

# DEVELOPING MODERN APPLICATIONS WITH SCALA

Hot Recipes for Scala Development



**ANDRIY REDKO**



**Java Code Geeks**  
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

# **Developing Modern Applications with Scala**

# Contents

<b>1</b>	<b>Build with SBT</b>	<b>1</b>
1.1	Introduction	1
1.2	Build Tools: Solid Foundation of Every Project	1
1.3	SBT: (not so) Simple Build Tool	1
1.4	Project Structure	2
1.5	Build Definitions	2
1.6	Single-Module Projects	2
1.7	Multi-Module Projects	4
1.8	Migrating from Maven	5
1.9	Interactivity at Heart	5
1.10	Integrations with IDEs	6
1.11	Conclusions	6
1.12	What's next	6
<b>2</b>	<b>Testing</b>	<b>7</b>
2.1	Introduction	7
2.2	ScalaCheck: the Power of Property-Based Testing	7
2.3	ScalaTest: Tests as Specifications	8
2.4	Specs2: One to Rule Them All	10
2.5	Conclusions	12
2.6	What's next	12
<b>3</b>	<b>Reactive Applications</b>	<b>13</b>
3.1	Introduction	13
3.2	Being Reactive	13
3.3	Reactive Streams Specification	13
3.4	Reactive Streams in the Wild	14
3.5	Akka Streams: Reactive Streams Implementation	14
3.5.1	Basic Concepts	14
3.5.2	Materialization	14
3.5.3	Sources and Sinks	15

---

3.5.4	Flows	16
3.5.5	Graphs and BidiFlows	16
3.5.6	Back-Pressure	18
3.5.7	Handling Errors	18
3.5.8	Testing	19
3.6	Conclusions	20
3.7	What's next	20
<b>4</b>	<b>Database Access with Slick</b>	<b>21</b>
4.1	Introduction	21
4.2	Database Access, the Functional Way	21
4.3	Configuration	21
4.4	Table Mappings	22
4.5	Repositories	23
4.6	Manipulating Schemas	24
4.7	Inserting	24
4.8	Querying	24
4.9	Updating	25
4.10	Deleting	25
4.11	Streaming	25
4.12	SQL	26
4.13	Testing	26
4.14	Conclusions	27
4.15	What's next	27
<b>5</b>	<b>Console Applications</b>	<b>28</b>
5.1	Introduction	28
5.2	UI-less and Command Line Oriented	28
5.3	Driven by Arguments	28
5.4	The Power of Interactivity	30
5.5	Conclusions	34
5.6	What's next	34
<b>6</b>	<b>Concurrency and parallelism with Akka</b>	<b>35</b>
6.1	Introduction	35
6.2	Threads	35
6.3	Reactors and Event Loops	37
6.4	Actors and Messages	37
6.5	Meet Akka	38
6.6	Supervision	39

---

6.7	Patterns	39
6.8	Typed Actors	40
6.9	Scheduler	41
6.10	Event Bus	41
6.11	Remoting	42
6.12	Testing	43
6.13	Conclusions	44
6.14	What's next	44
<b>7</b>	<b>Web Applications with Play Framework</b>	<b>45</b>
7.1	Introduction	45
7.2	MVC and the Power of Patterns	45
7.3	Play Framework: enjoyable and productive	46
7.4	Controllers, Actions and Routes	47
7.5	Views and Templates	47
7.6	Action Compositions and Filters	50
7.7	Accessing Database	51
7.8	Using Akka	51
7.9	WebSockets	51
7.10	Server-Sent Events	52
7.11	Running Play Applications	54
7.12	Secure HTTP (HTTPS)	54
7.13	Testing	54
7.14	Conclusions	56
7.15	What's next	56
<b>8</b>	<b>Web APIs with Akka HTTP</b>	<b>57</b>
8.1	Introduction	57
8.2	Being REST(ful)	57
8.3	From Spray to Akka Http	57
8.4	Staying on the Server	57
8.4.1	Routes and Directives	58
8.4.2	Marshalling and Unmarshalling	58
8.4.3	Directives in Depth	60
8.4.4	When things go wrong	61
8.4.5	Startup / Shutdown	61
8.4.6	Secure HTTP (HTTPS)	62
8.4.7	Testing	63
8.5	Living as a Client	63
8.6	Conclusions	64
8.7	What's next	64

---

Copyright (c) Exelixis Media P.C., 2016

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

# Preface

Scala is a general-purpose programming language. It has full support for functional programming and a very strong static type system. Designed to be concise, many of Scala's design decisions were inspired by criticism of Java's shortcomings. Scala source code is intended to be compiled to Java bytecode, so that the resulting executable code runs on a Java virtual machine. Java libraries may be used directly in Scala code and vice versa (language interoperability).

Like Java, Scala is object-oriented, and uses a curly-brace syntax reminiscent of the C programming language. Unlike Java, Scala has many features of functional programming languages like Scheme, Standard ML and Haskell, including currying, type inference, immutability, lazy evaluation, and pattern matching. It also has an advanced type system supporting algebraic data types, covariance and contravariance, higher-order types (but not higher-rank types), and anonymous types. Other features of Scala not present in Java include operator overloading, optional parameters, named parameters, raw strings, and no checked exceptions. (Source: <https://bit.ly/2c0QjjA>)

In this ebook, we provide a framework and toolset so that you can develop modern Scala applications. We cover a wide range of topics, from SBT build and reactive applications, to testing and database access. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

## About the Author

Andriy completed his Master Degree in Computer Science at Zhitomir Institute of Engineering and Technologies, Ukraine. For the last fifteen years he has been working as the Consultant/Software Developer/Senior Software Developer/Team Lead for a many successful projects including several huge software systems for customers from North America and Europe.

Through his career Andriy has gained a great experience in enterprise architecture, web development (ASP.NET, Java Server Faces, Play Framework), software development practices (test-driven development, continuous integration) and software platforms (Sun JEE, Microsoft .NET), object-oriented analysis and design, development of the rich user interfaces (MFC, Swing, Windows Forms/WPF), relational database management systems (MySQL, SQL Server, PostgreSQL, Oracle), NoSQL solutions (MongoDB, Redis) and operating systems (Linux/Windows).

Andriy has a great experience in development of distributed (multi-tier) software systems, multi-threaded applications, desktop applications, service-oriented architecture and rich Internet applications. Since 2006 he is actively working primarily with JEE / JSE platforms.

As a professional he is always open to continuous learning and self-improvement to be more productive in the job he is really passionate about.



# Chapter 1

## Build with SBT

### 1.1 Introduction

For many experienced Java developers, Scala programming language is not a stranger. It's been around for quite a while now (officially, since first public release in 2004) and gained quite a lot of traction in the recent years.

There are many reasons why one is going to pick Scala over Java, Clojure, Groovy, Kotlin, Ceylon, ... and we are not going to discuss that in this tutorial. However, what we are going to talk about is the ecosystem of tools, frameworks and libraries which Scala community has developed over the years to provide a native Scala experience for the developers.

We are going to start from the basics, like build tools and testing frameworks, talking about principles of reactive applications, accessing data storages, concurrency and parallelism, and finish up by outlining the typical choices in building console applications, web services and full-fledged web applications.

### 1.2 Build Tools: Solid Foundation of Every Project

One of the first things every software project is facing at a quite early stage is how it is going to be built and manage its dependencies. The choice of tools ranges from custom made shell scripts, [make](#), XML-based [Apache Ant](#), [Apache Maven](#), [Apache Ivy](#), to really sophisticated ones like for example [Gradle](#) and [Apache Buildr](#).

### 1.3 SBT: (not so) Simple Build Tool

It comes as no surprise that Scala community has own view on the projects build and dependency management and [SBT](#), or simple build tool, is the direct confirmation of that. [SBT](#) differs from any other build tool out there, first of all because it uses Scala language for describing the build definitions.

In its current version **0.13.11**, [SBT](#) is still on its undergoing journey to **1.0 release** (which is going to happen soon, hopefully). The simplest way to install [SBT](#) is to [download it as a platform-independent archive](#), extract it into some location and include this location into your operating system **PATH** environment variable.

Please notice that appropriate Java version (at least Java 7 but Java 8 is the recommended distribution these days) should be installed before.

Although **S** in [SBT](#) stays for “**simple**”, for many developers it does not sound like it is. May be because of the way how project builds are being described (essentially, key/value pairs), or may be because there are multiple ways of creating build definitions (using **.sbt** and **.scala** files). Nonetheless, in this section we are going to demystify some [SBT](#) concepts, hopefully making this powerful tool really simple to understand and use.

To be noted here, [SBT](#) is really very capable and feature-rich tool, worth of a complete tutorial to be written. We are going to talk about some basic concepts so to let us get started quickly however do not hesitate to look at the [comprehensive documentation available online](#) any time.

---

## 1.4 Project Structure

Most of the build tools try to impose some conventions regarding how the project layout should look like. **SBT** does not introduce yet another one but instead follows the same project layout conventions as **Apache Maven**, for example:

```
<project-name>
| - *project*;
| - src
| - main
|   | - scala
|   | - resources
| - test
|   | - scala
|   | - resources
```

The only stranger in this layout is the **project** subfolder. It is specific to **SBT** and in many cases contains only two files:

- **properties** with desired **SBT** version, for example: `sbt.version=0.13.11`
- **sbt** with the list of additional **SBT** plugins, for example: `addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")`

Although **plugins.sbt** is still widely used, more recent **SBT** versions are lean towards using a little bit different approach as described in the **Using Plugins** section of the **SBT** documentation.

## 1.5 Build Definitions

As we already briefly touched upon, **SBT** support multiple ways of creating build definitions: using **.sbt** files and **.scala** files. Furthermore, with regards to **.sbt** files, there are so called **bare .sbt** files which should not be used anymore but we are going to discuss them a bit as we may encounter them in the wild quite often.

Essentially, no matter what approach you are going to use, at the end the result is always the same: provide the description of each project as a set of key / value pairs, which are often called **settings**. Along with **settings**, **SBT** has very important notions of **tasks** (for example **compile**, **package**, **run**) and **configurations** (for example **Compile**, **Runtime**, **Test**).

With that being said, depending on task and/or configuration, every setting may have different value so **settings** (or more precisely, **setting keys**) are said to be associated with some scope (or just scoped). Under the hood, it is certainly more complex than it sounds but thinking about the build definition in this way is a good starting point. For the curious ones, **.sbt%20build%20definition[sbt build definition]** and **Scopes** sections of **SBT** online documentation go much deeper into details.

In the next couple of sections we are going to look at the different **SBT** build definitions using just simplest single-module projects as well as more complex multi-module projects. Both build definition styles, using **.sbt** or **.scala** files, are going to be discussed so we could pick the one closer to your heart.

## 1.6 Single-Module Projects

Projects which consist of a single module are extremely easy to build using just a few lines in **SBT** build definition **build.sbt** file, for example:

```
lazy val main = (project in file("."))
  .settings(
    name := "single-module-sbt",
    version := "0.0.1-SNAPSHOT",
    scalaVersion := "2.11.8",
    libraryDependencies += Seq(
      "ch.qos.logback" % "logback-classic" % "1.1.2"
```

```

),
  resolvers += "Typesafe" at "https://repo.typesafe.com/typesafe/releases/",
  mainClass in (Compile, run) := Some("com.javacodegeeks.single.App")
)

```

That basically is the complete build definition! In the nutshell, we just referred to the current directory as a project:

```
lazy val main = (project in file("."))
```

And defined a couple named pairs (key / value pairs or settings), like **name**, **version**, **scalaVersion**, **libraryDependencies**, **resolvers**. In this regards **mainClass** stands separately as it additionally includes configuration (**Compile**) and task (**run**) scopes.

As of now, this is the recommended way of using **SBT** with Scala projects (and projects written in other languages which **SBT** supports). Using **.scala** and **bare.sbt** styles are considered deprecated in favor of this one. Nonetheless, let us take a look on the examples of both, starting from **.scala**, as there are chances one or more of your current projects may already use them.

The traditional way the **.scala** based builds are being organized is to have **Build.scala** file in the **project** folder as an entry point. For example, the same build definition is going to look a little bit differently but still close enough:

```

import sbt._
import Keys._

object ProjectBuild extends Build {
  override val settings = super.settings ++ Seq(
    name := "single-module-sbt",
    version := "0.0.1-SNAPSHOT",
    scalaVersion := "2.11.8",
    libraryDependencies += Seq(
      "ch.qos.logback" % "logback-classic" % "1.1.2"
    ),
    resolvers += "Typesafe" at "https://repo.typesafe.com/typesafe/releases/"
  )

  lazy val main = Project(
    id = "single-module-scala",
    base = file("."),
    settings = Project.defaultSettings ++ Seq(
      mainClass in (Compile, run) := Some("com.javacodegeeks.single.App")
    )
  )
}

```

Comparing to the previous **.sbt** version, this build definition looks considerably more verbose, with quite a bit of code being written. Surely, it is still readable but the benefits of the recommend approach become obvious. To read more about **.scala** build definitions please take a look at the [official documentation](#).

To finish up, let us take a look at one of the simplest build definition possible, using so called these days just **bare.sbt** file.

```

name := "bare-module-sbt"

version := "0.0.1-SNAPSHOT"

scalaVersion := "2.11.8"

libraryDependencies += Seq(
  "ch.qos.logback" % "logback-classic" % "1.1.2"
)

resolvers += "Typesafe" at "https://repo.typesafe.com/typesafe/releases/"

mainClass in (Compile, run) := Some("com.javacodegeeks.bare.App")

```

It is still the same **build.sbt**, but defined using pure settings based approach (please notice, the new lines between each setting definition are mandatory). This build definition style is explained in the [official documentation](#) as well.

As it was already mentioned, **.scala** and **bare .sbt** build definitions are not recommended for use anymore so we are not going to get back to them in the tutorial.

## 1.7 Multi-Module Projects

Most real-world projects are organized in a set of modules to encourage code reuse and proper boundaries. Logically, in this case many build settings, tasks and properties could be shared between most (or even all) modules in the project and the choice of the right build tool becomes increasingly important.

**SBT** support multi-module project very naturally, making it no brainer to add new modules or maintain quite complex build definitions. Let us come up with a sample multi-module project which consists of 3 modules: **core-module**, **api-module** and **impl-module**.

```
<project-name>
| - *project*
| - *core-module*
|   | - src
|       | - main
|       ...
| - *api-module*
|   | - src
|       | - main
|       ...
| - *impl-module*
|   | - src
|       | - main
|       ...
```

There are quite a few ways to organize your build but in our case the project's root folder is the only one which contains **project** folder and **build.sbt** file.

```
lazy val defaults = Seq(
  version := "0.0.1-SNAPSHOT",
  scalaVersion := "2.11.8",
  resolvers += "Typesafe" at "https://repo.typesafe.com/typesafe/releases/"
)

lazy val core = (project in file("core-module"))
  .settings(defaults: _*)
  .settings(
    name := "core-module"
  )

lazy val api = (project in file("api-module"))
  .settings(defaults: _*)
  .settings(
    name := "api-module"
  )

lazy val impl = (project in file("impl-module"))
  .settings(defaults: _*)
  .settings(
    name := "impl-module",
    libraryDependencies += Seq(
      "ch.qos.logback" % "logback-classic" % "1.1.2"
    ),
    mainClass in (Compile, run) := Some("com.javacodegeeks.multi.App")
  )
```

```
)  
.dependsOn(core)  
.dependsOn(api)  
  
lazy val main = (project in file("."))  
.aggregate(core, api, impl)
```

Essentially, multi-module (or multi-project) build definition is just a set of individual projects, sharing some common settings (for example **lazy val defaults**) and declaring dependencies on each other using **dependsOn**:

```
lazy val impl = (project in file("impl-module"))  
...  
.dependsOn(core)  
.dependsOn(api)
```

**SBT** is taking care of everything else (like constructing dependency graph and figuring out the proper build sequence). For the curious ones, it is quite well explained in the [official documentation](#) section.

## 1.8 Migrating from Maven

Arguably, these days **Apache Maven** is still one of the most popular build tools in Java ecosystem. Likely, **SBT** provides an easy migration path from **Apache Maven** using **externalPom** configuration (assuming there is a **pom.xml** file in the along with **build.sbt**), for example:

```
lazy val main = (project in file("."))  
.settings(  
  organization := "com.javacodegeeks",  
  name := "pom-module-sbt",  
  version := "0.0.1-SNAPSHOT",  
  scalaVersion := "2.11.8",  
  mainClass in (Compile, run) := Some("com.javacodegeeks.maven.App"),  
  externalPom()  
)
```

It is important to mention that **externalPom** has limited capabilities and imports dependencies only. For more in depth details please take a look at the [official documentation](#).

## 1.9 Interactivity at Heart

One of the absolutely astonishing capabilities of **SBT** is interactivity though feature-rich shell. It is as simple as typing **sbt** from the root folder of your project.

```
$ sbt  
>
```

**SBT** let you to interact with your build definitions live, inspecting various settings and invoking arbitrary tasks. Not to forget the integration with Scala console through **sbt console** command (or just console with interactive shell).

```
$ sbt console  
  
[info] Starting scala interpreter...  
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_77).  
Type in expressions for evaluation. Or try :help.  
  
scala>
```

In the next sections of the tutorial we are going to use these **SBT** superpowers a lot, hopefully changing your mind about the whole concept of what build tool should be.

## 1.10 Integrations with IDEs

Last but not least, there is another very important subject to cover of how **SBT** integrates with popular Java IDEs. It is surprising how important such integrations are for developers in order to quickly get started with their projects.

**SBT** is very extensible due to its own plugins mechanism. More often than not the integrations with most popular Java IDEs are done through dedicated plugins, for example:

- **sbteclipse** is a plugin for **SBT** to create **Eclipse** project definitions
- **nbsbt** is a plugin for **SBT** to create **Netbeans** project definition
- **ensime-sbt** is a plugin for **SBT** to create **Ensime** project definition

Adding plugins to build definition is as simple as adding a couple of lines into **plugin.sbt** file in the **project** subfolder, for example in case of **sbteclipse**:

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

Consequently, the plugin functionality is accessible through the respective task (or tasks) from the command line (or from the interactive shell), for example for **sbteclipse** it looks like that:

```
$ sbt "eclipse with-source=true with-javadoc=true"
```

It is worth to mention that some IDEs like **JetBrains IntelliJ IDEA** do provide superior **SBT** through own plugins ecosystem.

