Modern Web Development with Scala

Denis Kalinin

Modern Web Development with Scala

A concise step-by-step guide to the Scala ecosystem

Contents

Preface

As a language, Scala emerged in 2003, but at that time it was known only to a limited circle of computer scientists and enthusiasts. A couple of years passed and the books started to appear. These were the books that concentrated mostly on Scala syntax and possibly the standard library. Then, in mid-2008, Twitter rewrote some of its core functionality in Scala. Foursquare, LinkedIn and Tumblr soon followed. All of a sudden, people realized that Scala is not a toy for enthusiasts, but a technology that can benefit today's businesses by allowing them to build high-quality and performant software.

Fast-forward to 2016. Lots of libraries, tools, frameworks written in Scala have appeared. People are not surprised anymore when they see another Scala success story on InfoQ. Recruiters and companies are actively looking for Scala developers and in Silicon Valley many big data start-ups don't even question the choose of the language defaulting to Scala as the obvious choice.

And yet, the majority of public appears to be living in 2004. It sounds crazy, but people still perceive Scala as one of many niche languages and express doubts about its *future*. As someone who spent last two years actively using Play and Akka for building Web apps, I can say that this future is already here. In fact, it's been here at least since the Scala 2.9 release in 2011. While Java stays a high-performant but relatively low-level language, Scala allows you to use very high-level syntax constructs without compromising on performance.

I decided to write this book to show how Scala is *used*. It is not about syntax or functional programming but about applying Scala for building real-world Web applications. First, we will learn just enough basics to get started. Then, we will learn some functional programming concepts that are used in the rest of the book. Finally, we will look at build tools and start using Play framework for developing a simple Web app. I'm not going to overwhelm you with sophisticated one-page long code samples but rather show a little bit of everything. After reading this book, you will be familiar with most aspects of Web development in Scala from database access to user authentication. You will also get a pretty good understanding how to integrate Play with modern frontend tools such as Webpack and React.

This book is intended for people who only want to start writing Web apps in Scala, so I don't expect any prior Scala experience. However, I do expect that the reader is familiar with basic Web concepts such as HTML, CSS, JavaScript and JSON. I also assume that the reader knows well at least one modern programming language like Java, C#, Ruby or Python.

The book should be read from start to finish, because later more advanced topics are always based on simpler ones explained earlier. The reader is welcome to try any examples and follow the app development, but it should also be possible to follow the narrative without a text editor.

Let's get started!

Before we begin

If you want to simply read a book in one go without trying to roll out an app of your own, you don't need any preparations. However, even if this is the case, I would still recommend flicking through this section, because it provides a good overview of what tools you can use to write Scala code and how to set them up.

Setting up your environment

I assume that you are already familiar with your operating system and its basic commands. I personally prefer using Ubuntu for Web development and if you want to give it a try, you may consider installing VirtualBox and then installing Ubuntu on top of it. If you want to follow this path, here are my suggestions:

- Download VirtualBox from https://www.virtualbox.org/[1] and install it
- Create a new virtual machine, give it about 2GB of RAM and attach 16GB of HDD (dynamically allocated storage)
- Download a lightweight Ubuntu distribution such as Xubuntu or Lubuntu. I wouldn't bother with 64bit images, plain i386 ones will work just fine. As for versions, 15.10 definitely works in latest versions of VirtualBox without problems, 15.04 probably will too.
- After installing Ubuntu, install Guest Additions, so that you will be able to use higher resolutions on a virtual machine.

By the way, according to StackOverflow surveys[2] about 20% of developers consistently choose Linux as their primary desktop operating system.

Installing Java Development Kit

Scala compiles into Java bytecode, so you will need to install the Java Development Kit (JDK) to run Scala programs. Fortunately, installing JDK is straightforward, just follow these steps:

[1]https://www.virtualbox.org/wiki/Downloads
[2]http://stackoverflow.com/research/developer-survey-2015#tech-os

- Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html[3] and download the latest JDK 8 (the full name will look something like “8u51”, which means “version 8, update 51”). If you are on Linux, I recommend downloading a tar.gz file;
- On Windows, just follow the instructions of the installer. On Linux, extract the contents of the archive anywhere (for example, to ~/DevTools/java8);
- Add the `bin` directory of JDK to your `PATH`, so that the following command works from any directory on your machine:

```
$ java -version
java version "1.8.0_51"
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
```

- Create an environment variable `JAVA_HOME` and point it to the JDK directory. On Linux, you can accomplish both goals if you add to your `~/.bashrc` the following:

```
JAVA_HOME=/home/user/DevTools/jdk1.8.0_51
PATH=$JAVA_HOME/bin:$PATH
```

Strictly speaking, Scala 2.11 will work perfectly fine with Java 7. However, Play starting with version 2.4 supports only Java 8, so we will stick to this version.

Installing Scala

You don’t need the Scala distribution in order to write Play applications, but you may want to install it to try some ideas in the interpreter (or REPL – read-eval-print loop). Just follow these steps:

- Get Scala binaries from http://www.scala-lang.org/download/[4]
- Extract the contents of the archive anywhere (for example, to `~/DevTools/scala`)
- Add the `bin` directory of the Scala distribution to your path:

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[4] http://www.scala-lang.org/download/

```
JAVA_HOME=/home/user/DevTools/jdk1.8.0_51
SCALA_HOME=/home/user/DevTools/scala-2.11.0
PATH=$JAVA_HOME/bin:$SCALA_HOME/bin:$PATH
```

- Start the interpreter in the interactive mode by typing `scala`.

Using REPL

When you type `scala` without arguments in the command line, Scala will start the interpreter in the interactive mode and greet you with a REPL prompt. REPL stands for Read-Eval-Print loop and it is a great way to learn new features or try out some ideas. You can start by typing

```
scala> println("Hello Scala")
```

and then you can press `Enter` and observe the output.

```
Hello Scala
```

Since the `println` function prints a message in `stdout` and the REPL session is hooked to `stdout`, you see the message in the console. Another option is to simply type

```
scala> "Hello World"
```

The REPL will respond by printing

```
res1: String = Hello World
```

in the console. The REPL evaluates an expression, determines its type and, if you haven't assigned it to anything, assigns it to an automatically-generated value.

By the way, The Scala Worksheet plugin in Atom works by intercepting `stdout` of the REPL and presenting its output as a comment in the editor area.

Feel free to come back to REPL later, when you learn more about the language, but for now here is the list of commands that you may find particularly useful:

`:quit`	exits the REPL
`:reset`	resets the REPL to its initial state
`:cp`	adds a specified library to the classpath
`:paste`	enters the `paste` mode, which allows you to define multiple things at once
`:history`	shows the list of previously typed expressions
`:help`	shows the list of available commands

Using IntelliJ IDEA

IntelliJ IDEA[5] accompanied with the official Scala plugin is the best way to write Scala code in 2016. And the good news is that the Community edition, which is free and open-source, will work just fine.

When installing IntelliJ there are several not-so-obvious things worth mentioning.

First, Windows versions come with an embedded JRE, while Linux versions use the already installed JDK, so if you have Java 8 installed, you may see messages like this when IDEA starts:

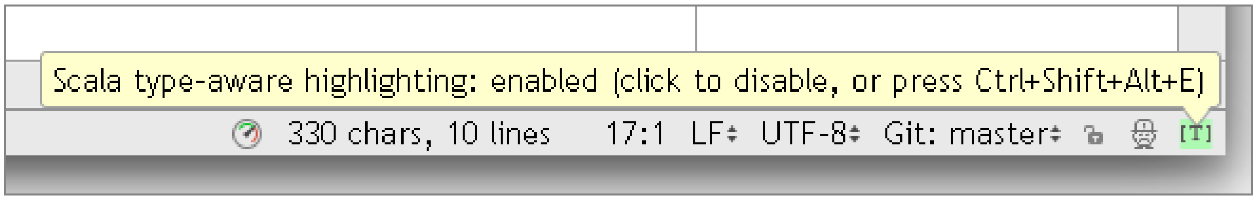
```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=350m; sup\
port was removed in 8.0
```

You can safely ignore them for now. By the way, you can use the `idea.vmoptions` file that reside in the `bin` directory to tweak, well, VM options. Two properties which are often tweaked are:

```
-Xms128m
-Xmx2048m
```

Here `-Xms` specifies the size of the initial memory pool and `-Xmx` specifies the maximum. And if you want to learn more about them, start with this question on StackOverflow[6].

Sometimes IntelliJ goes crazy and starts highlighting valid parts of your code as errors. When it happens, there are two ways to fix the IDE. The simplest is to deactivate "Scala type-aware highlighting" and then activate it again. This helps in many cases.


Scala type-aware highlighting: enabled (click to disable, or press Ctrl+Shift+Alt+E)

Scala type-aware highlighting toggle

A more radical solution is to click "File -> Invalidate Caches / Restart…". IDEA will have to reindex everything after restart, but sometimes it could be the only way to regain its senses.

[5] https://www.jetbrains.com/idea/
[6] http://stackoverflow.com/questions/14763079/

Using Atom Editor

Even though I strongly believe that using a full-blown IDE when working with Scala significantly increases your productivity, it's also worth mentioning a slightly more lightweight alternative - the Atom Editor. Atom is created by Github, based on such technologies as V8 and React and at the moment it's becoming more and more popular.

The official site is atom.io[7], but if you're using Ubuntu, you may want to install it from a PPA repository. Just add `ppa:webupd8team/atom` to the list of repositories via "Software & Updates" or, if you prefer command-line, simply type:

```
$ sudo add-apt-repository ppa:webupd8team/atom
$ sudo apt-get update
$ sudo apt-get install atom
```

There are many packages written for Atom and for trying examples from this book you may want to install the following:

- `language-scala` - adds syntax highlighting for Scala
- `scala-worksheet-plus` - enables the worksheet mode, which executes your code in the Scala interpreter behind the scenes and shows results as comments
- `react` - adds support for React including JSX files

Just go to "Packages -> Settings View -> Install Packages/Themes".

Why use Atom Editor?

Atom is based on some relatively heavy technologies, so it's not as lightweight as say Notepad++[8]. Moreover, it doesn't help you much when it comes to adding package imports and in this respect it's inferior to tools like IntelliJ IDEA, so why use it at all?

Well, there are several situations when you may actually prefer Atom to IntelliJ:

- When you browse the contents of someone else's project, often you want to get the overview of its directory structure and possibly quickly edit some files. In this case, you could simply type "`atom .`" and start hacking in a couple of seconds. IntelliJ would take at least half a minute to load and then it would require you to import the project, which may take a couple of minutes even on decent hardware.

[7] http://atom.io
[8] https://notepad-plus-plus.org/

- Everything related to frontend is usually top quality in Atom. At the same time, full support of JavaScript and CSS is considered a paid feature in IntelliJ. The Community version will paint HTML markup in different colors and even highlight JS files, but if you want support for JSX templates or CSS preprocessors, you should start looking at the Ultimate version or switch to Atom.
- Lastly, I found the Scala Worksheet plugin in Atom to be more stable than its counterparts in IntelliJ or Eclipse. Besides, even if you encounter a bug in a plugin, all source code is in `~/.atom`, so it's usually fixable.

I suggest you use Atom for the first parts of the book and then switch to IntelliJ for working with Play. Please refer to Appendix B for additional instructions on importing Scala projects in IntelliJ.

Language fundamentals

In this section, I will demonstrate the basics of the Scala programming language. This includes things that are common to most programming languages as well as several features which are specific to Scala. I'm going to provide the output of the interpreter here, but feel free to start you own REPL session and try it yourself.

Defining values

There are three main keywords for defining everything in Scala:

`val`	defines a constant (or value)
`var`	defines a variable, very rarely used
`def`	defines a method

By the way, there are several ways to define a function in Scala and using `def` is only one of them. Read on!

Defining a constant is usually as simple as typing

```
scala> val num = 42
num: Int = 42
```

Notice that Scala guessed the type of the constant by analyzing its value. This feature is called *type inference*.

Sometimes you want to specify the type explicitly. If in the example above you want to end up with a value of type `Short`, simply type

```
scala> val num: Short = 42
num: Short = 42
```

Variables are rarely needed in Scala, but they are supported by the language and could be useful sometimes:

```
scala> var hello = "Hello"
hello: String = Hello
```

Just as with types you may explicitly specify the type or allow the type inferrer to do its work.

As we will see later, all statements in Scala evaluate to some value. This feature essentially allows you to write code using `val`s and resort to `var`s only occasionally.

When defining methods it is required to specify argument types. Specifying the return type is recommended but optional:

```
scala> def greet(name: String) = "Hello" + name
greet: (name: String)String
```

It's worth mentioning that the body doesn't require curly braces and could be placed on the same line as the signature. At the same time, relying on the type inferrer for guessing the return type is not recommended for a number of reasons:

- If you rely on the type inferrer, it basically means that the return type of your function depends on its implementation, which is dangerous as implementation is often changed;
- The readers of your code don't want to spend time guessing the resulting type.

This is particularly true when you work on public APIs.

The return type can be specified after the parentheses:

```
scala> def greet(name: String): String = "Hello " + name
greet: (name: String)String
```

The value that is returned from the method is the last expression of its body, so in Scala you don't need to use the `return` keyword. Remember, however, that the assignment operator doesn't return a value.

If you want to create a procedure-like method that doesn't return anything and is only invoked for side effects, you should specify the return type as `Unit`. This is a Scala replacement for `void` (which, by the way, is not even a keyword in Scala):

```
scala> def greet2(): Unit = println("Hello World")
greet2: ()Unit
```

Not so long ago, the official documentation recommended omitting the equals sign when defining procedure-like methods. Not anymore! Starting with version 2.10, the recommendation is to use the equals sign, always. Check this StackOverflow question[9] or the docs[10].

It's also possible to define a method without parentheses, in which case it is called a *parameterless* method:

```
scala> def id = Math.random
id: Double

scala> id
res10: Double = 0.15449866168176285
```

This makes the invocation of such a method look like accessing a variable/constant and supports the uniform access principle[11]. If you want to define your method like this, you should ensure that it doesn't have side effects, otherwise it will be extremely confusing for users.

Functional types

It's possible to define a function and assign it to a variable (or constant). For example, the `greet` method from the example above could also be defined the following way:

```
scala> var greetVar: String => String = (name) => "Hello " + name
greetVar: String => String = <function1>
```

As REPL output shows, `String => String` is the type of our function. It can be read as *a function from String to String*. Again, you don't have to type the return type, but if you want the type inferrer to do the work, you'll have specify the type of parameters:

```
scala> var greetVar = (name: String) => "Hello " + name
greetVar: String => String = <function1>
```

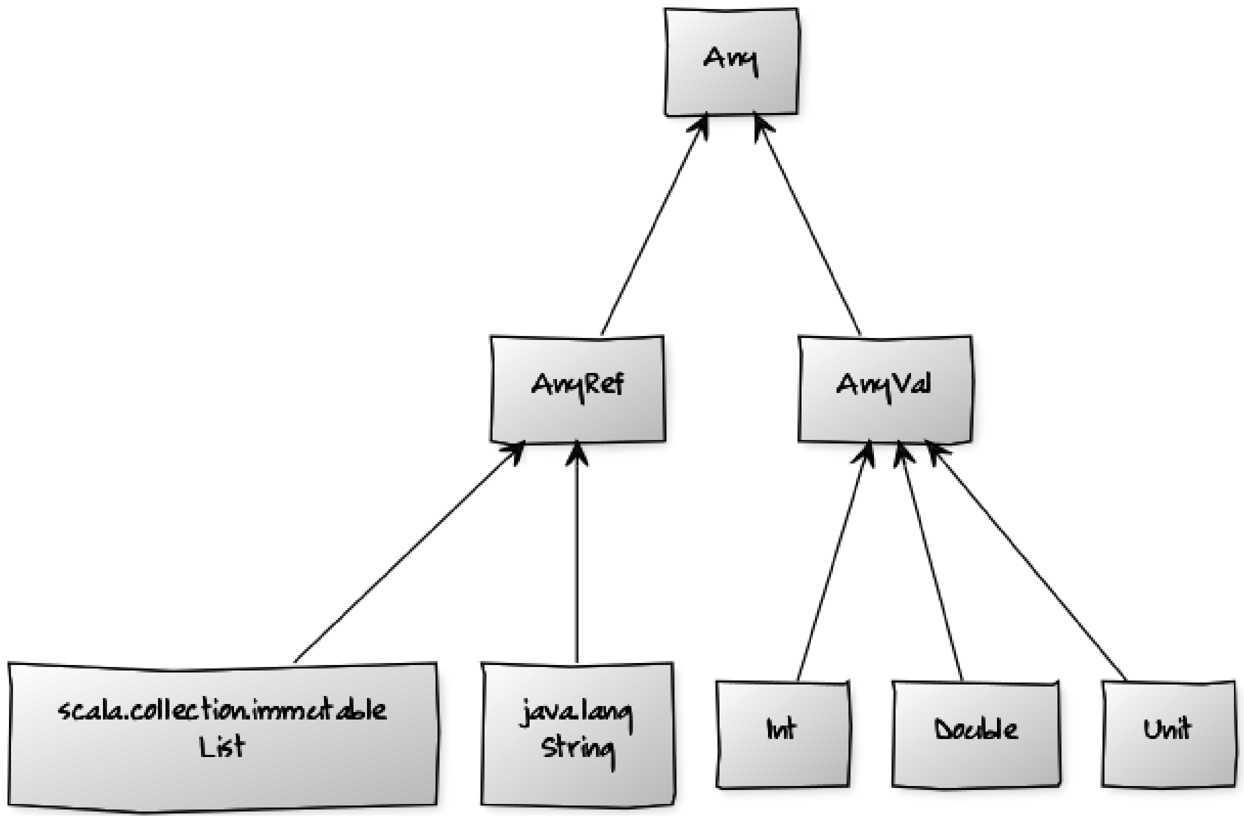
Assigning methods to variables is also possible, but requires using special syntax to tell the compiler that you want to *reference* the function rather than *call* it:

[9] http://stackoverflow.com/questions/944111/
[10] http://docs.scala-lang.org/style/declarations.html#procedure_syntax
[11] http://stackoverflow.com/questions/7600910/

```
scala> val gr = greet _
gr: String => String = <function1>
```

Type hierarchy

Since we are on the topic of types, let's take a look at a very simplified type hierarchy:



Scala type hierarchy - top

Unlike Java, literally everything has its place here. Primitive types inherit from `AnyVal`, reference types (including user-defined classes) inherit from `AnyRef`. The absence of value has its own type `Unit`, which belongs to the primitives group. Even functions have their place in this hierarchy: for example, a function from String to String has type `Function1[String, String]`:

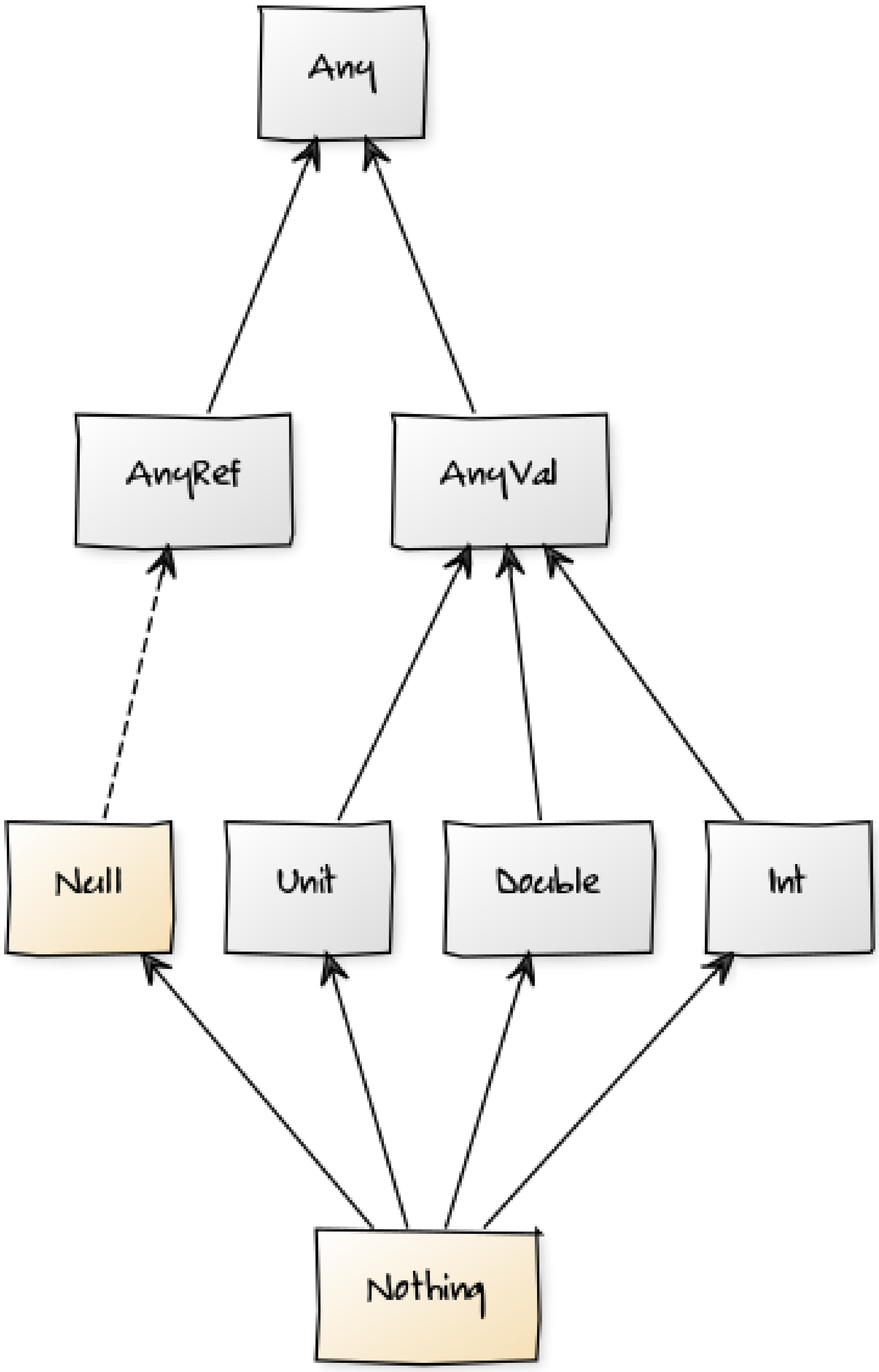
```
scala> val greetFn = (name: String) => "Hello " + name
greetFn: String => String = <function1>

scala> val fn: Function1[String, String] = greetVar
fn: String => String = <function1>
```

You can use `isInstanceOf` to check the type of a value:

```
scala> val str = "Hello"
str: String = Hello

scala> str.isInstanceOf[String]
res20: Boolean = true

scala> str.isInstanceOf[AnyRef]
res21: Boolean = true
```

By the way, another name of `scala.AnyRef` is `java.lang.Object`, so `str.isInstanceOf[Object]` returns `true`

Scala also introduces the concept of so-called *bottom types* which implicitly inherit from all other types:



Scala type hierarchy - bottom types

In the diagram above, `Null` implicitly inherits from all reference types (including user-defined ones, of course) and `Nothing` from all types including primitives and `Null` itself. You are unlikely to use bottom types directly in your programs, but they are useful to understand type inference. More on this later.

Collections

Unlike Java, Scala uses the same uniform syntax for creating and accessing collections. For example, you can use the following code to instantiate a new list and then get its first element:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list(0)
res24: Int = 1
```

For comparison, here is the same code that uses an array instead of a list:

```
scala> val array = Array(1,2,3,4)
array: Array[Int] = Array(1, 2, 3, 4)

scala> array(0)
res25: Int = 1
```

For performance reasons, Scala will map the latter collection to a Java array. It happens behind the scenes and is completely transparent for a developer.

The collections API was completely rewritten in Scala 2.8, which in turn was released in 2009. Even though many regard this rewrite as a significant improvement, some people were quite skeptical back then[12].

Another important point is that Scala collections always have a type. If the type wasn't specified, it will be inferred by analyzing provided elements. In the example above we ended up with the list and array of `Int`s because the elements of these collections looked like integers. If we wanted to have, say, a list of `Short`s, we would have to set the type explicitly like so:

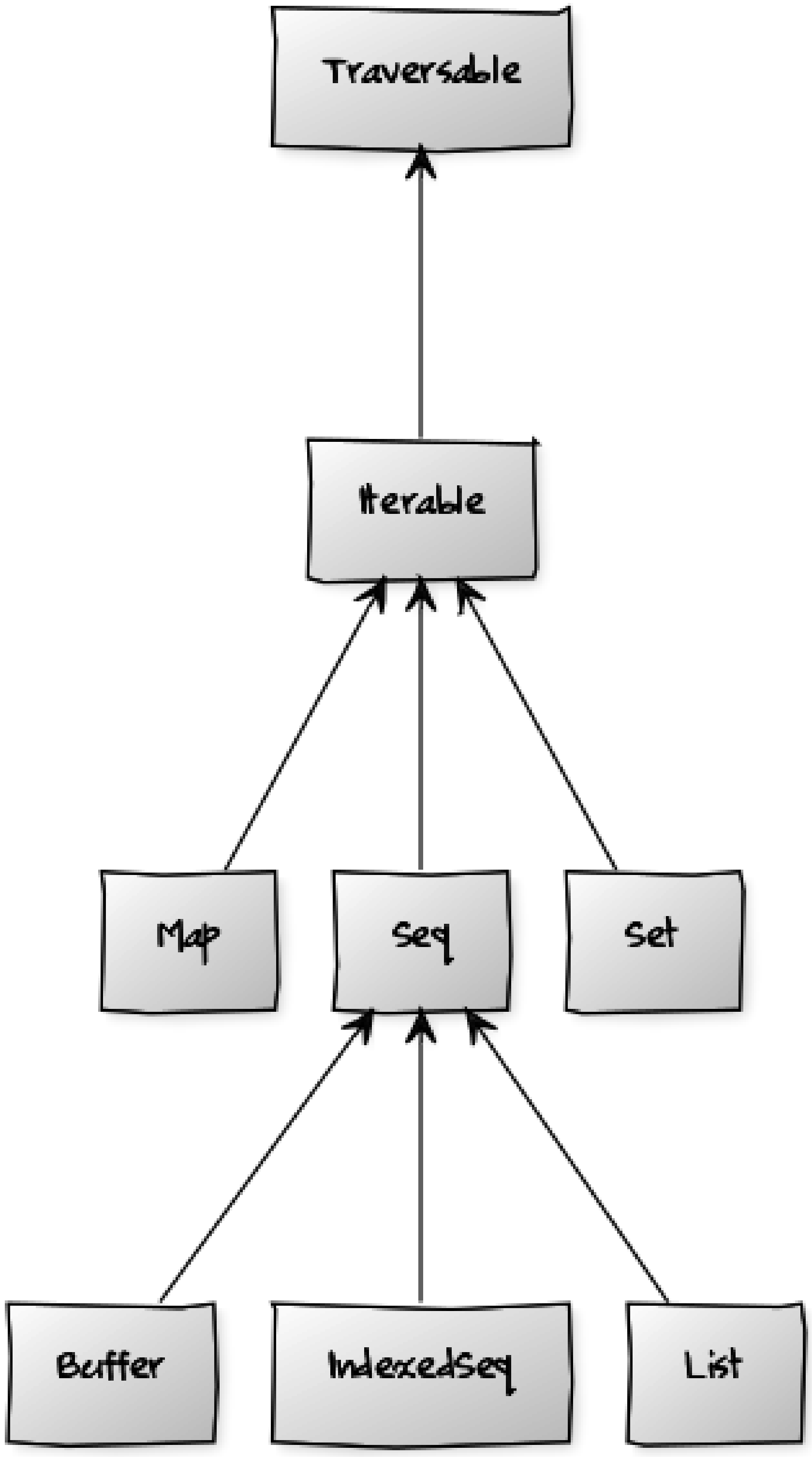
```
scala> val list = List[Short](1,2,3,4)
list: List[Short] = List(1, 2, 3, 4)
```

If the type inferrer cannot determine the type of the collection, compilation will fail:

[12]http://stackoverflow.com/questions/1722726/

```
scala> var list: List = null
<console>:7: error: type List takes type parameters
       var list: List = null
                 ^
```

A greatly simplified Collections hierarchy is shown below:



Collections Hierarchy

It's worth mentioning that most commonly used methods are already defined on very basic collection types. In practice, this means that when you need a collection, usually you can simply use `Seq` and its API will probably be sufficient most of the time.

Collections also come in two flavours - mutable and immutable. Scala defaults to the immutable flavour, so when in the previous example we typed `List`, we actually ended up with the immutable `List`:

```
scala> list.isInstanceOf[scala.collection.immutable.List[Short]]
res29: Boolean = true
```

Immutable collections don't have methods for altering the original collection, but they have methods that return new collections. For example, the `:+` method returns a new list that contains all the elements of the original list and one new element:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list :+ 5
res33: List[Int] = List(1, 2, 3, 4, 5)
```

On the other hand, mutable collections can alter themselves. A great example of a mutable list is `ListBuffer`, which is often used to accumulate values and then build an immutable list from these values:

```
scala> val buffer = scala.collection.mutable.ListBuffer.empty[Int]
buffer: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buffer += 1
res35: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1)

scala> buffer += 2
res36: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2)

scala> buffer.toList
res37: List[Int] = List(1, 2)
```

This is similar to using `StringBuilder` for constructing strings in Java.

