enable you to create Spark Streaming queries that detect arbitrage opportunities (when the price of a product is cheaper in one market than in another).

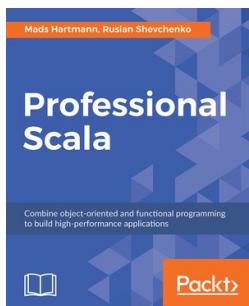
The building blocks we implemented in this chapter can be also used in a Lambda architecture. A Lambda architecture is a data processing architecture designed to handle large volumes of data by using both batch processing and streaming methods. Many Lambda architectures involve having different code bases for the batch and streaming layers, but with Spark, this negative point can be greatly reduced. You can find out more about Lambda architectures on this website: <http://lambda-architecture.net/>.

This completes the final chapter in our book. We hope you enjoyed reading it as much as we enjoyed writing it. You will be empowered by the ins and outs of various concepts of Scala. Scala is a language that is well designed and is not an end by himself, this is the basement to build more interesting concepts, like the type theory and for sure the category theory, I encourage your curiosity to look for more concepts to continually improve the readability and the quality of your

code. You will also be able to apply it to solve a variety of real-world problems.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Professional Scala

Mads Hartmann, Ruslan Shevchenko

ISBN: 978-1-78953-3-835

- Understand the key language syntax and core concepts for application development
- Master the type system to create scalable type-safe applications while cutting down your time spent debugging
- Understand how you can work with advanced data structures via built-in features such as the Collections library
- Use classes, objects, and traits to transform a trivial chatbot program into a useful assistant
- Understand what are pure functions, immutability, and higher-order functions
- Recognize and implement popular functional programming design patterns



Scala Design Patterns - Second Edition

Ivan Nikolov

ISBN: 978-1-78847-130-5

- Immerse yourself in industry-standard design patterns—structural, creational, and behavioral—to create extraordinary applications
- See the power of traits and their application in Scala
- Implement abstract and self types and build clean design patterns
- Build complex entity relationships using structural design patterns
- Create applications faster by applying functional design patterns

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!