## 1.11 Conclusions

**SBT** is very powerful build tool which, when used accordingly, can significantly boost your productivity. This section talked about very basic concepts but mostly every other part of the tutorial is going to refer to some new **SBT** features (or just rely on the ones we already know).

## 1.12 What's next

In the next section of the tutorial we are going to talk about different testing frameworks adopted by Scala development community. We are also going to see in action how **SBT** could be used to run individual tests or test suites and watch for changes.

## Chapter 2

# Testing

### 2.1 Introduction

In this section of the tutorial we are going to talk about testing frameworks which are widely adopted by majority of the **Scala** application developers. Although the hot debates around effectiveness and usefulness of the **test-driven development** (or just **TDD**) practices are going on for years, this section is based on a true belief that tests are great thing which makes us better software developers and improves the quality and maintainability of the software systems we have worked or are working on.

Why to talk about testing so early? Well, most of the future sections will introduce us to different frameworks and libraries, and as a general rule we are going to spend some time discussing the testing strategies applicable in their contexts. Plus, it is much easier and safer way to show off a new language or framework capabilities to your team (or organization) when it does not affect production code anyhow.

### 2.2 ScalaCheck: the Power of Property-Based Testing

The first framework we are going to touch upon is **ScalaCheck** which serves the purpose of automated property-based testing. In case you are not familiar with property-based testing, it is quite intuitive and powerful technique: it basically verifies (or better to say, checks) that assertions about the output results of your code hold true for a number of auto-generated inputs.

Let us start from very simple example of `Customer` case class so the idea and benefits of property-based testing becomes apparent.

```
case class Customer(firstName: String, lastName: String) {  
  def fullName = firstName + " " + lastName  
}
```

Basically, the traditional way of ensuring that **full name** is assembled from **first and last names** would be to write a test case (or multiple test cases) by explicitly providing first and last name values. With **ScalaCheck** (and property-based testing) it is looking much simpler:

```
object CustomerSpecification extends Properties("Customer") {  
  property("fullName") = forAll { (first: String, last: String) =>  
    val customer = Customer(first, last)  
    customer.fullName.startsWith(first) &&  
    customer.fullName.endsWith(last)  
  }  
}
```

The inputs for **first and last names** are automatically generated, while the only thing we need to do is to construct the `Customer` class instance and define our checks which must be true for all possible inputs:

---

```
customer.fullName.startsWith(first) &&  
  customer.fullName.endsWith(last)
```

Looks really simple, but we can do even better than that. **ScalaCheck** by default provides the generators for all primitive types, giving the option to define your own. Let us go one step further and define the generator for `Customer` class instances.

```
implicit val customerGen: Arbitrary[Customer] = Arbitrary {  
  for {  
    first <- Gen.alphaStr  
    last  <- Gen.alphaStr  
  } yield Customer(first, last)  
}
```

With this generator in place, our test case becomes even more trivial and readable:

```
property("fullName") = forAll { customer: Customer =>  
  customer.fullName.startsWith(customer.firstName) &&  
    customer.fullName.endsWith(customer.lastName)  
}
```

Very elegant, isn't it? Surely, **ScalaCheck** has much more to offer in terms of supporting flexible checks (and verification failure reporting) for quite complex test scenarios as we are going to see in the next sections of this tutorial. Nonetheless, the **official documentation** has a number of good references in case you would like to learn more about **ScalaCheck**.

Having test cases is good, but how to run them? There are multiple ways to run **ScalaCheck** tests but the most convenient one is to use **SBT**, the tool we have learned about in the first section of this tutorial, using its **test** task.

```
$ sbt test  
...  
[info] + Customer.fullName: OK, passed 100 tests.  
[info] + CustomerGen.fullName: OK, passed 100 tests.  
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2  
  
[success] Total time: 1 s
```

As the **SBT** output shows, **ScalaCheck** generated 100 different tests out of our single test case definition, saving quite a lot of our time.

## 2.3 ScalaTest: Tests as Specifications

**ScalaTest** is a great example of **Scala**-based variant of the full-fledged testing framework which could be found in many other languages (like **Spock Framework** or **RSpec** just to name a few). In the core of the **ScalaTest** are test specifications which support different styles of writing tests. In fact, this variance of testing styles is extremely useful feature as it lets the developers coming to **Scala** universe from other languages to follow the style they might be already familiar and comfortable with.

Before rolling up the sleeves and exploring **ScalaTest**, it is worth to mention that the current stable release branch is **2.2** but the next version **3.0** is about to be released (hopefully) very soon, being in **RC3** stage right now. As such, the latest **3.0-RC3** is going to be the **ScalaTest** version we will build our test cases upon.

Let us get back to the `Customer` case class and show off different flavors of **ScalaTest** specifications, starting from the basic one, **FlatSpec**:

```
class CustomerFlatSpec extends FlatSpec {  
  "A Customer" should "have fullName set" in {  
    assert(Customer("John", "Smith").fullName == "John Smith")  
  }  
}
```



Although the usage of assertions is known and understandable by most of us, there are better, more human-friendly ways to express your expectations. In **ScalaTest** there are matchers which serve this purpose. Let us take a look on another version of the test, still using the **FlatSpec**:

```
class CustomerMatchersFlatSpec extends FlatSpec with Matchers {
  "A Customer" should "have fullName set" in {
    Customer("John", "Smith").fullName should be ("John Smith")
  }
}
```

With just a minor change where the explicit **assert** has been replaced by convenient **should be** matcher, provided by **Matchers** trait, the test case could be now read fluently, as a story.

Aside from **FlatSpec**, **ScalaTest** includes many other favors in a form of **FunSuite**, **FunSpec**, **WordSpec**, **FreeSpec**, **Spec**, **PropSpec** and **FeatureSpec**. You are certainly encouraged to look on those and pick your favorite but the one we are going to discuss at last is **FeatureSpec**.

**Behavior-driven development** (or simply **BDD**) is another testing methodology which emerged from **TDD** and became quite popular in the recent years. In **BDD** the test scenarios (or better to say acceptance criteria) should be written for every feature being developed and should follow **Given / When / Then** structure. **ScalaTest** supports this kind of testing with **FeatureSpec** style. So let us take a look on **BDD** version of the test scenario we have implemented so far.

```
class CustomerFeatureSpec extends FeatureSpec with GivenWhenThen with Matchers {
  info("As a Customer")
  info("I should have my Full Name composed from first and last names")

  feature("Customer Full Name") {
    scenario("Customer has correct Full Name representation") {
      Given("A Customer with first and last name")
      val customer = Customer("John", "Smith")
      When("full name is queried")
      val fullName = customer.fullName
      Then("first and last names should be returned")
      fullName should be ("John Smith")
    }
  }
}
```

Certainly, for such a simple test scenario the **BDD** style may look like overkill, but it is actually very expressive way to write acceptance criteria which could be understood by business and engineering people at the same time.

With no doubts, **ScalaTest** has first-class **SBT** support and with all our test suites in place, let us run them to make sure they all pass:

```
$ sbt test
...
[info] CustomerFeatureSpec:
[info] As a Customer
[info] I should have my Full Name composed from first and last names
[info] Feature: Customer Full Name
[info]   Scenario: Customer has correct Full Name representation
[info]     Given A Customer with first and last name
[info]     When full name is queried
[info]     Then first and last names should be returned
[info] CustomerMatchersFlatSpec:
[info] A Customer
[info] - should have fullName set
[info] CustomerFlatSpec:
[info] A Customer
[info] - should have fullName set
[info] Run completed in 483 milliseconds.
[info] Total number of tests run: 3
[info] Suites: completed 3, aborted 0
```

```
[info] Tests: succeeded 3, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[success] Total time: 1 s
```

As we can see in **SBT** output, the **FeatureSpec** test suite formats and prints out feature and scenario descriptions, plus the details about every **Given / When / Then** step performed.

## 2.4 Specs2: One to Rule Them All

The last (but not least) **Scala** testing framework we are going to discuss in this section is **specs2**. From the features perspective, it is quite close to **ScalaTest** but **specs2** takes a slightly different approach. Essentially, it is also based on test specifications, but there are only two of them: mutable (or **unit specification**) and immutable (or **acceptance specification**).

Let us see what those two styles really mean in practice by developing same kind of test cases for our **Customer** case class.

```
class CustomerSpec extends Specification {
  "Customer" >> {
    "full name should be composed from first and last names" >> {
      Customer("John", "Smith").fullName must_== "John Smith"
    }
  }
}
```

It looks like quite simple test suite with familiar structure. Not much different from **ScalaTest** except it extends `org.specs2.mutable.Specification` class. It is worth to mention that **specs2** has extremely **rich set of matchers**, providing usually different variations of them depending on your preferred style. In our case, we are using `must_==` matcher.

Another way to create test suites with **specs2** is to extend `org.specs2.Specification` class and write down the specification as a pure text. For example:

```
class CustomerFeatureSpec extends Specification { def is = s2"""
  Customer
    should have full name composed from first and last names $fullName
  """

  val customer = Customer("John", "Smith")
  def fullName = customer.fullName must_== "John Smith"
}
```

This kind of specifications became possible since introduction of the **string interpolation capabilities** to **Scala** language and compiler. As you could guess at this point, such relaxed way of defining test specifications is a great opportunity for writing test cases following **BDD** style. Surely, **specs2** enables that but goes even a bit further by providing additional support in term of steps and scenarios. Let us take a look at the example of such test specification.

```
class CustomerGivenWhenThenSpec extends Specification
  with GWT with StandardDelimitedStepParsers { def is = s2"""
  As a Customer
  I should have my Full Name composed from first and last names  ${scenario.start}
  Given a customer last name {Smith}
  And first name {John}
  When I query for full name
  Then I should get: {John Smith}                                ${scenario.end}
  """

  val scenario =
    Scenario("Customer")
      .given(aString)
      .given(aString)
      .when() { case s :: first :: last :: _ =>
        Customer(first, last)
```

```

    }
    .andThen(aString) { case expected :: customer :: _ =>
      customer.fullName must be_== (expected)
    }
  }
}

```

Essentially, in this case the test specification consists of two parts: the actual scenario definition and its interpretation (sometimes called code-behind approach). The first and last names are extracted from the definition (using step parsers), **Customer** case class instance is created in background, and at the last step the expected full name is also extracted from the definition and compared with the customer's one. If you find it a bit cumbersome, this is **not the only way** to define **Given / When / Then** style specifications using **specs2**. But be advised than other alternatives would require you to track and maintain the state between different **Given / When / Then** steps.

And finishing up with **specs2** it would be incomplete not to mention its seamless integration with **ScalaCheck** framework just by extending `org.specs2.ScalaCheck` trait, for example:

```

class CustomerPropertiesSpec extends Specification with ScalaCheck { def is = s2"""
  Customer
    should have full name composed from first and last names ${fullName}
  """

  val fullName: Prop = forAll { (first: String, last: String) =>
    val customer = Customer(first, last)
    customer.fullName.startsWith(first) &&
    customer.fullName.endsWith(last)
  }
}

```

The result is exactly what you would expect from running a pure **ScalaCheck**: the checks are going to be run against generated first and last names. Using our friend **SBT**, let us run all the test suites to ensure we did a good job.

```

$ sbt test
...
[info] CustomerPropertiesSpec
[info]   + Customer should have full name composed from first and last names
[info]
[info] Total for specification CustomerPropertiesSpec
[info] Finished in 196 ms
[info] 1 example, 100 expectations, 0 failure, 0 error
[info]
[info] CustomerSpec
[info]
[info] Customer
[info]   + full name should be composed from first and last names
[info]
[info] Total for specification CustomerSpec
[info] Finished in 85 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] CustomerFeatureSpec
[info]   Customer
[info]     + should have full name composed from first and last names
[info]
[info] Total for specification CustomerFeatureSpec
[info] Finished in 85 ms
[info] 1 example, 0 failure, 0 error
[info]
[info] CustomerGivenWhenThenSpec
[info]   As a Customer
[info]     I should have my Full Name composed from first and last names
[info]

```

```
[info]      Given a customer last name Smith
[info]      And first name John
[info]      When I query for full name
[info]      + Then I should get: John Smith
[info]
[info] Total for specification CustomerGivenWhenThenSpec
[info] Finished in 28 ms
[info] 1 example, 4 expectations, 0 failure, 0 error
[info] Passed: Total 4, Failed 0, Errors 0, Passed 4
[success] Total time: 10 s
```

The output from **SBT** console is very similar to what we have seen for **ScalaTest**, with all test suites details printed out in compact and readable format.

## 2.5 Conclusions

**Scala** is extremely powerful language designated to solve a large variety of the software engineering problems. But as every other program written in a myriad of programming languages, **Scala** applications are not immutable to bugs. **TDD** (and recently **BDD**) practices are widely adopted by **Scala** developers community, which led to invention of such a great testing frameworks as **ScalaCheck**, **ScalaTest** and **specs2**.

## 2.6 What's next

In the next section of the tutorial we are going to talk about reactive applications, in particular focusing our attention on reactive streams as their solid foundation, briefly touching upon **The Reactive Manifesto** as well.

## Chapter 3

# Reactive Applications

### 3.1 Introduction

In the last couple of years many software systems, used by millions and even billions of people every day, have started to face unprecedented scalability requirements. In many regard the traditional software architectures were pushed to its limits, unveiling the urgent need to come up with other architectural styles which better suit the demands of the modern world.

This was the moment where another paradigm, **reactive programming**, has started to emerge and widespread very fast. Along with **functional programming**, **reactive programming** has shaken the industry and uncovered a whole new class of applications, which we call **reactive applications** these days.

### 3.2 Being Reactive

Building applications in accordance to **reactive programming** paradigm implies to follow a different architectural style and design principles, which are best described in **The Reactive Manifesto**, published by **Jonas Boner** around 2013.

- **Responsive:** the system responds in a timely manner if at all possible.
- **Resilient:** the system stays responsive in the face of failure.
- **Elastic:** the system stays responsive under varying workload.
- **Message Driven:** reactive systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

Needless to say that each and every of those principles makes a lot of sense and all together they represent a perfect recipe for building modern software applications. But certainly, there is no silver bullet in there. There is no magic which will suddenly make any application or system reactive. It is a combination of asynchronous and non-blocking programming, message-passing concurrency complemented by immutability and back pressure, to name a few key ones.

Along this tutorial we are going to learn about all the necessary building blocks for developing truly reactive applications using **Scala** programming language and ecosystem, focusing in this section on the first one of them, **reactive streams**.

### 3.3 Reactive Streams Specification

The purpose of **reactive streams** is to serve as a solid foundation for asynchronous stream processing with non-blocking back pressure. In this regards, **reactive streams specification** stands out as an ongoing effort to provide an interoperable standard which conforming implementations are going to follow.

The **reactive streams specification** is written strictly following the principles outlined by **The Reactive Manifesto** and the first official version **1.0.0** for JVM platform **has been already released**.

---

## 3.4 Reactive Streams in the Wild

Once the official **reactive streams** API for JVM platform went public, a number of very popular open source projects announced the immediate availability of compliant implementations. Although the **complete list** includes a dozen of those, here are just a few best-known ones:

- **Project Reactor**, member of **Spring** projects portfolio
- **RxJava**, an implementation of more broad **Reactive Extensions** library
- **Akka Streams**, the part of the **Akka Toolkit** distribution

It is worth to mention that **RxJava** is one of the first and most advanced JVM libraries which introduced the powerful principles of the **reactive programming** paradigm to Java developers. Although it also has a port to **Scala** language, called **RxScala**, we are going to focus on **Akka Streams**, pure **Scala**-based implementation of the **reactive streams** specification.

## 3.5 Akka Streams: Reactive Streams Implementation

As we already briefly mentioned **Akka Streams** is the just a part of more comprehensive **Akka Toolkit** distribution. The latest released version of **Akka Toolkit** at the moment of writing is **2.4.8** and this is the one we are going to use in the rest of the section.

However, do not panic if your version of **Akka Toolkit** is not the latest one, the implementation of **reactive streams** specification is provided by **Akka Streams** since quite a long time.

### 3.5.1 Basic Concepts

**Akka Streams** is built on top of a just a few basic pieces, which interoperate with each other and allow to describe very complex stream processing pipelines.

- **Source**: something with exactly one output stream (conceptually, represents a **Publisher**)
- **Sink**: something with exactly one input stream (conceptually, represents a **Subscriber**)
- **Flow**: something with exactly one input and one output stream (conceptually, represents a **Processor**)
- **BidiFlow**: something with exactly two input streams and two output streams
- **Graph**: a stream processing topology that accepts certain inputs and exposes certain outputs

For the curious readers, **Akka Streams** fully implements **reactive streams** specification but hides this fact behind more concise user-facing API abstractions, introducing own basic primitives. That is why if you look at the **reactive streams** API for JVM, you may not find the straightforward matches to **Akka Streams** ones, although the respective transformations are supported.

One of the key design goals of the **Akka Streams** is reusability. All the building blocks described above could be shared or/and composed into more complex stream processing topologies as we are going to see pretty soon.

### 3.5.2 Materialization

**Akka Streams** makes a very clear separation between stream definition and actual stream execution. The mechanism of taking an arbitrary stream definition and providing all the necessary resources it needs to run is called materialization in **Akka Streams** terminology.

Essentially, the materializer implementation could be anything but by default **Akka Streams** provides **ActorMaterializer** which basically maps different processing stages using **Actors**. It also implies that in general stream processing is going to be fully asynchronous and message-driven.

In the upcoming section of the tutorial, “**Concurrency and parallelism: Akka**”, we are going to talk about **Actors** and **Actor Model** in a great details. Luckily, **Akka Streams** does a really great job by hiding from us the unnecessary abstractions so the actors will not pop up anywhere else beside **ActorMaterializer**.