Note how array and list elements are accessed by using parentheses. Note also that both collections were created without using the `new` keyword. This syntax is not collection-specific and we will return to this topic after discussing classes and objects.

Packages and imports

Scala classes are organized into packages similarly to Java or C#. For example, the `ListBuffer` type was defined in a package called `scala.collection.mutable` so in order to use it in your program you need to either always type a so-called fully qualified name[13] (FQN for short) or import the class:

```
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer

scala> val buffer = ListBuffer.empty[Int]
buffer: scala.collection.mutable.ListBuffer[Int] = ListBuffer()
```

Just like in Java or C#, we can import all classes from a particular package using a wildcard. The difference is that Scala uses _ instead of `*`:

```
scala> import scala.collection.mutable._
import scala.collection.mutable._

scala> val set = HashSet[Int]()
set: scala.collection.mutable.HashSet[Int] = Set()
```

In addition to the imports defined by a developer, Scala automatically imports the following:

- `java.lang._`, which includes `String`, `Exception` etc.
- `scala._`, which includes `Option`, `Any`, `AnyRef`, `Array` etc.
- `scala.Predef`, which conveniently introduces aliases for commonly used types and functions such as `List`, `Seq`, `println` etc.

Imports in Scala are not restricted to the beginning of the file and can appear inside functions or classes.

Defining classes

Users can define their own types using the omnipresent `class` keyword:

[13] https://en.wikipedia.org/wiki/Fully_qualified_name

```
scala> class Person {}
defined class Person
```

Needless to say, the `Person` class defined above is completely useless. You can create an instance of this class but that's about it:

```
scala> val person = new Person
person: Person = Person@794eeaf8
```

Note that the REPL showed `Person@794eeaf8` as a value of this instance. Why is that? Well, in Scala all user-defined classes implicitly extend `AnyRef`, which is the same as `java.lang.Object`. The `java.lang.Object` class defines a number of methods including `toString`:

```
def toString(): String
```

By convention, the `toString` method is used to create a String representation of the object, so it makes sense that REPL uses this method to print a value of a user-defined class in the console.

A slightly better version of this class can be defined as follows:

```
scala> class Person(name: String) {}
defined class Person
```

By typing `name: String` inside the parentheses we defined a constructor with a parameter of type `String`. Although it is possible to use `name` inside the class body, the class still doesn't have a field to store it. Fortunately, it's easily fixable:

```
scala> class Person(val name: String) {}
defined class Person
```

Much better! Adding `val` in front of the parameter name defines an immutable field. Similarly `var` defines a mutable field. In any case, this field will be initialized with the value passed to the constructor when the object is instantiated. What's interesting, the class defined above is roughly equivalent of the following Java code:

```
class Person {
  private final String mName;
  public Person(String name) {
    mName = name;
  }
  public String name() {
    return mName;
  }
}
```

By the way, if you don't have anything inside the body of your class, you don't need curly braces.

To define a method simply use the `def` keyword inside a class:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
  def apply(): String = name
  def sayHello(): String = "Hi! I'm " + name
}
```

When working with classes and objects in REPL, it is often necessary to type several lines of code at once. Remember that REPL provides the paste mode, which can be activated by the `:paste` command.

In the example above we defined two methods - `apply` and `sayHello` and overrode (thus keyword `override`) existing method `toString`. Of these three, the simplest one is `sayHello` as it only prints a phrase to `stdout`:

```
scala> val joe = new Person("Joe")
joe: Person = Person(Joe)

scala> joe.sayHello()
res49: String = Hi! I'm Joe
```

Notice that REPL used our `toString` implementation to print information about the instance to `stdout`. As for the `apply` method, there are two ways to invoke it.

```
scala> joe.apply()
res50: String = Joe

scala> joe()
res51: String = Joe
```

Yep, using parentheses on the instance of a class actually calls the `apply` method defined on this class. This approach is widely used in the standard library as well as in third-party libraries.

Defining objects

Scala doesn't have the `static` keyword but it does have syntax for defining *singletons*[14]. If you need to define methods or values that can be accessed on a type rather than an instance, use the `object` keyword:

```
object RandomUtils {
  def random100 = Math.round(Math.random * 100)
}
```

After `RandomUtils` is defined this way, you will be able to use method `random100` without creating any instances of the class:

```
scala> RandomUtils.random100
res62: Long = 67
```

You can define the `apply` method on an object and then call it using the name of the object followed by parentheses. One common pattern is to define an object that has the same name as the original class and define the `apply` method with the same constructor parameters as the original class:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
}
object Person {
  def apply(name: String): Person = new Person(name)
}
```

If an object has the same name as a class, then it's called a *companion object*. Companion objects are often used in Scala for defining additional methods and implicit values. You will see a concrete example when we get to serializing objects into JSON.

Now you can use the `apply` method on the object to create instances of class Person:

[14] https://en.wikipedia.org/wiki/Singleton_pattern

```
scala> val person = Person("Joe")
person: Person = Person(Joe)
```

Essentially, this eliminates the need for using the `new` keyword. Again, this approach is widely used in the standard library and in third-party libraries. In later chapters I usually refer to this syntax as calling the *constructor* even though it is actually calling the `apply` method on a companion object.

Type parametrization

Classes can be parametrized, which makes them more generic and possibly more useful:

```
class Cell[T](val contents: T) {
    def get: T = contents
}
```

We defined a class called `Cell` and specified one field `contents`. The type of this field is not yet known. Scala doesn't allow *raw types*, so you cannot simply create a new `Cell`, you must create a `Cell` *of something*. Fortunately, the type inferrer can help with that:

```
scala> new Cell(1)
res71: Cell[Int] = Cell@1c0d3eb6
```

Here we passed `1` as an argument and it was enough for the type inferrer to decide on the type of this instance. We could also specify the type ourselves:

```
scala> new Cell[Short](1)
res73: Cell[Short] = Cell@751fa7a3
```

Collections syntax revisited

Now it's becoming more and more clear that there is absolutely nothing in Scala syntax that is specific to collections. In fact, now we can explain what happens behind the scenes when we are creating and accessing `Lists`:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list(0)
res24: Int = 1
```

First, we can conclude that `List` is a parametrized class that has the `apply` method. This method accepts an index as an argument and returns the element that has this index. Second, there is also an *object* called `List` which has the `apply` method. This method probably calls the `List` constructor to instantiate the actual object. Finally, the `toString` method is overridden to return the word List and its elements in parentheses.

This is actually a recurring idea in Scala design. Rather than introduce specifics, Scala always tries to solve a particular problem using a more universal or generic approach.

Functions and implicits

A parameter of a function can have a default value. If this is the case, users can call the function without providing the value for the argument:

```
scala> def greet(name: String = "User") = "Hello " + name
greet: (name: String)String

scala> greet()
res77: String = Hello User
```

Compiler *sees* that the argument is absent, but it also *knows* that there is a default value, so it takes this value and then invokes the function as usual.

We can go even further and declare the `name` parameter as *implicit*:

```
scala> def greet(implicit name: String) = "Hello " + name
greet: (implicit name: String)String
```

Here we're not setting any default values in the function signature. As a result, in the absence of the argument, Scala will look for a value of type `String` marked as `implicit` and defined somewhere in scope. So, if we try to call this function immediately, it will not work:

```
scala> greet
<console>:9: error: could not find implicit value for parameter name: String
              greet
              ^
```

However, after defining an implicit value with the type `String`, it will:

```
scala> implicit val n = "User"
n: String = User

scala> greet
res79: String = Hello User
```

This code works because there is exactly one value of type `String` marked as `implicit` and defined in the scope. If there were several `implicit` `String`s, the compilation would fail due to ambiguity.

Implicits and companion objects

You can define your own `implicit` values or introduce already defined `implicits` into scope via `import`. However, there is one more place where Scala will look for an `implicit` value when it needs one. This is the companion object of the parameter type. For example, when we are defining a new class called `Person`, we may decide to create a default value and put it into the companion object:

```scala
class Person(val name: String)

object Person {
  implicit val person: Person = new Person("User")
}
```

Let's also create a method with one parameter of type `Person` and mark it as `implicit`:

```scala
def sayHello(implicit person: Person): String = "Hello " + person.name
```

Even if we don't create any instances of class `Person` in the scope, we will still be able to use the `sayHello` method without providing any arguments:

```
scala> sayHello
res0: String = Hello User
```

This works because companion objects are another place where the compiler looks for implicits. The point here is that the type of the method parameter is the same as the name of the companion object.

Of course, we can always pass a regular argument *explicitly*:

```
scala> sayHello(new Person("Joe"))
res1: String = Hello Joe
```

Some people argue that implicits make code unmaintainable. In my experience, however, it has never been the case. In fact, most professional Scala developers agree that implicits actually make code clearer.

Loops and conditionals

Unlike Java with *if-statements*, in Scala *if-expressions* always result in a value. In this respect they are similar to Java's ternary operator `?:`. Let's define a function that uses the if expression to determine whether the argument is an even number:

```
def isEven(num: Int) = {
  if (num % 2 == 0)
    true
  else
    false
}
```

When this method is defined in REPL, the interpreter responds with `isEven: (num: Int)Boolean`. Even though we haven't specified the return type, the type inferrer determined it as the type of the last (and only) expression, which is the *if expression*. How did it do that? Well, by analyzing the types of both branches:

if-branch	`Boolean`
else-branch	`Boolean`
whole expression	`Boolean`

If an argument is an even number, the *if branch* is chosen and the result type has type `Boolean`. If an argument is an odd number, the *else branch* is chosen but result still has type `Boolean`. So, the whole expression has type `Boolean` regardless of the "winning" branch, no rocket science here.

But what if branch types are different, for example `Int` and `Double`? In this case the nearest common supertype will be chosen. For `Int` and `Double` this supertype will be `AnyVal`, so `AnyVal` will be the type of whole expression.

If you give it some thought, it makes sense, because the result type must be able to hold values of both branches. After all, we don't know which one will be chosen until runtime.

Bottom types revisited

Now it's time to recall two types which reside at the bottom of the Scala type hierarchy - `Null` and `Nothing`.

There is only one object of type `Null` - `null` and its meaning is the same as it was in Java or C#: the object is not initialized. Any reference object can be `null`, so it's only logical that any reference type (i.e `AnyRef`) is a supertype of `Null`. This brings us to the following conclusion:

if-branch	`AnyRef`
else-branch	`Null`
whole expression	`AnyRef`

OK, what if the else-branch never returns and instead throws an exception? In this case, the type of the *else-branch* is considered to be `Nothing` and the whole expression will have the type of the *if-branch*.

if-branch	`Any`
else-branch	`Nothing`
whole expression	`Any`

There's not much you can do with bottom types, but they are included in Scala hierarchy and they make the rules that the type inferrer uses more clear.

while loops

The `while` loop is almost a carbon copy of its Java counterpart, so in Scala it looks like an outcast. Instead of returning a value, it's called for a side effect. Moreover, it almost always utilizes a `var` for iterations:

```
var it = 5
while (it > 0) {
  println(it)
  i -= 1
}
```

It feels almost *awkward* to use it in Scala and as a matter of fact you almost never need to. We will look at some alternatives when we get to the Functional programming section but for now here is more Scala-like code that does essentially the same thing:

```
1.to(5).reverse.foreach { num => println(num) }
```

String interpolation

String interpolation was one of the features introduced in Scala 2.10. It allows to execute Scala code inside string literals. In order to use it, simply put `s` in front of a string literal and `$` in front of any variable or value you want to interpolate:

```
val name = "Joe"
s"Hello $name"
```

If an interpolated expression is more complex, e.g contains dots or operators, it needs to be taken in curly braces:

```
val person = new Person("Joe")
s"Hello ${person.name}"
```

Traits

Traits in Scala are similar to mixins in Ruby in a sense that you can use them to add functionality to your classes. Traits can contain both abstract (not implemented) and concrete (implemented) members. If you mix in traits with abstract members then you must either implement them or mark your class as `abstract`, so the Java rule about implementing interfaces still holds.

In Java, a class can implement many interfaces, but if you want to make your class concrete (i.e. allow to instantiate it), you need to provide implementations for all methods defined by all interfaces. Unlike Java interfaces, though, Scala traits can and often do have concrete methods.

Let's look at a rather simplistic example:

```
trait A { def a(): Unit = println("a") }

trait B { def b(): Unit }
```

We defined two traits so that trait `A` has one concrete method and trait `B` has one abstract method (the absence of a body that usually comes after the equals sign means exactly that). If we want to create a new class `C` that inherits functionality from both traits, we will have to implement the `b` method:

```
class C extends A with B { def b(): Unit = println("b") }
```

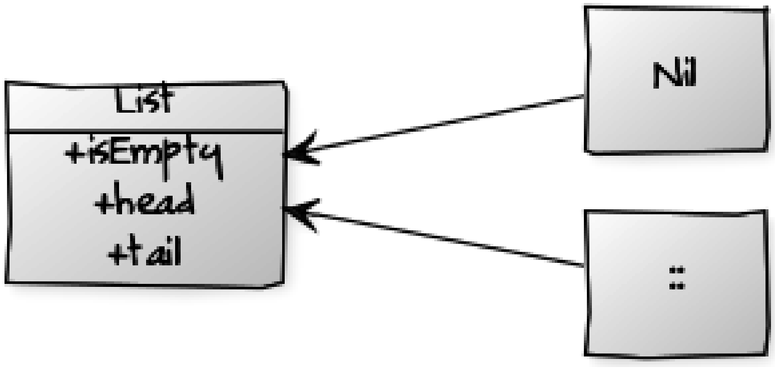
If we don’t implement `b`, we will have to make the `C` class `abstract` or get an error.

Functional programming

In this section we will examine a number of features which support functional programming and are available in Scala. As we will see, all of them are actively used by library designers and developers in real-world applications.

Algebraic data types

Even though it sounds like rocket science, *algebraic data types* (ADT) are actually pretty simple. An ADT is a type formed by combining other types. For example, `List` is an ADT, because it's formed by combining two other types - empty list (represented by singleton object `Nil`) and non-empty list (represented by `::` and pronounced *cons*):



List as an ADT

The important thing here is that the list may be either empty or non-empty, i.e `Nil` and `::` form disjoint sets. In addition to using the `List` object's `apply` method, you can use the following approach to construct a new list:

```
val list = 1 :: 2 :: 3 :: Nil
```

The `::` method is called `prepend` and it's available on all lists including the empty one. In Scala, if a method's name ends with colon (`:`), the argument of this method (when the method invocation is written in the *operator style*) must be placed on left, so the above code transforms into following:

```
val list = Nil.::(3).::(2).::(1)
```

Basically, we start with the empty list `Nil` and then prepend the numbers, so that the resulting list will be `List(1, 2, 3)`.

As we already discussed, there are many illustrations of generic design in Scala. Another example is using methods instead of operators. For instance, the `+` operator in expression `2 + 2` is actually a method `+` defined on class `Int`. The *operator style* syntax we used here is allowed by Scala and in fact is equivalent of `2.+(2)`.

In order to determine the type of the list we could use the `isInstanceOf` method available on all types:

```
list.isInstanceOf[List[Any]]    // true
list.isInstanceOf[Nil.type]     // false
list.isInstanceOf[::[Int]]      // true
```

Note that since Scala doesn't allow raw types, we had to specify `Any` and `Int` even though we weren't interested in such details. On the other hand, `Nil` is defined as object (singleton) and represents the empty list regardless of the element type.

A more Scala-like approach is to use *pattern matching*.

Pattern matching

You can think of pattern matching as Java's `switch` statement on steroids. The main differences are that pattern matching in Scala always returns a value and it is *much* more powerful.

```
list match {
  case Nil => println("empty")
  case ::(head, tail) => println("non-empty")
}
```

The first case block simply checks whether the `list` object is the `Nil` singleton object. The second one is more interesting, though. Not only does it check that the `list` has type `::`, it also extracts `head` (the first element) and `tail` (all elements except the first one). If you want your class to have similar capabilities, you must either provide the `unapply` method on the companion object or turn your class into a `case class`.

Case classes

Remember our `Person` class from the section about objects? Let's revisit that code here:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
}
object Person {
  def apply(name: String): Person = new Person(name)
}
```

Here we're creating a class, defining a field, overriding the `toString` method, defining a companion object. All of this could be achieved by the following line:

```
case class Person(name: String)
```

On top of the goodies mentioned above, this definition also does the following:

- overrides the `hashcode` method used by some collections from the standard library
- overrides the `equals` method, which makes two objects with the same of values equal, so `new Person("Joe") == new Person("Joe")` will return `true`
- defines the `unapply` method used in pattern matching

As a result, even though the word `case` suggests that case classes are somehow related to pattern matching, many developers create them to quickly get hassle-free types with `hashcode`/`equals`/`apply` methods.

Higher-order functions

Functions that accept other functions as their arguments are called *higher-order functions*[15]. They are supported by virtually any modern programming language including JavaScript, Ruby, Python, PHP, Java, C# and so on. Since they are such a common concept, which is probably already familiar to you, we're not going to spend much time discussing the theory behind higher-order functions. Instead we will look at some Scala specifics.

Curly braces instead of parentheses

If your function accepts only one argument, then you are free to call your function using curly braces instead of parentheses:

[15]https://en.wikipedia.org/wiki/Higher-order_function

```
scala> def greet(name: String) = "Hello " + name
greet: (name: String)String

scala> greet{"Joe"}
res74: String = Hello Joe
```

It may not be that useful when the only argument is a `String`, but what if the function accepts another function as the argument? If we had to always use parentheses, there would be to many of them:

```
scala> def invokeBlock(block: () => Unit): Unit = block()
invokeBlock: (block: () => Unit)Unit

scala> invokeBlock( () => println("Inside the block!") )
Inside the block!
```

But since the `invokeBlock` method accepts only one argument, we can use the following syntax:

```
invokeBlock { () =>
  println("Inside the block!")
}
```

The block spreads to several lines and using curly braces makes the code actually look like a DSL (Domain-Specific Language).

map, filter, foreach

Let's create a simple list, that we will use to demonstrate common higher-order functions from the Collections API:

```
val list = List(1, 2, 3, 4)
```

Arguably, the most used method in the entire Collection API is `map`. It accepts a function that will be used to transform collection elements one by one. For example:

```
list.map { el => el * el }              // List(1, 4, 9, 16)
```

Here each element is multiplied by itself. Note that the original list stays untouched because instead of changing it, `map` returns a new list.

Another method is `filter`, which allows to keep only those elements that satisfy the condition:

```
list.filter { el => el % 2 == 0 }   // List(2, 4)
```

Here we created a new collection containing only even numbers.

If we don't need a new collection, but instead we want to perform some action on each element, we can use `foreach`:

```
list.foreach { el => print(el) }    // prints 1234
```

Here we're simply printing each element's value.

flatMap

Let's assume that in addition to the list defined above - `List(1,2,3,4)` - we have another list of strings:

```
val list2 = List("a", "b")
```

What if we wanted to combine each element of the first list with each element of the second list? Let's try to solve this task by nesting two `map`s:

```
list.map { el1 =>
  list2.map { el2 =>
    el1 + el2
  }
}
```

The resulting collection will have the following elements:

```
List[List[String]] = List(List(1a, 1b), List(2a, 2b), List(3a, 3b), List(4a, 4b))
```

We ended up with a list of lists. Not quite what we expected, right? The problem here is the external `map`. If you look at the Scala documentation, you will see that the `map` method accepts a function that takes one element and returns another element. On the other hand, we want to pass a function that takes an element and returns a list. Is there such a function defined on `List`? It turns out that yes, and it's called `flatMap`. Using `flatMap` the problem can be solved easily:

```
list.flatMap { el1 =>
  list2.map { el2 =>
    el1 + el2
  }
}
```

The result is exactly what we needed:

```
List[String] = List(1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b)
```

for comprehensions

In order to simplify writing expressions that use `map`, `flatMap`, `filter` and `foreach`, Scala provides so-called *for comprehensions*. Oftentimes *for comprehensions* make your code clearer and easier to understand.

Let's rewrite the previous examples with `for` expressions:

```
for { el <- list } yield el * el                 // List(1, 4, 9, 16)
for { el <- list if el % 2 == 0 } yield el       // List(2,4)
for { el <- list } println(el)                   // prints 1234
```

The `yield` keyword is used when the whole expression returns some value. With `foreach` we are simply printing values on the screen, thus there's no `yield` keyword.

The `flatMap` example translates into the following:

```
for {
  el1 <- list
  el2 <- list2
} yield el1 + el2
```

In my opinion, the last snippet can be easily understood even by people who don't know what `flatMap` is and even that this function exists.

For comprehensions are the central part of the Scala programming language. They are used in many situations and you will see a lot of examples with the `for` keyword throughout this book.

If you want to use *your* classes in for comprehensions, you must provide at least two methods - `map` and `flatMap`. The other two are necessary only if you plan on filtering and iterating with `for`.

Currying

A *curried function* is a function that has each of the parameters in its own pair of parentheses. The process of transforming a regular function into a curried one is called *currying*[16]. A simple function that sums two integers

```
def sum(a: Int, b: Int): Int = a + b
```

can be transformed into the following curried function:

```
def sum(a: Int)(b: Int): Int = a + b
```

If the function is curried, it must be called with each argument in its own pair of parentheses:

```
sum(2)(2)        // 4
```

foldLeft

The `foldLeft` method from the Collections API is a curried method that takes two parameters. First parameter - the initial value - is used as the staring point of the computation and the second parameter is a function that describes how to compute the next value using the previous one and the accumulator. It may sound terribly complicated, but in reality `foldLeft` is absolutely straightforward to use.

For example, we can use `foldLeft` to calculate the sum of elements in our list - `List(1,2,3,4)`:

```
list.foldLeft(0) { (acc, next) =>
  acc + next
}
```

Here we replaced parentheses around the second argument with curly braces to make it look like a proper function.

Another (more advanced) reason for making `foldLeft` curried is type inference. By looking at the first argument, the type inferrer gets more insight into the types used in the second argument. Check this question on StackOverflow[17] if you want more details.

[16]https://en.wikipedia.org/wiki/Currying
[17]http://stackoverflow.com/questions/4915027/

Laziness

By default, all values in Scala are initialized *eagerly* at the moment of declaration. This approach is widely known and in many programming languages there is no other way. In Scala, however, values can be initialized *lazily*, which means they remain uninitialized until the first use.

Let's create a fictional service that we will use for demonstration:

```
1 class ConnectionService {
2   def connect = println("Connected")
3 }
```

The ConnectionService has only one method. Obviously, the service must be instantiated somewhere before it's used. The following code reminds us that the order of initialization matters:

```
1 var service: Service = null
2 val serviceRef = service
3 service = new Service
4 serviceRef.connect
```

The code will fail at runtime with NullPointerException because when we declared the serviceRef, the service itself hadn't been initialized. Ironically, when we first tried to use it, the service was up and running, so the problem was the order of initialization, not the service being uninitialized. If only we could postpone the initialization of serviceRef! It turns out that in Scala we can do exactly that by putting the lazy keyword in front of the val declaration (line 2).

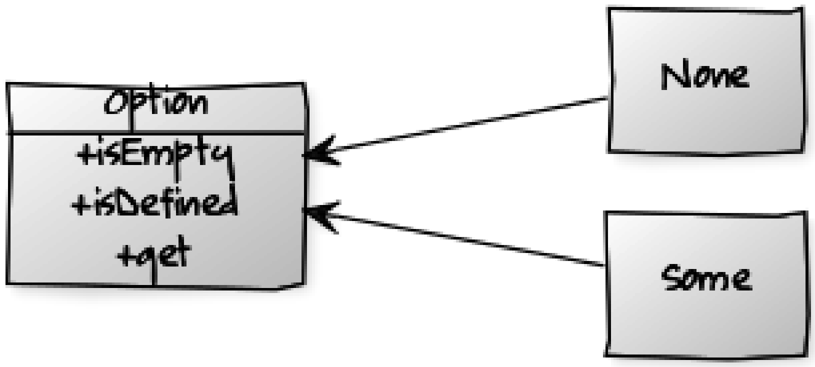
```
2 lazy val serviceRef = service
```

In this case, the serviceRef is not initialized until serviceRef first referenced. By this time, the service is already up and running, so everything works fine.