---

### 3.5.3 Sources and Sinks

The input data is the starting point of any stream processing. It could be anything: file, collection, network socket, stream, future, you name it. In the **Akka Streams** API, this input is represented by parameterized **Source[+Out, +Mat]** class, where:

- **Out** is the type of the elements which source outputs
- **Mat** is the type of the some additional value which source may produce (often set to **NotUsed** but more about that later)

For convenience, **Source** object has a lot of factory methods which simplify the wrapping of the typical inputs into respective **Source** class instances, for example:

```
val source: Source[Int, NotUsed] = Source(1 to 10)

val source: Source[Int, NotUsed] = Source(Set(1, 2, 3, 4, 5))

val source: Source[String, NotUsed] = Source.single("Reactive Streams")

val source: Source[ByteString, _] = FileIO.fromPath(file)

val source: Source[Int, _] = Source.tick(1 second, 10 seconds, Random.nextInt())
```

Having the input is already enough to start simple stream processing. But as we already know, defining **Source** will not actually do anything till the materialization moment. The **Source** class (and some others like **Flow** f.e.) has a family of so called terminal functions: `run()` and `runWith()`. The call to any of this functions triggers materialization process, requiring the materializer to be provided implicitly or explicitly. For example:

```
implicit val system = ActorSystem("reactive-streams")
implicit val materializer = ActorMaterializer()

val numbers = List(100, 200, 300, 400, 500)
val source: Source[Int, NotUsed] = Source(numbers)

source
  .runForeach { println _ }
  .onComplete { _ => system.terminate() }
```

Once the execution of the stream processing terminates, each number is going to be printed out on a console:

```
100
200
300
400
500
```

Interestingly, in the code snippet above the call to `runForeach` under the hood is using another **Akka Streams** abstraction, **Sink**, which essentially is a consumer of the stream input at different stages. So our example could be rewritten like that:

```
source
  .runWith { Sink.foreach { println _ } }
  .onComplete { _ => system.terminate() }
```

And as you would certainly expect, **Source** class supports a wide range of functions to transform or process the stream elements, which are called processing stages in **Akka Streams**, for example:

```
source
  .map { _ * 2 }
  .runForeach { println _ }
  .onComplete { _ => system.terminate() }
```

Please notice that the processing stages never modify the current stream definition but rather return a new processing stage. And last, but not least: **Akka Streams** do not allow `null` to flow through the stream as an element, something the people coming from Java background should take extreme care of.

### 3.5.4 Flows

If **Source** and **Sink** are just abstractions over outputs and inputs, **Flow** is this kind of glue which essentially hooks them up together. Let us get back to the example with the numbers but this time we are going to read them from the file.

```
val file: Path = Paths.get(getClass.getResource("/numbers.txt").toURI())
val source: Source[ByteString, _] = FileIO.fromPath(file)
```

The **numbers.txt** is just a plain old text file, where every line contains some arbitrary number, for example:

```
100
200
300
400
500
```

It might sound trivial but let us take a look on the **Out** type of the **Source**: it is actually **ByteString** (more precisely, the file is going to be read in **ByteString** chunks). This is not really what we want, we would like to read file line by line, number by number, but how we can do that?

Luckily, **Akka Streams** has out of the box support for that in form of **Framing** and we only need to define the transformation from the stream of **ByteString** to stream of regular integer numbers:

```
val flow: Flow[ByteString, Int, _] = Flow[ByteString]
  .via(Framing.delimiter(ByteString("\r\n"), 10, true))
  .map { _.utf8String.toInt }
```

Here, we have just defined a **Flow**! It is not attached nor to input, nor to output. However, it could be reused by any stream processing definition, for example:

```
source
  .via(flow)
  .filter { _ > 200 }
  .runForeach { println _ }
```

Or we can define yet another stream processing pipeline just to count how many lines we have been processed overall. This is where the **Mat** type is going to be handy as we are going to use the value from one of the **Sinks** as the final result, for example:

```
val future: Future[Int] = source
  .via(flow)
  .toMat(Sink.fold[Int, Int](0) { (acc, _) => acc + 1 }) (Keep.right)
  .run

future.onSuccess { case count =>
  println(s"Lines processed: $count")
}
```

Pretty simple, isn't it? It is worth to make one important note regarding elements ordering guarantees: **Akka Streams** preserves the input order of elements in most cases (but some processing stages may not do that).

### 3.5.5 Graphs and BidiFlows

While **Flow** is really powerful abstraction, it is limited to only one input and only one output. It could be enough for many use cases but complex stream processing scenarios require more flexibility. Let us get introduced to **Graphs** and **BidiFlows**.

**Graph** may use arbitrary number of inputs and outputs and form really complex topologies but in the nutshell, they are composed out of simple **Flows**. To illustrate **Graphs** in action, let us consider this example. The organization is ready to pay its employees annual bonus but all management positions gets 10k bonus to a base salary, while others will get only 5k. The complete stream processing could be illustrated by the image below.





Figure 3.1: Graph

With expressive **Graph DSL**, **Akka Streams** makes it very simple to build really complex scenarios, although ours is pretty naive.

```

val employees = List(
  Employee("Tom", "manager", 50000),
  Employee("Bob", "employee", 20000),
  Employee("Mark", "employee", 20000),
  Employee("John", "manager", 55000),
  Employee("Dilan", "employee", 35000)
)

val graph = GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  val in = Source(employees)
  val out = Sink.foreach { println _ }

  val broadcast = builder.add(Broadcast[Employee](2))
  val merge = builder.add(Merge[Employee](2))

  val manager = Flow[Employee]
    .filter { _.position == "manager" }
    .map { e => e.copy(salary = e.salary + 10000) }

  val employee = Flow[Employee]
    .filter { _.position != "manager" }
    .map { e => e.copy(salary = e.salary + 5000) }

  in ~> broadcast ~> manager ~> merge ~> out
    broadcast ~> employee ~> merge

  ClosedShape
}

```

The **ClosedShape** at the end of the **Graph** means that we have defined a fully connected graph, where all inputs and outputs are plugged in. Fully connected graph could be converted to **RunnableGraph** and actually executed, for example:

```

RunnableGraph
  .fromGraph(graph)
  .run

```

We will see the expected output in the console, once the graph execution completes. Everyone has a respective bonus added to his/her salary:

```

Employee(Tom,manager,60000)
Employee(Bob,employee,25000)
Employee(Mark,employee,25000)
Employee(John,manager,65000)
Employee(Dilan,employee,40000)

```

As most of other basic pieces, **Graph** and **RunnableGraph** are freely shareable. One of the consequences of that is the ability to construct partial graphs and combine different **Sources**, **Sinks**, and **Flows** together.

In the examples we have seen so far the data flows through the stream in one direction only. **BidiFlow** is a special case of the graph where there are two flows which go in opposite directions. The best illustration of **BidiFlow** is a typical request / response communication, for example:

```
case class Request(payload: ByteString)
case class Response(payload: ByteString)

val server = GraphDSL.create() { implicit builder =>
  import GraphDSL.Implicits._

  val out = builder.add(Flow[Request].map { _.payload.utf8String })
  val in = builder.add(Flow[String].map { s => Response(ByteString(s.reverse)) })

  BidiShape.fromFlows(out, in)
}
```

In this simplistic example the request's payload is just reversed and wrapped up as response payload. The server is actually a graph and to create a **BidiFlow** instance we have to use `fromGraph` factory method, like this:

```
val bidiFlow = BidiFlow.fromGraph(server)
```

We are now ready to materialize and run the **BidiFlow** instance by supplying a simple request and wiring the both `bidiFlow` flows together directly.

```
Source
  .single(Request(ByteString("BidiFlow Example")))
  .via(bidiFlow.join(Flow[String]))
  .map(_.payload.utf8String)
  .runWith(Sink.foreach { println _ })
  .onComplete { _ => system.terminate() }
```

Not surprising but the reversed version of the **"BidiFlow Example"** string is going to be printed out in the console:

```
elpmaxE wolFidiB
```

### 3.5.6 Back-Pressure

The concept of back-pressure is one of the foundational in the **reactive streams** philosophy. **Akka Streams** tries very hard to keep stream processing healthy by guaranteeing that publisher will never emit more elements than subscriber is able to handle. In addition to controlling the demand and propagating back-pressure from downstream flows to upstream ones, **Akka Streams** supports usage of buffering and throttling for more fine-grained and advanced back-pressure management.

### 3.5.7 Handling Errors

With any more or less real-life scenario, the stream processing pipelines built with **Akka Streams** would refer to application specific logic, including database access or external service calls. Errors may and will happen, resulting in premature stream processing termination.

Luckily, **Akka Streams** provides at least three strategies to handle exceptions raised as part of the application code execution:

- **Stop:** the stream is completed with failure (the default strategy)
- **Resume:** the element is dropped and the stream continues
- **Restart:** the element is dropped and the stream continues after restarting the stage

Those are heavily influenced by the [Actor Model](#) which [Akka Streams](#) uses to materialize the stream processing flows and also are referred to as supervision strategies. To illustrate how it works, let us consider a simple stream which uses numbers as a source and raises an exception every time it encounters an even number.

```
val source = Source
  .unfold(0) { e => Some(e + 1, e + 1) }
  .map { e => if (e % 2 != 0) e else throw new IllegalArgumentException("Only odd numbers ←
    are allowed") }
  .withAttributes(ActorAttributes.supervisionStrategy(_ => Supervision.Resume))
  .take(10)
  .runForeach { println _ }
  .onComplete { _ => system.terminate() }
```

Without supervision, the stream will end once the number 2 is emitted. But with resuming strategy, the stream continues an execution from where it left, skipping the problematic element. As expected, we should see only odd numbers in the console:

```
1
3
5
7
9
11
13
15
17
19
```

### 3.5.8 Testing

As we are going to see, most of the frameworks and libraries developed by [Flows](#) section.

```
val flow: Flow[ByteString, Int, _] = Flow[ByteString]
  .via(Framing.delimiter(ByteString("\r\n"), 10, true))
  .map { _.utf8String.toInt }
```

It sounds like a good idea to have a test case which verifies that framing works exactly as we expected so let us create one.

```
class FlowsSpec extends TestKit(ActorSystem("reactive-streams-test"))
  with SpecificationLike with FutureMatchers with AfterAll {

  implicit val materializer = ActorMaterializer()
  def afterAll = system.terminate()

  "Stream" >> {
    "should return an expected value" >> {
      val (publisher, subscriber) = TestSource.probe[ByteString]
        .via(flow)
        .toMat(TestSink.probe[Int]) (Keep.both)
        .run()

      subscriber.request(2)
      publisher.sendNext(ByteString("20"))
      publisher.sendNext(ByteString("0\r\n"))
      publisher.sendComplete()

      subscriber.expectNext() must be_==(200)
    }
  }
}
```

The classes like `TestSource` and `TestSink` (and many more) give a complete control over the stream processing so it is possible to test a very sophisticated pipelines. Also, `Akka Streams` does not mandate the testing framework to use, so the example we came up with is a typical `Specs2` specification.

## 3.6 Conclusions

This section was just a shallow introduction into the world of `reactive programming` and `reactive streams` in particular. Despite the fact that we have talked about `Akka Streams` quite a lot, we have only touched a tiny part of it, just scratching the tip of the iceberg: a long list of features and intrinsic details stayed uncovered. `Akka Streams official documentation` is a great source of the comprehensive knowledge about the subject, full of examples. Please do not hesitate to go through it.

## 3.7 What's next

In the next section of the tutorial we are going to talk about accessing relation databases from within your `Scala` applications. The topics of this section are going to be of great use as we will see the importance of the `reactive streams` in backing certain database access patterns.

## Chapter 4

# Database Access with Slick

We are certainly leaving at the epoch of flourishing data stores. Myriads of **NoSQL** and **NewSQL** solutions have emerged over the last couple of years, and even these days the new ones are popping up here and there from time to time.

Nonetheless, long-time players in the form of **relational database** are still being used in the vast majority of software systems. Bullet-proof and battle-tested, they are number one choice for storing critical to business data.

### 4.1 Introduction

When we talk about Java and JVM platform in general, **JDBC** is a standard way of interfacing with **relational databases**. As such, mostly every single **relational database** vendor provides a **JDBC** driver implementation so it becomes possible to connect to the database engine from the application code.

In many respects, **JDBC** is old-fashioned specification which hardly fits into modern **reactive programming** paradigms. Although there are some discussions to revamp the specification, in reality no active work is happening in this direction, at least publicly.

### 4.2 Database Access, the Functional Way

Inability to speed up the changes in specification does not mean nothing could be done. There many, many different frameworks and libraries available on JVM to access **relational databases**. Some of them aim to be as close to **SQL** as possible, others going further than that offering sort of “seamless” mapping of the **relational model** to programming language constructs (so called class of **ORM** or **object-relational mapping** solutions). Although to be fair, most of them are built on top of **JDBC** abstractions nonetheless.

In this regards, **Slick** (or in full, **Scala Language-Integrated Connection Kit**) is a library to provide access to **relational databases** from **Scala** application. It is heavily based on **functional programming** paradigm and as such is often being referred as **functional relational mapping** (or **FRM**) library. **Slick**’s ultimate promise is to reinvent the way to access **relational databases** by means of regular operations over collections, so familiar to every **Scala** developer, with strong emphasize on type safety.

Despite having **3.1.1 release** out not so long ago, it is relatively very young library, still on its way to reach a certain level of maturity. Many early adopters do remember how significant the difference between versions **2.x** and **3.x** was. Luckily, things are getting more stable and smoother, with the first milestone of upcoming **3.2** release being just a few weeks old.

**Slick** goes with the times and is fully asynchronous library. And, as we are going to see very soon, implements **reactive streams specification** as well.

### 4.3 Configuration

**Slick** fully supports **quite a number** of popular open-source and commercial **relational database** engines. To demonstrate that we are going to use at least two of them: **MySQL** for production deployment and **H2** for integration testing.

---

To give **Slick** the credits, it is very easy to get started with when the application is developed against single **relational database** engine. But it is a little bit tricky to configure and use **Slick** in **JDBC**-driver independent fashion, due to differences in the database's capabilities. As we are targeting at least two different engines, **MySQL** and **H2**, we are certainly going to take this route.

Typical way of configuring database connections in **Slick** is to have a dedicated named section in the `application.conf` file, for example:

```
db {
  driver = "slick.driver.MySQLDriver$"

  db {
    url = "jdbc:mysql://localhost:3306/test?user=root&password=password"
    driver = com.mysql.jdbc.Driver
    maxThreads = 5
  }
}
```

The configuration should look familiar to JVM developers working with **relational database** over **JDBC**. It is worth to mention that **Slick** supports database connection pooling out of the box using brilliant **HikariCP** library. The `maxThreads` setting hints **Slick** to configure connection pool of the maximum size of 5.

If you are curious why there are two driver settings in the configuration, here is the reason. The first driver setting identifies the **Slick**-specific **JDBC** profile (**Slick** driver), while the second one points out to **JDBC** driver implementation to use.

To take care of this configuration we are going to define a dedicated `DbConfiguration` trait, although the purpose of introducing this trait may not be so obvious for now:

```
trait DbConfiguration {
  lazy val config = DatabaseConfig.forConfig[JdbcProfile]("db")
}
```

## 4.4 Table Mappings

Arguably the first thing to start with in **relational databases** universe is data modeling. Essentially, it translates to creation of the database schema, tables, their relations and constraints. Luckily, **Slick** makes it extremely easy to do.

As an exercise, let us build a sample application to manage users and their addresses, represented by those two classes.

```
case class User(id: Option[Int], email: String,
  firstName: Option[String], lastName: Option[String])

case class Address(id: Option[Int], userId: Int,
  addressLine: String, city: String, postalCode: String)
```

In turn, our relation data model is going to be constituted from just two tables, `USERS` and `ADDRESSES`. Let us use **Slick** capabilities to shape that out in **Scala**.

```
trait UsersTable { this: Db =>
  import config.driver.api._

  private class Users(tag: Tag) extends Table[User](tag, "USERS") {
    // Columns
    def id = column[Int]("USER_ID", O.PrimaryKey, O.AutoInc)
    def email = column[String]("USER_EMAIL", O.Length(512))
    def firstName = column[Option[String]]("USER_FIRST_NAME", O.Length(64))
    def lastName = column[Option[String]]("USER_LAST_NAME", O.Length(64))

    // Indexes
    def emailIndex = index("USER_EMAIL_IDX", email, true)
```

```
// Select
def * = (id.?, email, firstName, lastName) <> (User.tupled, User.unapply)
}

val users = TableQuery[Users]
}
```

For the people familiar with **SQL** language, there is definitely a very close resemblance with `CREATE TABLE` statement. However, **Slick** also has a way to define seamless conversion between domain entity represented by **Scala** class (`User`) to table row (`Users`) and vice versa, using `*` projection (literally translates to `SELECT * FROM USERS`).

The one subtle detail we haven't touched upon yet is `Db` trait (referenced by `this: Db => construct`). Let us take a look on how it is defined:

```
trait Db {
  val config: DatabaseConfig[JdbcProfile]
  val db: JdbcProfile#Backend#Database = config.db
}
```

The `config` is the one from `DbConfiguration` while `db` is a new database instance. Later on in the `UsersTable` trait the respective types for the relevant **JDBC** profile are introduced into the scope using `import config.driver.api._` declaration.

The mapping for the `ADDRESSES` table looks very much the same, except the fact we need a foreign key reference to the `USERS` table.

```
trait AddressesTable extends UsersTable { this: Db =>
  import config.driver.api._

  private class Addresses(tag: Tag) extends Table[Address](tag, "ADDRESSES") {
    // Columns
    def id = column[Int]("ADDRESS_ID", O.PrimaryKey, O.AutoInc)
    def addressLine = column[String]("ADDRESS_LINE")
    def city = column[String]("CITY")
    def postalCode = column[String]("POSTAL_CODE")

    // ForeignKey
    def userId = column[Int]("USER_ID")
    def userFk = foreignKey("USER_FK", userId, users)
      (_.id, ForeignKeyAction.Restrict, ForeignKeyAction.Cascade)

    // Select
    def * = (id.?, userId, addressLine, city, postalCode) <>
      (Address.tupled, Address.unapply)
  }

  val addresses = TableQuery[Addresses]
}
```

The `users` and `addresses` members serve as a facade to perform any database access operations against respective tables.

## 4.5 Repositories

Although repositories are not specific to **Slick** per se, defining a dedicated layer to communicate with database engine is always a good design principle. There would be only two repositories in our application, `UsersRepository` and `AddressesRepository`.

```
class UsersRepository(val config: DatabaseConfig[JdbcProfile])
  extends Db with UsersTable {
```

```
import config.driver.api._
import scala.concurrent.ExecutionContext.Implicits.global

...
}

class AddressesRepository(val config: DatabaseConfig[JdbcProfile])
  extends Db with AddressesTable {

  import config.driver.api._
  import scala.concurrent.ExecutionContext.Implicits.global

  ...
}
```

All data manipulations we are going to show case later on are going to be part of one of those classes. Also, please notice the presence of `Db` trait in the inheritance chain.

## 4.6 Manipulating Schemas

Once the table mappings (or simplify database schema) are defined, **Slick** has a capability to project it to a sequence of **DDL** statements, for example:

```
def init() = db.run(DBIOAction.seq(users.schema.create))
def drop() = db.run(DBIOAction.seq(users.schema.drop))

def init() = db.run(DBIOAction.seq(addresses.schema.create))
def drop() = db.run(DBIOAction.seq(addresses.schema.drop))
```

## 4.7 Inserting

In the simplest scenarios adding a new row to the table is as easy as adding an element to `users` or `addresses` (instances of **TableQuery**), for example:

```
def insert(user: User) = db.run(users += user)
```

That works fine when primary keys are assigned from the application code. However, in case when primary keys are generated on database side (for example using **auto-increments**), like for `Users` and `Addresses` tables, we have to ask for these primary identifiers to be returned to us:

```
def insert(user: User) = db
  .run(users returning users.map(_._id) += user)
  .map(id => user.copy(id = Some(id)))
```

## 4.8 Querying

Querying is one of the **Slick** distinguishing features which really shines. As we already mentioned, **Slick** tries hard to allow using **Scala** collection semantics over database operations. However it works surprisingly well please note that you are not working with the standard **Scala** types but the lifted ones: the technique known as **lifted embedding**.

Let us take a look on this quick example on the one of the possible ways to retrieve user from the table by its primary key:

```
def find(id: Int) =
  db.run((for (user <- users if user.id === id) yield user).result.headOption)
```



Alternatively to `for` comprehension we could just use filtering operation, for example:

```
def find(id: Int) = db.run(users.filter(_.id === id).result.headOption)
```

The results (and generated **SQL** query by the way) are exactly the same. In case we need to fetch user and its address, we could use a couple of query options here as well, starting with a typical join:

```
def find(id: Int) = db.run(
  (for ((user, address) <- users join addresses if user.id === id)
    yield (user, address)).result.headOption)
```

Or, alternatively:

```
def find(id: Int) = db.run(
  (for {
    user <- users if user.id === id
    address <- addresses if address.userId === id
  } yield (user, address)).result.headOption)
```

**Slick** querying capabilities are really very powerful, expressive and readable. We have just looked at a couple of typical examples but please [glance through official documentation](#) to find much more.

## 4.9 Updating

Updates in **Slick** are represented as a combination of a query (which basically outlines what should be updated) and essentially the update itself. For example, let us introduce a method to update user's first and last names:

```
def update(id: Int, firstName: Option[String], lastName: Option[String]) = {
def update(id: Int, firstName: Option[String], lastName: Option[String]) = {
  val query = for (user <- users if user.id === id)
    yield (user.firstName, user.lastName)
  db.run(query.update(firstName, lastName)) map { _ > 0 }
}
```

## 4.10 Deleting

Similarly to updates, the delete operation is basically just a query to filter out the rows to be removed, for example:

```
def delete(id: Int) =
  db.run(users.filter(_.id === id).delete) map { _ > 0 }
```

## 4.11 Streaming

**Slick** offers the capability to stream results of the database query. Not only that, its streaming implementation fully supports [reactive streams specification](#) and could be used right away in conjunction with **Akka Streams**.

For example, let us stream the results from `users` table and collect them as a sequence using `Sink.fold` processing stage.

```
def stream(implicit materializer: Materializer) = Source
  .fromPublisher(db.stream(users.result.withStatementParameters(fetchSize=10)))
  .to(Sink.fold[Seq[User], User](Seq())(_ :+ _))
  .run()
```

Please be advised that **Slick**'s streaming feature is really very sensitive to **relational database** and **JDBC** driver you are using and may require more exploration and tuning. Definitely do some extensive testing to make sure the data is streamed properly.

## 4.12 SQL

In case there is a need to run a custom **SQL** queries, **Slick** has nothing against that and as always tries to make it as painless as possible, providing useful macros. Let say we would like to read user's first and last names directly using plain old **SELECT** statement.

```
def getNames(id: Int) = db.run(
  sql"select user_first_name, user_last_name from users where user_id = #$id"
  .as[(String, String)].headOption)
```

It is as easy as that. In case the shape of the **SQL** query is not known ahead of time, **Slick** provides the mechanisms to customize the result set extraction. In case you are interested, official documentation has very good section dedicated to **plain old SQL queries**.

## 4.13 Testing

There are multiple ways you can approach testing of the database access layer when using **Slick** library. The traditional one is by using in-memory database (like **H2** for example), which in our case translates into minor configuration change inside `application.conf`:

```
db {
  driver = "slick.driver.H2Driver$"

  db {
    url = "jdbc:h2:mem:test1;DB_CLOSE_DELAY=-1"
    driver=org.h2.Driver
    connectionPool = disabled
    keepAliveConnection = true
  }
}
```

Please notice that if in production configuration we turned database connection pooling on, the test one uses just single connection and pool is explicitly disabled. Everything else essentially stays the same. The only thing we have to take care of is creating and dropping the schema between test runs. Luckily, as we saw in section **Manipulating Schemas**, it is very easy to do with **Slick**.

```
class UsersRepositoryTest extends Specification with DbConfiguration
  with FutureMatchers with OptionMatchers with BeforeAfterEach {

  sequential

  val timeout = 500 milliseconds
  val users = new UsersRepository(config)

  def before = {
    Await.result(users.init(), timeout)
  }

  def after = {
    Await.result(users.drop(), timeout)
  }

  "User should be inserted successfully" >> { implicit ee: ExecutionEnv =>
    val user = User(None, "a@b.com", Some("Tom"), Some("Tommyknocker"))
    users.insert(user) must be_== (user.copy(id = Some(1))).awaitFor(timeout)
  }
}
```

Very basic **Specs2** test specification with single test step to verify that new user is properly inserted into database table.

In case for any reasons you are developing your own database driver for [Slick](#), there is a helpful [Slick TestKit](#) module available along with example driver implementation.

## 4.14 Conclusions

[Slick](#) is extremely expressive and powerful library to access [relational databases](#) from [Scala](#) applications. It is very flexible and in most cases offers multiple alternative ways of accomplishing things at the same time trying hard to maintain the balance between making developers highly productive and not hiding the fact that they deal with [relational model](#) and [SQL](#) under the hood.

Hopefully, we all are [Slick](#)-infected right now and are eager to try it out. [Official documentation](#) is a very good place to begin learning [Slick](#) and get started.

## 4.15 What's next

In the next section of the tutorial we are going to talk about developing command line (or simply console) [Scala](#) applications.

## Chapter 5

# Console Applications

### 5.1 Introduction

Needless to say that Web and mobile have penetrated very deeply into our lives, affecting a lot our day to day habits and expectations about things. As such, overwhelming majority of the applications being developed these days are either mobile apps, or web APIs or full-fledged web sites and portals.

Classic, old style, console-based applications have largely faded away. They are living their lives primarily on Linux / Unix operating systems, being at the core of their philosophies. However, console-based applications are extremely useful in solving a wide range of problems and by no means should be forgotten.

In this section of the tutorial we are going to talk about developing console (or to say it a bit differently, command line) applications using **Scala** programming language and ecosystem. The sample application we are about to start building will do only one simple thing: fetch the data from provided URL address. To make it a little bit more interesting, the application will require to provide timeout and, optionally, output file to store the content of the response.

### 5.2 UI-less and Command Line Oriented

Although there are some exceptions, console applications usually do not have any kind of graphical UI or pseudo-graphical interface.

Their input is either command line arguments passed to execute the application, or just a piped stream from another command or source. Their output is typically printed out in the console (that is why those applications are often called console apps) or, to be more precise, there could be multiple output streams like for example standard output (console) and error output.

One of the most useful capabilities of the console applications is pipelining: the output of one application could be piped as an input to another application. Such extremely power composition allows to express very complex processing pipelines with ease, for example:

```
ps -ef | grep bash | awk '{ print "PID="$2; }'
```

### 5.3 Driven by Arguments

Along this section we are going to get introduced into two powerful **Scala** frameworks and develop two versions of the sample console application we have outlined before. The first library, **scopt**, aims to help us a lot by taking care of parsing and interpreting command line arguments (and their combinations) so let us take a closer look at it.

As we already know all the requirements, let us start from restating what we would like to achieve in terms of command line input. Our application would require the first argument to be URL to fetch, the timeout is also required and should be specified using `-t` (or alternatively `--timeout`) argument, while output file is optional and could be provided using `-o` (or alternatively `--out`) argument. So the complete command line looks like that:

```
java -jar console-cli.jar <url> -t <timeout> [-o <file>]
```

Or like that, while using the verbose argument names (please notice that a combination of both is totally legitimate):

```
java -jar console-cli.jar <url> --timeout <timeout> [--out <file>]
```

Using terrific **scopt** library this task becomes rather trivial. First, let us introduce a simple configuration class which reflects command line arguments (and their semantics):

```
case class Configuration(url: String = "", output: Option[File] = None, timeout: Int = 0)
```

Now, the problem we are facing is how to get from the command line arguments to the instance of the configuration we need? With **scopt**, we start from creating the instance of **OptionParser** where all our command line options are described:

```
val parser = new OptionParser[Configuration]("java -jar console-cli.jar") {
  override def showUsageOnError = true

  arg[String]("")
    .required()
    .validate { url => Right(new URL(url)) }
    .action { (url, config) => config.copy(url = url) }
    .text("URL to fetch")

  opt[File]('o', "out")
    .optional()
    .valueName("")
    .action((file, config) => config.copy(output = Some(file)))
    .text("optionally, the file to store the output (printed in console by default)")

  opt[Int]('t', "timeout")
    .required()
    .valueName("")
    .validate { _ match {
      case t if t > 0 => Right(Unit)
      case _ => Left("timeout should be positive")
    } }
    .action((timeout, config) => config.copy(timeout = timeout))
    .text("timeout (in seconds) for waiting HTTP response")

  help("help").text("prints the usage")
}
```

Let us walk over this parser definition and match each code snippet to respective command line argument. The first entry, `arg[String]("<url>")`, describes `<url>` option, it has no name and comes as-is right after the application name. Please notice that it is required and should represent a valid URL address as per validation logic.

The second entry, `opt[File]('o', "out")`, is used for specifying the file to store the response to. It has short (o) and long (out) variations and is marked as optional (so it could be omitted). In similar fashion, `opt[Int]('t', "timeout")`, allows to specify the timeout and is required argument, moreover it must be greater than zero. And last but not least, special `help("help")` entry prints out the details about command line options and arguments.

Once we have parser definition, we can apply it to command line arguments using `parse` method of the **OptionParser** class.

```
parser.parse(args, Configuration()) match {
  case Some(config) => {
    val result = Await.result(Http(url(config.url) OK as.String),
      config.timeout seconds)

    config.output match {
      case Some(f) => new PrintWriter(f) {
```

```

        write(result)
        close
    }
    case None => println(result)
}

Http.shutdown()
}
case _ => /* Do nothing, just terminate the application */
}

```

The result of the parsing is either a valid instance of the `Configuration` class or application is going to terminate, outputting encountered errors in the console. For example, if we do not specify any arguments, here is what is going to be printed:

```

$ java -jar console-cli-assembly-0.0.1-SNAPSHOT.jar

Error: Missing option --timeout
Error: Missing argument
Usage: console-cli [options]

<url>                                URL to fetch
-o, --out <file>                     optionally, the file to store the output (printed on the  ↵
    console by default)
-t, --timeout <seconds>             timeout (in seconds) for waiting HTTP response
--help                               prints the usage

```

Out of curiosity you can try to run this command providing only some command line arguments, or passing the invalid values, `scopt` will figure this one out and complain. However, it is going to keep silence if everything is fine and let the application to fetch the URL and print out the response in the console, for example:

```

$ java -jar console-cli-assembly-0.0.1-SNAPSHOT.jar https://freegeoip.net/json/www.google. ↵
    com -t 1

{
  "ip": "216.58.219.196",
  "country_code": "US",
  "country_name": "United States",
  "region_code": "CA",
  "region_name": "California",
  "city": "Mountain View",
  "zip_code": "94043",
  "time_zone": "America/Los_Angeles",
  "latitude": 37.4192, "longitude": -122.0574,
  "metro_code": 807
}

```

Awesome, isn't it? Although we have played with only basic use cases, it is worth noting that `scopt` is capable of supporting quite sophisticated combinations of command line options and arguments while keeping the parser definitions readable and maintainable.

## 5.4 The Power of Interactivity

Another class of the console applications is the ones which offer an interactive, command-driven shell, which could range from somewhat trivial (like `ftp`) to quite sophisticated (like `sbt` or `Scala REPL`).

Surprisingly, all the necessary building blocks are already available as a part of the `sbt` tooling (which by itself offers very powerful interactive shell). `sbt` distribution provides the `foundation` as well as a dedicated `launcher` to run your applications from anywhere. Following the requirements we have set for ourselves, let us embody them as interactive console applications using `sbt` scaffolding.

In the core of **sbt**-based applications lays **xsbti.AppMain** interface which others (in our example, `ConsoleApp` class) should implement. Let us take a look at the typical implementation.

```
class ConsoleApp extends AppMain {
  def run(configuration: AppConfig) =
    MainLoop.runLogged(initialState(configuration))

  val logFile = File.createTempFile("console-interactive", "log")
  val console = ConsoleOut.systemOut

  def initialState(configuration: AppConfig): State = {
    ...
  }

  def globalLogging: GlobalLogging =
    GlobalLogging.initial(MainLogging.globalDefault(console), logFile, console)

  class Exit(val code: Int) extends xsbti.Exit
}
```

The most important function in the code snippet above is `initialState`. We left it blank for now, but don't worry, once we understand the basics, it will be full of code pretty quickly.

**State** is the container of all available information in **sbt**. Performing some action may require to introduce the modifications of the current **State**, producing a new **State** thereafter. One class of such actions in **sbt** is the **command** (although there are **more**).

It sounds like a good idea to have a dedicated command which fetches the URL and prints out the response so let us introduce it:

```
val FetchCommand = "fetch"
val FetchCommandHelp = s"""$FetchCommand

    Fetches the    and prints out the response
"""

val fetch = Command(FetchCommand, Help.more(FetchCommand, FetchCommandHelp)) {
  ...
}
```

Looks simple but we need to somehow supply URL to fetch. Luckily, commands in **sbt** may have own arguments but it is a responsibility of the command to tell what the shape of its arguments is by defining the instance of **Parser** class. From that, **sbt** takes care of feeding the input to the parser and either extracting the valid arguments or failing with an error.

In case of our `fetch` command, we need to provide a parser for URL (however, **sbt** significantly simplifies our task by defining `basicUri` parser in the **sbt.complete.DefaultParsers** object which we can reuse).

```
lazy val url = (token(Space) ~> token(basicUri, "")) <~ SpaceClass.*
```

Great, now we have to modify our command instantiation a little bit to hint **sbt** that we expect some arguments to be passed and essentially provide the command implementation as well.

```
val fetch = Command(FetchCommand, Help.more(FetchCommand, FetchCommandHelp))
  (_ => mapOrFail(url)(_.toURL()) !!! "URL is not valid") { (state, url) =>
    val result = Await.result(Http(dispatch.url(url.toString()) OK as.String),
      state.get timeout getOrElse 5 seconds)
    state.log.info(s"${result}")
    state
  }
```

Excellent, we have just defined our own command! However, an attentive reader may notice the presence of `timeout` variable in the code snippet above. The **State** in **sbt** may contain additional **attributes** which could be shared. Here is how the `timeout` is being defined:

```
val timeout = AttributeKey[Int]("timeout",
    "The timeout (in seconds) to wait for HTTP response")
```

With that, we have covered the last piece of the puzzle and are ready to provide the implementation of the `initialState` function.

```
def initialState(configuration: AppConfig): State = {
    val commandDefinitions = fetch +: BasicCommands.allBasicCommands
    val commandsToRun = "iflast shell" +: configuration.arguments.map(_.trim)

    State(
        configuration,
        commandDefinitions,
        Set.empty,
        None,
        commandsToRun,
        State.newHistory,
        AttributeMap(AttributeEntry(timeout, 1)),
        globalLogging,
        State.Continue
    )
}
```

Please notice how we included our `fetch` command into the initial state (`fetch +: BasicCommands.allBasicCommands`) and specified the default timeout value of 1 second (`AttributeEntry(timeout, 1)`).

The last topic which needs some clarification is how to launch our interactive console application? For this purposes `sbt` provides a **launcher**.

In its minimal form, it is just a single file `sbt-launch.jar` that should be downloaded and used to launch the applications by resolving them through **Apache Ivy** dependency management. Each application is responsible for supplying its **launcher configuration** which for our simple example may look like this one (stored in `console.boot.properties` file):

```
01 [app]
02   org: com.javacodegeeks
03   name: console-interactive
04   version: 0.0.1-SNAPSHOT
05   class: com.javacodegeeks.console.ConsoleApp
06   components: xsbti
07   cross-versioned: binary
08
09 [scala]
10   version: 2.11.8
11
12 [boot]
13   directory: ${sbt.boot.directory}-${sbt.global.base}-${user.home}/.sbt/boot/}
14
15 [log]
16   level: info
17
18 [repositories]
19   local
20   maven-central
21   typesafe-ivy-releases: http://repo.typesafe.com/typesafe/ivy-releases
22   typesafe-releases: http://repo.typesafe.com/typesafe/releases
```

Figure 5.1: Launcher configuration

Nothing prevents us from running the sample application anymore so let us do that by publishing it to the local **Apache Ivy** repository first:



```
$ sbt publishLocal
```

And running via **sbt** launcher right after:

```
$ java -Dsbt.boot.properties=console.boot.properties -jar sbt-launch.jar
Getting com.javacodegeeks console-interactive_2.11 0.0.1-SNAPSHOT ...
:: retrieving :: org.scala-sbt#boot-app
   confs: [default]
   22 artifacts copied, 0 already retrieved (8605kB/333ms)
>
```

Awesome, we are now in the interactive shell of our application! Let us type `help fetch` to make sure our own command is there.

```
> help fetch
fetch <url>

Fetches the <url> and prints out the response

>
```

How about fetching the data from some real URL addresses?

```
> fetch https://freegeoip.net/json/www.google.com
[info] {"ip":"216.58.219.196","country_code":"US","country_name":"United States","
      region_code":"CA","region_name":"California","city":"Mountain View","zip_code":"94043","
      time_zone":"America/Los_Angeles","latitude":37.4192,"longitude":-122.0574,"metro_code":
      807}
>
```

It works perfectly fine! But what if we made some typo and our URL address is not valid? Would our `fetch` command figure this one out? Let us see ...

```
> fetch http://freegeoip.net/json/www.google.com
[error] URL is not valid
[error] fetch http://freegeoip.net/json/www.google.com
[error] ^
>
```

As expected, it did and promptly reported about the error. Nice but could we execute the `fetch` command without the need to run the interactive shell? The answer is “Yes, sure!”, we just need to pass the command to be executed as an argument to **sbt** launcher, wrapped in double quotes, for example:

```
$ java -Dsbt.boot.properties=console.boot.properties -jar sbt-launch.jar "fetch https://
      freegeoip.net/json/www.google.com"

[info] {"ip":"172.217.4.68","country_code":"US","country_name":"United States","region_code":
      "CA","region_name":"California","city":"Mountain View","zip_code":"94043","time_zone":
      "America/Los_Angeles","latitude":37.4192,"longitude":-122.0574,"metro_code":807}
```

The results printed out in the console are exactly the same. You may notice that we did not implement optional support for writing the output into the file however for console-based applications we can use a simple trick with stream redirection:

```
$ java -Dsbt.boot.properties=console.boot.properties -jar sbt-launch.jar "fetch https://
      freegeoip.net/json/www.google.com" > response.json
```

Although fully functional, our simple interactive application uses just a tiny piece of **sbt** power. Please feel free to explore the [documentation section](#) of this great tool related to creation of the command line applications.