Lazy values are often used in Scala for configuration and *dependency injection* (DI). In Java building a dependency graph was a task performed by various DI frameworks. In Scala this task can be delegated to the compiler.

Option

`Option` is a container that may or may not contain a value. It is another example of an algebraic data type, because it is formed by two subtypes: `None` and `Some`.



Option as ADT

Options are a great way to communicate that a particular value may be absent. This helps to always avoid `NullPointerExceptions` if you play by the rules. Options are used everywhere in the standard library, in third-party libraries and they should be used in user code as well.

The safest way to create a value of type `Option` is to specify the type and then pass the value as the only argument:

```
scala> Option[String]("Joe")
res108: Option[String] = Some(Joe)
```

This way you can guarantee that the type will be `Option[String]` and you never end up with `null`-values inside. Even if you pass `null`, you will get `None`:

```
scala> Option[String](null)
res107: Option[String] = None
```

In many situations you can also pass a value in `Some`, but this approach doesn't guarantee that the type will be what you expect:

```
scala> Some(1)
res109: Some[Int] = Some(1)
```

In the example above we expected to get more generic `Option[Int]` but instead received more narrow `Some[Int]`. Moreover, we may end up with `null`s inside, which is a very dangerous situation and brings back the possibility of `NullPointerExceptions`:

```
scala> Some(null)
res104: Some[Null] = Some(null)
```

Never put `null` into `Some`!

There are lots and lots of methods on the `Option` class, which makes `Options` such a pleasure to work with. Here are only some of the methods:

method	description
`isDefined`	checks whether the value is present
`isEmpty`	checks whether the value is absent
`getOrElse`	returns the value if it's there or the provided default value if it's empty
`get`	returns the value if it's there or throws an exception if it is not; very rarely used
`map`	allows to transform one option into another
`foreach`	allows to use existing value without returning anything

Let's look at one concrete example to demonstrate how options are used. Imagine that we have a map that may contain information about the user. Here we're interested in two particular keys `firstName` and `lastName` and we want to use them to construct a value for full name.

```scala
val data = Map(
 "firstName" -> "Joe",
 "lastName" -> "Black"
)
```

There are several ways to initialize a map and this is only one of them. Note, that using `->` for creating key-value pairs is a neat trick that is sometimes used in Scala. If you want to learn more about this particular use case, check out this question on StackOverflow[18].

The `apply` method on `Map` returns the value associated with the requested key if the key is present or throws an exception if it is not. So, if we know for sure that this map contains both keys, constructing the full name is trivial:

```scala
val fullName = data("firstName") + " " + data("lastName")
```

If we don't know what information is inside the map, then we need to use the `get` method, which returns an `Option`.

[18]http://stackoverflow.com/questions/4980515/

```
val maybeFirstName = data.get("firstName")
val maybeLastName = data.get("lastName")
```

Obviously, we cannot concatenate two `Options` as we would `Strings`, but we can obtain `Strings` by using the `get` method on `Option`:

```
val maybeFullName = if (maybeFirstName.isDefined &&
    maybeLastName.isDefined) {
    Some(maybeFirstName.get + " " + maybeLastName.get)
} else None
```

When working with `Options`, we must first check whether the value actually exists and then use the `get` method.

A slightly more Scala-like approach would involve pattern matching:

```
val maybeFullName = (maybeFirstName, maybeLastName) match {
  case (Some(firstName), Some(lastName)) =>
    Some(firstName + " " + lastName)
  case _ => None
}
```

Here we're grouping two values together by enclosing them in parentheses. Then we match the newly created pair against two `case` blocks. The first block works if both `Options` have type `Some` and therefore contain some values inside. The second block defines a so-called *catch-all* case, which is used if nothing else succeeded so far.

We can also achieve the same result by using `map`/`flatMap` combination:

```
val maybeFullName = maybeFirstName.flatMap { firstName =>
  maybeLastName.map { lastName =>
    firstName + " " + lastName
  }
}
```

Doesn't it look familiar? It certainly does! When we were working with lists, we found out that this construct can be written as a for comprehension:

```scala
val maybeFullName = for {
    firstName <- maybeFirstName
    lastName <- maybeLastName
} yield firstName + " " + lastName
```

Great, isn't it? This looks almost as straightforward as `String` concatenation. Please, memorize this construct, because as we will see later, it is used in Scala for doing a great deal of seemingly unrelated things like working with dangerous or asynchronous code.

Try

Let's write a fictitious service that works roughly 60% of the time:

```scala
object DangerousService {
  def queryNextNumber: Long = {
    val source = Math.round(Math.random * 100)
    if (source <= 60)
      source
    else throw new Exception("The generated number is too big!")
  }
}
```

If we start working with this service directly without taking any precautions, sooner or later our program will blow up:

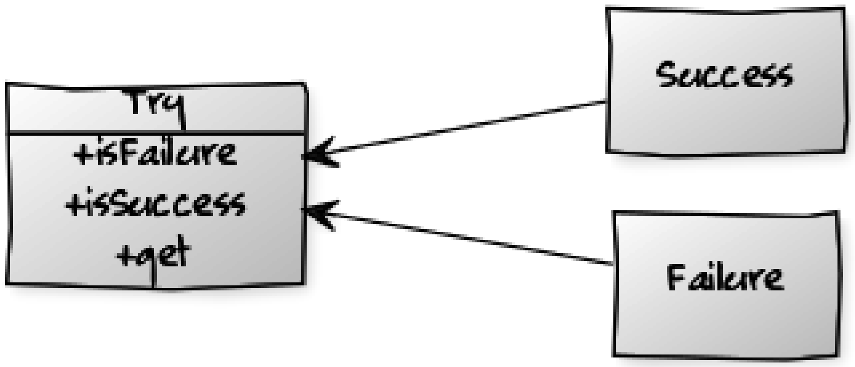
```
scala> DangerousService.queryNextNumber
res118: Long = 27

scala> DangerousService.queryNextNumber
java.lang.Exception: The generated number is too big!
  at DangerousService$.queryNextNumber(<console>:14)
  ... 32 elided
```

To work with dangerous code in Scala you can use `try-catch-finally` blocks. The syntax slightly differs from Java/C#, but the main idea is the same: you wrap dangerous code in the `try` block and then if an exception occurs, you can do something about it in the `catch` block.

```scala
val number = try {
  DangerousService.queryNextNumber
} catch { case e: Exception =>
  e.printStackTrace
  60
}
```

The good news about `try` blocks in Scala is that they return a value, so you don't need to define a `var` before the block and then assign it to some value inside (this approach is extremely common in Java). The bad news is that even if you come up with some reasonable value to return in case of emergency, this will essentially swallow the exception without letting anyone know what has happened. In theory, we could return an `Option`, but the problem here is that `None` will not be able to store information about the exception.

A better alternative is to use `Try` from the standard library. The `Try` type is an algebraic data type that is formed by two subtypes - `Success` and `Failure`.



Try as ADT

The `Try` class provides many utility methods that make working with it very convenient:

method	description
`isSuccess`	checks whether the value was successfully calculated
`isFailure`	checks whether the exception occurred
`get`	returns the value if it is a `Success` or throws an exception if it is a `Failure`; very rarely used
`toOption`	returns `Some` if it is a `Success` or `None` if it is a `Failure`
`map`	allows to transform one `Try` into another
`foreach`	allows to use existing value without returning anything

The `Try` type is defined in `scala.util`, so you need to use `import scala.util.Try` if you want to use `Try` without typing its fully qualified name.

When working with `Try`, the dangerous code can be simply put in the `Try` constructor, which will catch all exceptions if any occur:

```scala
val number1T = Try { DangerousService.queryNextNumber }
val number2T = Try { DangerousService.queryNextNumber }
```

If we want to sum two values wrapped in `Try`s, we can use the already familiar `flatMap`/`map` combination:

```scala
val sumT = number1T.flatMap { number1 =>
  number2T.map { number2 =>
    number1 + number2
  }
}
```

As always, we can write it as a for comprehension:

```scala
val sumT = for {
  number1 <- number1T
  number2 <- number2T
} yield number1 + number2
```

Note that a for comprehension *yields* the same container type (in our case `Try`) that was used initially. This happens because `map`s and `flatMap`s can merely transform the value inside the container, but they cannot change the type of the container itself. If at some point you need to convert `Try` into something else, you will need to use utility methods but not `map`s or `flatMap`s.

Future

The final piece of the standard library that we are going to examine here is the `Future` class. `Future`s allow you to work with asynchronous code in a type-safe and straightforward manner without resorting to concurrent primitives like threads or semaphores.

Many methods defined on the `Future` class are curried and accept an `implicit` argument. For example, take a look at the `map` signature:

```scala
def map[S](f: (T) => S)(implicit executor: ExecutionContext): Future[S]
```

This definition allows users of this class to declare or import some default `ExecutionContext` and then work with `Future`s as if they were regular Scala containers like `Option`s or `Try`s. The global `ExecutionContext` is already defined in the standard library and the only thing that's left is to import it:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Let's add a pause to our service to make it slower and therefore simulate working with something very remote:

```
object DangerousAndSlowService {
  def queryNextNumber: Long = {
    Thread.sleep(2000)
    val source = Math.round(Math.random * 100)
    if (source <= 60)
      source
    else throw new Exception("The generated number is too big!")
  }
}
```

If we try to use this service synchronously, *our* program, which uses this service, will be slowed down. In order to avoid it, we need to wrap our calls to `DangerousAndSlowService` in a `Future`:

```
val number1F = Future { DangerousAndSlowService.queryNextNumber }
val number2F = Future { DangerousAndSlowService.queryNextNumber }
```

The important thing here is that both `number1F` and `number2F` will be initialized immediately. However, we will not be able to use the actual numbers until they are returned from the service.

The `Future` class provides many methods that you can use:

method	**description**
`onComplete`	applies the provided callback when the `Future` completes, successfully or not
`onFailure`	applies the provided callback when the `Future` completes with an exception
`onSuccess`	applies the provided callback when the `Future` completes successfully
`mapTo`	casts the type of the `Future` to the given type
`map`	allows to transform one `Future` into another
`foreach`	allows to use existing value without returning anything

In theory, it is possible to work with `Futures` relying solely on callbacks:

```scala
number1F.onSuccess { case number1 =>
  number2F.onSuccess { case number2 =>
    println(number1 + number2)
  }
}
```

However, this approach makes code very complicated extremely quickly by introducing so-called *callback hell*. Besides, even after your callback is executed it's usually not obvious how to communicate the result (or lack thereof) to the rest of the program.

A better solution is to utilize `map` and `flatMap`:

```scala
val sumF = number1F.flatMap { number1 =>
  number2F.map { number2 =>
    number1 + number2
  }
}
```

Ana again, the same code could also be written as a for comprehension:

```scala
val sumF = for {
  number1 <- number1F
  number2 <- number2F
} yield number1 + number2
```

Usually, if you have only one container (`Option`, `Try`, `Future`, `Seq` and so on), it's easier to simply use `map`. If you have several containers, for comprehensions will be a better choice.

You've probably noticed that in several examples we used functional blocks that start with the `case` keyword. These functional blocks are called *partial functions*[19]. A function is called *partial* if it is defined only for a limited set of arguments. In practice, just remember that if you want to use a higher-order function that accepts a parameter of type `PartialFunction[T, R]`, you are expected to provide a functional block that starts with `case`.

[19]https://en.wikipedia.org/wiki/Partial_function

Build tools

Using the Scala Worksheet plugin or working in the interpreter may be fun, but at some point you will need to build your project. In this section we're going to look at different approaches to building Scala code starting with the ubiquitous command line.

Command line

The command line tools provide basic functionality for building and running Scala applications. It is not common to use these tools for building serious apps, but it's still worth learning them.

scalac

`scalac` is the Scala Compiler, i.e. a tool that is used to compile source code into byte code. In order to test the compiler, let's create a simple file called `HelloWorld.scala` with the following contents:

```scala
object HelloWorld {
  def main(args: Array[String]): Unit = {
    println("Hello World")
  }
}
```

The `main` method, just as in many other languages, is the starting point of the application. To compile this file, simply type in the command line:

```
$ scalac HelloWorld.scala
```

There will be no output, but `scalac` will create two new files in the current directory:

```
$ ls
HelloWorld.class  HelloWorld$.class  HelloWorld.scala
```

These two files contain regular Java bytecode.

scala

`scala` is the Scala interpreter. If it is invoked without arguments, it will start the REPL session. It can also be used to execute compiled Scala code. If we want to execute the `main` method of the recently compiled `HelloWorld` object, we need to type the following:

```
$ scala HelloWorld
Hello World
```

Another possibility is to execute the source file directly without compiling:

```
$ scala HelloWorld.scala
HelloWorld
```

This way, Scala can be used as a scripting language.

Gradle

Gradle is one of the most popular build tools for Java, but it can also be used for building Groovy, Scala and even C++ projects. It combines best parts of *Ant* and *Maven* (previous generation build tools) and uses a Groovy-based DSL for describing the build.

You can download a binary distribution from from the official website[20]. The latest version at the time of this writing is 2.10 and it works perfectly well with Scala.

After downloading the zip archive, you can install it anywhere (for example, to ~/DevTools/gradle) and then add the bin directory to your path. A good way to do it is to edit your `.bashrc`:

```
GRADLE_HOME=/home/user/DevTools/gradle
PATH=$JAVA_HOME/bin:$SCALA_HOME/bin:$GRADLE_HOME/bin:$PATH
```

To try Gradle, let's create a new directory called `gradle-build` and initialize a new Gradle project there:

```
$ cd gradle-build
$ gradle init --type scala-library
Starting a new Gradle Daemon for this build (subsequent builds will be faster).
:wrapper
:init

BUILD SUCCESSFUL

Total time: 5.563 secs
```

Gradle will create several files and directories including `build.gradle`. Remove both *.scala files from the `src` directory and copy our `HelloWorld.scala` to `src/main/scala`:

[20]http://gradle.org/gradle-download/

```
$ rm src/main/scala/Library.scala
$ rm src/test/scala/LibrarySuite.scala
$ cp ../HelloWorld.scala src/main/scala/
```

Finally, remove all the comments and test dependencies from the `gradle.build` file so that it looks like following:

```
apply plugin: 'scala'

repositories {
    jcenter()
}

dependencies {
    compile 'org.scala-lang:scala-library:2.11.7'
}
```

If we want to build the project, we need to type:

```
$ gradle build
```

Gradle will put the compiled class files into `build/classes/main`. In addition, it will pack these files into a jar called `gradle-build.jar` (you can find it in `build/libs`). JAR files contain the compiled bytecode of a particular application or library and some metadata. The `-cp` option of the Scala interpreter allows you to modify the default classpath, which adds one more way to start the app:

```
$ scala -cp build/libs/gradle-build.jar HelloWorld
Hello World
```

The class files that were created during compilation are regular Java bytecode files. Since they are no different from the files produced by `javac` (the Java compiler), we could try to run our Scala app using `java` (the Java interpreter). Let's try it:

```
$ java -cp build/libs/gradle-build.jar HelloWorld
```

Java will respond with the following error:

```
Exception in thread "main" java.lang.NoClassDefFoundError: scala/Predef$
        at HelloWorld$.main(HelloWorld.scala:4)
        at HelloWorld.main(HelloWorld.scala)
```

What happened? Well, Java loaded our jar file and started loading the `HelloWorld` object. The problem is that all Scala files implicitly import definitions from the `scala.Predef` object, which is part of the Scala standard library. Java wasn't able to find it, so it terminated with an error. This reasoning actually leads us to the solution to our problem - we need to add the Scala library to the classpath:

```
$ java -cp $SCALA_HOME/lib/scala-library.jar:build/libs/gradle-build.jar HelloWo\
rld
Hello World
```

What if we want our jar file to be self-sufficient? It turns out, this desire is quite common in the Java world and the solution is to build a so-called uber-jar[21] file. Building uber-jar files in Gradle is supported via the ShadowJar plugin[22]. In order to add it to our project, we need to replace the `apply plugin` line in `build.gradle` file with the following:

```
plugins {
  id 'scala'
  id 'com.github.johnrengelman.shadow' version '1.2.2'
}
```

We don't need to change anything else, because the Scala library is already in the dependencies section. The ShadowJar plugin will analyze this section and copy class files from provided libraries into an uber-jar file. This file can be built by invoking the following command:

```
$ gradle shadowJar
```

After that, we will be able to start our app without adding any additional libraries to the classpath:

```
$ java -cp build/libs/gradle-build-all.jar HelloWorld
Hello World
```

All of this can bring us to the conclusion that we actually don't need to install Scala on production servers to "run Scala code". The Scala distribution may be useful during the development stage, but after our application is built, organized into several jar files and copied to the production server, we will only need the Java runtime to run it.

[21]http://stackoverflow.com/questions/11947037/
[22]https://github.com/johnrengelman/shadow

sbt

sbt is the ultimate Scala build tool. It is the core component of Typesafe Activator, which is used for building Play Web applications, and the most popular build tool in the Scala World. It is quite different from Gradle, so we will discuss it in detail.

There are many ways to install sbt. The most universal one is to download the zip archive from the official sbt website[23], extract it and add the `sbt` executable to the PATH:

```
SBT_HOME=/home/user/DevTools/sbt
PATH=$JAVA_HOME/bin:$SCALA_HOME/bin:$GRADLE_HOME/bin:$SBT_HOME/bin:$PATH
```

When extracting the sbt archive you may notice that there isn't much in there:

```
$ tree ~/DevTools/sbt/bin -L 1

├── sbt
├── sbt.bat
├── sbt-launch.jar
└── sbt-launch-lib.bash

0 directories, 4 files
```

That's right - only a tiny launcher comes with the sbt distribution. Everything else will be downloaded when it's needed. The interesting outcome of this is that one `sbt` executable can work as different versions of the tool.

Let's create a directory called `sbt-build` and start `sbt` from there:

```
$ mkdir sbt-build
$ cd sbt-build
$ sbt
```

sbt will download required files and start itself in the interactive mode. In this mode you can type commands (by the way, Tab-completion is fully supported) and observe the output in the console. Since there are no files in our sbt project at the moment, we can start by observing and changing some project settings. Here is the summary of four standard settings we will be interested in:

[23] http://www.scala-sbt.org/download.html

setting	description	current value
name	the name of the project	sbt-build
organization	a project namespace, typically a reverse domain name of your company	default
version	the current version of the project	0.1-SNAPSHOT
scalaVersion	the version of Scala used for building the project	2.10.5

You can use the `show` command to inspect a setting:

```
> show scalaVersion
[info] 2.10.5
```

The setting value can be changed with the `set` command:

```
> set scalaVersion := "2.11.7"
[info] Defining *:scalaVersion
[info] The new value will be used by *:allDependencies, *:crossScalaVersions and\
 12 others.
```

Notice the assignment operator and the message about using the new value in several other settings. Essentially, the whole sbt build is just a set of different settings. By changing them you change how your build works. Let's assign new values to some of the settings as described in the following table:

setting	old value	new value
organization	default	com.appliedscala.sbt
version	0.1-SNAPSHOT	1.0-SNAPSHOT
scalaVersion	2.10.5	2.11.7

After it's done, you will probably want to save the results of your work, so that the new values will comprise the definition of our build. There is actually a command that does exactly this:

```
> session save
[info] Reapplying settings...
```

Now we can `exit` from `sbt` and check that all these values found their place in a newly created `build.sbt` file:

```
scalaVersion := "2.11.7"

organization  := "com.appliedscala.sbt"

version := "1.0-SNAPSHOT"
```

sbt also created a new directory called `project`. If we want to fix the version of `sbt` itself, we need to create a new file called `build.properties` and put it in this directory. The file will have the following content:

```
sbt.version = 0.13.11
```

Next time, when someone tries to run our build, the sbt launcher will make sure that the correct version is used.

multi-module builds

In order to demonstrate how sbt works with multi-module projects we will create two fictitious projects - *common* and *backend*. The common project will contain a service implementation and the backend project will use this service for performing its tasks.

sbt requires that each project resides in its own directory, so let's create two new folders inside the `sbt-build` directory:

```
$ mkdir common
$ mkdir backend
```

These projects will need to store Scala source files, so let's also create the source directories:

```
$ mkdir -p common/src/main/scala
$ mkdir -p backend/src/main/scala
```

Since sbt uses a Scala-based DSL, it's possible to use Scala inside build.sbt. Adding two sub-projects to the build definition can be achieved by introducing two values:

```
lazy val common = (project in file("common"))

lazy val backend = (project in file("backend"))
```

It's usually better to make values *lazy* as this approach allows to delegate determining the correct order of initialization to the Scala compiler.

Let's create our fictitious service by putting the following code to `GreetingService.scala` in the `common` subproject:

```
object GreetingService {
  def getGreeting(name: String): String = {
    s"Hello $name"
  }
}
```

The GreetingPrintService from the `backend` subproject will simply print whatever `GreetingService` returns:

```
object GreetingPrintService {
 def printGreeting(name: String): Unit = {
   println(GreetingService.getGreeting(name))
 }
}
```

After both files are created and stored in the `src/main/scala` directories of their projects, we can start sbt and inspect what has changed in the interactive mode. There we will be able to inspect settings of both subprojects and even compile `common`:

```
> show common/name
[info] common
> show backend/name
[info] backend
> common/compile
[success] Total time: 2 s, completed Feb 1, 2016 11:37:11 PM
```

However, the compilation of the `backend` project will fail:

```
> backend/compile
[info] Updating {file:/home/user/Startups/Creativity/src/sbt-build/}backend...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
...
[error]    println(GreetingService.getGreeting(name))
[error]            ^
[error] one error found
[error] (backend/compile:compileIncremental) Compilation failed
[error] Total time: 1 s, completed Feb 1, 2016 11:38:18 PM
```

If you think about it, this failure actually makes sense. At the moment, the `backend` project is not aware that it depends on the `common` project, so when the compiler couldn't find the `GreetingService` definition, it failed. Fortunately, specifying relationships between projects is very easy in sbt:

```
lazy val common = (project in file("common"))

lazy val backend = (project in file("backend")).
  dependsOn("common")
```

If you make this change without restarting `sbt`, you will need to use the `reload` command first, so that `sbt` will reread the build definition. After that, the backend project will compile without errors:

```
> reload
[info] Set current project to sbt-build (in build file:/home/user/Startups/Creat\
ivity/src/sbt-build/)
> backend/compile
[info] Updating {file:/home/user/Startups/Creativity/src/sbt-build/}backend...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Compiling 1 Scala source to /home/user/Startups/Creativity/src/sbt-build/\
backend/target/scala-2.10/classes...
[success] Total time: 1 s, completed Feb 2, 2016 12:16:10 PM
```

Note that `sbt` has put compiled files in `target/scala-2.10/classes`. How come it is 2.10 and not 2.11 as we specified earlier? Let's inspect the scalaVersion setting of the backend project:

```
> show backend/scalaVersion
[info] 2.10.5
```

I turns out that when we changed settings in the build.sbt file, these changes were only applied to the root project but not subprojects. A simple logic suggests that most settings would be relevant to both subprojects and we only need to tell sbt to apply them. Let's put all common settings in one `Seq`:

```
val commonSettings = Seq(
  scalaVersion := "2.11.7",
  organization  := "com.appliedscala.sbt",
  version := "1.0-SNAPSHOT"
)

lazy val common = (project in file("common")).
  settings(commonSettings)

lazy val backend = (project in file("backend")).
  settings(commonSettings).
  dependsOn("common")
```

If we check the `scalaVersion` setting right now, we will see that not both subprojects are compiled using Scala 2.11. However, the root project switched back to Scala 2.10:

```
> show scalaVersion
[info] backend/*:scalaVersion
[info]         2.11.7
[info] common/*:scalaVersion
[info]         2.11.7
[info] sbt-build/*:scalaVersion
[info]         2.10.5
```

The problem is that when we extracted common settings in a new list, we robbed the root project of these definitions. Let's fix this problem by creating a new lazy value that will represent our root project:

```
lazy val root = (project in file(".")).
  settings(commonSettings)
```

If we inspect the `scalaVersion` setting, we will see that its value is 2.11.7 for all projects:

```
> scalaVersion
[info] 2.11.7
```

There is one thing that's left. The default root project used to aggregate its subprojects, so previously we could invoke the `clean` command and both `common` and `default` projects would be cleaned. After creating the root value this default behaviour was lost. In order to tell sbt that `root` is still the the parent, we need to use the `aggregate` method:

```
lazy val root = (project in file(".")).
  settings(commonSettings).
  aggregate(common, backend)
```

Now the `clean` command will clean all projects and the `compile` command will compile them.

Working with Play

In this section we are going to examine what the Play framework has to offer when it comes to developing Web applications, discuss why it's such a convenient tool and which third-party library you may want to add.

Typesafe Activator

The recommended way to start working with Play is to use Typesafe Activator. Activator is a very thin wrapper around sbt, so everything you've learned previously is equally relevant here. Activator, however, needs to be installed separately. Simply download the mini-package from the official website[24], install it anywhere (for example, to ~/DevTools/activator) and add the directory to the PATH.

When started for the first time, Activator will download lots of packages from the Internet. If you find it easier to grab everything in one zip archive, you can ignore the mini-package and download the 500Mb version.

As always, you can add the `ACTIVATOR_HOME` environment variable by editing the `.bashrc` file:

```
ACTIVATOR_HOME=/home/user/DevTools/activator
PATH=$JAVA_HOME/bin:$ACTIVATOR_HOME:$SCALA_HOME/bin:$SBT_HOME/bin:$GRADLE_HOME/b\
in:$PATH
```

Then you can check that the installation was successful by typing the following:

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

The Play sample application

Activator provides the `new` option for creating skeletons for new projects. In order to create a new project called `scala-web-project` using the officially supported Play Scala template, simply type:

[24] https://www.typesafe.com/activator/download

```
$ activator new scala-web-project play-scala
```

The project will be initialized in a newly created directory with the same name.

In code snippets, I usually omit unchanged lines and import statements to prevent them from cluttering the main logic. Most of the time the surrounded text provides enough insight into what to import, but when it doesn't, please refer to Appendix A where all third-party classes are listed alphabetically along with their fully-qualified names. Alternatively, you can always download the code from the GitHub repo[25].

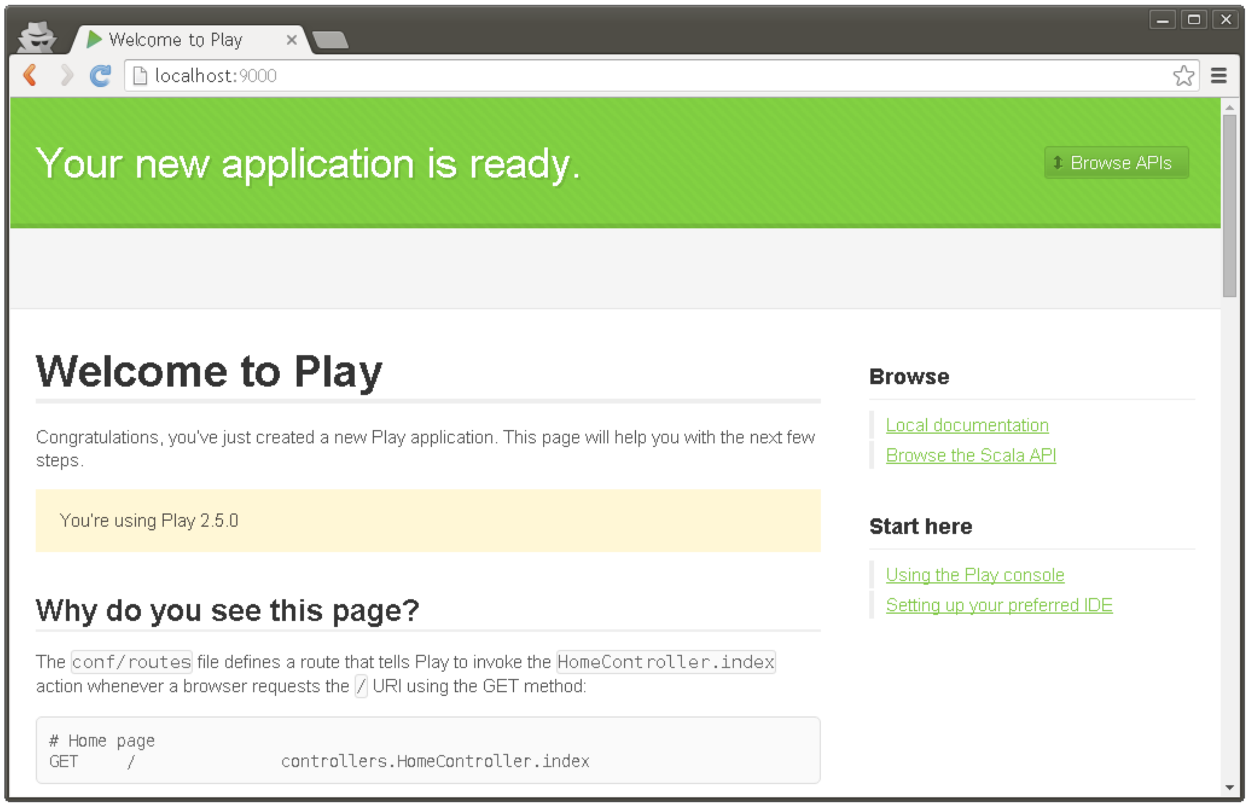
It's worth trying to run the app to make sure it's working before making any changes.

```
$ cd scala-web-project
$ activator
[scala-web-project] $ run
```

Remember that Activator is only a wrapper and after typing `activator` without arguments you will get to the interactive mode with already familiar sbt commands. Also, you need to remember to call `reload` wherever you change something in `build.sbt`. The most used command, though, is probably `run` that starts the app on port 9000. It will also start waiting for changes, so the code will be recompiled if necessary on page refresh.

The app starts on port 9000, so if you try navigating to `localhost:9000` you will see something similar to this:

[25] https://github.com/denisftw/modern-web-scala



Your new application is ready.

Welcome to Play

Congratulations, you've just created a new Play application. This page will help you with the next few steps.

You're using Play 2.5.0

Why do you see this page?

The `conf/routes` file defines a route that tells Play to invoke the `HomeController.index` action whenever a browser requests the `/` URI using the GET method:

```
# Home page
GET     /               controllers.HomeController.index
```

Play Welcome page

All this stuff that Play created for our app is certainly impressive, but we are not going to use most of it. Here is the table that shows what we need to do before starting to make our own changes:

file	what to do
build.sbt	Add `pipelineStages := Seq(digest)` somewhere in the file
project/plugins.sbt	Remove all web plugins except "sbt-digest" as we're going to use Webpack for handling frontend assets
conf/application.conf	Remove all comments so that only the lines containing `play.crypto.secret` and `play.i18n.langs` remain
conf/routes	See below
views/index.scala.html	Replace its content with main.scala.html and then edit as described below
views/main.scala.html	Delete the file
app/Filters.scala	Delete the file
app/Module.scala	Delete the file
app/controllers/AsyncController.scala	Delete the file
app/controllers/CountController.scala	Delete the file
app/controllers/HomeController.scala	See below
app/filters	Delete the directory

file	what to do
app/services	Delete the directory
public/javascripts	Delete the directory
public/stylesheets	Delete the directory
test	Delete the directory

In `index.scala.html` you need to remove the template's parameters and frontend assets, as we are going to build a better workflow later:

```
@()
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
        <link rel="shortcut icon" type="image/png"
           href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello Play</h1>
    </body>
</html>
```

In the `conf/routes` file, replace its contents with the following:

```
GET   /               controllers.Application.index
GET   /assets/*file   controllers.Assets.versioned(path="/public", file: Asset)
```

Finally, rename file `HomeController.scala` to `Application.scala` and replace its contents with the following:

```
package controllers

import play.api._
import play.api.mvc._

class Application extends Controller {
  def index = Action {
    Ok(views.html.index())
  }
}
```

Application structure

Now that we've cleaned up the sample app, let's take a look at its structure. Here are the main parts:

part	description
build.sbt	contains typical sbt stuff like library dependencies, enabled plugins, version etc
activator files	allows other developers to work on the app without having to install Typesafe Activator
public	contains static assets like images and fonts and compiled frontend assets like minified scripts and stylesheets
project/build.properties	specifies sbt version
project/plugins.sbt	specifies necessary Play plugins including Play itself
conf/application.conf	contains configuration for the app, for example, database connection properties
conf/logback.xml	contains logging configuration
routes	specifies exposed routes as HTTP endpoints and maps them to Scala controller methods
app/views	contains Play templates
app/*.scala	the backend part of the application

As we progress, we will add many more files to this structure, but for now this is enough to form the basis of our application. If you look at the `build.sbt` file, you may find all familiar definitions from the previous section. This file, however, uses two sbt operators that haven't been discussed, so let's take a look at how `libraryDependencies` and `resolvers` are defined:

```
name := """scala-web-project"""

version := "1.0-SNAPSHOT"

lazy val root = (project in file(".")).enablePlugins(PlayScala)

scalaVersion := "2.11.7"

pipelineStages := Seq(digest)

libraryDependencies ++= Seq(
  jdbc,
  cache,
  ws,
  "org.scalatestplus.play" %% "scalatestplus-play" % "1.5.0-RC1" % Test
)

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

The += operator adds a new value to a list setting. The ++= works in a similar manner but instead of adding one value, it adds all values from the provided sequence. So, in the above example, four libraries were added to the list of dependencies and one new resolver was added to the list of resolvers.

> *Resolvers* determine where sbt looks for libraries. By default, sbt already includes the Maven repository, which contains most Java or Scala libraries you will need. Since we are not going to use ScalaZ in our project, you can safely remove this line.

A typical record in the `conf/routes` file looks like this:

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

The first column specifies the HTTP-method (usually GET or POST). The second column specifies URL relative to the root of the app. Together they form an *endpoint*. The last column points to a controller action that will be invoked when a given endpoint is reached. So, in the example above, when you navigate the browser to `http://localhost:9000/`, the `index` method from the `Application` controller will serve this request:

```scala
def index = Action {
  Ok(views.html.index())
}
```

The only thing that the index method does is take the Play template and send it to the caller (in our case, to the browser).

Play templates

You can see Play templates (sometimes called *Twirl templates*) as HTML-files with fragments of Scala code. During compilation, Play transforms them into Scala methods that return HTML markup. The `@()` directive at the top forms the method's signature.

The `@import` directive allows to import packages inside the template:

```
@import scala.concurrent.Future
```

The `@` character itself starts a Scala expression. For example, the following snippet will result in rendering a random number inside the `div` tags:

```
    <body>
        <h1>Hello Play</h1>
        <div>@{Math.random}</div>
    </body>
```

By the way, the `routes.Assets.versioned` method is used to add fingerprinting to static assets. However, if you check the generated HTML that was received by the browser, you will see that it isn't working:

```
<link rel="shortcut icon" type="image/png" href="/assets/images/favicon.png">
```

First, we need to make sure that the `digest` Play module is added to the assets pipeline in the build.sbt file:

```
pipelineStages := Seq(digest)
```

Second, we need to check that the corresponding Play plugin is defined in `project/plugins.sbt`:

```
addSbtPlugin("com.typesafe.sbt" % "sbt-digest" % "1.1.0")
```

Finally, as fingerprinting is disabled in the development mode, if you want to see fingerprinted URLs, you must start the app in the production mode by typing `activator testProd`. Note that before doing that, you need to change the `play.crypto.secret` from the `application.conf`. In order to generate a new secret sequence, simply type:

```
$ $ activator playGenerateSecret
[info] Generated new secret: _AC9P4rt66K7^cB=[g9`MoI9pGu;q2wB`cp19cBY0FN0x:864`M\
1OFR_i=t^D2YL
[success] Total time: 0 s, completed Mar 5, 2016 5:26:44 PM
```

Then, in `application.conf`, replace "changeme" with a newly generated string and start the application in the production mode:

```
$ activator testProd
(Starting server. Type Ctrl+D to exit logs, the server will remain in background)
[info] - play.api.Play - Application started (Prod)
[info] - play.core.server.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9\
000
```

After that all URLs created by `routes.Assets.versioned` will become fingerprinted:

```
<link rel="shortcut icon" type="image/png"
   href="/assets/images/84a01dc6c53f0d2a58a2f7ff9e17a294-favicon.png">
```

In the next section we are going to start making changes to the sample application we've just created. If you want to follow this process, please refer to Appendix B that shows how to clone the book's repo and import the project in IntelliJ IDEA. If you are already following, you can use *the starting point*[26] release as a reference.

Now that we know fingerprinting is working, we can stop the production server and return back to the development mode.

Using template parameters

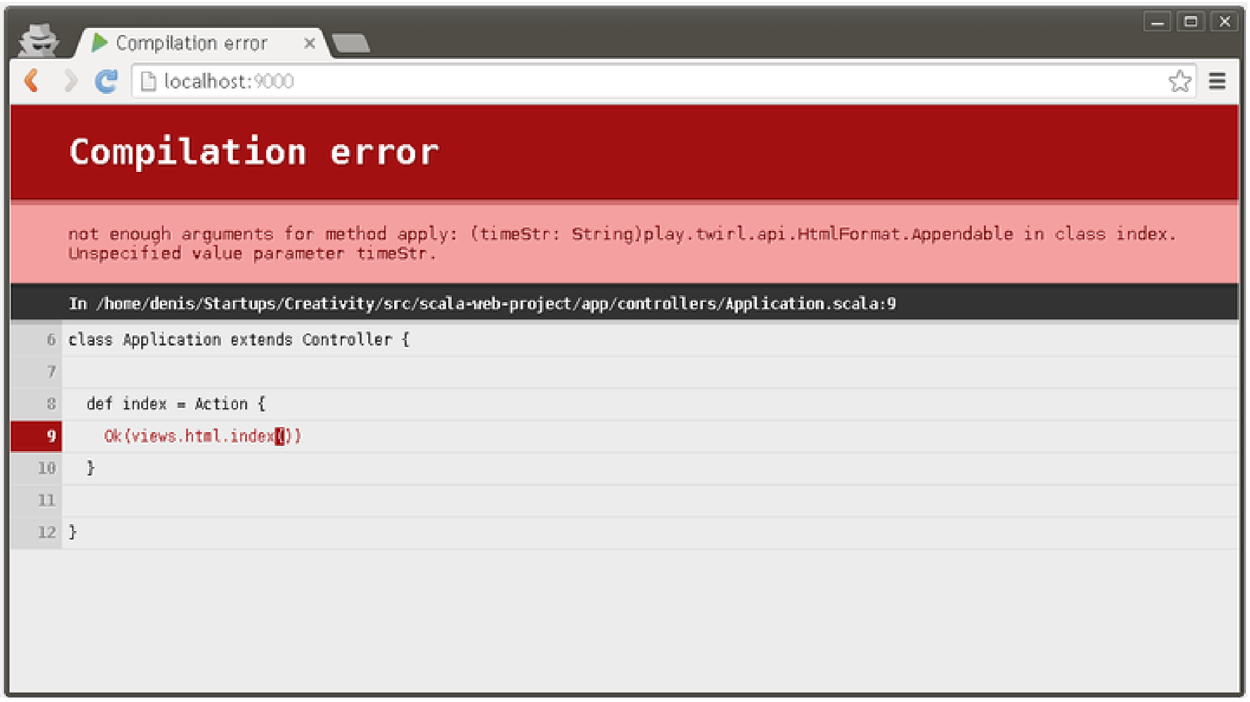
Since Play templates are actually Scala methods, we can define parameters and pass arguments. Let's render the current time (in your default time zone) as a simple formatted string inside the `div` tag on our home page.

First, we need to add a new parameter in `index.scala.html`:

```
@(timeStr: String)
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
        <link rel="shortcut icon" type="image/png"
           href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello Play</h1>
        <div>Current time: @timeStr</div>
    </body>
</html>
```

Note that we defined the parameter as `String`. As Scala is a statically typed language, passing arguments of any other type will fail at compile time. In fact, if we try refreshing the page now, we will see an error, because at the moment the `index` method of the `Application` controller still tries to call the template method without passing any arguments:

[26] https://github.com/denisftw/modern-web-scala/releases/tag/starting-point-play-2.5.0



Compilation error

```
not enough arguments for method apply: (timeStr: String)play.twirl.api.HtmlFormat.Appendable in class index.
Unspecified value parameter timeStr.
```

```
In /home/denis/Startups/Creativity/src/scala-web-project/app/controllers/Application.scala:9
```

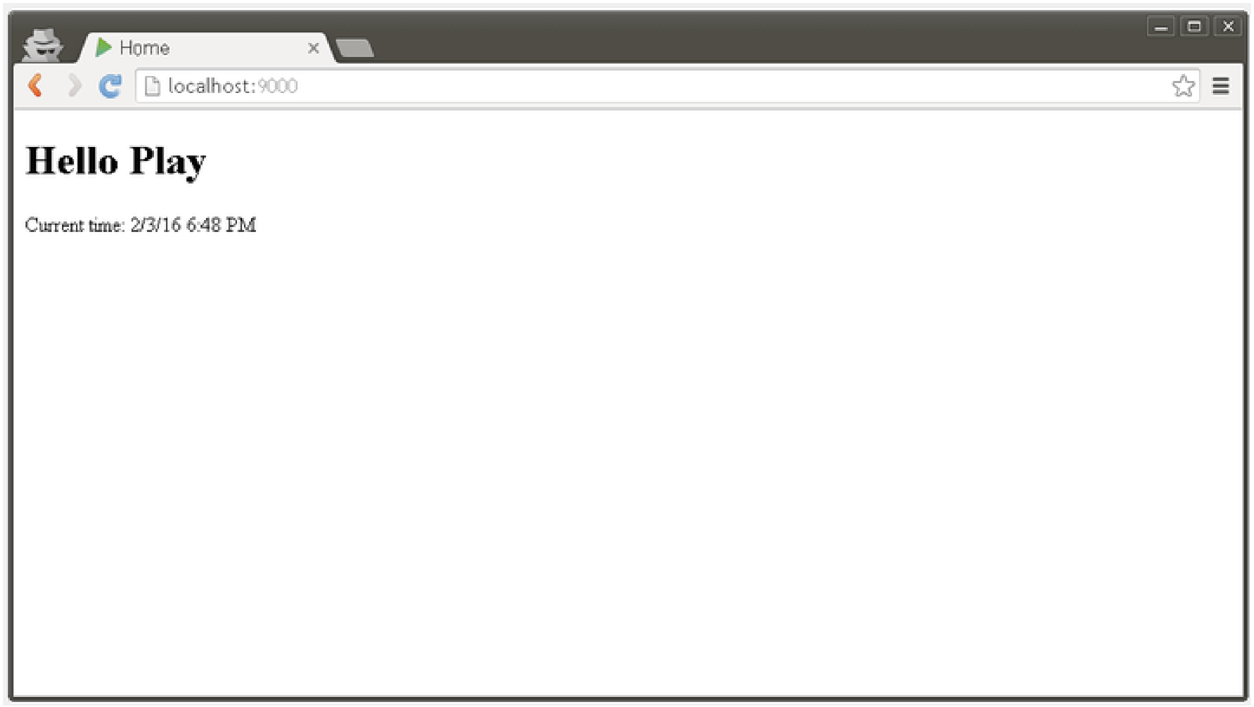
```
class Application extends Controller {

  def index = Action {
    Ok(views.html.index())
  }

}
```

Missing parameters

The simplest way to obtain current time in Scala is to create a new `java.util.Date` instance. Once we have this object, we can format it using `SimpleDateFormatter` and pass it to the template:

```
  def index = Action {
    import java.util.Date
    import java.text.SimpleDateFormat

    val date = new Date()
    val dateStr = new SimpleDateFormat().format(date)
    Ok(views.html.index(dateStr))
  }
```

After that, everything should work just fine:



Hello Play

Current time: 2/3/16 6:48 PM

Showing current time

Serving asynchronous results

Instead of simply creating a `Date` instance, let's query some information from a remote server. In Play, such a task can be completed easily thanks to a built-in HTTP client called `WS`, which is based on Async Http Client for Java[27].

In the first examples, we will use `WS` as a singleton object, which is not recommended. The proper way would be to inject `WSClient` via a built-in runtime dependency injection provided by Play. This issue will be fixed after we introduce a compile-time dependency injection, so don't worry about deprecation warnings for now.

As querying remote servers can be slow, the WS client returns a `Future`. We already learned how to work with `Future`s in previous sections, but using them in controllers presents a new challenge. How can we return a proper result of type `play.api.mvc.Result`[28] if we don't know when the Future object will be resolved?

This seemingly impossible challenge is actually very easy to resolve. Since Play provides a first-class support for asynchronous programming, a controller action can return a `Future[Result]` and the rest will be handled by the framework itself. There's one thing that we need to change, though, and it is using `Action.async` instead of `Action.apply`.

[27]https://github.com/AsyncHttpClient/async-http-client
[28]https://www.playframework.com/documentation/2.5.x/ScalaActions

```
  def index = Action.async {
    import java.util.Date
    import java.text.SimpleDateFormat
    import scala.concurrent.Future

    val date = new Date()
    val dateStr = new SimpleDateFormat().format(date)
    Future.successful { (views.html.index(dateStr)) }
  }
```

You can make these changes and then refresh the page to check that it's still working.

When you create a `Future` using its constructor-like `apply` method, you need to have an implicit `ExecutionContext` in scope. However, using the `successful` method doesn't require an `ExecutionContext` as it simply wraps the result of an expression in a `Future` without scheduling the computation in a different thread.

Adding sunrise and sunset times

We are going to extend our application by showing information about sunrise and sunset on the home page. As an example I will use Sydney, but feel free to use any location you want. According to TravelMath.com[29] Sydney's latitude is roughly -33.8830 and the longitude is 151.2167.

A great service called "Sunrise-Sunset"[30] exposes free API which allows to query information about sunrise and sunset times for any location in the world. You can try using the service right in the browser by navigating to:

```
http://api.sunrise-sunset.org/json?lat=-33.8830&lng=151.2167&formatted=0
```

The response will have the following format:

[29]http://www.travelmath.com/cities/Sydney,+Australia
[30]http://sunrise-sunset.org/

```
{"results":{"sunrise":"2016-02-03T19:19:22+00:00","sunset":"2016-02-04T08:58:37+\
00:00","solar_noon":"2016-02-04T02:08:59+00:00","day_length":49155,"civil_twilig\
ht_begin":"2016-02-03T18:52:34+00:00","civil_twilight_end":"2016-02-04T09:25:24+\
00:00","nautical_twilight_begin":"2016-02-03T18:20:12+00:00","nautical_twilight_\
end":"2016-02-04T09:57:46+00:00","astronomical_twilight_begin":"2016-02-03T17:46\
:05+00:00","astronomical_twilight_end":"2016-02-04T10:31:54+00:00"},"status":"OK\
"}
```

The response looks messy, but since it's a valid JSON, it can be parsed and formatted easily. To format it, you can use one of numerous online services, for example CodeBeautify.org[31]. This is how the formatted result is supposed to look:

```
{
    "results": {
        "sunrise": "2016-02-03T19:19:22+00:00",
        "sunset": "2016-02-04T08:58:37+00:00",
        "solar_noon": "2016-02-04T02:08:59+00:00",
        "day_length": 49155,
        "civil_twilight_begin": "2016-02-03T18:52:34+00:00",
        "civil_twilight_end": "2016-02-04T09:25:24+00:00",
        "nautical_twilight_begin": "2016-02-03T18:20:12+00:00",
        "nautical_twilight_end": "2016-02-04T09:57:46+00:00",
        "astronomical_twilight_begin": "2016-02-03T17:46:05+00:00",
        "astronomical_twilight_end": "2016-02-04T10:31:54+00:00"
    },
    "status": "OK"
}
```

Using the WS object we can obtain the same JSON response programmatically:

```
val responseF = WS.url("http://api.sunrise-sunset.org/json?" +
  "lat=-33.8830&lng=151.2167&formatted=0").get()
```

Then we can extract a particular field using \ method (yes, in Scala nothing prevents you from creating a method called "\") defined on `JsValue`.

[31] http://codebeautify.org/jsonviewer

```
val json = response.json
val sunriseTimeStr = (json \ "results" \ "sunrise").as[String]
val sunsetTimeStr = (json \ "results" \ "sunset").as[String]
```

To keep things clean, we will create a new case class in the `model` package and call it `SunInfo`. It will be used to store information about sunrise and sunset times:

```
case class SunInfo(sunrise: String, sunset: String)
```

We will be forming `SunInfo` objects in our controller and passing them to the template like so:

```
def index = Action.async {
    val responseF = WS.url("http://api.sunrise-sunset.org/json?" +
      "lat=-33.8830&lng=151.2167&formatted=0").get()
    responseF.map { response =>
      val json = response.json
      val sunriseTimeStr = (json \ "results" \ "sunrise").as[String]
      val sunsetTimeStr = (json \ "results" \ "sunset").as[String]
      val sunInfo = SunInfo(sunriseTimeStr, sunsetTimeStr)
      Ok(views.html.index(sunInfo))
    }
  }
```

Note that in order for this method to work, we need to add two new imports:

```
import play.api.Play.current
import scala.concurrent.ExecutionContext.Implicits.global
```

The `Play.current` introduces an implicit `Application`, which is used by the `WS` client. The `Implicits.global` introduces an implicit `ExecutionContext`, which is used by `Futures`. Finally, we need to modify the index template, so it accepts and uses a parameter of type `SunInfo`:

```
@(sunInfo: model.SunInfo)
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
        <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello Play</h1>

        <div>Sunrise time: @sunInfo.sunrise</div>
        <div>Sunset time: @sunInfo.sunset</div>
    </body>
</html>
```

Try refreshing the page now, it should work now. The page is showing the following information:

```
Sunrise time: 2016-02-03T19:19:22+00:00
Sunset time: 2016-02-04T08:58:37+00:00
```

The sunrise at 7:19 PM and sunset at 8:58 AM certainly look strange. Besides, the information about the dates seems redundant here, so let's address these issues next.

Using JodaTime

The problem is that right now all times are shown with the default timezone of the Sunrise-Sunset service, which has the offset of 0. Since we are presenting Sydney information, it would be better to show times using the Sydney timezone. Obviously, as long as we work with plain strings we will not be able to change the timezone, so let's convert them into proper date representations.

In theory, we could use `java.util.Date`, but this type was so badly designed that it is regarded as a disaster by both Java and Scala community. Historically, most Scala projects had been using an alternative third-party library called JodaTime, so Play developers simply included it into the framework. If you ever need to use this library outside of Play app, you can find its Maven coordinates in the dependencies section of the official JodaTime website[32].

Strings that we receive from Sunrise-Sunset are formatted using the ISO 8601 standard[33]. The JodaTime library can parse these strings easily and after that we will format the resulting `DateTime` instances once again. This time, though, we will add the "Australia/Sydney" timezone and drop the date part as we don't need it. The final version of the index method is this:

[32]http://www.joda.org/joda-time/dependency-info.html
[33]https://en.wikipedia.org/wiki/ISO_8601

```
def index = Action.async {
  val responseF = WS.url("http://api.sunrise-sunset.org/json?" +
    "lat=-33.8830&lng=151.2167&formatted=0").get()
  responseF.map { response =>
    val json = response.json
    val sunriseTimeStr = (json \ "results" \ "sunrise").as[String]
    val sunsetTimeStr = (json \ "results" \ "sunset").as[String]
    val sunriseTime = DateTime.parse(sunriseTimeStr)
    val sunsetTime = DateTime.parse(sunsetTimeStr)
    val formatter = DateTimeFormat.forPattern("HH:mm:ss").
      withZone(DateTimeZone.forID("Australia/Sydney"))
    val sunInfo = SunInfo(formatter.print(sunriseTime),
      formatter.print(sunsetTime))
    Ok(views.html.index(sunInfo))
  }
}
```

You can try refreshing the page and you'll see that now the information is presented in the following way:

```
Sunrise time: 06:19:22
Sunset time: 19:58:37
```

Sunrise and sunset times are presented in the local time zone, so the numbers make more sense now.

Adding weather information

Wouldn't it be great to show current temperature in addition to sunrise/sunset times? The weather information can be obtained from the OpenWeatherMap[34] service. One of their API methods accepts geographic coordinates (latitude and longitude), which suits us perfectly:

```
api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}
```

In all examples that involve using OpenWeatherMap API I'm omitting the `appId` query parameter. If you want to use their API, you need to register a free account on their website[35] and then use a generated application identifier.

Passing Sydney coordinates returns the following JSON (formatted, irrelevant information omitted):

[34] http://openweathermap.org/
[35] http://openweathermap.org/appid

```
{
    "main": {
        "temp": 21.51,
        "pressure": 1021.02,
        "humidity": 82,
        "temp_min": 21.51,
        "temp_max": 21.51,
        "sea_level": 1027.05,
        "grnd_level": 1021.02
    }
    // ...
}
```

We already know how to query this information with `WS` and extract the temperature from `JsValue`:

```scala
val weatherResponseF = WS.url("http://api.openweathermap.org/data/2.5/" +
  "weather?lat=-33.8830&lon=151.2167&units=metric").get()
weatherResponseF.map { weatherResponseF =>
  val json = weatherResponseF.json
  val temperature = (json \ "main" \ "temp").as[Double]
}
```

Adding a new `Double` parameter to the index template is also trivial:

```html
@(sunInfo: model.SunInfo, temperature: Double)
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
        <link rel="shortcut icon" type="image/png"
         href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello Play</h1>

        <div>Sunrise time: @sunInfo.sunrise</div>
        <div>Sunset time: @sunInfo.sunset</div>
        <div>Current temperature: @temperature</div>
    </body>
</html>
```

We already discussed how to combine two `Future` objects using `map/flatMap` combinations or for comprehensions. As you've probably guessed, this wasn't a purely classroom exercise - for comprehensions are used in real-world Scala, a lot. Let's now put everything together and rewrite the `index` method so that it uses data from different remote services:

```scala
def index = Action.async {
  val responseF = WS.url("http://api.sunrise-sunset.org/" +
    "json?lat=-33.8830&lng=151.2167&formatted=0").get()
  val weatherResponseF = WS.url("http://api.openweathermap.org/data/2.5/" +
    "weather?lat=-33.8830&lon=151.2167&units=metric").get()

  for {
    response <- responseF
    weatherResponse <- weatherResponseF
  } yield {
    val weatherJson = weatherResponse.json
    val temperature = (weatherJson \ "main" \ "temp").as[Double]
    val json = response.json
    val sunriseTimeStr = (json \ "results" \ "sunrise").as[String]
    val sunsetTimeStr = (json \ "results" \ "sunset").as[String]
    val sunriseTime = DateTime.parse(sunriseTimeStr)
    val sunsetTime = DateTime.parse(sunsetTimeStr)
    val formatter = DateTimeFormat.forPattern("HH:mm:ss").
      withZone(DateTimeZone.forID("Australia/Sydney"))
    val sunInfo = SunInfo(formatter.print(sunriseTime),
      formatter.print(sunsetTime))
    Ok(views.html.index(sunInfo, temperature))
  }
}
```

As you can see, our date manipulations and JSON processing moved inside the `yield` block, but apart from that the code mostly remained unchanged. If you try refreshing the page, you will see that information about current temperature is showing below sunrise and sunset times:

```
Sunrise time: 06:19:22
Sunset time: 19:58:37
Current temperature: 22.0
```

Excellent!

Moving logic into services

If you take a look at our `index` method, you will see that it contains a lot of logic. Right now, it's querying remote services, parsing responses, reformatting dates. Strictly speaking, this kind of logic doesn't belong to controllers, because it makes them more complicated than they should be. Besides, what if we want to use this logic somewhere else? It's much better to put this logic into *services*, which controllers can use to obtain needed data.

We can start refactoring the `index` method by extracting logic related to current temperature into the `WeatherService`. The service itself can be placed into the `services` package. We can also extract latitude and longitude as method parameters and therefore make the service more generic:

```scala
class WeatherService {
  def getTemperature(lat: Double, lon: Double): Future[Double] = {
    val weatherResponseF = WS.url(
      "http://api.openweathermap.org/data/2.5/weather?" +
        s"lat=$lat&lon=$lon&units=metric").get()
    weatherResponseF.map { weatherResponse =>
      val weatherJson = weatherResponse.json
      val temperature = (weatherJson \ "main" \ "temp").as[Double]
      temperature
    }
  }
}
```

Using the same approach, we can also take logic related to sunrise and sunset times from the `index` method and put it into a newly created `SunService`. Here we can also make the service more generic by extracting latitude and longitude as method parameters:

```scala
class SunService {
  def getSunInfo(lat: Double, lon: Double): Future[SunInfo] = {
    val responseF = WS.url("http://api.sunrise-sunset.org/json?" +
      s"lat=$lat&lng=$lon&formatted=0").get()
    responseF.map { response =>
      val json = response.json
      val sunriseTimeStr = (json \ "results" \ "sunrise").as[String]
      val sunsetTimeStr = (json \ "results" \ "sunset").as[String]
      val sunriseTime = DateTime.parse(sunriseTimeStr)
      val sunsetTime = DateTime.parse(sunsetTimeStr)
      val formatter = DateTimeFormat.forPattern("HH:mm:ss").
        withZone(DateTimeZone.forID("Australia/Sydney"))
      val sunInfo = SunInfo(formatter.print(sunriseTime),
        formatter.print(sunsetTime))
      sunInfo
    }
  }
}
```

It's worth mentioning that since we are using the `WS` service and `Futures` inside our newly created services, both `Play.current` and `Implicits.global` need to be passed into their scopes:

```
import play.api.Play.current
import scala.concurrent.ExecutionContext.Implicits.global
```

In order to use the services, we must have the references to their instances in the `Application` controller. For now, let's simply instantiate both services and make them regular fields of the `Application` class:

```
class Application extends Controller {

  val sunService = new SunService
  val weatherService = new WeatherService

  // ...
}
```

Finally, the `index` method, which now can delegate most of its logic to the services, becomes quite simple:

```
def index = Action.async {
  val lat = -33.8830
  val lon = 151.2167
  val sunInfoF = sunService.getSunInfo(lat, lon)
  val temperatureF = weatherService.getTemperature(lat, lon)
  for {
    sunInfo <- sunInfoF
    temperature <- temperatureF
  } yield {
    Ok(views.html.index(sunInfo, temperature))
  }
}
```

If you try refreshing the page now, you will see that the app is still working as usual.

Injecting dependencies with MacWire

Our last refactoring was definitely a success, but the code that we have now presents a new challenge. We extracted the logic into services on the premise of reusing it somewhere else. Now, however, we are instantiating both `SunService` and `WeatherService` inside the `Application` controller. What if we need to use them in another controller? We could instantiate them once again but this will obviously lead to unnecessary waste of memory. Basically, we need only one instance of each service and a way to inject them where they are needed.

This is one of the reasons why people use *dependency injection* (DI) frameworks. They also help with resolving a so-called *dependency graph* and determining the correct order of initialization. Play comes with a very popular Java DI framework - Google Guice (pronounced “juice”). It relies heavily on Java annotations and builds the dependency graph in runtime. If you want to ever use it in your applications, it’s well documented on the official Play website[36].

We will, however, go another way by using an alternative compile-time dependency injection framework written specifically for Scala and called MacWire[37]. It is dead-simple, introduces zero-overhead in runtime and reduces boilerplate code with the `wire` macro. Moreover, when we are done, our services will remain completely unaware that they were injected by Macwire!

If you want to try a runtime dependency injection but don’t feel like using Guice annotations, I recommend you take a look at Scaldi[38] - a lightweight library that utilizes a more Scala-like approach to managing dependencies.

Lets’ start by adding MacWire libraries to the dependencies section of `build.sbt`:

```
"com.softwaremill.macwire" %% "macros" % "2.2.0" % "provided",
"com.softwaremill.macwire" %% "util" % "2.2.0"
```

Note that the macros library is marked as `provided`. This is because MacWire macros are only used at compile time and they are not needed when the app is running.

The second thing is to tell Play that we are going to use compile-time DI by specifying the application loader in `application.conf`:

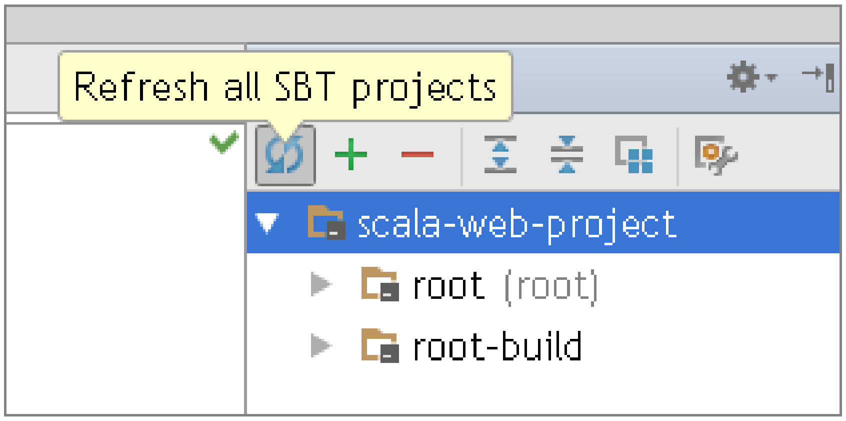
```
play.application.loader = "AppApplicationLoader"
```

The next step is obviously the AppApplicationLoader class. You can simply create a file called `AppLoader.scala` in the root package (i.e. right in the `app` directory) with the following content:

[36]https://www.playframework.com/documentation/2.5.x/ScalaDependencyInjection
[37]https://github.com/adamw/macwire
[38]http://scaldi.org/

```
import controllers.{Application, Assets}
import play.api.ApplicationLoader.Context
import play.api._
import play.api.routing.Router
import router.Routes
import com.softwaremill.macwire._

class AppApplicationLoader extends ApplicationLoader {
  def load(context: Context) = {
    LoggerConfigurator(context.environment.classLoader).foreach { configurator =>
      configurator.configure(context.environment)
    }
    (new BuiltInComponentsFromContext(context) with AppComponents).application
  }
}

trait AppComponents extends BuiltInComponents {
  lazy val assets: Assets = wire[Assets]
  lazy val prefix: String = "/"
  lazy val router: Router = wire[Routes]
  lazy val applicationController = wire[Application]
}
```

If IntelliJ cannot find the `softwaremill` package, it means that you need to reimport the project after making changes to `build.sbt`:

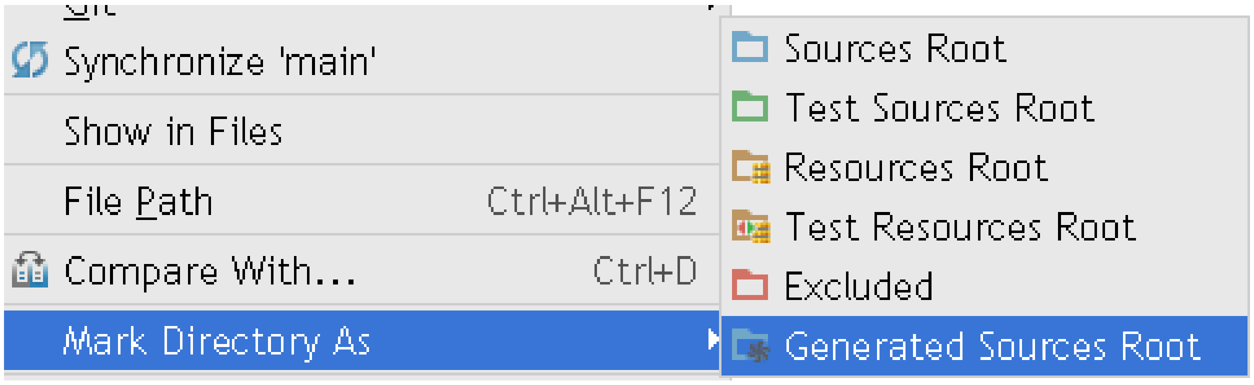


Reimporting the project

Sometimes IntelliJ mistakenly highlights the following line as an error:

```
import router.Routes
```

This happens because the `target/scala-2.11/routes/main` folder is marked as "Excluded" instead of "Generated Sources Root". If you want to help IDEA, simply select this folder and then use the context menu to mark it as generated source:



Marking "Generated Sources Root"

In `AppLoader.scala`, we are creating the `AppApplicationLoader` class so that it implements a built-in trait called `ApplicationLoader`. The `load` method will be invoked by Play every time our application is reloaded. The `AppComponents` trait initializes components that our application uses. This is basically where DI happens. The `wire` macro reduces boilerplate by liberating you from instantiating objects manually and passing necessary constructor parameters explicitly while the `lazy` keyword ensures the correct order of initialization. So far, so good.

It's important to understand that creating our own application loader means that we also need to handle Play's built-in services. It's not difficult but Play is not going to do it for us anymore. For example, the `WS` will no longer work as a singleton. Instead, we will need to use the `WSClient` type, which is defined in the `AhcWSComponents` trait.

There are many built-in traits whose names end with *Components* such as BuiltInComponents, AhcWSComponents, HikariCPComponents and so on. They are meant to be added to your main `AppComponents` trait when you need a particular built-in service. In our case the `AhcWSComponents` trait gives us an implementation of `WSClient`.

We will need to make the following changes to both our services:

- Create a constructor parameter `wsClient` of type `WSClient`
- Use this parameter instead of `WS` singleton object

Let's start with `WeatherService`:

```scala
class WeatherService(wsClient: WSClient) {
  def getTemperature(lat: Double, lon: Double): Future[Double] = {
    val weatherResponseF = wsClient.url(
      "http://api.openweathermap.org/data/2.5/" +
        s"weather?lat=$lat&lon=$lon&units=metric").get()
    // ...
  }
}
```

And then make the same changes to `SunService`:

```scala
class SunService(wsClient: WSClient) {
  def getSunInfo(lat: Double, lon: Double): Future[SunInfo] = {
    val responseF = wsClient.url("http://api.sunrise-sunset.org/json?" +
      s"lat=$lat&lng=$lon&formatted=0").get()
    // ...
  }
}
```

In the `Application` class, instead of creating both services, we can pass references to them via constructor:

```scala
class Application(sunService: SunService,
    weatherService: WeatherService) extends Controller {

  def index = Action.async {
    // ...
  }
}
```

The only thing that is left is to create the actual service instances in `AppComponents` and wire everything together:

```scala
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  lazy val sunService = new SunService(wsClient)
  lazy val weatherService = new WeatherService(wsClient)
}
```

What about the `wire` macro? Well, during compilation this macro will be replaced with a real instantiation, so instead of typing `new SunService(wsClient)` we can use `wire[SunService]` (the `wsClient` argument will be passed automatically) and save a couple keystrokes:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  lazy val sunService = wire[SunService]
  lazy val weatherService = wire[WeatherService]
}
```

Remember that you need to `reload` the application in the activator console every time you change `build.sbt`. You can always stop the app without exiting `activator` by pressing Ctrl+D.

If you refresh the page now, you will see that the app is still working. Now, however, it's a perfectly modular app.

Managing application lifecycle

What if you want to perform some task when the application stops or starts? Before version 2.4, you could extend the `GlobalSettings` trait and override the `onStart`/`onStop` methods. With the introduction of dependency injection, this approach was deprecated.

If you look at our `AppComponents` trait, you will see that it extends `BuiltInComponents`, which in turn, has a field called `applicationLifecycle`. This field allows you to register a callback that will be invoked when the app is about to stop:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  applicationLifecycle.addStopHook { () =>
    Logger.info("The app is about to stop")
    Future.successful(Unit)
  }
}
```

Note that the callback must return `Future[Unit]`. This means that we could wrap the entire method body in `Future` and perform our tasks asynchronously. Here, however, we're simply logging a message and returning a successful `Future` to comply with the method signature.

If you need to perform some tasks when the app is starting, you can simply create a non-lazy field inside the `AppComponents` traits and initialize it with a code block:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  val onStart = {
    Logger.info("The app is about to start")
  }
}
```

After refreshing the page the following messages will appear in the log:

```
[info] - application - The app is about to start
[info] - play.api.Play - Application started (Dev)
[info] - application - The app is about to stop
```

Intercepting requests with filters

In the next two sections, we are going to start counting the number of requests happened since the last application reload and showing this information on the home page. The first task is to increment the counter when a new request enters the system.

Many Web frameworks, including classic Java Servlets, allow to register so-called *HTTP filters* which will be invoked after the request is received by the system but before it gets into the controller. Play also allows to intercept incoming requests by registering `Filter` implementations.

Creating a new filter is surprisingly easy. All we need to do is to extend the `play.api.mvc.Filter` trait and register our newly created filter in the request pipeline.

Since Play 2.5.0 is tightly integrated with Akka Streams, all `Filters` are required to have an implicit `Materializer` defined. For us, though, it's not going to be a problem, because we can simply define a `Materializer` value as a constructor parameter and MacWire will take care of the rest.

Let's start with creating a class called `StatsFilter`:

```
class StatsFilter(implicit val mat: Materializer) extends Filter {
  override def apply(nextFilter: (RequestHeader) => Future[Result])
                    (header: RequestHeader): Future[Result] = {
    Logger.info(s"Serving another request: ${header.path}")
    nextFilter(header)
  }
}
```

As you can see, the `apply` method is a curried function and inside of it you have access to the request header as well as the next action in the chain. Invoking `nextFilter(header)` transfers control to the next action and request processing continues as usual.

Since we are using compile-time DI, we also need to tell Play to include our filter in its chain. This can be achieved by overriding the `httpFilters` field from the `BuildInComponents` trait:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  lazy val statsFilter: Filter = wire[StatsFilter]
  override val httpFilters = Seq(statsFilter)
  // ...
}
```

After that, each time the app is serving a request, you will see messages like these in the log:

```
[info] - application - Serving another request: /
[info] - application - Serving another request: /assets/images/favicon.png
```

Note that when a browser requests an image, this request goes through the usual filter chain including user-defined filters as well. The fact that this request is not processed by a user-defined controller action doesn't matter - it will still be processed by a built-in Play controller called `Assets`.

Using Akka

When you need to simply perform an asynchronous task, you should probably stick to using `Futures`. If, however, you need to also manage mutable state, it's better to use another abstraction called *Actor*. Actors are basically objects that communicate with each other and the outside world using immutable messages. Actors themselves can have mutable state and usually they do.

One popular implementation of the actor model is called Akka[39]. Akka is written in Scala and can be used from any JVM language. It is very well documented and high performant. Besides, it's integrated into Play, so using actors in Play is absolutely straightforward.

When you create an actor, you need to extend `akka.actor.Actor` and implement the `receive` method by supplying a `PartialFunction[Any, Unit]`. It's also very common to create a companion object for your actor. This object usually defines the messages that the actor accepts and some constants.

We will need three messages. During the start-up phase of the application, the start-up code will send a `Ping` message to the actor. The `Ping` itself doesn't do anything other than ensures that the actor is started and ready to accept other messages. When a new request comes, the `StatsFilter` will send

[39]http://akka.io/

another message called `RequestReceived`. The `StatsActor` reacts to this message by incrementing the counter. Finally, the `Application` controller can send a message called `GetStats` and the actor will respond by sending the current value of the counter back. In addition to these three messages, the actor needs a name and path. The name is used during the creation of an actor and the path is used for obtaining a reference to it.

Let's define what we've just discussed inside the companion object:

```scala
object StatsActor {
  val name = "statsActor"
  val path = s"/user/$name"

  case object Ping
  case object RequestReceived
  case object GetStats
}
```

Using case objects is a very common approach that is used when your message doesn't need any parameters. As for the `path`, when a new actor is created with a given name, Akka assigns to this actor a path that starts with `"/user/"` followed by this name.

Using already defined messages, we can write an implementation for the actor itself:

```scala
class StatsActor extends Actor {
  var counter = 0

  override def receive: Receive = {
    case Ping => ()
    case RequestReceived => counter += 1
    case GetStats => sender() ! counter
  }
}
```

The `sender()` method is available within `Actor` and can be used to get the reference to the sender of the request. The `!` method, also known as `tell`, follows the "fire-and-forget" semantics, i.e sends the message without expecting anything in response.

The `actorSystem` (available as a field in `BuiltInComponents`) is responsible for creating and managing actors, so it's only logical to put actor initialization logic in the `AppComponents` trait:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  lazy val statsActor = actorSystem.actorOf(
    Props(wire[StatsActor]), StatsActor.name)
}
```

Here we're using Akka's `Props` class to create the actor. After the actor is created, we can initialize it by sending it the `Ping` message:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...
  val onStart = {
    Logger.info("The app is about to start")
    statsActor ! Ping
  }
}
```

When `StatsFilter` gets notified about another accepted request, it should send the `RequestReceived` message to `StatsActor`:

```
class StatsFilter(actorSystem: ActorSystem,
  implicit val mat: Materializer) extends Filter {
  override def apply(nextFilter: (RequestHeader) => Future[Result])
                    (header: RequestHeader): Future[Result] = {
    // ...
    actorSystem.actorSelection(StatsActor.path) ! StatsActor.RequestReceived
    nextFilter(header)
  }
}
```

Note that we need to pass a reference to an `ActorSystem` as a constructor parameter. After that we will be able to select the required actor by its path.

In the `Application` controller we need to obtain the current counter value from the actor. This time, however, we cannot use the "fire-and-forget" approach of the `!` method. Instead we need to use another method called `?` (also known as `ask`). There are several things that you need to keep in mind when working with the ask pattern in Akka:

- the `ask` pattern is not there by default and must be imported separately from `akka.pattern`
- an `implicit` value specifying the timeout must be in scope
- the `?` returns an untyped `Future` (i.e `Future[Any]`) that must be cast manually with `mapTo`

With these considerations in mind, let's make necessary changes in the `Application` controller:

```
class Application(sunService: SunService,
    weatherService: WeatherService,
    actorSystem: ActorSystem) extends Controller {

  def index = Action.async {
    // ...

    implicit val timeout = Timeout(5, TimeUnit.SECONDS)
    val requestsF = (actorSystem.actorSelection(StatsActor.path) ?
      StatsActor.GetStats).mapTo[Int]

    for {
      sunInfo <- sunInfoF
      temperature <- temperatureF
      requests <- requestsF
    } yield {
      Ok(views.html.index(sunInfo, temperature, requests))
    }
  }
}
```

Note, that since we are using MacWire, we don't need to change initialization lines that contain `wire[Application]` and `wire[StatsFilter]`. The new constructor parameters will be populated by the macro automatically using the `actorSystem` value from the `BuildInComponents` trait.

On the home page, we simply need to add a new parameter to the template and show it in a new `div`:

```
@(sunInfo: model.SunInfo, temperature: Double, requests: Int)
<!DOCTYPE html>
<html lang="en">
    <!-- the same -->
    <body>
        <!-- the same -->
        <div>Requests: @requests</div>
    </body>
</html>
```

If you refresh the page several times, you will see that the counter is increasing.

One interesting application of using Akka with Play is performing tasks on schedule. Refer to official documentation[40] if you need details.

[40]https://www.playframework.com/documentation/2.5.x/ScalaAkka

Frontend integration

Let's take a break from the backend development and see how we can integrate Play with modern frontend tools. After all, using Scala templates is fine and good, but in many cases it's not enough. Users want more interactivity and developers respond with writing single-page applications (SPAs).

Today it's hard to imagine any serious Web development without NodeJS. Even though SBT plugins performing frontend related tasks exist, they usually lag behind the original libraries written for Node. Besides, we are going to use NodeJS only during development and installing it on production servers is completely unnecessary.

Setting up nvm

Since NodeJS is developing very rapidly and new versions appear very often, it's a good idea to install the Node Version Manager (`nvm`). The node version manager will allow you to use different versions of NodeJS in different projects and switch between them in a matter of seconds.

Here is what you need to do in order to install `nvm`:

- Go to https://github.com/creationix/nvm[41] and scroll down to the Install script section
- Copy the wget or curl script and run it in the terminal
- After installation is complete, restart the terminal

Before we continue, I just wanted to make one important clarification. When people talk about NodeJS in the context of Web development, they usually mean two things:

1. node - the Node JavaScript interpreter based on V8 engine
2. npm - the Node Package Manager, which comes with Node and helps with downloading and installing node-packages, running JS scripts

Fortunately, `nvm` manages the versions of both tools.

Choosing the versions

After `nvm` is installed, you can start using it.

To see the list of available NodeJS versions type:

[41]https://github.com/creationix/nvm

```
$ nvm ls-remote
```

It will connect to the remote server and show the list of NodeJS versions available for installation. I used version v5.3.0 while working on this book, so you can do the same. Simply type:

```
$ nvm install v5.3.0
```

`nvm` will download necessary packages and store the specified version of NodeJS locally, so it won't need to download it again. To start using it, type:

```
$ nvm use v5.3.0
```

By default, when you open a new session in the terminal, you need to type the `use` command to make `npm` and `node` executables available. If you don't want to do it every time, just decide on one good version of NodeJS that will be used for most projects and then fix this version by adding the following line to the very end of your `.bashrc`:

```
nvm use v5.3.0 > /dev/null
```

After that you can check in a new terminal window that both `npm` and `node` commands reveal their versions:

```
$ npm --version
3.3.12
$ node --version
v5.3.0
```

package.json

The main file for `npm` is `package.json`. It contains information about the project such as project name, version, description, author etc. Since we're not going to publish our project, most of these fields are not important.

`npm` will create a new `package.json` file if you type the following in an empty directory :

```
$ npm init
```

There are three sections which are relevant for us. These sections are:

- `dependencies`

- `devDependencies`
- `scripts`

The `dependencies` section lists all dependencies that our application has. For example, if your application uses React, you'll need to add React to this section. Usually, you don't edit this section manually, but instead you should instruct `npm` to install the required package and save it as an application dependency. To do that simply type:

```
$ npm i react -S
```

The `-S` option instructs `npm` to add the package as an application dependency.

The `devDependencies` section lists all dependencies that your project uses during development. For example, if you want to use Webpack to build your app bundle, you will need to add `webpack` to this section via the following command:

```
$ npm i webpack -D
```

By the way, `npm` downloads all these dependencies to a directory called `node_modules`. This directory is usually not under version control, but it can be easily recreated by `npm` when you invoke the following:

```
$ npm i
```

Although we're not using this feature in this book, `npm` can also install packages globally with the `-G` flag. After installing a package globally, it will be accessible from any directory. For example, this way you can install a NodeJS-based HTTP-server to then start it wherever you want:

```
$ npm i http-server -G
```

The `scripts` section allows you to run scripts. The trick here is that these scripts will have access to packages from `node_modules`. For example, if you want to run Webpack from command-line, you will either need to install it globally (which is not recommended) or add the script to the `scripts` section:

```
{
  // ...
  "scripts": {
    "watch": "webpack --watch"
  }
  // ...
}
```

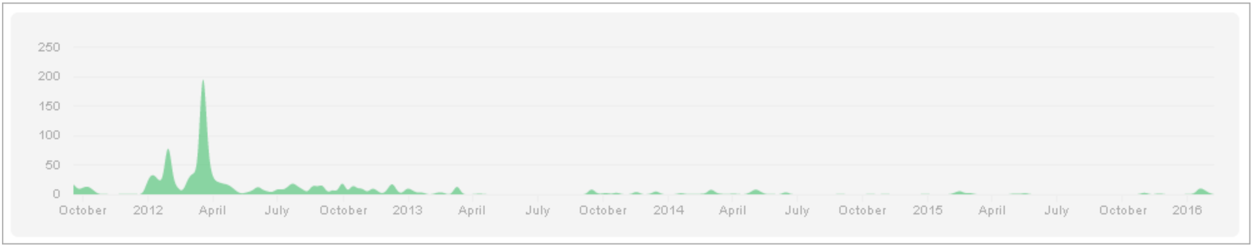
After doing this, you will be able to start Webpack by typing in the command line the following:

```
$ npm run watch
```

GruntJS and Gulp

Two popular build tools for JavaScript are GrutnJS and Gulp.

GruntJS was one of the pioneers of JavaScript automation and quickly became very popular in 2011-2012. It used plugins to perform common tasks and allowed to write build definitions in JavaScript. Even though GruntJS is not that popular anymore, these two ideas are still widely used.

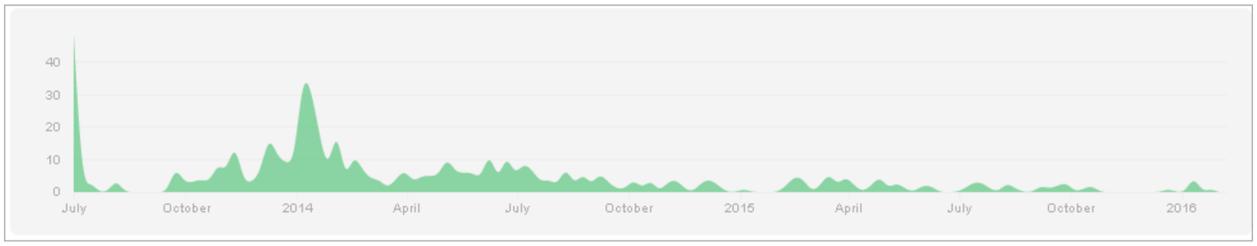


GruntJS commit chart

Gulp, released in 2013, took some ideas from GruntJS and improved them by leveraging pipes and streams. With Gulp, developers don't have to write intermediate files to disk. As an example of using pipes, here is a portion of `gulpfile.js` that describes a style sheet processing task:

```
gulp.task('minify-styles', function() {
    return gulp.src(paths.stylesSrc)
        .pipe(less())
        .pipe(minifyCss())
        .pipe(rename('styles.css'))
        .pipe(gulp.dest(paths.assetsDest));
});
```

The `minify-styles` task takes the contents of the directory referenced by `paths.stylesSrc`, translates the files from LESS to CSS, minifies the resulting CSS file, renames it and finally saves it to disk.



Gulp commit chart

As many surveys show, Gulp is still the most popular JavaScript build tool.

Module bundlers

When JavaScript was released, it didn't offer much support for modularity. Developers would reference several script files on the web page, and all definitions from all these files would be combined by the browser. Over time, several alternative solutions for bundling JavaScript files appeared and then finally official syntax was standardized as part of EcmaScript 6 specification.

At the moment, the two most popular approaches are *CommonJS* and *ES6 modules*. Loading modules with CommonJS looks like this:

```
var React = require('react');
var _ = require('lodash');
var HelloWorld = require('./helloworld.jsx');
```

Using EcmaScript 6 syntax looks like this:

```
import React from 'react';
import _ from 'lodash';
import HelloWorld from './helloworld.jsx';
```

In any case, you split your source code into a number of modules that need to be tied together in order for you application to work. This is what *module bundlers* are for. Essentially, module bundlers analyze your source files and form one or several *bundles* that could be executed by the browser.

Again, there are many different choices but usually people either use Browserify[42] with Gulp or Webpack[43]. It is possible to use Webpack as a Gulp plugin, but it's not necessary as Webpack can easily handle most frontend tasks.

Since we're aiming at *modern* Web development, we're going to use Webpack as the main frontend build tool.

[42]http://browserify.org/
[43]https://webpack.github.io/

Using EcmaScript 6 with Babel

Changing language specification is a difficult process. Changing the language that is continuously in production since 1995 and has different implementations is even more challenging. There is a good news, though. As it turned out JavaScript is extremely flexible, so it's often possible to implement the features of the next specification using the old one.

One of the most popular projects that allow you to use new features of the next version JavaScript today is Babel[44]. It takes your source code written using EcmaScript 6 (or EcmaScript 7) syntax and turns it into JS code that modern browsers understand.

Babel uses plugins to transform the code. You can select which syntactical features you need or use a preset which combines a number of those features. The usual recommendation is to start with the ES2015 preset and we are going to follow it.

Explaining React

React[45] is a JavaScript library developed by Facebook and open-sourced in 2013. It combines several unconventional approaches to typical frontend problems, which appeals to many developers. In particular, React promotes the idea of writing HTML-tags alongside JavaScript code. It may look crazy at first, but many agree that this approach gives a lot of flexibility. The support for JSX files is available in Atom via the React plugin[46] and in IntelliJ Ultimate out of the box.

Unlike more feature-rich solutions like AngularJS, React is only concerned with the *View* in MVC (Model-View-Controller), which can be both a good thing or bad thing. On the bright side, it means that React is very unopinionated about the technology stack you are going to use and thus compatible with almost everything. The bad news is that many familiar components of a typical JavaScript framework are not provided out of the box.

For example, AngularJS[47] comes with a built-in service called `$http`, which allows you to query your backend from the client easily:

```
$http( { method: 'GET', url: '/api/data' } ).
  then(function successCallback(response) {
    console.log('Data received: ', response);
  }, function errorCallback(response) {
    console.log('Error occurred!');
  });
```

[44] https://babeljs.io/
[45] http://facebook.github.io/react/
[46] https://atom.io/packages/react
[47] https://angularjs.org/

The `$http` service returns a *promise*, which resembles Scala's Future. Once you have a reference to the promise, you can register callbacks or transform it.

React doesn't have have the `$http` service, but you can use the `fetch` function from EcmaScript 6, which is available natively in Chrome and in other browsers through polyfill:

```
fetch('/api/data').then(function(response) {
    return response.json();
  }).then(function(json) {
    console.log('Data received: ', json);
  });
```

As you will see, the lack of built-in service like `$http` can be easily overcome and in the end you will have a system which is more flexible and customizable than an all-in-one solution.

Separating the frontend

Right now we have the following directory structure:

```
$ tree -L 1
.
├── activator
├── activator.bat
├── activator-launch-1.2.3.jar
├── app
├── build.sbt
├── conf
├── LICENSE
├── logs
├── project
├── public
├── README
├── target
└── test
```

Most of these files and directories are part of the backend. The `public` directory contains static frontend assets, which Play uses. Basically, it means that in our setup Play views already compiled, bundled and possibly minified JavaScript files as static resources. When the browser encounters a script tag like this one:

```
<script src="/assets/compiled/bundle.js"></script>
```

it simply sends a request to the server, i.e Play. Play *knows* that when the path starts with "/assets/", it needs to send back a file from the public directory. Play *doesn't know* how this file was created - either typed by a developer in a text editor or compiled from EcmaScript 6-source with Webpack.

We're going to keep this separation of concerns by creating a directory called `ui`:

```
$ mkdir ui
```

We will store our source JavaScript (ES6) files in this directory. During the frontend compilation stage Webpack will combine them to create a single `bundle.js` file and put it into `public/compiled`. Let's create this directory as well:

```
$ mkdir public/compiled
```

Initiliazing the frontend project

Even though we are going to store frontend sources in the `ui` directory, we can still initialize our frontend project in the root directory. Let's start by creating the `package.json` file:

```
$ npm init
```

`npm` will ask several questions, which are all irrelevant for us, so we can simply press Enter several times until it finishes. Then we need to install some NodeJS-packages, including Webpack and Babel, as development dependencies:

```
$ npm i babel-core babel-loader babel-preset-es2015 babel-preset-react exports-l\
oader imports-loader webpack -D
```

After development dependencies are installed, we need to install React and the `fetch` polyfill as application dependencies:

```
$ npm i react react-dom whatwg-fetch -S
```

Finally, we can add a new `scripts` entry to the `package.json`:

```
{
  // ...
  "scripts": {
    "watch": "webpack --watch"
  }
  // ...
}
```

webpack.config.js explained

Now that we have everything in place, let's create a new file in the project root directory called `webpack.config.js` with the following content:

```
var webpack = require('webpack');

module.exports = {
  entry: './ui/entry.js',
  output: { path: __dirname + '/public/compiled', filename: 'bundle.js' },
  module: {
    loaders: [
      { test: /\.jsx?$/, loader: 'babel-loader',
        include: /ui/, query: { presets: ['es2015', 'react'] } }
    ]
  },
  plugins: [
    new webpack.ProvidePlugin({
      'fetch': 'imports?this=>global!exports?global.fetch!whatwg-fetch'
    })
  ]
}
```

First, we're importing Webpack definitions using CommonJS syntax. Then we're specifying the main entry of our application. Webpack will start building a bundle with the `ui/entry.js` file. The output will be placed to `public/compiled` and named `bundle.js`. So far so good. Then it gets more interesting:

```
// ...
module: {
    loaders: [
      {
        test: /\.jsx?$/,
        loader: 'babel-loader',
        include: /ui/,
        query: { presets: ['es2015', 'react'] }
      }
    ]
 //...
```

Out of the box, Webpack can work only with standard JavaScript files and it uses loaders to work with everything else. Here we're specifying the regex that limits the `babel-loader` to process only `js` and `jsx` files. The query part instructs Babel to use ES2015 and React presets.

The last part looks quite cryptic but in essence it allows us to use the `fetch` function as if it was already released and available as part of standard JavaScript.

Using React

Let's start using React by creating a new `SunWeatherComponent` in a new file called `sun-weather-component.jsx` (in `ui/scripts`):

```
import React from 'react';

module.exports = React.createClass({
  getInitialState: function() {
    return {
      sunrise: undefined,
      sunset: undefined,
      temperature: undefined,
      requests: undefined
    }
  },
  render: function() {
    return <div>
      <div>Sunrise time: {this.state.sunrise}</div>
      <div>Sunset time: {this.state.sunset}</div>
      <div>Current temperature: {this.state.temperature}</div>
      <div>Requests: {this.state.requests}</div>
    </div>
  }
});
```

The `getInitialState` function defines the initial value of the `this.state` field. This field is used in the `render` function to render the component. The React template is partially copied from the `index.scala.html`, in which we can replace the informational part with an empty div:

```
@(sunInfo: model.SunInfo, temperature: Double, requests: Int)
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
        <link rel="shortcut icon" type="image/png"
           href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello Play</h1>
        <div id="reactView"></div>
    </body>
    <script src="@routes.Assets.versioned("compiled/bundle.js")" ></script>
</html>
```

Here we also added a script tag. We are not going to use the template parameters, but let's keep them for now.

We will need two more files - `ui/scripts/main.js` and `ui/entry.js`. The `entry.js` is simple. All it does is import `main.js`:

```
import './scripts/main.js';
```

The `main.js` itself is slightly more sophisticated:

```
import React from 'react';
import ReactDOM from 'react-dom';
import SunWeatherComponent from './sun-weather-component.jsx';

ReactDOM.render(<SunWeatherComponent />, document.getElementById('reactView'));
```

Here we are importing everything we need, rendering the SunWeatherComponent and putting it in place of the empty `div` we've just created in `index.scala.html`.

Try running webpack via npm:

```
$ npm run watch
Hash: 7e374197371e6f7099d0
Version: webpack 1.12.13
Time: 1185ms
    Asset     Size  Chunks             Chunk Names
bundle.js   678 kB       0  [emitted]  main
    + 161 hidden modules
```

If you refresh the page now, you will see that it basically works, but all numbers have disappeared.

Sending data via JSON

The simplest way to get our numbers back is to send them to the client via JSON. In order to do that, it's better to create a new class that will combine data from `SunInfo` with temperature information and the number of requests. So, here is the `CombinedData` class:

```scala
case class CombinedData(sunInfo: SunInfo, temperature: Double, requests: Int)
```

If we want to be able to send objects of this class via JSON, we need to tell Play how to serialize them. There are several ways to do that, but probably the easiest one is to create an `implicit` value of type `Writes[CombinedData]` in the companion object:

```scala
object CombinedData {
  implicit val writes = Json.writes[CombinedData]
}
```

The `Json.writes` helper method already knows how to serialize primitives and collections, but for the `SunInfo` class we also need to add an implicit `Writes[SunInfo]`:

```scala
case class SunInfo(sunrise: String, sunset: String)

object SunInfo {
  implicit val writes = Json.writes[SunInfo]
}
```

The next step is to move all data related logic into a new controller method that will return JSON instead of a full-blown web page:

```scala
class Application(sunService: SunService,
    weatherService: WeatherService,
    actorSystem: ActorSystem) extends Controller {

  def index = Action {
    Ok(views.html.index())
  }

  def data = Action.async {
    val lat = -33.8830
    val lon = 151.2167
    val sunInfoF = sunService.getSunInfo(lat, lon)
    val temperatureF = weatherService.getTemperature(lat, lon)

    implicit val timeout = Timeout(5, TimeUnit.SECONDS)
    val requestsF = (actorSystem.actorSelection(StatsActor.path) ?
      StatsActor.GetStats).mapTo[Int]

    for {
      sunInfo <- sunInfoF
      temperature <- temperatureF
      requests <- requestsF
    } yield {
      Ok(Json.toJson(CombinedData(sunInfo, temperature, requests)))
    }
  }
}
```

Basically, we're moving everything from `index` to `data`. The difference is the return value: the `data` method sends `CombinedData` serialized as JSON. The `index` method doesn't need all this information anymore and simply returns the `index` page. Therefore, we can finally remove the template parameters from the `index.scala.html`:

```html
@()
<!DOCTYPE html>
<html lang="en">
<!-- omitted -->
</html>
```

We also need to add the following line to `conf/routes` in order to expose the JSON endpoint:

```
GET     /data                         controllers.Application.data
```

Finally, we can use the `fetch` function inside of our React component to load the data:

```
module.exports = React.createClass({
  // ...
  componentDidMount: function() {
    let reactRef = this;
    fetch('/data').then(function(response) {
      return response.json();
    }).then(function(json) {
      reactRef.setState({
        sunrise: json.sunInfo.sunrise,
        sunset: json.sunInfo.sunset,
        temperature: json.temperature,
        requests: json.requests
      });
    });
  },
  // ...
});
```

We're using the `componentDidMount` callback function that is invoked after the initial rendering happened. First, we're saving `this` in a variable and after we get our JSON parsed, we're changing the state of the component. React will receive these changes and re-render the component. If you refresh the page now, you will see that all numbers are back.

Using Sass with Webpack

It's hard to imagine a Web application that doesn't use CSS in some way, so let's add some styling. We are going to use Sass - one of the most popular preprocessors. CSS preprocessors introduce useful things that are absent in standard CSS such as variables, functions and mixins. They make it easier to target multiple browsers and use experimental features which are only available with prefixes. On top of Sass we will use the ConciseCSS[48] framework. It is not as feature rich as Bootstrap or Foundation, but it offers a great deal of functionality while staying relatively lightweight.

Sass is available in two flavours - *original sass* and *scss*. SCSS uses curly braces for structuring while SASS uses indents. We are going to stick to SCSS as this syntax is more popular among developers.

First, we need to create a new directory in the `ui` folder:

[48]http://concisecss.com/

```
mkdir -p ui/styles/vendor/concise
```

By putting all third-party libraries in the `vendor` folder we will be able to distinguish between their code and our code.

The easiest way to install ConciseCSS is to simply download the latest release as a ZIP file. Once you have the archive, extract it anywhere and copy the insides of the `src` directory to `ui/styles/vendor/concise`, so that the `styles` directory has the following structure:

```
$ tree -L 3
.
└── vendor
    └── concise
        ├── addons
        ├── concise.scss
        ├── core
        └── custom

5 directories, 1 file
```

After that, we need to create a new file called `style.scss`. This file will work as an aggregator and simply reference other SCSS files. Right now, it will have only one line:

```
@import 'vendor/concise/concise';
```

Finally, we need to add `style.scss` to `entry.js`, so that Webpack will know that it needs to process styles as well:

```
import './scripts/main.js';
import './styles/style.scss';
```

This looks strange because we're referencing a SCSS file from a JavaScript file, but since we've specified the `entry.js` file as the entry of our bundle, it makes sense for Webpack. One challenge remains, though. Webpack doesn't know how to handle SCSS files. Nor does it know about CSS files. Nor styles in general. So, as with JSX files, we need to use loaders. Let's install them:

```
$ npm i sass-loader node-sass css-loader style-loader -D
```

If you look at the Sass-loader GitHub page[49], you will see that it is recommended to apply three loaders - `style`, `css`, `sass` - sequentially:

[49]http://github.com/jtangelder/sass-loader

```
var webpack = require('webpack');

module.exports = {
  // ...
  module: {
    loaders: [
      // ...
      { test: /\.scss$/, loaders: ["style", "css", "sass"]}
    ]
  },
  // ...
}
```

Note that loaders are applied from right to left. So, first Webpack will use *sass-loader* to transform Sass to CSS, then it will use *css-loader* to transform CSS to style. Finally, it will use *style-loader* to apply the styles to you web-page using JavaScript. If you try relaunching Webpack now, it will emit the updated `bundle.js` file, which now contains not only React logic, but also styling information!

Try refreshing the browser to see that it actually works.

Extracting styles

Most developers prefer to separate styling information from application logic. Fortunately, it's certainly possible to do it in Webpack with the help of *ExtractTextPlugin*. Let's start with installing this plugin via npm:

```
npm i extract-text-webpack-plugin -D
```

Then we need to adjust our `webpack.cofig.js` so that Webpack knows that it needs to extract styling information into a separate file called `styles.css`:

```
var webpack = require('webpack');
var ExtractTextPlugin = require("extract-text-webpack-plugin");

module.exports = {
  entry: './ui/entry.js',
  output: { path: __dirname + '/public/compiled', filename: 'bundle.js' },
  module: {
    loaders: [
      { test: /\.jsx?$/, loader:
        'babel-loader', include: /ui/, query: { presets: ['es2015', 'react'] } },
      { test: /\.scss$/, loader:
        ExtractTextPlugin.extract( "style", "css!sass") }
    ]
  },
  plugins: [
    new webpack.ProvidePlugin({
      'fetch': 'imports?this=>global!exports?global.fetch!whatwg-fetch'
    }),
    new ExtractTextPlugin("styles.css")
  ]
}
```

Here we're importing the `ExtractTextPlugin` and specifying the output file that will contain styling information. In the loaders section we are using interesting syntax - `css!sass`. This is a shortcut for applying *css-loader* to the output of *sass-loader*.

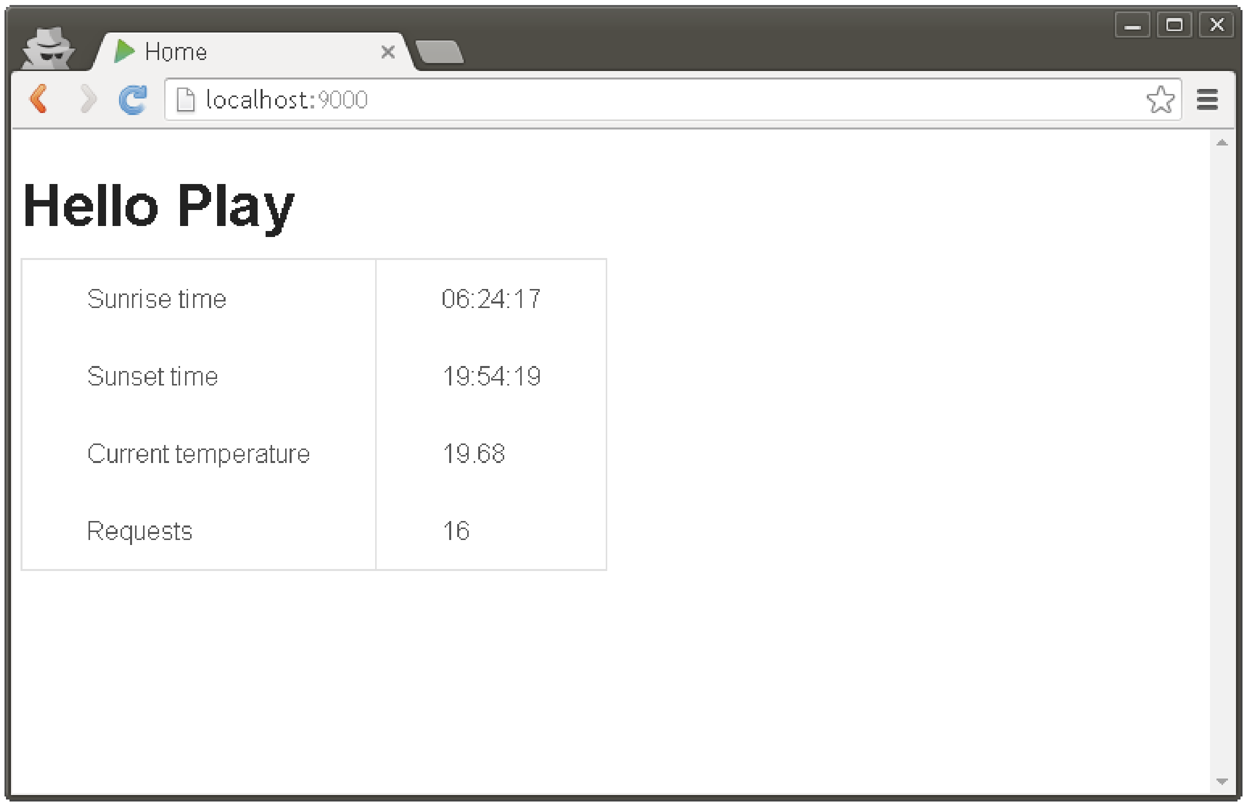
If you try relaunching Webpack, you will see that now it emits two files - `bundle.js` and `styles.css`. It means that in order to get new styles we need to reference the `styles.css` in the `index.scala.html`:

```
@()
<!DOCTYPE html>
<html lang="en">
    <!-- omitted -->
        <link rel="stylesheet"
          href="@routes.Assets.versioned("compiled/styles.css")">
        <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.versioned("images/favicon.png")">
    <!-- omitted -->
</html>
```

Let's also convert our `div` tags into a table. This will demonstrate that our styling is indeed working as ConciseCSS provide styles for the `table` tag which are different from default:

```
import React from 'react';

module.exports = React.createClass({
  // ...
  render: function() {
    return <table>
      <tbody>
        <tr>
          <td>Sunrise time</td>
          <td>{this.state.sunrise}</td>
        </tr>
        <tr>
          <td>Sunset time</td>
          <td>{this.state.sunset}</td>
        </tr>
        <tr>
          <td>Current temperature</td>
          <td>{this.state.temperature}</td>
        </tr>
        <tr>
          <td>Requests</td>
          <td>{this.state.requests}</td>
        </tr>
      </tbody>
    </table>
  }
});
```

After this change, the home page should look like following:



Hello Play

Sunrise time	06:24:17
Sunset time	19:54:19
Current temperature	19.68
Requests	16

Presenting data in a table

You may also notice that the font was changed. This is because ConciseCSS overrides the `font-face` CSS property making it *sans-serif* instead of *serif*.

Authenticating users

In this part, we will apply everything we've learned and implement a simple authentication system. There are several third-party solutions available, but we are going to roll out our own authenticating mechanism using only Play API. Since we need some way to store user information, we will start by looking at database support in Scala.

Installing PostgreSQL

Relational databases are probably the most popular tool when it comes to storing your data. They are fast, reliable and supported by most programming languages and frameworks. Among existing open-source databases, PostgreSQL stands out as one of the most advanced.

If you are using Ubuntu 15.X, you can find PostgreSQL 9.4 in the official distribution repository. If you prefer slightly older versions of the OS, you can add the official PostgreSQL PPA (personal package archive) and install version 9.4 from there. Relevant instructions can be found in the Ubuntu Downloads section of the official PostgreSQL website[50].

We will need two packages - the core server and the contrib package. I would also recommend installing `pgAdmin`, a graphical tool that allows to query the database and visualize the results:

```
$ sudo apt-get install postgresql-9.4 postgresql-contrib-9.4 pgadmin3
```

After installation is completed, Ubuntu will automatically start the server. Our next step is to create a database and user. The tricky part here is to use the `postgres` Linux user:

```
$ /usr/bin/sudo -u postgres psql --command "CREATE USER scalauser WITH SUPERUSER\
 PASSWORD 'scalapass';"
$ /usr/bin/sudo -u postgres createdb -O scalauser scaladb
```

Here we're creating a new user called `scalauser` and a database called `scaladb`.

Adding ScalikeJDBC

Since all database-related tools in Scala use JDBC (Java Database Connectivity), we need to add PostgreSQL JDBC drivers to the list of dependencies in `build.sbt`:

[50] http://www.postgresql.org/download/linux/ubuntu/

```
// ...

libraryDependencies ++= Seq(
  // ...
  "org.postgresql" % "postgresql" % "9.4.1207.jre7"
)

// ...
```

Since we're using Java 8, we can safely use a version built for JRE7.

You can always find actual versions of drivers on MVN Repository[51].

For database access we are going to use ScalikeJDBC[52]. This is a great library that has everything we need. It gives us an intuitive yet safe way to write SQL queries, maps database `NULLs` to Scala `Nones`, handles transactions and so on. Above all, configuration is also straightforward.

Let's add ScalikeJDBC library files to the list of dependencies:

```
// ...

libraryDependencies ++= Seq(
  // ...
  "org.scalikejdbc" %% "scalikejdbc"        % "2.3.5",
  "org.scalikejdbc" %% "scalikejdbc-config"  % "2.3.5",
  "ch.qos.logback"  %  "logback-classic"    % "1.1.3"
)

// ...
```

After all required libraries are there, we can edit the `application.conf` file to specify database connection properties:

[51]http://mvnrepository.com/artifact/org.postgresql/postgresql
[52]http://scalikejdbc.org/