## 5.5 Conclusions

In this section of the tutorial we have talked about building console applications using terrific **Scala** programming language and libraries. The value and usefulness of the plain old command line applications is certainly much underestimated and hopefully this section proves the point. Either you are developing a simple command-line driven tool or interactive one, **Scala** ecosystem is here to offer the full support.

## 5.6 What's next

In the next section we are going to talk a lot about concurrency and parallelism, more precisely discussing the ideas and concepts behind **Actor model** and continuing our acquaintance with other parts of the awesome **Akka** toolkit.

The complete projects [are available for download](#).

---

## Chapter 6

# Concurrency and parallelism with Akka

It would be fair to say that the epoch of computers with single CPU (or just one core) has mostly become a forgotten history these days. Now, most of the devices, no matter how small are they, have been built using multi-core CPU architectures aimed to increase the overall computational power.

Such advances and innovations on hardware front forced us to rethink the approaches of how to develop and run software systems in order to effectively utilize all available resources.

### 6.1 Introduction

In this section of the tutorial we are going to talk about two core concepts which modern software systems are relying upon: Concurrency and parallelism. Although those are very close to each other, there is a slight but important distinction. Concurrency describes the process when multiple tasks could make a progress over time while parallelism describes the process when multiple tasks are being executed simultaneously.

### 6.2 Threads

Traditionally, most of concurrent programming models were built around threads, and Java was not an exception. A typical JVM application (run as a process) may spawn many threads in order to execute some work concurrently or, ideally, in parallel.

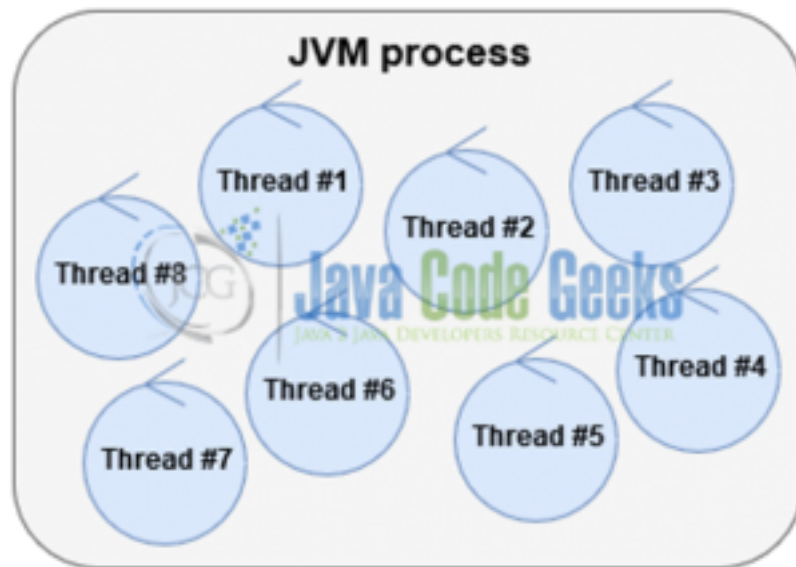


Figure 6.1: Typical JVM process spawns a couple of threads

This model worked somewhat well for a while, but there is at least one fundamental flaw of thread-based concurrency models: state or resource sharing. In most cases, threads need to exchange share some data or utilize another shared resource(s) in order to accomplish their work. Without proper mechanisms in place, uncontrolled access to shared resources or modification of the shared state by multiple threads (known as **race conditions**) leads to data corruption and often causes application to crash.

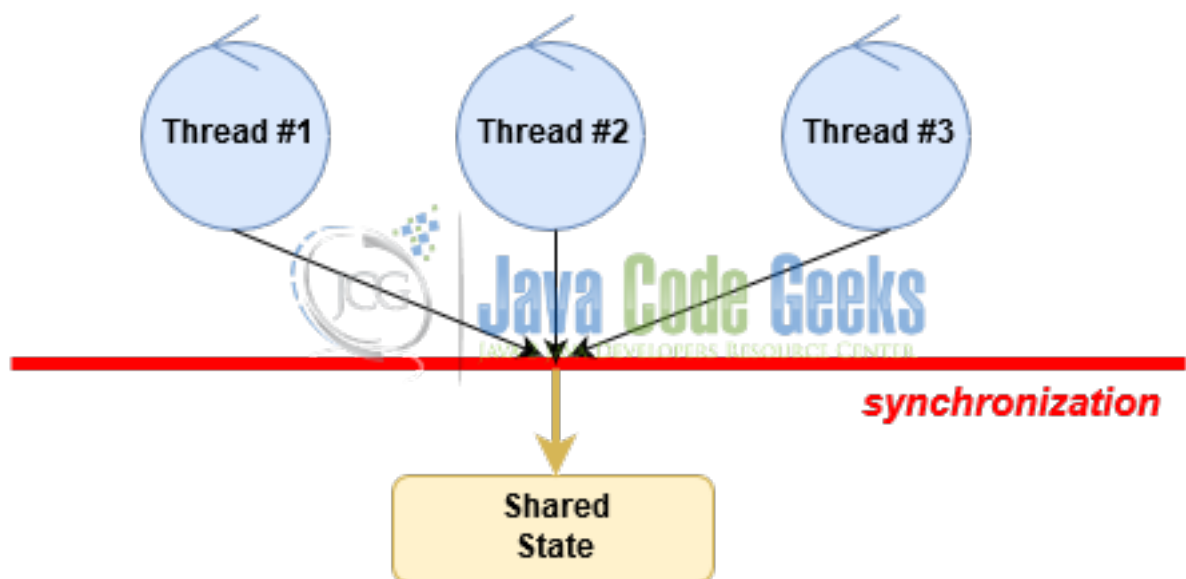


Figure 6.2: Using synchronization to access shared state

Proper usage of the synchronization primitives (locks, mutexes, semaphores, monitors, ...) solves the problem of accessing shared state (or resource), however the price to pay is really high. Not only it complicates the programming models, non-deterministic nature of the multi-threaded execution flows makes the process of troubleshooting the issues very time-consuming and difficult. Moreover, the whole new class of the problems have arisen: lock contention, thread starvation, deadlocks, livelocks, and more.

Along with that, threads and thread management consumes quite a lot of resources of the underlying operating system and essentially, at some point creation of more threads will actually have a negative impact on the application performance and

scalability.

## 6.3 Reactors and Event Loops

The flaws of thread-based concurrency models turned industry's attention into search of alternatives which more adequately address the demands of modern software systems. **Reactor pattern** and **event handling**, the foundations of the modern asynchronous and non-blocking programming paradigms, are some of the emerging programming models to deal with concurrency at scale.

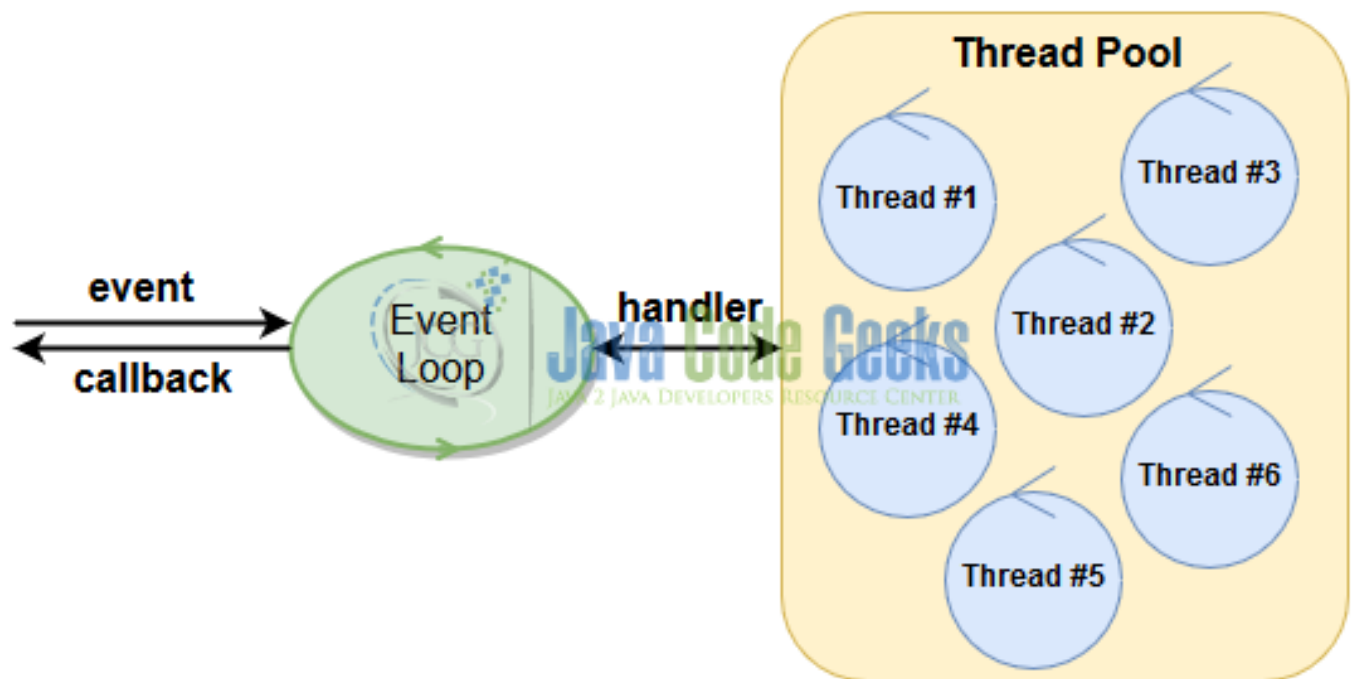


Figure 6.3: The Reactor pattern (simplified)

Please notice that this is a very simplified visualization of one of the possible implementations of the **Reactor pattern** however it illustrates its key elements pretty well. In the core of **Reactor** is a single-threaded event loop. Under the hood, event handling may use one or more threads (often grouped in a pools) to do the actual work, however the purpose and usage of threads in this model are quite different and shielded from applications completely.

## 6.4 Actors and Messages

**Message-passing**, or to be more precise, asynchronous message-passing is another very interesting and powerful concurrency model which gained a lot of traction recently. **Actor Model**, originated back in 1973, is one of the best examples of asynchronous message-passing concurrency.

Actors are the core entities of **Actor Model**. Each actor has own message queue (called mailbox), single-threaded message handler and communicates with other actors only by means of asynchronous messages (not to omit the fact that actors may create another actors as well).

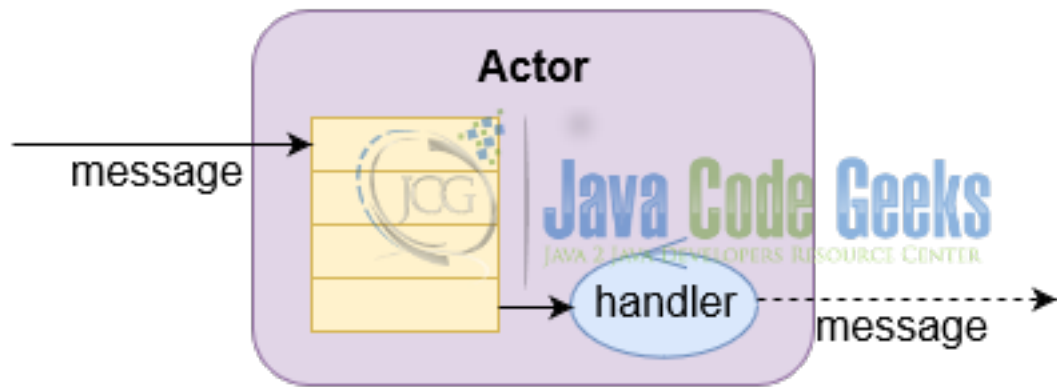


Figure 6.4: Actor in Akka

This is typical share-nothing architecture: actors may have own state but they never share anything with any other actor. Actors may live within same JVM process, or multiple JVM processes on the same physical node, or even be spread over the network, it does not really matter as far as they are able to reference each other and communicate over messages.

## 6.5 Meet Akka

We already met [Akka toolkit](#) when we talked about reactive applications and learnt about [Akka Streams](#). However, going back into the history a little bit, it is worth to mention that [Akka toolkit](#) had started as an [Actor Model](#) implementation on JVM platform. Since then it had seen many releases (with most recent one being 2.4.10), gained a lot of additional features and capabilities but nonetheless actors are the bare bones of [Akka](#) even today.

[ActorSystem](#) is the entry point into [Akka](#) actors' universe. It is a single place in the application to create and manage actors.

```
implicit val system = ActorSystem("akka-actors")
```

With that, we are ready to create new actors! In [Akka](#), every actor should subclass (or mix with) [Actor](#) trait and implement receive function, for example:

```
import akka.actor.Actor

class SampleActor extends Actor {
  def receive = {
    ...
  }
}
```

But more often than usual, you would also include [ActorLogging](#) trait into the mix to have access to a dedicated logger using log reference, for example:

```
import akka.actor.Actor
import akka.actor.ActorLogging

class SampleActor extends Actor with ActorLogging {
  def receive = {
    case _ => log.info("Received some message!")
  }
}
```

Although our `SampleActor` does not do much at the moment, we could instantiate it and send literally any message to it. As we already mentioned, actors are created only through [ActorSystem](#) instance rather than using `new` operator, for example:

```
val sampleActor = system.actorOf(Props[SampleActor], "sample-actor")
sampleActor ! "Message!"
```

If you think that `sampleActor` variable is the instance of `SampleActor` class, you are not quite right. It is actually a reference to `SampleActor` actor, represented as `ActorRef` class. This is the only mechanism to address particular actor in **Akka**, direct access to underlying actor class instances is not available.

What is going to happen when we run the application? Not much except the fact that we should see something like that in the console:

```
[INFO] [akka-actors-akka.actor.default-dispatcher-2] [akka://akka-actors/user/sample-actor] ←
    Received some message!
```

## 6.6 Supervision

Interesting but very important property of actors in **Akka** is that they are organized in hierarchies. Naturally, actors may spawn child actors in order to split the work in smaller pieces and as such, form a parent/child hierarchy.

It sounds like a minor detail but it is not because in **Akka** parent actors may watch their children, the process known as **supervision**. In this case, parent actors essentially become the supervisors and may apply different strategies in case when child actors encounter failures (or to be more specific, terminate with an exception).

**Official documentation** discusses defaults and different supervision strategies in great details but let us have a look at quick example. Our `ChildActor` is defined in such a way that always throws an exception upon receiving any `Message`.

```
class ChildActor extends Actor with ActorLogging {
  def receive = {
    case Message(m) =>
      throw new IllegalStateException("Something unexpected happened")
  }
}
```

Consequently, our `ParentActor` actor creates the instance of `ChildActor` actor and forwards any message it receives to `ChildActor` instance.

```
class ParentActor extends Actor with ActorLogging {
  val child = context.actorOf(Props[ChildActor])

  override val supervisorStrategy = OneForOneStrategy() {
    case _: IllegalStateException => Resume
    case _: Exception => Escalate
  }

  def receive = {
    case _ => child ! Message("Message from parent")
  }
}
```

According to default supervision strategy, the actor which raises an `Exception` is going to be restarted (which is probably the desired behavior in most cases). In our example however, we overwrote this policy (using `supervisorStrategy` property of the `ParentActor`) to resume the normal message processing of the supervising actors (`ChildActor`) in case of `IllegalStateException` only.

## 6.7 Patterns

In its basic form, actors in **Akka** communicate via asynchronous, one-way messages. It certainly works, however many real-world scenarios require more complex interactions, for example using request/reply communication, circuit breakers, piping messages

between actors and others. Luckily, `akka.pattern` package provides a set of commonly used `Akka` patterns, ready to be applied. For example, let us change a little bit the `SampleActor` implementation to handle the messages of type `Message` and, once received, reply with `MessageReply`.

```
case class Message(message: String)
case class MessageReply(reply: String)

class SampleActor extends Actor with ActorLogging {
  def receive = {
    case Message(m) => sender ! MessageReply(s"Reply: $m")
  }
}
```

Now, we can employ the `ask` pattern in order to send the message to the actor and wait for reply as well, for example:

```
import akka.pattern.ask
import akka.util.Timeout

implicit val timeout: Timeout = 1 second
val reply = (sampleActor ? Message("Please reply!")).mapTo[MessageReply]
```

In this case, the sender sends a message to an actor and waits for the reply back (with 1 second timeout). Please take a note that typical `Akka` actors do not support any type safety semantics regarding messages: anything could be sent out as well as received as a response. It becomes the responsibility of the sender to make a proper type casting (for example, using `mapTo` method). Similarly, if the sender sends the message to an actor which it does not know how to handle, the message ends up in `dead letters`.

## 6.8 Typed Actors

As it was mentioned before, actors in `Akka` do not offer any type safety regarding messages they accept or reply with. But for quite some time now `Akka` includes an experimental support of `Typed Actors`, where the contracts are explicit and are going to be enforced by compiler.

Definition of the `Typed Actors` is very different from the regular `Akka` actors and resembles a lot the way we used to build `RPC-style systems`. First of all, we have to start by defining the interface and its implementation, for example:

```
trait Typed {
  def send(message: String): Future[String]
}

class SampleTypedActor extends Typed {
  def send(message: String): Future[String] = Future.successful("Reply: " + message)
}
```

In turn, the way `Typed Actors` are instantiated requires a bit more code, although still using `ActorSystem` under the hood.

```
implicit val system = ActorSystem("typed-akka-actors")

val sampleTypedActor: Typed =
  TypedActor(system).typedActorOf(TypedProps[SampleTypedActor]())

val reply = sampleTypedActor
  .send("Hello Typed Actor!")
  .andThen { case Success(r) => println(r) }
```

At this moment the logical question may hit your mind: shouldn't `Typed Actors` be used everywhere? Good point, but the short answer is: no, probably not. If you are curious, please take some time to go over [this great discussion](#) about pros and cons of using `Typed Actors` versus regular, untyped ones.



## 6.9 Scheduler

Beyond providing superior implementation of **Actor Model**, **Akka** offers quite a few very helpful utilities around as well. One of them is **scheduling support** which provides the capability to send a message to a particular actor periodically or at some point in time.

```
implicit val system = ActorSystem("akka-utilities")
import system.dispatcher

val sampleActor = system.actorOf(Props[SampleActor], "sample-actor")
system.scheduler.schedule(0 seconds, 100 milliseconds, sampleActor, "Wake up!")
```

Needless to say, the ability to schedule some task executions is a requirement for most of the real-world applications so it is quite handy to have this feature available out of the box.

## 6.10 Event Bus

Another very useful utility which **Akka** contains generic **event bus** concept and its particular implementation provided by **ActorSystem** in a form of **event stream**.

If the actor-to-actor communication assumes that sender of the message somehow knows who the recipient is, with **event stream**, actors have an option to broadcast any events (messages of some type) to any other actor, without any prior knowledge who will receive it. In this case, the involved parties have to express their interest by subscribing to such events by their type.

For example, assume we have an important message which we simply name `Event`.

```
case class Event(id: Int)
```

Our `SampleEventActor` is designated to handle this kind of messages and prints out on the console the `id` of the message every time it receives one.

```
class SampleEventActor extends Actor with ActorLogging {
  def receive = {
    case Event(id) => log.info(s"Event with '$id' received")
  }
}
```

It looks easy, but nothing really exciting to the moment. Now, let us take a look at the **event stream** and publish/subscribe communication pattern in action.

```
implicit val system = ActorSystem("akka-utilities")

val sampleEventActor = system.actorOf(Props[SampleEventActor])
system.eventStream.subscribe(sampleEventActor, classOf[Event])

system.eventStream.publish(Event(1))
system.eventStream.publish(Event(2))
system.eventStream.publish(Event(3))
```

Our `sampleEventActor` expresses its interest in receiving messages of type `Event` by calling `system.eventStream.subscribe()` method. Now, every time `Event` is going to be published by means of `system.eventStream.publish()` invocation, the `sampleEventActor` is going to receive it, no matter who the publisher was. With logging turned on, we are going to see something like that in the console output:

```
[INFO] [akka-utilities-akka.actor.default-dispatcher-2] [akka://akka-utilities/user/$a] ←
      Event with '1' received
[INFO] [akka-utilities-akka.actor.default-dispatcher-2] [akka://akka-utilities/user/$a] ←
      Event with '2' received
[INFO] [akka-utilities-akka.actor.default-dispatcher-2] [akka://akka-utilities/user/$a] ←
      Event with '3' received
```

## 6.11 Remoting

All the examples we have seen so far dealt with just one **ActorSystem**, running in a single JVM within one node. But **Akka**'s networking extensions support multi-JVM / multi-node deployments so different **ActorSystems** may communicate with each other in a truly distributed environment.

To enable **ActorSystem**'s **remote capabilities** we would need to change its default **actor reference provider** and enable network transport. All that is easy to accomplish using `application.conf` configuration file:

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
  }

  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty {
      tcp {
        hostname = "localhost"
        port = ${port}
      }
    }
  }
}
```

As an exercise, we are going to run two **ActorSystem** instances, one with name `akka-remote-1` on port 12000, and another one, `akka-remote-2` on port 12001. We also are going to define one actor to communicate with, `SampleRemoteActor`.

```
class SampleRemoteActor extends Actor with ActorLogging {
  def receive = {
    case m: Any => log.info(s"Received: $m")
  }
}
```

On the first **ActorSystem**, `akka-remote-1`, we are going to create an instance of the `SampleRemoteActor` and send one message to it.

```
implicit val system = ActorSystem("akka-remote-1")

val sampleActor = system.actorOf(Props[SampleRemoteActor], "sample-actor")
sampleActor ! "Message from Actor System #1!"
```

However on the second one, `akka-remote-2`, we are going to send a message to the `SampleRemoteActor` instance using its remote reference, which among other things includes **ActorSystem** name (`akka-remote-1`), host (`localhost`), port (12000) and given actor name (which in our case is `sample-actor`):

```
implicit val system = ActorSystem("akka-remote-2")

val sampleActor = system.actorSelection(
  "akka.tcp://akka-remote-1@localhost:12000/user/sample-actor")
sampleActor ! "Message from Actor System #2!"
```

Quite straightforward, isn't it? Running both **ActorSystem** instances side by side will produce the following output in the console of `akka-remote-1` JVM process:

```
[INFO] [main] [akka.remote.Remoting] Starting remoting
[INFO] [main] [akka.remote.Remoting] Remoting started; listening on addresses :[akka.tcp:// ←
  akka-remote-1@localhost:12000]
[INFO] [main] [akka.remote.Remoting] Remoting now listens on addresses: [akka.tcp://akka- ←
  remote-1@localhost:12000]
```

```
[INFO] [akka-remote-1-akka.actor.default-dispatcher-2] [akka.tcp://akka-remote-1@localhost ←
:12000/user/sample-actor] Received: Message from Actor System #1!
[INFO] [akka-remote-1-akka.actor.default-dispatcher-4] [akka.tcp://akka-remote-1@localhost ←
:12000/user/sample-actor] Received: Message from Actor System #2!
```

Adding to what we have seen so far, not only one **actor system** may reference the actors from another **actor systems**, it can also create new actor instances **remotely**.

## 6.12 Testing

**Akka** includes a superior **testing support** to ensure that every single aspect of actors' behavior and interactions could be covered. In fact, **Akka TestKit** provides appropriate scaffolding for writing traditional unit tests as well as integration tests.

Unit testing techniques revolve around **TestActorRef** which is a simplification around regular **ActorRef** implementation, with no concurrency involved and access to actor state internals. Let us start with this one and come up with a simple unit test for our **SampleActor** using already familiar to us **specs2** framework.

```
class SampleActorTest extends Specification with AfterAll {
  implicit val timeout: Timeout = 1 second
  implicit lazy val system = ActorSystem("test")

  "Sample actor" >> {
    "should reply on message" >> { implicit ee: ExecutionEnv =>
      val actorRef = TestActorRef(new SampleActor)
      actorRef ? Message("Hello") must be_==(MessageReply("Reply: Hello")).await
    }
  }

  def afterAll() = {
    system.terminate()
  }
}
```

Unit testing is certainly a very good start but at the same time, it is often quite limited as it relies on a simplified view of the system. However, again thanks to **Akka TestKit**, there are more powerful testing techniques at our disposal.

```
class SampleActorIntegrationTest extends TestKit(ActorSystem("test"))
  with ImplicitSender with WordSpecLike with BeforeAndAfterAll {

  "Sample actor" should {
    "should reply on message" in {
      val actorRef = system.actorOf(Props[SampleActor])
      actorRef ! Message("Hello")
      expectMsg(MessageReply("Reply: Hello"))
    }

    "should log an event" in {
      val actorRef = system.actorOf(Props[SampleActor])
      EventFilter.info(
        message = "Event with '100' received", occurrences = 1) intercept {
        actorRef ! Event(100)
      }
    }
  }

  override def afterAll() = {
    shutdown()
  }
}
```

This time we have used **ScalaTest** framework perspective and took a slightly different approach relying on **TestKit** class which offers a **rich set of assertions** over message expectations. Not only we have an ability to spy on messages, we are also able to make assertions over expected log records using **EventFilter** class, backed by **TestEventListener** in the `application.conf` file.

```
akka {  
  loggers = [  
    akka.testkit.TestEventListener  
  ]  
}
```

Really nice, the test cases look simple, readable and maintainable. However, **Akka testing capabilities** do not stop here and are still evolving very fast. For example, it is worth to mention the availability of the experimental **multi node testing** support.

## 6.13 Conclusions

**Akka** is a terrific toolkit and serves as a solid foundation for many other libraries and frameworks. As **Actor Model** implementation, it yet offers another approach to concurrency and parallelism using asynchronous message passing and promoting immutability.

It is worth to mention that these days **Akka** is being actively developed and goes way beyond the **Actor Model**. With every new release it includes more and more tools (stable or/and experimental) for building highly concurrent and distributed systems. Many of its advanced features like **clustering**, **persistence**, **finite state machines**, **routing**, ... we have not touched upon at all but **official Akka documentation** is the best place to get familiarized with all of them.

## 6.14 What's next

In the next section of the tutorial we are going to talk about **Play! Framework**: powerful, highly productive and feature-rich framework for building scalable, full-fledged web applications in **Scala**.

The source code of all examples is **available here**.

---

## Chapter 7

# Web Applications with Play Framework

### 7.1 Introduction

It is been a long time since Web became a dominant, universally and globally accessible platform for myriad of different applications: either web sites, web portals, or web APIs. Started as a simple set of static HTML pages, web applications in leaps and bounds were catching up with their desktop counterparts, transforming into new class of what we used to call **rich Internet applications** (or just **RIA**). However, most of such advances would have not been possible without evolution (and in some cases revolution) of the web browsers.

In this section of the tutorial we are going to talk about developing rich web sites and portals (or to say it simply, modern web applications) using **Scala** programming language and ecosystem.

### 7.2 MVC and the Power of Patterns

There are many ways one can approach the design and development of the web applications. Nonetheless, a couple of patterns have been emerged out of the crowd and gained a widespread adoption in software development community.

**Model-View-Controller** (or **MVC**) is one of the most widely applied architectural patterns used for developing maintainable UI-based applications. It is very simple but provides quite sufficient level of the separation of concerns and responsibilities.

---

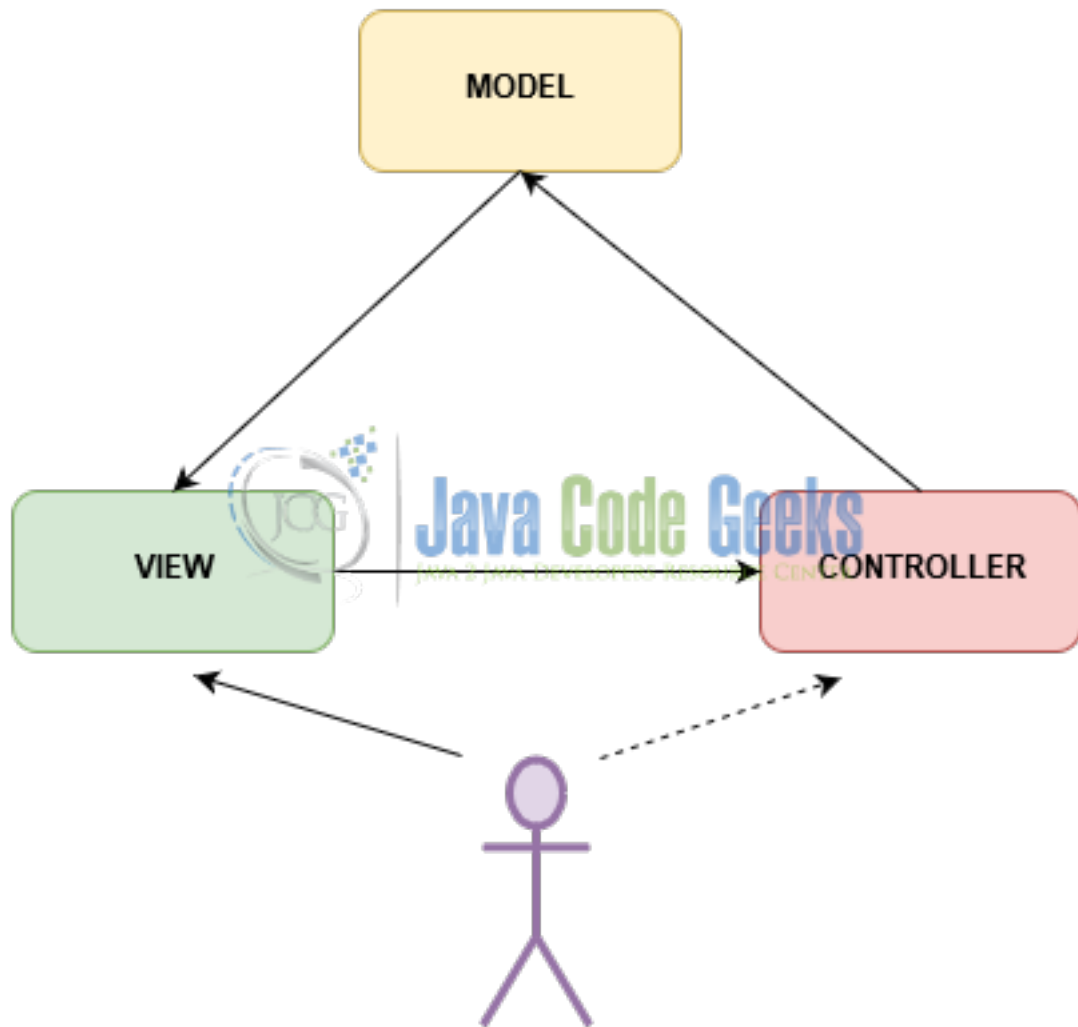


Figure 7.1: MVC pattern and its collaborators

In the essence, **MVC** outlines pretty well the collaborators and their roles. The **View** uses **Model** to render the desktop or web UI representation to the **User**. The **User** interacts with the **View**, which may lead to model updates (or retrievals) by means of using the **Controller**. In turn, **Controller**'s actions over **Model** may lead to the **View** being refreshed as well. In some cases, **User** may interact with the **Controller** directly, completely bypassing the **View**.

Many frameworks used these days for web applications development are designed around **MVC** pattern or one (or more) its derivatives. In **Scala** ecosystem, the **Play Framework** is the undoubtedly the best available option out there and it is what we are going to talk about in this section of the tutorial.

### 7.3 Play Framework: enjoyable and productive

**Play Framework** is modern, production-ready, high velocity, full-fledged web framework written in **Scala** (with Java-friendly API also available). It is architected to be fully asynchronous, lightweight and stateless and is built on top of **Akka Toolkit**, which we have discussed in details in the previous section of the tutorial. The latest released version of the **Play Framework** at the moment of this writing is 2.5.9.

Although **Play Framework** is not limited to support the development of the web applications only, we are going to focus mostly on this side of things, continuing the discussion about web APIs in the next section, dedicated specifically to that.

## 7.4 Controllers, Actions and Routes

**Play Framework** fully embraces the **MVC** model and right from the start introduces the concept of **controllers**. Following their responsibilities, **controllers** may generate some **actions**, returning some results back, for example:

```
@Singleton
class HealthController extends Controller {
  def check() = Action {
    Ok("OK")
  }
}
```

Please notice that by convention, controllers are stored under `controllers` package. Controller methods may be exposed directly as **HTTP** endpoints, using **HTTP** protocol semantics. In **Play Framework** such a mapping is called a **route** and all route definitions are placed in the `conf/routes` file, for example:

```
GET    /health          controllers.HealthController.check
GET    /assets/*file     controllers.Assets.versioned(path="/public", file: Asset)
```

Let the simplicity of this example not deceive you, **Play Framework** route definitions may include arbitrary number of quite sophisticated parameters and **URI** patterns as we are going to see later in the section.

The controllers in **Play Framework** extend **Controller** trait and may contain any (reasonable) number of methods which return **Action** instances. All actions are executed in an **asynchronous, non-blocking way** and it is very important to keep that in mind. Controllers should avoid execution of the blocking operations whenever possible and **Action** companion object offers convenient **async** methods family to seamlessly integrate with asynchronous execution flows, for example:

```
@Singleton
class UserController @Inject() (val service: UserService) extends Controller {
  import play.api.libs.concurrent.Execution.Implicits.defaultContext

  def getUsers = Action.async {
    service.findAll().map { users =>
      Ok(views.html.users(users))
    }
  }
}
```

Besides asynchronicity, this short code snippet also illustrates how controllers introduce yet another **MVC** collaborator, the **view** of the model. Let us talk about that for a moment.

## 7.5 Views and Templates

The views in **Play Framework** are usually based on regular **HTML** markup but backed by **Twirl**, extremely powerful **Scala**-based template engine. Under the hood, **templates** are transformed to **Scala** classes along with companion objects. They can have arguments and are compiled as standard **Scala** code.

Let us have a quick example of how model could be passed and rendered in **Play Framework** view template by introducing a User case class:

```
case class User(id: Option[Int], email: String,
  firstName: Option[String], lastName: Option[String])
```

If it looks familiar, you are not mistaken: this is exactly the same case class we have seen in **Database Access with Slick** section of the tutorial. So let us create a `users.scala.html` template to print out the list of users as **HTML** table, stored by convention in `views` folder (which also serves as a package name):

```
@(users: Seq[model.User])

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Manage Users</title>
  </head>
  <body>

    <div class="panel panel-default">
      <div class="panel-heading">Users
        < table class="table">
          <thead>
            <th>Id</th>
            <th>First Name</th>
            <th>Last Name</th>
            <th>Email</th>

          </thead>
          @for(user <- users) {
            |@user.id|@user.firstName|@user.lastName|@user.email
          }
        </ table>
      </div>
    </div>
  </body>
</html>
```

By and large, this is just raw **HTML** markup, so close to heart of any front-end developer. The first line declares template arguments, `@(users: Seq[model.User])` which is just list of users. The only place we use this argument is when render the rows of the table by applying **Scala**-like expressions:

```
@for(user <- users) {
  |@user.id|@user.firstName|@user.lastName|@user.email
}
```

And that's it! And because all the templates are compiled down to byte code, any errors related, for example, to non-existing properties or inappropriate expressions usage will be caught at compile time! With such a help from compiler, any kind of refactorings become much easier and safer.

Users			
Id	First Name	Last Name	Email
1	Tom	Tommyknocker	a@b.com
2	Bob	Bobinec	b@c.com

Figure 7.2: Users table

To close the loop, in the **Controllers** section we have already seen how the templates could be instantiated and send out to the browser using controller action:

```
def getUsers = Action.async {
  service.findAll().map { users =>
    Ok(views.html.users(users))
  }
}
```



Aside from simple components, **templates** may include **HTML forms** which could be backed directly by controller methods. As an example, let us implement the adding of new user functionality, as such wiring view, controller, and model together. First, on the controller side we have to add `Form` definition, including all validation constraints:

```
val userForm = Form(
  mapping(
    "id" -> ignored[Option[Int]](None),
    "email" -> email.verifying("Maximum length is 512", _.size <= 512),
    "firstName" -> optional(text(maxLength = 64)),
    "lastName" -> optional(text(maxLength = 64))
  ) (User.apply) (User.unapply)
)
```

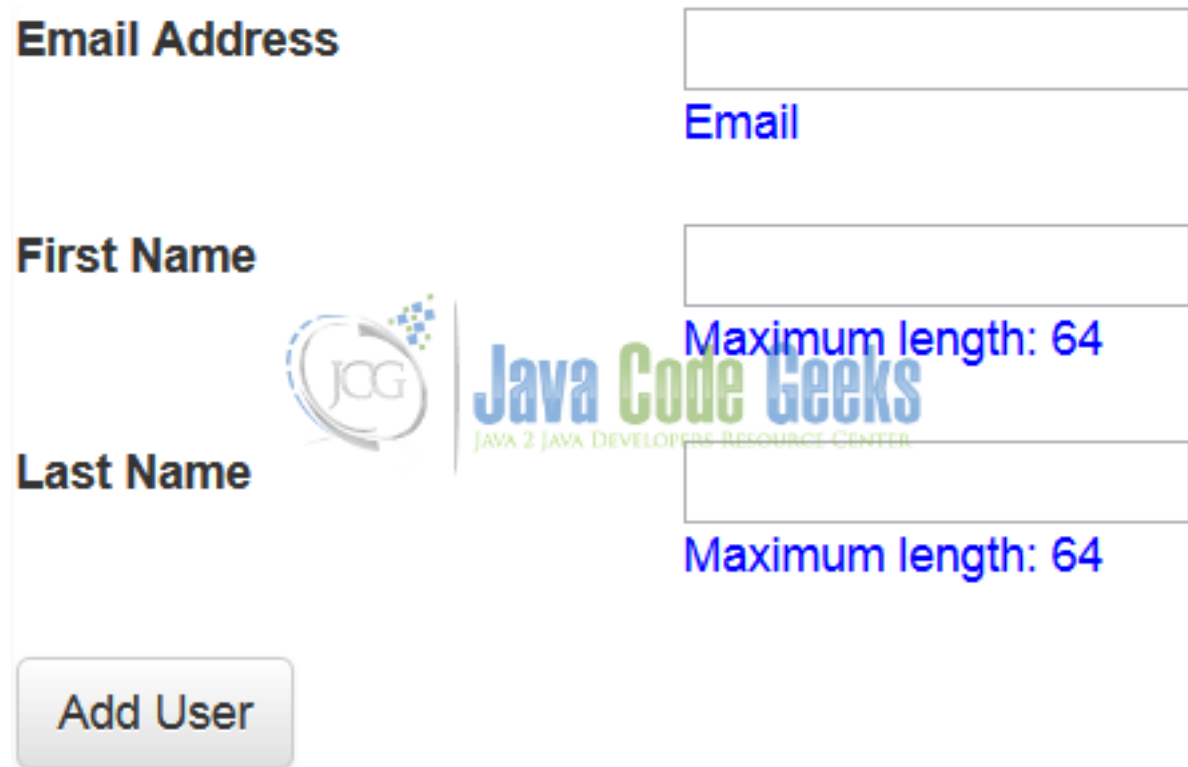
Second, we are going to create a dedicated endpoint in the controller to add new user as the result of form submission:

```
def addUser = Action.async { implicit request =>
  userForm.bindFromRequest.fold(
    formWithErrors => {
      service.findAll().map { users =>
        BadRequest(views.html.users(users) (formWithErrors))
      }
    },
    user => {
      service.insert(user).map { user =>
        Redirect(routes.UserController.getUsers)
      } recoverWith {
        case _ => service.findAll().map { users =>
          BadRequest(views.html.users(users) (userForm
            .withGlobalError(s"Duplicate email address: ${user.email}")))
        }
      }
    }
  )
}
```

Please notice how `userForm.bindFromRequest.fold` in one shot does the bindings of the form parameters from the request along with performing all validation checks. Next thing, we have to replicate the `userForm` into its **HTML** presentation, using view template for that:

```
@helper.form(action = routes.UserController.addUser) {
  @helper.inputText(userForm("email"), '_label -> "Email Address")
  @helper.inputText(userForm("firstName"), '_label -> "First Name")
  @helper.inputText(userForm("lastName"), '_label -> "Last Name")
  <button class="btn btn-default" type="submit">Add User</button>
}
```

The usage of `@helper` simplifies a lot forms construction as all the relevant types and validation constraints will be taken from `Form` definition and hinted to the user. But as mostly everywhere in **Play Framework**, you are not obliged to use this approach: any JavaScript/CSS framework of your choice could be used instead. Here is a sneak peak on how this form looks in the browser.



The image shows a web form for adding a user. It has three input fields: 'Email Address', 'First Name', and 'Last Name'. The 'Email Address' field is labeled 'Email' in blue text and has a 'Maximum length: 64' constraint. The 'First Name' field is labeled 'First Name' and also has a 'Maximum length: 64' constraint. The 'Last Name' field is labeled 'Last Name' and also has a 'Maximum length: 64' constraint. There is a 'JOG' logo and a 'Java Code Geeks' watermark in the background. At the bottom, there is a button labeled 'Add User'.

Figure 7.3: Add User form

And as a final step, the routing table should be updated to have this new endpoint listed as well. Luckily, it is just one liner:

```
POST / controllers.UserController.addUser
```

Please notice that **Play Framework** scaffolding takes care of all validation part and reporting it back to the view, for example, submission of the form with invalid email address will not be accepted:



The image shows the same 'Add User' form as in Figure 7.3, but with an error message. The 'Email Address' field contains the text 'wdwq'. Below the field, there is a red error message that says 'Valid email required'.

Figure 7.4: Error in case incorrect email address is specified

Surely, **Play Framework** gives enough flexibility to report back other types of errors, not necessarily tied to validation. For example, in the controller we have used `userForm.withGlobalError` method to signal duplicate email address.

## 7.6 Action Compositions and Filters

There is often a need to perform some actions before or after controller's method invocation, possibly even altering the response. The examples of that could be logging, security verifications or support of the **cross-origin resource sharing** (CORS).

Out of the box **Play Framework** provides a couple of ways to hijack into the processing pipeline: using **filters** and **action compositions**.

## 7.7 Accessing Database

Any more or less real-world web application would need to manage some data and in many cases well-known relation data stores are the perfect choice. **Play Framework** offers **superior integration** with a couple of the JDBC-based libraries but we already learned quite a lot of great things about **Slick** and surely, **Play Framework** integrates **with Slick** seamlessly.

To make things even simpler and familiar, we are going to reuse the same data model we have built in **Database Access with Slick** part, with mostly no modification whatsoever. The minor change that we would need to do is affecting **UserRepository** only: injecting **DatabaseConfigProvider** for default database configuration and use its `provider.get[JdbcProfile]` method to get the corresponding **JdbcProfile** instance.

```
@Singleton
class UserRepository @Inject() (val provider: DatabaseConfigProvider)
  extends HasDatabaseConfig[JdbcProfile] with UsersTable {
  val dbConfig = provider.get[JdbcProfile]

  import dbConfig.driver.api._
  import scala.concurrent.ExecutionContext.Implicits.global

  ...
}
```

And we are done. **Play Framework** allows to manage multiple named database instances and configure them through `application.conf` file. For convenience, single database web applications may use the specially treated default one.

```
slick {
  dbs {
    default {
      driver="slick.driver.H2Driver$"
      db {
        driver="org.h2.Driver"
        url="jdbc:h2:mem:users;DB_CLOSE_DELAY=-1"
      }
    }
  }
}
```

One of the most common problems which application developers face every time while dealing with relation databases is schema management. The data model evolves over time as so does the database: new tables, columns and indexes are often added, unused ones get removed. **Database evolutions** is yet another terrific feature **Play Framework** provides out of the box.

## 7.8 Using Akka

The **Play Framework** stands on **Akka Toolkit** foundation and as such actors are the first class-citizens in there. Any **Play Framework** web application has a **dedicated actor system** created right when the application starts (and restarts automatically when the application restarts).

Fully embracing the reactive paradigm since the very beginning, the **Play Framework** is one of the earliest adopters of the **Akka Streams** implementation. Going even further, **Play Framework** provides quite **a few useful utility classes** to bridge together **Akka Streams** with web application specific technologies, which we are going to talk about.

## 7.9 WebSockets

Arguably, **WebSockets** are one of the most interesting and rapidly spreading communication protocols these days. Essentially, **WebSockets** take the web client/web server interactions to the next level, providing the full-duplex communication channel established over HTTP protocol.

To put **WebSockets** in perspective of real applications, let us implement the feature to show the notification in our web application every time new user is added. Internally, this fact is represented by `UserAdded` event.

```
case class UserAdded(user: User)
```

Using **Akka's event stream**, which we talked about in the previous part of the tutorial, we can subscribe to this event by creating a dedicated actor, let us call it `UsersWebSocketActor`.

```
class UsersWebSocketActor(val out: ActorRef) extends Actor with ActorLogging {
  override def preStart() = {
    context.system.eventStream.subscribe(self, classOf[UserAdded])
  }

  override def postStop() = {
    context.system.eventStream.unsubscribe(self)
  }

  def receive() = {
    case UserAdded(user) => out ! user
  }
}
```

It looks exceptionally simple but the mysterious `out` actor reference. Let us see where it comes from. **Play Framework** always had superior supports for **WebSockets**, however closer integration with **Akka Streams** made it **much, much better**. Here is the **WebSockets** endpoint to broadcast the notifications about new users to the client.

```
def usersWs = WebSocket.accept[User, User] { request =>
  ActorFlow.actorRef(out => Props(new UsersWebSocketActor(out)))
}
```

It is just a few lines of code for such a complex feature, really amazing! Please take a note that the `out` actor reference essentially represents the web client side and is provided by **Play Framework** out of the box. The changes in routing table are minimal as well.

```
GET /notifications/users controllers.NotificationController.usersWs
```

On the browser side of things, this is a standard piece of JavaScript code, which could be injected right into the view template.

```
<script type="text/javascript">
  var socket = new WebSocket(
    "@routes.NotificationController.usersWs().websocketURL()"
  )

  socket.onmessage = function(event) {
    var user = jQuery.parseJSON(event.data);
    ...
  }
</script>
```

As was mentioned, **WebSockets** are bi-direction communication channel: not only web server can send data to web client, web client can initiate some messages as well. We have not covered this part in the example here but **Play Framework documentation** discusses it in details.

## 7.10 Server-Sent Events

**WebSockets** are extremely powerful but often the web application needs could be backed by much simpler implementation. In case the full-duplex channel is not required, web server can rely on **server-sent events** (or **SSE**) to just send data to the web client in one-way fashion. In **Play Framework** (and many other frameworks as well), it is implemented by supporting chunked (or streaming) responses with special content type `text/event-stream`.

```
def usersSse = Action {
  Ok.chunked(
    Source.actorPublisher(Props[UsersSseActor]) via EventSource.flow[User]
  ).as(ContentType.EVENT_STREAM)
}
```

In this case the sever may use a full-fledged **Akka Streams** data processing pipelines and deliver data to the client using **EventSource** scaffolding. To illustrate yet another interesting feature of **Akka Streams**, we are using `UsersSseActor` actor, functionally similar to `UsersWebSocketActor`, as the stream source.

```
class UsersSseActor extends ActorPublisher[User] with ActorLogging {
  var buffer = Vector.empty[User]

  override def preStart() = {
    context.system.eventStream.subscribe(self, classOf[UserAdded])
  }

  override def postStop() = {
    context.system.eventStream.unsubscribe(self)
  }

  def receive = {
    case UserAdded(user) if buffer.size < 100 => {
      buffer :=+ user
      send()
    }

    case Request(_) => send()
    case Cancel => context.stop(self)
  }

  private[this] def send(): Unit = if (totalDemand > 0) {
    val (use, keep) = buffer.splitAt(totalDemand.toInt)
    buffer = keep
    use foreach onNext
  }
}
```

It is a bit more complex due to the fact that we have to follow **Akka Streams** convention and APIs to have a well-behaved publisher, but essentially it also uses **event stream** to subscribe to the notifications. Again, on a view side, just bare bone JavaScript:

```
<script type="text/javascript">
var event = new EventSource(
  "@routes.NotificationController.usersSse().absoluteURL()");

event.addEventListener('message', function(event) {
  var user = jQuery.parseJSON(event.data);
  ...
});
</script>
```

And not to forget about adding yet another entry into routing table:

```
GET /notifications/sse controllers.NotificationController.usersSse
```

## 7.11 Running Play Applications

There are multiple ways to run our **Play Framework** application, but probably the easiest one is to use the **sbt** tool we already are quite familiar with:

```
sbt run
```

The better way however, still using **sbt**, would be to run the application in continuous edit-compile-(re)deploy cycle, to (mostly) instantaneously reflect the modifications in the source files:

```
sbt ~run
```

By default, every **Play** application is running on HTTP port **9000**, so feel free to navigate your browser to <https://localhost:9000> to play with users or to <https://localhost:9000/notifications> to see **WebSockets** and **server-sent events** in action.

## 7.12 Secure HTTP (HTTPS)

Using **secure HTTP (HTTPS)** in production is a must-have rule for modern web sites and portal these days. But very often there is a need to run your **Play Framework** application with **HTTPS** support during the development as well. It is usually being done by generating **self-signed certificates** and importing them into **Java Key Store**, which is just one command away:

```
keytool -genkeypair -v
  -alias localhost
  -dname "CN=localhost"
  -keystore conf/play-webapp.jks
  -keypass changeme
  -storepass changeme
  -keyalg RSA
  -keysize 4096
  -ext KeyUsage:critical="keyCertSign"
  -ext BasicConstraints:critical="ca:true"
  -validity 365
```

The `conf/play-webapp.jks` key store could be used to **configure Play Framework application** to run with **HTTPS** support, for example:

```
sbt run -Dhttps.port=9443 -Dplay.server.https.keyStore.path=conf/play-webapp.jks -Dplay.↵
  server.https.keyStore.password=changeme
```

Now we could navigate to <https://localhost:9443/> to get the list of the users (the same one we would see by using <https://localhost:9000/>). Very simple and easy, isn't it?

## 7.13 Testing

In the web applications world, testing has many forms and faces, but **Play Framework** does a really good job by providing the necessary scaffolding to simplify those. Moreover, both **ScalaTest** and **specs2** frameworks are equally supported.

Probably, the simplest and fastest way to approach testing in **Play Framework** is by using unit tests. For example, let us take a look on this **specs2** suite for testing **UserController** methods.

```
class UserControllerUnitSpec extends PlaySpecification with Mockito {
  "UserController" should {
    "render the users page" in {
      val userService = mock[UserService]
      val controller = new UserController(userService)

      userService.findAll() returns Future.successful(Seq(
```

```

        User(Some(1), "a@b.com", Some("Tom"), Some("Tommyknocker")))
    val result = controller.getUsers() (FakeRequest())

    status(result) must equalTo(OK)
    contentAsString(result) must contain("a@b.com")
  }
}

```

Helpful **Play Framework** scaffolding for **specs2 integration** makes writing a unit tests as easy as breathing. Going up one level in **testing pyramid**, we may need to consider writing integration tests and in this case **Play Framework** would be better used with **ScalaTest** due to **dedicated additional features** provided out of the box.

```

class UserControllerSpec extends PlaySpec with OneAppPerTest with ScalaFutures {
  "UserController" should {
    "render the users page" in {
      val userService = app.injector.instanceOf[UserService]

      whenReady(userService.insert (User (None, "a@b.com",
        Some("Tom"), Some("Tommyknocker")))) { user =>
        user.id mustBe defined
      }

      val users = route(app, FakeRequest (GET, "/")).get
      status(users) mustBe OK
      contentType(users) mustBe Some("text/html")
      contentAsString(users) must include("a@b.com")
        .and(include ("Tommyknocker"))
    }
  }
}

```

In this case there is a full-fledge **Play Framework** application being created, including database instance configured with all evolutions applied. However, if you would like to get as close as possible to real deployment, you may consider to add web UI (with or without browser) test cases, and again, **ScalaTest integration** is offering the necessary pieces.