```
play.crypto.secret = "changeme"
play.i18n.langs = [ "en" ]
play.application.loader = "AppApplicationLoader"

db.default.driver=org.postgresql.Driver
db.default.url="jdbc:postgresql://localhost:5432/scaladb"
db.default.username=scalauser
db.default.password=scalapass

db.default.poolInitialSize=1
db.default.poolMaxSize=5
db.default.ConnectionTimeoutMillis=1000

play.evolutions.autoApply=true
```

Here we're also specifying the connection pool settings. Using *Connection pooling*, ScalikeJDBC will be able to reuse existing connections instead of opening new ones thus improving the performance of the app.

Note that the word `default` (for example in `db.default.driver`) specifies how Play will refer to our database.

Since we're using the `scalikejdbc-config` helper package, the initialization is extremely simple. We only need to call `DBs.setupAll()` when the app is starting and `DBs.closeAll()` when it's stopping. Obviously, the best place for this code is the `AppComponents` trait:

```scala
trait AppComponents extends BuiltInComponents with AhcWSComponents {
  // ...

  applicationLifecycle.addStopHook { () =>
    Logger.info("The app is about to stop")
    DBs.closeAll()
    Future.successful(Unit)
  }

  val onStart = {
    Logger.info("The app is about to start")
    DBs.setupAll()
    statsActor ! Ping
  }
}
```

Storing passwords

Before creating the `Users` table, let's think a little bit about the information we need to store. We obviously need to store login names or user codes. Since we're dealing with relational database we also need to store some internal identifier as a key and UUIDs (Universally unique identifiers[53]) will be good for that. What about passwords?

Experience shows that storing passwords in plain text is a really bad idea. In case our database is stolen, all the users will have to change their passwords. However, as it turns out, we actually don't need to store user passwords in the database. It's enough to store the hash of the password and we will be able to identify users just fine. Here is how it works.

- A user chooses a password and supplies it during the registration process
- On the server side, we take this password and calculate its hash using, not surprisingly, a hash function. Hash functions are one-way functions, i.e. calculating the output by the given input is easy, but reconstructing the input from the output is virtually impossible.
- Then we store the hash in the database against the user code
- Next time the user tries to login, they supply their original password once again. We again calculate its hash and then check whether it matches the one stored in the database. If it does, we authenticate the user.

This approach has only one little problem. Most hash functions are well-known and there are tables (called *rainbow tables*) that contain already calculated hashes for millions of passwords. Using these tables, it becomes possible to find an original password by its hash. This, however, is easy to counteract by appending some additional (possibly randomly generated) text, called *salt*, to the original password before hashing. The important part here is to also store salt as it's needed for authentication. The complete algorithm, therefore, looks like this:

- A user chooses a password and supplies it during the registration process. No changes here.
- On the server side, we take this password, generate some salt, mix them together and calculate the hash of this mix.
- Then we store both password and salt in the database
- Next time the user tries to login, we take their supplied password, find salt in the database, mix them together and calculate the hash of this mix. By matching it with the already stored hash, we can decide whether to authenticate the user or not.

As for the hash function itself, we are going to use a library called jBCrypt[54]. It's written in Java, so we can use it easily in Scala. Let's add the library's Maven coordinates[55] to the list of dependencies:

[53]https://en.wikipedia.org/wiki/Universally_unique_identifier
[54]http://www.mindrot.org/projects/jBCrypt/
[55]http://mvnrepository.com/artifact/de.svenkubiak/jBCrypt

```
// ...

libraryDependencies ++= Seq(
  // ...
  "de.svenkubiak" % "jBCrypt" % "0.4.1"
)

// ...
```

Before staring to write actual code that uses a new library, it is always good to try it in the Scala interpreter. We can switch to the interpreter easily from Activator (or sbt). Simply press Ctrl+D to stop the server, `reload` the project to update dependencies and type `console`:

```
[scala-web-project] $ reload
[scala-web-project] $ console
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.7.
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

The great thing about starting the interpreter this way is that you will have access to all application code that you have written and all libraries listed in the dependencies section. Since we added jBCrypt to `build.sbt`, we can use it here:

```
scala> import org.mindrot.jbcrypt.BCrypt
import org.mindrot.jbcrypt.BCrypt

scala> val hash = BCrypt.hashpw("password123", BCrypt.gensalt())
hash: String = $2a$10$niF.amAexQMHaevqlkganeSjvMHfTq/OdISyj8/5BQy1FHvlbi3Ne
```

Here we are using BCrypt to hash a simple password. BCrypt also generates some salt and appends it to the resulting string. This is the exact string that will be stored in the `Users` table. If we want to check the password, we can use the `checkpw` function:

```
scala> BCrypt.checkpw("password123", hash)
res0: Boolean = true

scala> BCrypt.checkpw("password321", hash)
res1: Boolean = false
```

Note that supplying the `checkpw` with incorrect password makes this function return `false`.

Managing database evolutions

It's finally time to create the `Users` table. In theory, we could start pgAdmin3 or some other graphical tool and use it to make changes to the database. However, there is a better approach. We can delegate managing the evolution of the database schema to Play. In this case, schema changes become part of source code and could be put into a version control system. Moreover, other developers will get the same database schema once they update their source code and start the app.

In order for Play evolutions to work, we need to put our SQL files in one specific location and make some additional steps. Since we specified connection properties to the database as `db.default`, we need to put evolution scripts in the `conf/evolutions/default`, so let's create a file called `1.sql` with the following content:

```
# --- !Ups

CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE users
(
    user_id UUID DEFAULT uuid_generate_v4() PRIMARY KEY,
    user_code VARCHAR(250) NOT NULL,
    password VARCHAR(250) NOT NULL
);

INSERT INTO users VALUES (uuid_generate_v4(), 'stest',
  '$2a$10$niF.amAexQMHaevqlkganeSjvMHfTq/OdISyj8/5BQy1FHvlbi3Ne');

# --- !Downs

DROP TABLE users;

DROP EXTENSION "uuid-ossp";
```

First, we're enabling the UUID extension as we want to use UUIDs as our primary keys. Then we're creating a table and finally inserting a test user. The test user will have the login "stest" and password "password123". Since jBCrypt appends salt to the hash, we don't need a separate column for storing salt.

We also have to add the `evolutions` Play module to the list of dependencies in `build.sbt`:

```
// ...
libraryDependencies ++= Seq(
  jdbc,
  cache,
  ws,
  evolutions,
  // ...
)
// ...
```

And finally, we need to enable evolutions in the `AppComponents` trait. The reasoning here is not obvious, so I'm showing it in detail, step-by-step:

- We need to apply evolutions when the app starts. In Play, our evolutions are applied when the `applicationEvolutions` field from the `EvolutionComponents` is resolved. Therefore, in order to get this field, we need to extend the `EvolutionComponents` trait.
- The `EvolutionsComponents` trait adds several abstract members, namely `dynamicEvolutions` (of type `DynamicEvolutions`) and `dbApi` (of type `DBApi`)
- The `dynamicEvolutions` abstract field is not a problem - we can simply create a new field and initialize it using the default constructor of the `DynamicEvolutions` class.
- The `dbAPi` instance can be found in the `DBComponents` trait, so let's extend it as well.
- Now we have `dbApi`, but `DBComponents` defines another abstract field called `connectionPool` (of type `ConnectionPool`).
- The `ConnectionPool` instance can be found in the `HikariCPComponents` trait and this trait doesn't introduce new abstract fields, so we are done here.
- Finally, since the `applicationEvolutions` field is marked as `lazy` we need to reference it in our `onStart` initialization code to make things work.

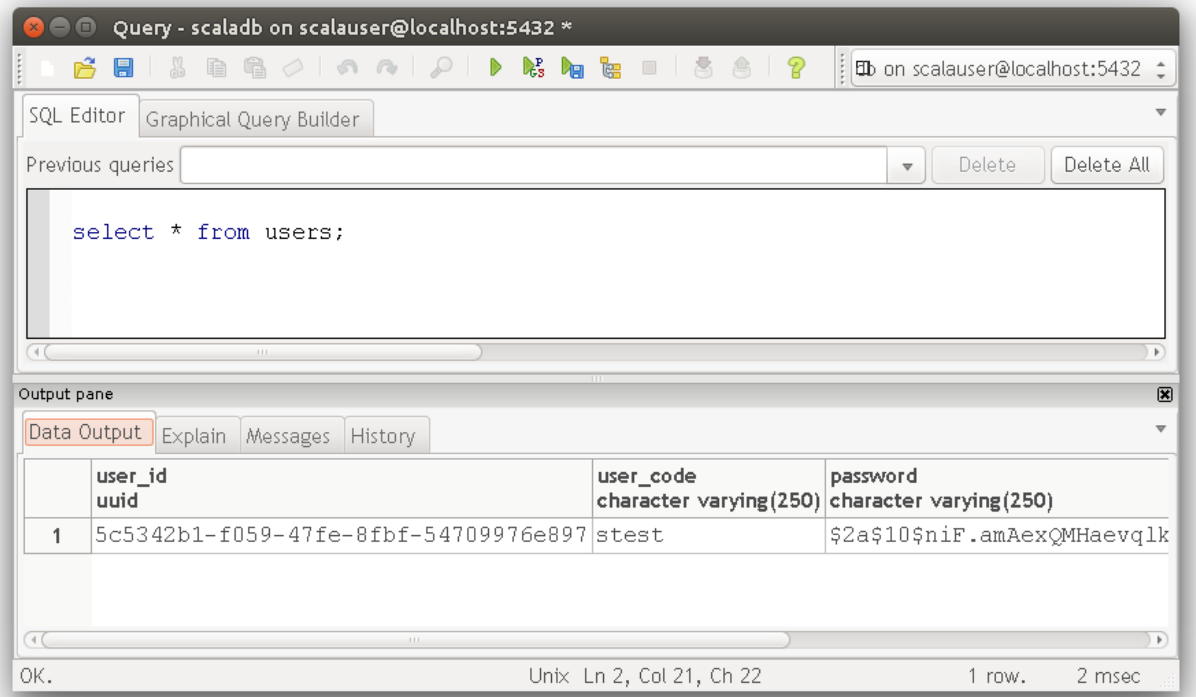
The reasoning involved a lot of words, but the code itself will be surprisingly concise:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents
 with EvolutionsComponents with DBComponents with HikariCPComponents {

  // ...
  lazy val dynamicEvolutions = new DynamicEvolutions

  // ...
  val onStart = {
    Logger.info("The app is about to start")
    applicationEvolutions
    DBs.setupAll()
    statsActor ! Ping
  }
}
```

If you refresh the browser page now, Play will apply the SQL script and a new table will be created. You won't find any indication about that, but you can try the following command in the console:

```
$ /usr/bin/sudo -u postgres psql -d scaladb --command "SELECT user_code,password\
 FROM users;" | more
 user_code |                           password
-----------+--------------------------------------------------------------
 stest     | $2a$10$niF.amAexQMHaevqlkganeSjvMHfTq/OdISyj8/5BQy1FHvlbi3Ne
(1 row)
```

Alternatively, you can use pgAdmin3 and see similar results:



```
select * from users;
```

```
   user_id                                  user_code                 password
   uuid                                     character varying(250)    character varying(250)
1  5c5342b1-f059-47fe-8fbf-54709976e897     stest                     $2a$10$niF.amAexQMHaevqlk
```

```
OK.    Unix  Ln 2, Col 21, Ch 22    1 row.    2 msec
```

pgAdmin showing the Users table

Authetication roadmap

Now that we have a test user in the database, let's outline how we are going to proceed with implementing the authentication system.

- We will obviously need a login page. This is where unauthenticated users are redirected when they try to access a restricted resource.
- The login page will collect user information (a username and password) and send it to a login endpoint.
- The login endpoint checks whether this information is valid or not. If it's valid, we create a session cookie associated with the authenticated user and add it to the response. We also store this cookie in a cache.
- As long as a cookie from a request matches the one from the cache, we consider the user authenticated and don't prevent them from accessing restricted resources.

I hope this doesn't sound terribly complicated, so let's get started.

The login page

The login page could be a simple form that sends user information to the login endpoint. We don't need any JavaScript there, but we need ConciseCSS styling to make our form look nice[56]. Let's start by creating a new Scala template called `login.scala.html` that consists of two fields and a button:

```
@()
<html lang="en">
    <head>
        <title>Login page</title>
        <link rel="stylesheet"
          href="@routes.Assets.versioned("compiled/styles.css")">
        <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <form method="post" action="/login">
            <fieldset>
                <legend>Login</legend>
                <p>
                    <label for="username">Username:</label>
                    <input id="username" type="text" name="username"  />
                </p>
                <p>
                    <label for="password">Password:</label>
                    <input id="password" type="password" name="password"  />
                </p>
                <button type="submit" class="button--sm">Login</button>
            </fieldset>
        </form>
    </body>
</html>
```

The controller action simply returns the login page, so it's trivial:

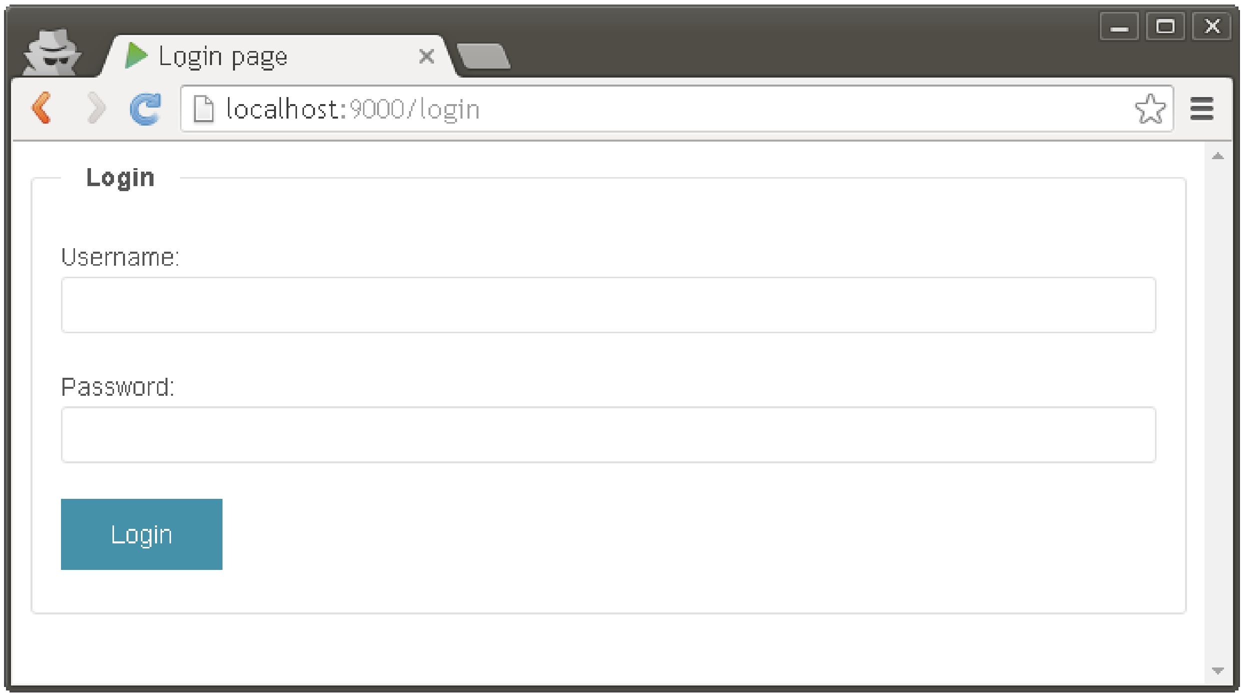
[56]http://concisecss.com/documentation/core/forms

```
class Application(sunService: SunService,
  //...
  def login = Action {
    Ok(views.html.login())
  }
}
```

Finally, we need to add the login page to the `routes` file:

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

After this, the login page will be available at `http://localhost:9000/login`:



Login page

Implementing the authentication service

Let's leave the login page for a moment and concentrate on main authentication logic. First, we will obviously need the `User` class that contains information about a user.

```scala
package model

import java.util.UUID

case class User(userId: UUID, userCode: String, password: String)
```

Since the `User` class reflects information from the database, we can also define a helper method that creates a new instance of the `User` class from a result set:

```scala
object User {
  def fromRS(rs: WrappedResultSet): User = {
    User(UUID.fromString(rs.string("user_id")),
      rs.string("user_code"), rs.string("password"))
  }
}
```

The `WrappedResultSet` class (defined in `scalikejdbc`) has a number of methods for retrieving values from result sets. There are methods that return a value of a particular type, for example `string` returns `String`, `short` returns `Short` and so on. There are also methods whose names end with `Opt`, for example `stringOpt`. These return `Option` values, i.e `None` when the database value is `NULL` and `Some` when it's not. Since we have only non-nullable columns in our table, we don't need to use `Opt`-methods.

The core of our authentication system is the `services.AuthService` class. Its public API will include the `login` method, to which we will pass user credentials as arguments. Depending on the validity of the passed user credentials, we will return either a new cookie (wrapped as `Some[Cookie]`) or `None`. This brings us to the `login` method with the following signature:

```scala
class AuthService {
  def login(userCode: String, password: String): Option[Cookie] = ???
}
```

The `login` method itself can be split into two parts - verifying user credentials and generating a new cookie. Following this logic, each part should be extracted as a separate helper method:

```scala
class AuthService {
  // ...
  private def checkUser(userCode: String, password: String): Option[User] = ???
  private def createCookie(user: User): Cookie = ???
}
```

Why does the `checkUser` method returns an `Option` while the `createCookie` a regular value? Well, users may provide wrong credentials, thus the check may fail. On the other hand, after the user identity is established, creating cookie cannot fail.

We also need to store generated cookies somewhere for future checks. We could use a simple `Map[String, User]` here, but in this case we would have to handle cookie expiration ourselves. An easier way is to use Play Cache API that will handle cookie expiration for us. To use cache in the `AuthService` we need to pass it as a constructor parameter:

```scala
class AuthService(cacheApi: CacheApi) {
  // ...
}
```

Internally Play uses EhCache[57] - a popular Java library - to implement `CacheApi`. The `cache` module is already in the `build.sbt` file, so we don't need to add anything here:

```scala
libraryDependencies ++= Seq(
  jdbc,
  cache,
  ws,
  // ...
)
```

We do need, however, to add the `EhCacheComponents` trait to our `AppComponents`:

```scala
trait AppComponents extends BuiltInComponents with AhcWSComponents
 with EvolutionsComponents with DBComponents with HikariCPComponents
 with EhCacheComponents {
  // ...
  lazy val authService = new AuthService(defaultCacheApi)
  // ...
}
```

The `EhCacheComponents` trait introduces two members of type `CacheApi` into the scope. As the `wire` macro chooses constructor arguments based on their type, it isn't working here and we're instantiating the `AuthService` manually.

Let's get back to generating cookies and think for a moment what some properties of good security tokens are. Since we're using the token for authentication, the token must be very hard (ideally impossible) to fabricate. Different users must have different tokens, so we should probably use some user identifier during cookie generation. Taking all of this into account, we may come up with an implementation similar to this:

[57] http://ehcache.org/

```scala
class AuthService(cacheApi: CacheApi) {
  val mda = MessageDigest.getInstance("SHA-512")
  val cookieHeader = "X-Auth-Token"

  // ...

  private def createCookie(user: User): Cookie = {
    val randomPart = UUID.randomUUID().toString.toUpperCase
    val userPart = user.userId.toString.toUpperCase
    val key = s"$randomPart|$userPart"
    val token = Base64.encodeBase64String(mda.digest(key.getBytes))
    val duration = Duration.create(10, TimeUnit.HOURS)
    cacheApi.set(token, user, duration)
    Cookie(cookieHeader, token, maxAge = Some(duration.toSeconds.toInt))
  }
}
```

The presence of the random part makes third-party cookie fabrication virtually impossible. We are also applying a hash function to make all tokens equal length. The generated cookie will be valid for 10 hours, so we are instructing both the Play Cache and the client browser to remove the expired cookie after this period is passed.

The `checkUser` helper method must perform two tasks. It must find a user record in the database and then check whether the provided password matches the one we have in the `Users` table.

For querying the database ScalikeJDBC exposes an object called `DB` that comes with a number of methods. We're going to use the `readOnly` method that has the following signature:

```scala
def readOnly[A](execution: DBSession => A)
  (implicit context: CPContext = NoCPContext): A
```

This is a curried function that takes two arguments - a functional block and connection pool context marked as `implicit`. The default connection pool context is brought into the scope via `import scalikejdbc._`, so we can safely ignore the second parameter. As for the `execution` part, this is where we are using the provided `DBSession` parameter for querying the database. The easiest way to do it is to use ScalikeJDBC's SQLInterpolation:

```scala
val maybeUser = sql"select * from users where user_code = $userCode".
    map(User.fromRS).single().apply()
```

There are several important things worth mentioning about SQLInterpolation:

- Even though it looks like a String, it translates into JDBC's type-safe `PreparedStatements`;

- The `map` method transforms data from database rows to domain objects, so it accepts function `WrappedResultSet => A` where `A` can be anything (e.g. `model.User`);
- The `single` method is used if you are interested in maximum one result. Since it's always possible to get back an empty result set, the return type of `single` is `Option`;
- The `apply` method does the actual work and therefore takes `DBSession` as an implicit parameter.

As for checking the password, we can still use `BCrypt.checkpw` that we discussed earlier. Let's apply this reasoning and implement the `checkUser` method:

```scala
class AuthService(cacheApi: CacheApi) {

  // ...

  private def checkUser(userCode: String, password: String): Option[User] =
      DB.readOnly { implicit session =>
    val maybeUser = sql"select * from users where user_code = $userCode".
      map(User.fromRS).single().apply()
    maybeUser.flatMap { user =>
      if (BCrypt.checkpw(password, user.password)) {
        Some(user)
      } else None
    }
  }
}
```

Here we're marking the `session` parameter as `implicit`, so it will be passed to the `apply` method automatically.

Finally, it's time to assemble the `login` method. As most work is already done in helper methods, implementing it will be a simple task:

```scala
class AuthService(cacheApi: CacheApi) {

  // ...

  def login(userCode: String, password: String): Option[Cookie] = {
    for {
      user <- checkUser(userCode, password)
      cookie <- Some(createCookie(user))
    } yield {
      cookie
    }
  }
}
```

The for comprehension here is used to combine two sequential tasks. If the user passes the check, we call the `createCookie` method and return the result. If the user fails, we simply return `None` and `createCookie` is never called.

Using Play Forms API

In 2016, HTML forms may not be as hot as they were ten years ago, but they still have their uses. And so does the Play *Forms API*. Let's start by importing two packages in the `Application.scala` file:

```
import play.api.data.Form
import play.api.data.Forms._
```

Then we will need a case class to store form data and a `Form` field to perform actual mapping:

```
case class UserLoginData(username: String, password: String)

class Application(sunService: SunService,
  weatherService: WeatherService,
  actorSystem: ActorSystem) extends Controller {

  // ...

  val userDataForm = Form {
    mapping(
      "username" -> nonEmptyText,
      "password" -> nonEmptyText
    )(UserLoginData.apply)(UserLoginData.unapply)
  }
}
```

Play provides many mappings including `text`, `number`, `date` (maps to `java.util.Date`), `jodaDate` (maps to `org.joda.time.DateTime`) and so on. It also ensures that types match and constraints are satisfied. The last piece of the puzzle is the `apply/unapply` part. This basically tells Play how to convert your class to a sequence of values and vice versa. Since we marked the `UserLoginData` class as `case class` both methods are implemented automatically and we only need to reference them.

Since business logic is already implemented in the `AuthService` class, the `doLogin` method itself is surprisingly simple:

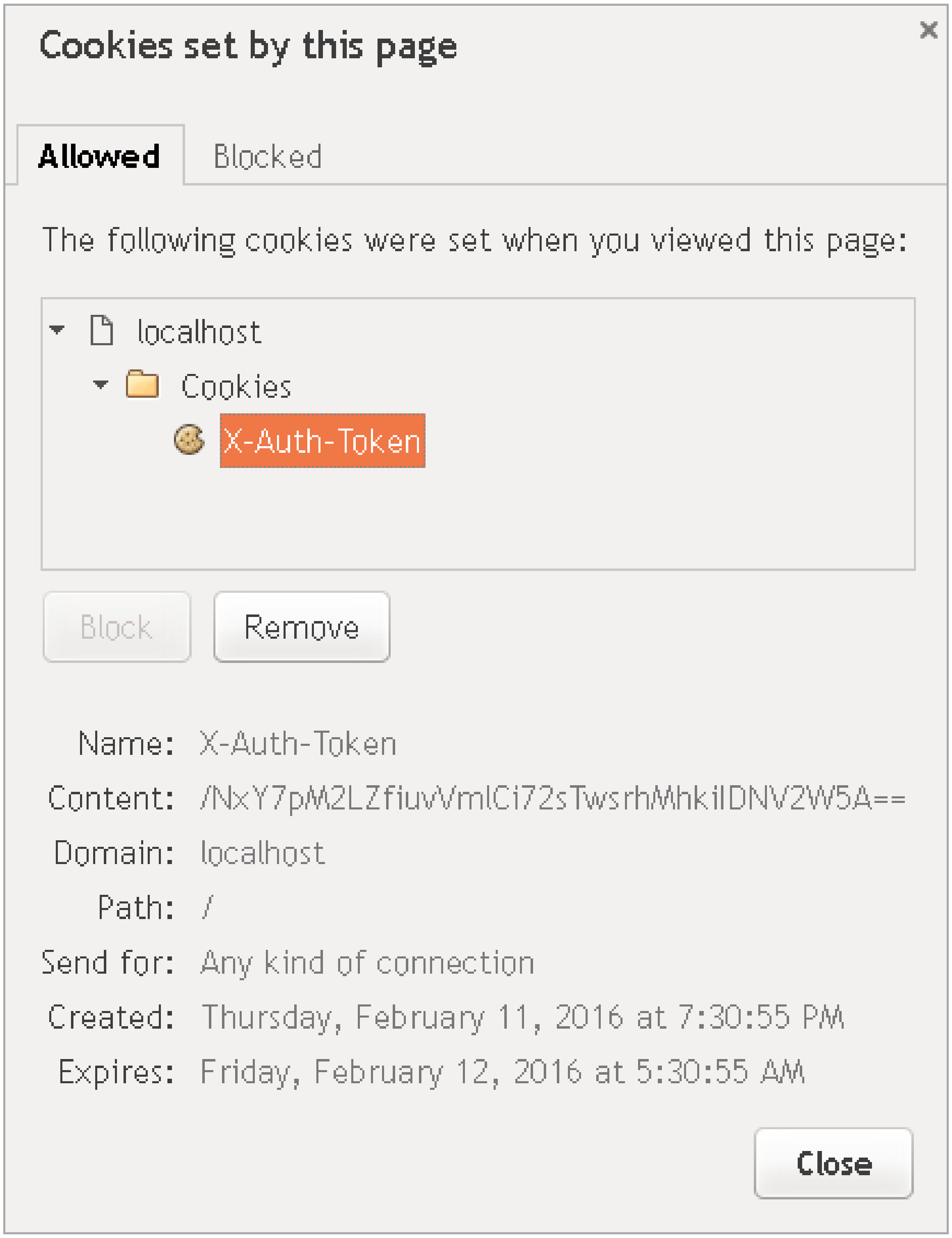
```scala
class Application(sunService: SunService,
  weatherService: WeatherService,
  actorSystem: ActorSystem,
  authService: AuthService) extends Controller {