```

class UserControllerBrowserSpec extends PlaySpec
  with OneServerPerSuite with OneBrowserPerSuite
  with HtmlUnitFactory {
  "Users page" must {
    "should show empty users" in {
      go to s"https://localhost:$port/"
      pageTitle mustBe "Manage Users"

      textField("email").value = "a@b.com"
      submit()

      eventually { pageTitle mustBe "Manage Users" }
      find(xpath("./*[@class='table']/tbody/tr[1]/td[4]")) map {
        _.text mustBe ("a@b.com")
      }
    }
  }
}

```

This well-known testing strategy is based on **Selenium web browser automation** which is nicely wrapped into **OneBrowserPerSuite** along with browser-less **HtmlUnitFactory**.

## 7.14 Conclusions

Without any doubts, **Play Framework** brings back the pleasure and the joy of web application development on JVM platform. With **Scala** and **Akka** in its core, modern, extremely feature-rich and productive, built on top of **reactive programming** paradigm, all that makes **Play Framework** a choice you would never regret. Not to forget that excellent separation between frontend and backend lets you to bridge the best parts of two worlds together, resulting into creation of beautiful and maintainable web applications.

## 7.15 What's next

In the next section of the tutorial we are going to talk about **REST(ful) web APIs** development using **Akka HTTP** module.

The complete source code is available for [download](#).

---



## Chapter 8

# Web APIs with Akka HTTP

### 8.1 Introduction

How often these days you hear phrases like “**Web APIs are eating the world**”? Indeed, credits to [Marc Andreessen](#) for nicely summarizing that, but APIs become more and more important in backing [business-to-business](#) or [business-to-consumer](#) conversations, particularly in web universe.

For many businesses the presence of the web APIs is must-have competitive advantage and often is a matter of survival.

### 8.2 Being REST(ful)

In the today’s web, [HTTP](#) is the king. Over the years many different attempts have been made to come up with standard, uniform high-level protocol to expose services over it. [SOAP](#) was probably the first one which gained widespread popularity (particularly in enterprise world) and served as de-facto specification for web services and APIs development for number of years. However, its verbosity and formalism often served as a source of additional complexity and that is one of the reasons for [Representational state transfer](#) (or just [REST](#)) to emerge.

These days most of the web services and APIs are developed following [REST](#) architectural styles and as such are called [REST\(ful\)](#). There are myriads of great frameworks available for building web services and APIs following [REST\(ful\)](#) principles and constraints, but in the world of [Scala](#), there is an unquestionable leader: [Akka HTTP](#).

### 8.3 From Spray to Akka Http

Many of you may be familiar with [Akka HTTP](#) by means of its terrific predecessor, very popular [Spray Framework](#). It is still quite widely used in the wild but since last year or so, [Spray Framework](#) is being actively migrated under [Akka HTTP](#) umbrella and is going to be discontinued eventually. As of moment of writing, the latest stable version of [Akka HTTP](#) (distributed as part of beloved [Akka Toolkit](#)) was 2.4.11.

[Akka HTTP](#) stands on shoulders of [Actor Model](#) (provided by [Akka](#) core) and [Akka Streams](#) and as such, fully embraces [reactive programming](#) paradigm. However, please take a note that some parts of the [Akka HTTP](#) are still wearing the experimental label as the migration of the [Spray Framework](#) to the new home is ongoing and some contracts very likely will change.

### 8.4 Staying on the Server

In general, when we are talking about web services and APIs, there are at least two parties involved: the server (provider) and the client (consumer). Unsurprisingly, [Akka HTTP](#) has the support for both so let us start from the most interesting part, the server-side.

---

**Akka HTTP** offers quite a few layers of the APIs, ranging from pretty **low-level request/response processing** up to the beautiful **DSLs**. Along this part of the tutorial, we are going to use **Akka HTTP server DSL** only, as it is the fastest (and prettiest) way to start building your **REST(ful)** web services and APIs in **Scala**.

### 8.4.1 Routes and Directives

In the core of the **Akka HTTP** servers is **routing** which in context of **HTTP**-based communications could be described as the process of selecting best paths to handle incoming requests. **Routes** are composed and described using **directives**: the building blocks of **Akka HTTP** server-side **DSL**.

For example, let us develop a simple **REST(ful)** web API to manage users, somewhat functionally similar to the one we have done in **Web Applications with Play Framework** section. Just to remind, here is how our `User` case class looks like (not surprisingly, it is the same one we have used everywhere):

```
case class User(id: Option[Int], email: String,
  firstName: Option[String], lastName: Option[String])
```

Probably, the first route we may need our web API to handle is to return the list of all users, so let it be our starting point:

```
val route: Route = path("users") {
  pathEndOrSingleSlash {
    get {
      complete(repository.findAll)
    }
  }
}
```

As simple as that! Every time the client will issue GET request to the `/users` endpoint (as per `path("users")` and `get` directives), we are going to fetch all the users from underlying data storage and return them back (as per `complete` directive). In reality, routes could be quite complex, requiring extraction and validation of different parameters but luckily, **routing DSL** has all that built-in, for example:

```
val route: Route = pathPrefix("users") {
  path(IntNumber) { id =>
    get {
      rejectEmptyResponse {
        complete(repository.find(id))
      }
    }
  }
}
```

Let us take a closer look on this code snippet. As you may already guess, we are providing a web API to retrieve a user by its integer identifier, which is supplied as **URI** path parameter (using `path(IntNumber)` directive), for example `/users/101`. However, the user with such identifier may or may not exist (generally speaking, `repository.find(id)` returns `Option[User]`). In this case, the presence of the `rejectEmptyResponse` directive instructs our endpoint to return a **404 HTTP status code** (rather than empty response) in case user was not found in the data storage.

Looks really easy and concise, but curious readers may be wondered what data format will be used to represent the users?

### 8.4.2 Marshalling and Unmarshalling

**Akka HTTP** uses the process of marshallng and unmarshalling to convert objects into the representation, transferable over the wire. But practically, most of the time when **REST(ful)** web APIs concerned, we are talking about **JSON** format and **Akka HTTP** has a superior support for that.

```
import akka.http.scaladsl.marshallers.sprayjson.SprayJsonSupport._
import spray.json.DefaultJsonProtocol._

implicit val userFormat = jsonFormat4(User)
```

The use of predefined `jsonFormat4` function introduces implicit support for marshalling `User` case class into **JSON** representation and unmarshalling it back to `User` from **JSON**, for example:

```
val route: Route = pathPrefix("users") {
  pathEndOrSingleSlash {
    post {
      entity(as[User]) { user =>
        complete(Created, repository.insert(user))
      }
    }
  }
}
```

Looks not bad so far but **REST(ful)** web API practitioners may argue that semantics of `POST` action should include `Location` header to point out to the newly created resource. Let us fix that, it would require to restructure the implementation just a bit, introducing `onSuccess` and `respondWithHeader` directives.

```
val route: Route = pathPrefix("users") {
  pathEndOrSingleSlash {
    post {
      entity(as[User]) { user =>
        onSuccess(repository.insert(user)) { u =>
          respondWithHeader(Location(uri.withPath(uri.path / u.id.mkString))) {
            complete(Created, u)
          }
        }
      }
    }
  }
}
```

Excellent, last but not least, there is a dedicated support for **streaming the HTTP responses** in **JSON** format, particularly when **Akka Streams** are engaged. In the **“Database Access with Slick”** section we have already learned how to stream the results from data store using awesome **Slick** library, so this code snippet should look very familiar.

```
class UsersRepository(val config: DatabaseConfig[JdbcProfile]) extends Db with UsersTable {
  ...
  def stream(implicit materializer: Materializer) = Source
    .fromPublisher(db.stream(
      users.result.withStatementParameters(fetchSize = 10)))
  ...
}
```

The `Source[T, _]` could be directly returned from `complete` directive, assuming that `T` (which in our case is `User`) has implicit support for **JSON** marshalling, for example:

```
implicit val jsonStreamingSupport = EntityStreamingSupport.json()

val route: Route = pathPrefix("users") {
  path("stream") {
    get {
      complete(repository.stream)
    }
  }
}
```

Although it may look no different from other code snippets we have seen so far, in this case **Akka HTTP** is going to use **chunked transfer encoding** to deliver the response.

Also, please notice that **JSON** support is currently in transitioning phase and still resides in the **Spray Framework** packages. Hopefully, the eventual **Akka HTTP** abstractions would be very similar and migration would be just a matter of package change (fingers crossed).

### 8.4.3 Directives in Depth

Directives in **Akka HTTP** are able to do mostly everything with request or response, and there is a really impressive list of **predefined ones**. In order to understand the mechanics better, we are going to take a look on two more examples: logging and security.

The `logRequestResult` directive becomes of great help if you need to troubleshoot your **Akka HTTP** web APIs by inspecting the complete snapshots of the requests and responses.

```
val route: Route = logRequestResult("user-routes") {
  ...
}
```

Here just a quick illustration of how this information may look like in the log output when a POST request is issued against `/users` endpoint:

```
[akka.actor.ActorSystemImpl(akka-http-webapi)] user-routes: Response for
Request : HttpRequest(HttpMethod(POST),https://localhost:58080/users,List(Host: localhost ←
:58080, User-Agent: curl/7.47.1, Accept: */*, Timeout-Accept: ),HttpEntity.Strict( ←
application/json,{"email": "a@b.com"}),HttpProtocol(HTTP/1.1))
Response: Complete(HttpResponse(200 OK,List(),HttpEntity.Strict(application/json,{"id":1, ←
"email": "a@b.com"}),HttpProtocol(HTTP/1.1)))
```

Please take a note that all request and response headers along with the payload are included. In case there is a sensitive information passed around (like passwords or credentials), it would be a good idea to implement some kind of filtering or masking on top.

Another interesting example is related to securing your **Akka HTTP** web APIs using `authenticateXxx` directives family. At the moment, there are only two authentication flows supported: **HTTP Basic Auth** and **OAuth2**. As an example, let us introduce yet another endpoint to our web API to allow user modification (typically done using PUT request).

```
val route: Route = pathPrefix("users") {
  path(IntNumber) { id =>
    put {
      authenticateBasicAsync(realm = "Users", authenticator) { user =>
        rejectEmptyResponse {
          entity(as[UserUpdate]) { user =>
            complete(
              repository.update(id, user.firstName, user.lastName) map {
                case true => repository.find(id)
                case _ => Future.successful(None)
              }
            )
          }
        }
      }
    }
  }
}
```

This endpoint is protected using **HTTP Basic Auth** and would verify if user's email already exists in the data storage. For simplicity, the password is always hard-coded to "password" (please never do that in real applications).

```
def authenticator(credentials: Credentials): Future[Option[User]] = {
  credentials match {
    case p @ Credentials.Provided(email) if p.verify("password") =>
      repository.findByEmail(email)
    case _ => Future.successful(None)
  }
}
```

Really nice, but probably the best part of the **Akka HTTP** design is extensibility built right into the core: it is very easy to introduce your own directives in case there is nothing predefined to satisfy your needs.

### 8.4.4 When things go wrong

For sure, most of the time your web APIs are going to work perfectly fine, servicing happy clients. However, from time to time bad things happen and it is better to be prepared to deal with them. Essentially, there could be many reasons for failure: violation of business constraints, database connectivity, unavailability of the external dependencies, garbage collection, ... Under the hood we could classify them into two different buckets: exceptions and execution duration.

In light of the web API we are developing, the typical example of exceptional situation would be the creation of the user with duplicate email address. On the data storage level it would lead to unique constraint violation and needs a specific treatment. As you may expect, there is a dedicated directive for that, `handleExceptions`, for example:

```
val route: Route = pathPrefix("users") {
  pathEndOrSingleSlash {
    post {
      handleExceptions(exceptionHandler) {
        extractUri { uri =>
          entity(as[User]) { user =>
            onSuccess(repository.insert(user)) { u =>
              complete(Created, u)
            }
          }
        }
      }
    }
  }
}
```

The `handleExceptions` directive accepts an `ExceptionHandler` as an argument which maps the particular exception to response. In our case, it would be `SQLException` mapped to `HTTP` response code 409, indicating conflict.

```
val exceptionHandler = ExceptionHandler {
  case ex: SQLException => complete(Conflict, ex.getMessage)
}
```

That's great, but what about execution time? Luckily, `Akka HTTP` supports a variety of `different timeouts` to protect your web APIs. Request timeout is one of those which allows to limit the maximum amount of time a route may take to return a response. If this timeout is exceeded, `HTTP` response code 503 will be returned, signaling that web API is not available at the moment. All these timeouts could be configured `using global settings` in the `application.conf` or using `timeout directives`, for example:

```
val route: Route = pathPrefix("users") {
  pathEndOrSingleSlash {
    post {
      withRequestTimeout(5 seconds) {
        ...
      }
    }
  }
}
```

The ability to use such a fine-grained control is a really powerful option as not all web APIs are equally important and the execution expectations may vary.

### 8.4.5 Startup / Shutdown

`Akka HTTP` uses `Akka`'s extension mechanisms and provides an extension implementation for `Http` support. There are quite a few ways to initialize and use it but the simplest one would be using `bindAndHandle` function.

```
implicit val system = ActorSystem("akka-http-webapi")
implicit val materializer = ActorMaterializer()
```

```
val repository = new UsersRepository(config)
Http().bindAndHandle(new UserApi(repository).route, "0.0.0.0", 58080)
```

The shutdown procedure is somewhat tricky but in the nutshell involves two steps: unbinding `Http` extension and shutting down the underlying actor system, for example:

```
val f = Http().bindAndHandle(new UserApi(repository).route, "0.0.0.0", 58080)
f.flatMap(_._unbind) onComplete {
  _ => system.terminate
}
```

Probably you would rarely encounter a need to use programmatic termination however it is good to know the mechanics of that nonetheless.

### 8.4.6 Secure HTTP (HTTPS)

Along with plain `HTTP`, `Akka HTTP` is ready for production deployments and supports secure communication over `HTTPS` as well. Similarly to what we have learned in “[Web Applications with Play Framework](#)” section of the tutorial, we would need a `Java Key Store` with certificate(s) imported into it. For the development purposes, generating `self-signed certificates` is good enough option to start with:

```
keytool -genkeypair -v
  -alias localhost
  -dname "CN=localhost"
  -keystore src/main/resources/akka-http-webapi.jks
  -keypass changeme
  -storepass changeme
  -keyalg RSA
  -keysize 4096
  -ext KeyUsage:critical="keyCertSign"
  -ext BasicConstraints:critical="ca:true"
  -validity 365
```

However, configuring `server-side HTTPS support` requires quite a bit of code to be written, luckily it is very well `documented`. The `SslSupport` trait serves as an example of such a configuration.

```
trait SslSupport {
  val configuration = ConfigFactory.load()
  val password = configuration.getString("keystore.password").toCharArray()

  val https: HttpsConnectionContext =
    managed(getClass.getResourceAsStream("/akka-http-webapi.jks")).map { in =>
      val keyStore = KeyStore.getInstance("JKS")
      keyStore.load(in, password)

      val keyManagerFactory = KeyManagerFactory.getInstance("SunX509")
      keyManagerFactory.init(keyStore, password)

      val tmf = TrustManagerFactory.getInstance("SunX509")
      tmf.init(keyStore)

      val sslContext = SSLContext.getInstance("TLS")
      sslContext.init(keyManagerFactory.getKeyManagers, tmf.getTrustManagers,
        new SecureRandom)

      ConnectionContext.https(sslContext)
    }.opt.get
}
```

Now we can use `https` connection context variable to create a `HTTPS` binding by passing it to `bindAndHandle` function as an argument, for example:

```
Http().bindAndHandle(new UserApi(repository).route, "0.0.0.0", 58083,
  connectionContext = https)
```

Please notice that you may have **HTTP** support, **HTTPS** support or both at the same time, **Akka HTTP** gives you a full freedom in making these kinds of choices.

### 8.4.7 Testing

There are many different strategies of how to approach web APIs testing. It comes with no surprise that as every other **Akka** module, **Akka HTTP** has dedicated test scaffolding, seamlessly integrated with **ScalaTest** framework (unfortunately **specs2** is not supported out of the box yet).

Let us start with very simple example to make sure that `/users` endpoint is going to return an empty list of users when GET request is sent.

```
"Users Web API" should {
  "return all users" in {
    Get("/users") ~> route ~> check {
      responseAs[Seq[User]] shouldEqual Seq()
    }
  }
}
```

Very simple and easy, the test case looks close to perfection! And those test cases are amazingly fast to execute because they are run against route definition, without bootstrapping the full-fledged **HTTP** server instance.

A bit more complicated scenario would be to develop a test case for user modification, which requires user to be created before as well as **HTTP Basic Authentication** credentials to be passed.

```
"Users Web API" should {
  "create and update user" in {
    Post("/users", User(None, "a@b.com", None, None)) ~> route ~> check {
      status shouldEqual Created
      header[Location] map { location =>
        val credentials = BasicHttpCredentials("a@b.com", "password")
        Put(location.uri, UserUpdate(Some("John"), Some("Smith"))) ~>
          addCredentials(credentials) ~> route ~> check {
            status shouldEqual OK
            responseAs[User] should have {
              'firstName (Some("John"))
              'lastName (Some("Smith"))
            }
          }
      }
    }
  }
}
```

Certainly, if you are practicing **TDD** or derivative methodologies (and I truly believe everyone should), you would enjoy the robustness and conciseness of the **Akka HTTP test scaffolding**. It feels just done right.

## 8.5 Living as a Client

I hope at this point we truly appreciated the powerful server-side support which **Akka HTTP** provides for **REST(ful)** web services and APIs development. But often the web APIs we built themselves become the clients for other external web services and APIs.

As we already mentioned before, **Akka HTTP** has excellent client-side support for communicating with external **HTTP**-based web services and APIs. Similarly to the server-side, there are **multiple levels of client APIs** available, but probably the easiest one to use is **request-level API**.

```
val response: Future[Seq[User]] =  
  Http()  
    .singleRequest(HttpRequest(uri = "https://localhost:58080/users"))  
    .flatMap { response => Unmarshal(response).to[Seq[User]] }
```

In this simple example there is just a single request issues to our `/users` endpoint and results are unmarshalled from **JSON** to `Seq[User]`. However, in most real-world applications it is costly to pay the price of establishing **HTTP** connections all the time so using the **connection pools** is a preferred and effective solution. It is good to know that **Akka HTTP** also has all the necessary building blocks to cover these scenarios as well, for example:

```
val pool = Http().superPool[String]()  
  
val response: Future[Seq[User]] =  
  Source.single(HttpRequest(uri = "https://localhost:58080/users") -> uuid)  
    .via(pool)  
    .runWith(Sink.head)  
    .flatMap {  
      case (Success(response), _) => Unmarshal(response).to[Seq[User]]  
      case _ => Future.successful(Seq[User]())  
    }
```

The end results of those two code snippets are exactly the same. But the latter one uses pool and shows one more time that **Akka Streams** are consistently supported by **Akka HTTP** and are quite handy in processing client-side **HTTP** communications. One important detail though: once finished, the **connection pools** should be shutdown manually, for example:

```
Http().shutdownAllConnectionPools()
```

In case the client and server communication relies on **HTTPS** protocol, **Akka HTTP** client-side API **supports TLS encryption** out of the box.

## 8.6 Conclusions

If you are building or thinking about building **REST(ful)** web services and APIs in **Scala** definitely give **Akka HTTP** a try. Its **routing DSL** is beautiful and elegant way of describing the semantic of your **REST(ful)** resources and plugging the implementation to service them. In comparison to **Play Framework**, there are quite a few overlaps indeed but for pure web API projects **Akka HTTP** is definitely a winner. The presence of some experimental modules may impose a few risks in adopting **Akka HTTP** right away however every release is getting closer and closer to deliver stable and concise contracts.

## 8.7 What's next

It is really sad to admit, but our tutorial is getting to its end. But I honestly believe that for most of us it becomes just a beginning of the exciting journey into the world of **Scala** programming language and ecosystem. Let the joy and success be with you along the way!

The complete source code is available for [download](#).