  // ...

  def doLogin = Action(parse.anyContent) { implicit request =>
    userDataForm.bindFromRequest.fold(
      formWithErrors => BadRequest,
      userData => {
        val maybeCookie = authService.login(userData.username, userData.password)
        maybeCookie match {
          case Some(cookie) =>
            Redirect("/").withCookies(cookie)
          case None =>
            Ok(views.html.login())
        }
      }
    )
  }
}
```

First, we're adding the `AuthService` to the list of constructor parameters. Then, inside the `doLogin` method we're using the `bindFromRequest` method from Forms API. The `bindFromRequest` method, as its name suggests, takes the `request` as an argument (passed implicitly in our case) and tries to build the `UserLoginData` object. If the user did something wrong, for example, supplied the empty string as a password, this will fail and the `formWithErrors` branch will be used. If everything is OK, we can safely perform our authentication logic and possibly redirect the user to the the home page adding a new cookie to the response.

Finally, let's not forget to add the new endpoint to the `routes` file:

```
POST    /login                          controllers.Application.doLogin
```

If you try to log in using the valid credentials (stest/password123), you will be redirected to the home page. The important thing is that now the browser has a new cookie called `X-Auth-Token`, which is valid for 10 hours:



Cookies set by this page

The following cookies were set when you viewed this page:

Name:	X-Auth-Token
Content:	/NxY7pM2LZfiuvVmlCi72sTwsrhMhkilDNV2W5A==
Domain:	localhost
Path:	/
Send for:	Any kind of connection
Created:	Thursday, February 11, 2016 at 7:30:55 PM
Expires:	Friday, February 12, 2016 at 5:30:55 AM

The X-Auth- Token cookie in Chrome

Showing errors on the page

At the moment, if the user supplies wrong credentials, the login page simply refreshes. No error messages, no popup windows, nothing. If the user sends a form with the empty field, the result is even worse. In this case, our server responds with *400 Bad Request* error, which makes the browser page go completely blank. We could certainly do better than that.

First, let's add a new parameter called `maybeErrorMessage` to the `login` template. As its name suggests this parameter may or may not contain an error message:

```
@(maybeErrorMessage: Option[String])
<html lang="en">
    <!-- omitted -->
</html>
```

Then, below the `legend` tag, we need to add the following code:

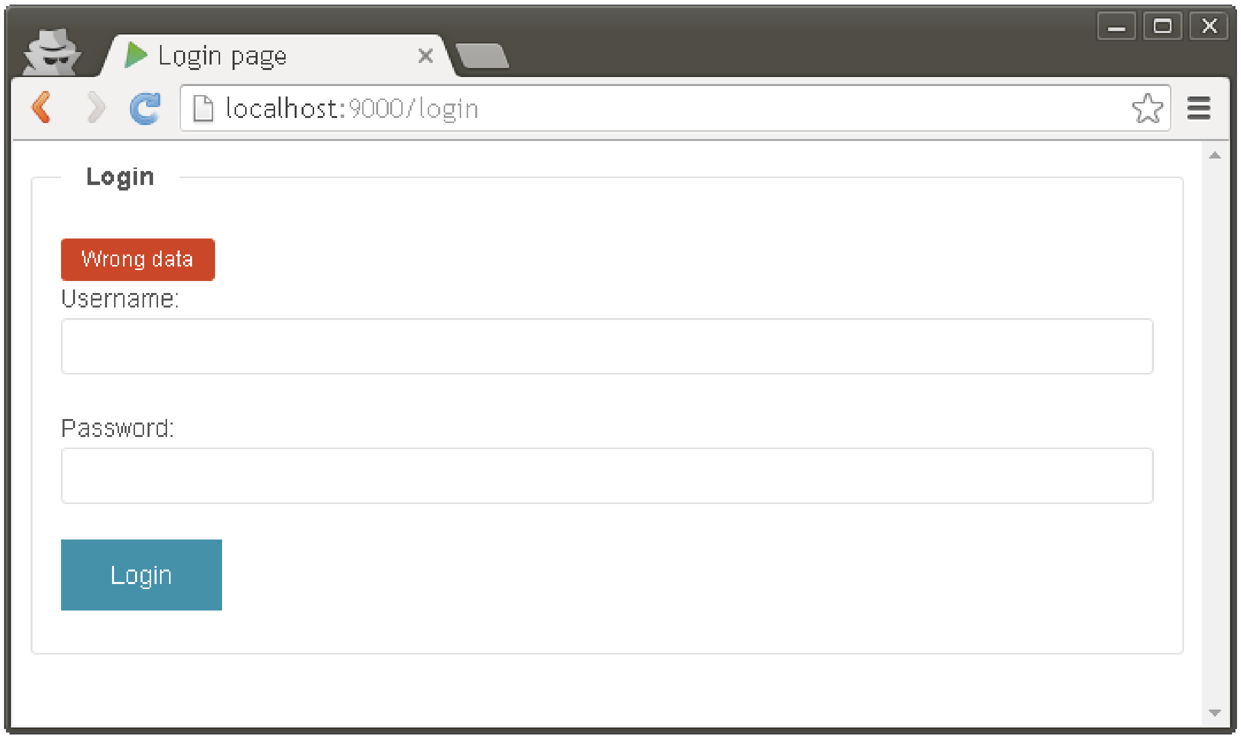
```
<!-- omitted -->
<legend>Login</legend>
@maybeErrorMessage.map { errorMessage =>
    <span class="label bg--error">@errorMessage</span>
}
<!-- omitted -->
<p>
```

Here we're using `@` to start a Scala expression. Since `maybeErrorMessage` is an `Option`, we can map it and the `span` fragment showing the error will appear only if the `Option` is not empty.

The next step is to fix the `Application` controller to reflect the fact that the `login` template now accepts a parameter:

```
class Application(sunService: SunService,
    weatherService: WeatherService,
    actorSystem: ActorSystem,
    authService: AuthService) extends Controller {

  // ...

  def login = Action {
    Ok(views.html.login(None))
  }

  def doLogin = Action(parse.anyContent) { implicit request =>
    userDataForm.bindFromRequest.fold(
      formWithErrors => Ok(views.html.login(Some("Wrong data"))),
      userData => {
          // ...
          case None =>
            Ok(views.html.login(Some("Login failed")))
        }
      }
    )
  }
}
```

When the users requests the login page for the first time (the `login` method), the form doesn't contain any errors, so we're passing `None` as an argument. In `doLogin`, if an error arises, we're sending back the login page with the error message. Try logging in without providing any credentials and you will see a page like this:



Wrong data error

Restricting access

The last part of our authentication mechanism is access restriction. In order to achieve our goal, we are going to use a specific Play feature called *action composition*. This feature allows developers to create their own `Actions` which have some additional functionality. In particular, our `UserAuthenticatedAction` will do the following:

- Check whether the user is authenticated by inspecting the `X-Auth-Token` cookie presented in `RequestHeader`;
- Redirect unauthenticated users to the login page;
- Grant authenticated users access to a requested restricted resource;
- If the user is authenticated, pass the `User` object into a controller action.

Since we already know that we will need to check cookies at some point, we can start by adding the `checkCookie` method to the `AuthService`:

```scala
class AuthService(cacheApi: CacheApi) {

  // ...

  def checkCookie(header: RequestHeader): Option[User] = {
    for {
      cookie <- header.cookies.get(cookieHeader)
      user <- cacheApi.get[User](cookie.value)
    } yield {
      user
    }
  }
}
```

Play exposes request cookies via the `cookies` field, which is a map-like structure. If the cookie is present in this collection, we're trying to find it in the cache and retrieve the associated `User` object along the way. Returning `None` will mean that the user is unauthenticated.

The `ActionBuilder` trait from the `play.api.mvc` package is parametrized with the request type. This enables developers to use their own request types instead of the standard `play.api.mvc.Request`. Let's create a file called `UserAuthAction` in package `services` and define a request wrapper called `UserAuthRequest`:

```scala
case class UserAuthRequest[A](user: User, request: Request[A])
  extends  WrappedRequest[A](request)
```

Our request wrapper adds a new field of type `User`, but in any other way it behaves just like the standard `Request`. Keeping this class parametrized will allow us to use it with different HTTP body content types (form, JSON etc).

Let's start implementing our `UserAuthAction` by extending the `ActionBuilder` trait (parametrized with the newly created `UserAuthRequest` type):

```scala
class UserAuthAction(authService: AuthService)
    extends ActionBuilder[UserAuthRequest] {

  def invokeBlock[A](request: Request[A],
             block: (UserAuthRequest[A]) => Future[Result]): Future[Result] = ???
}
```

We already know, that we will need the `AuthService` to check the cookie, so we're adding it as a constructor parameter.

The `invokeBlock` method is the only abstract method defined in `ActionBuilder`. Take a look at its signature. The first parameter represents the original request that is received by the server. The second parameter is a function from `UserAuthRequest` to `Future[Result]`. What is it? It turns out, this is the very block that we write when we define a new controller action. This already suggests how we may tackle the implementation of the `invokeBlock` method:

1. Pass the `RequestHeader` to the `checkCookie` method of the `AuthService`;
2. If the check fails, return with redirecting the user to the login page;
3. If the check succeeds, build a new instance of `UserAuthRequest` using the original `Request` and the `User` object obtained from `checkCookie`.
4. Invoke the `block` passing the instance of `UserAuthRequest` as an argument
5. Whatever the `block` invocation returns becomes the result of the `invokeBlock` method.

Translating this logic into code results in the following implementation:

```
class UserAuthAction(authService: AuthService)
    extends ActionBuilder[UserAuthRequest] {

  def invokeBlock[A](request: Request[A],
           block: (UserAuthRequest[A]) => Future[Result]): Future[Result] = {
    val maybeUser = authService.checkCookie(request)
    maybeUser match {
      case None => Future.successful(Results.Redirect("/login"))
      case Some(user) => block(UserAuthRequest(user, request))
    }
  }
}
```

Note that we're passing `Request` to our `checkCookie` method, because in Play API `RequestHeader` is a supertype of `Request`.

In order to battle-test the `UserAuthAction` we will need a new template that takes a `User` object as a parameter, so let's create a new file called `restricted.scala.html` with the following content:

```
@(user: model.User)
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Restricted</title>
        <link rel="stylesheet"
          href="@routes.Assets.versioned("compiled/styles.css")">
        <link rel="shortcut icon" type="image/png"
          href="@routes.Assets.versioned("images/favicon.png")">
    </head>
    <body>
        <h1>Hello @user.userCode</h1>
        <div>Your id is @user.userId</div>
    </body>
</html>
```

We should add `UserAuthAction` as a constructor parameter to the `Application` controller and after that we will be able to use it just like a regular `Action`:

```
class Application(sunService: SunService,
    weatherService: WeatherService,
    actorSystem: ActorSystem,
    authService: AuthService,
    userAuthAction: UserAuthAction) extends Controller {

  def restricted = userAuthAction { userAuthRequest =>
    Ok(views.html.restricted(userAuthRequest.user))
  }

  // ...
}
```

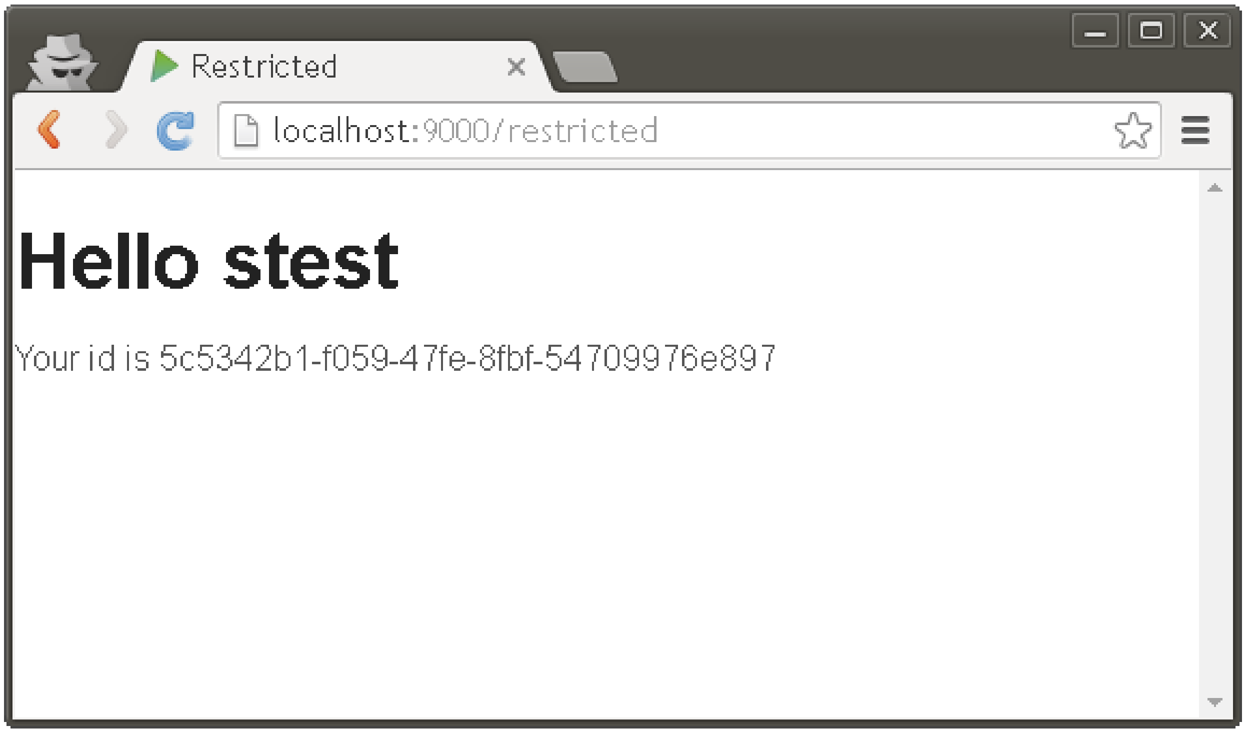
The important difference is that instead of the standard `Request` we have a `UserAuthRequest`, which comes with the already initialized `user` field. In order to pass the `UserAuthAction` instance to the `Application` controller, we need to initialize it in the `AppComponents` trait:

```
trait AppComponents extends BuiltInComponents with AhcWSComponents
 with EvolutionsComponents with DBComponents with HikariCPComponents
 with EhCacheComponents {

  // ...

  lazy val userAuthAction = wire[UserAuthAction]
}
```

Again, the `wire` macro takes care of passing an `AuthService` to the `UserAuthAction` constructor. The only thing that is left is to add the `restricted` route:

```
GET     /restricted                 controllers.Application.restricted
```

If you try accessing the page at `localhost:9000/restricted` from a new *incognito* browser window, you will be redirected to the login page. However, if you go to the restricted page after entering the correct credentials (i.e. stest/password123), you will be able to see the user information:



Hello stest

Your id is 5c5342b1-f059-47fe-8fbf-54709976e897

Restricted page with user information

Deployment

When you start the app using Activator, it works in the so-called *development mode*. However, when it is ready for deployment, it should be build for work on the production server.

Hosting considerations

Many hosting providers offer virtual private servers (VPS) for as little as $5 a month. This usually includes 512Mb of RAM, which is hardly enough for modern JVM-based applications, so I would recommend starting with 1Gb servers. As for virtualization technologies used by the provider, both KVM or Xen will work just fine. You can also try cloud-based solutions, but the price will be significantly higher.

Many providers offer "one-click installations" of Ubuntu Server, which is a great way to get your server up and running within minutes.

The only thing you need in addition to Ubuntu is Oracle JDK. The challenging part here is downloading the Java distribution without a browser. For this purpose you may use `wget` with the following options:

```
$ wget --no-check-certificate --no-cookies --header "Cookie: oraclelicense=accep\
t-securebackup-cookie" <url>
```

You can also refer to this StackOverflow question[58].

Preparing the distribution

Before using your app in production, you need to change the `play.crypto.secret` property from the `application.conf` file. Fortunately, it's very easy to do. Simply start `activator` and type:

```
[scala-web-project] $ playGenerateSecret
[info] Generated new secret: DkDuDgX;3zdZ1VY0<vNukRb5>II@KLHBtsO/u]c3?9w0GdVnMdJ\
1kXEC0M7B1R=t
[success] Total time: 0 s, completed Feb 13, 2016 3:21:54 PM
```

Then, the generated sequence needs to be copied to the configuration file.

Before building the production distribution, we also need to minify our frontend assets. Since we're using Webpack, we need to add a new script to the `package.json` file:

[58]http://stackoverflow.com/questions/10268583/

```
{
  // ...
  "scripts": {
    "watch": "webpack --watch",
    "prod": "webpack -p"
  },
  // ...
}
```

After that, running `npm run prod` will result in building minified files inside the same `public/compiled` directory:

```
$ npm run prod

Hash: 476d1972a6a6465fddfe
Version: webpack 1.12.13
Time: 8217ms
     Asset     Size  Chunks             Chunk Names
 bundle.js   196 kB       0  [emitted]  main
styles.css  30.6 kB       0  [emitted]  main
    + 166 hidden modules
```

Note that the minified `bundle.js` occupies less than 200 kB.

The final step is to build the app distribution:

```
$ activator clean compile stage
[success] Total time: 5 s, completed Feb 13, 2016 3:34:47 PM
```

Activator will place everything you need to run the app on production servers to the `target/universal/stage` directory:

```
$ cd target/universal/stage/
$ tree -L 1
.
├── bin
├── conf
├── lib
├── README
└── share
```

The `lib` directory contains a lot of JAR files including your application logic, the Scala standard library and even a Web server. The `application.conf` file along with other configuration files and evolution scripts goes to the `conf` directory. The `share` directory contains automatically generated documentation and the `bin` directory contains run scripts for Unix and Windows:

```
$ ./bin/scala-web-project
[info] - application - The app is about to start
[info] - application - Creating Pool for datasource 'default'
[info] - play.api.db.DefaultDBApi - Database [default] connected at jdbc:postgre\
sql://localhost:5432/scaladb
[info] - play.api.Play - Application started (Prod)
[info] - play.core.server.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:9\
000
```

Note that it's possible to override most of the options defined in `application.conf` using `-D`. For example, if you want to start your app on port 8080 instead of default 9000, you can use:

```
$ ./scala-web-project -Dhttp.port=8080
[info] - play.core.server.NettyServer - Listening for HTTP on /0:0:0:0:0:0:0:0:8\
080
```

Closing remarks

Now you know enough about Scala and its ecosystem to start building your own Web applications. The journey, however, doesn't end here. In fact, it's only the beginning of applied Scala.

If you are interested in getting deeper into Scala and want to learn more about its syntax and the standard library, I recommend that you obtain a copy of "Programming in Scala" by Martin Odersky, Lex Spoon and Bill Venners. It's about 800 pages, but it's read like a detective and, besides, this is probably the most comprehensive and authoritative work on the language today.

I hope that you've enjoyed reading this book and that it provided you with a lot of useful information. If you have any questions or feedback, you can reach me via email: `denis` AT `appliedscala.com`.

I wish you the best of luck in your future Scala endeavors.

And thank you for reading my book.

Denis Kalinin,

2016

Appendix A. Packages

Sometimes, it could be difficult to determine which package needs to be imported in order for a particular class to start working. IntelliJ certainly helps with this task, but it's still better to know for sure what to import and when. Here is the list of all third-party types and objects that we used throughout the book with their fully qualified names in alphabetic order:

type/object	fully qualified name
`?/ask`	`akka.pattern.ask`
`Action`	`play.api.mvc.Action`
`ActionBuilder`	`play.api.mvc.ActionBuilder`
`Actor`	`akka.actor.Actor`
`ActorSystem`	`akka.actor.ActorSystem`
`ApplicationLoader`	`play.api.ApplicationLoader`
`Base64`	`org.apache.commons.codec.binary.Base64`
`BCrypt`	`org.mindrot.jbcrypt.BCrypt`
`BuiltInComponents`	`play.api.BuiltInComponents`
`BuiltInComponentsFromContext`	`play.api.BuiltInComponentsFromContext`
`CacheApi`	`play.api.cache.CacheApi`
`Context`	`play.api.ApplicationLoader.Context`
`Controller`	`play.api.mvc.Controller`
`Cookie`	`play.api.mvc.RequestHeader`
`Date`	`java.util.Date`
`DateTime`	`org.joda.time.DateTime`
`DateTimeFormat`	`org.joda.time.format.DateTimeFormat`
`DateTimeZone`	`org.joda.time.DateTimeZone`
`DBComponents`	`play.api.db.DBComponents`
`DB`	`scalikejdbc.DB`
`DBs`	`scalikejdbc.config.DBs`
`Duration`	`scala.concurrent.duration.Duration`
`DynamicEvolutions`	`play.api.db.evolutions.DynamicEvolutions`
`EhCacheComponents`	`play.api.cache.EhCacheComponents`
`EssentialFilter`	`play.api.mvc.EssentialFilter`
`EvolutionsComponents`	`play.api.db.evolutions.EvolutionsComponents`
`Filter`	`play.api.mvc.Filter`
`Form`	`play.api.data.Form`
`Forms.mapping`	`play.api.data.Forms.mapping`
`Forms.nonEmptyText`	`play.api.data.Forms.nonEmptyText`
`Future`	`scala.concurrent.Future`
`HikariCPComponents`	`play.api.db.HikariCPComponents`
`Json`	`play.api.libs.json.Json`

type/object	**fully qualified name**
`Logger`	`play.api.Logger`
`LoggerConfigurator`	`play.api.LoggerConfigurator`
`Materializer`	`play.api.mvc.Materializer`
`MessageDigest`	`java.security.MessageDigest`
`AhcWSComponents`	`play.api.libs.ws.ahc.AhcWSComponents`
`Props`	`akka.actor.Props`
`Receive`	`akka.actor.Actor.Receive`
`Request`	`play.api.mvc.Request`
`RequestHeader`	`play.api.mvc.RequestHeader`
`Result`	`play.api.mvc.Result`
`Results`	`play.api.mvc.Results`
`Router`	`play.api.routing.Router`
`Routes`	`router.Routes`
`SimpleDateFormat`	`java.text.SimpleDateFormat`
`Timeout`	`akka.util.Timeout`
`TimeUnit`	`java.util.concurrent.TimeUnit`
`UUID`	`java.util.UUID`
`WrappedRequest`	`play.api.mvc.WrappedRequest`
`WrappedResultSet`	`scalikejdbc.WrappedResultSet`
`WS`	`play.api.libs.ws.WS`
`WSClient`	`play.api.libs.ws.WSClient`

When working with ScalikeJDBC, simply import the entire package via `import scalikejdbc._` to introduce all necessary types and implicits into the scope.

Appendix B. Using IntelliJ IDEA

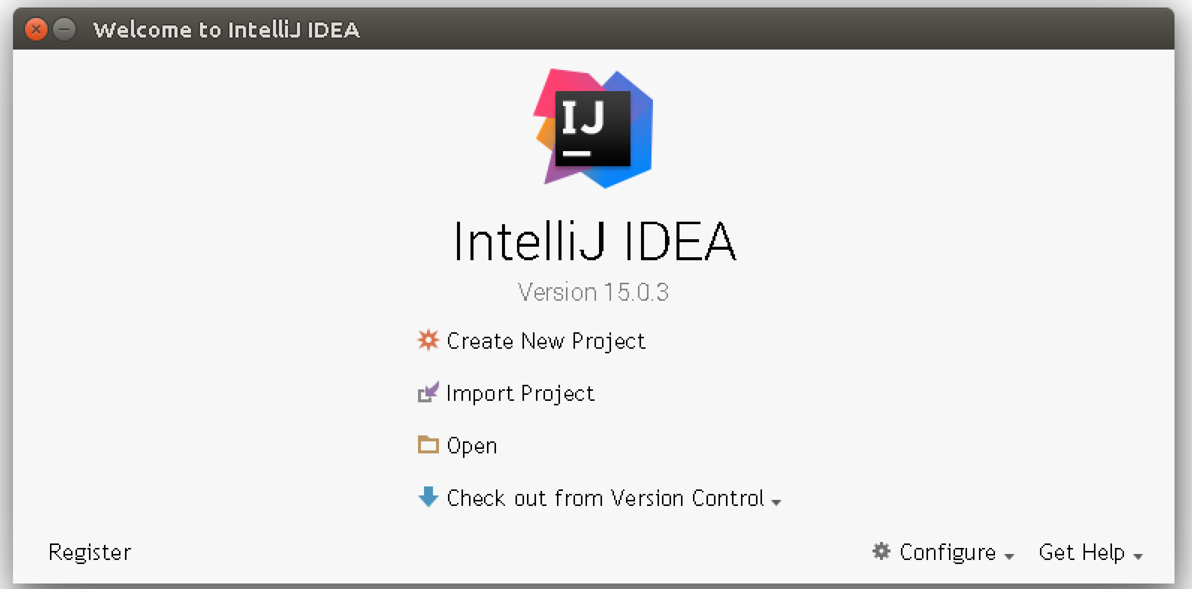
Before starting the IDE, you need to clone the book's GitHub repository and checkout the "starting-point-play-2.5.0" tag while initializing your own branch along the way. In order to do this you can use the following sequence of `git` commands:

```
$ git clone https://github.com/denisftw/modern-web-scala .
Cloning into '.'...
$ git checkout tags/starting-point-play-2.5.0 -b my-branch
Switched to a new branch 'my-branch'
```

After IntelliJ IDEA is unpacked in a directory of your choice, you can start it by invoking the `idea.sh` script from the `bin` directory:

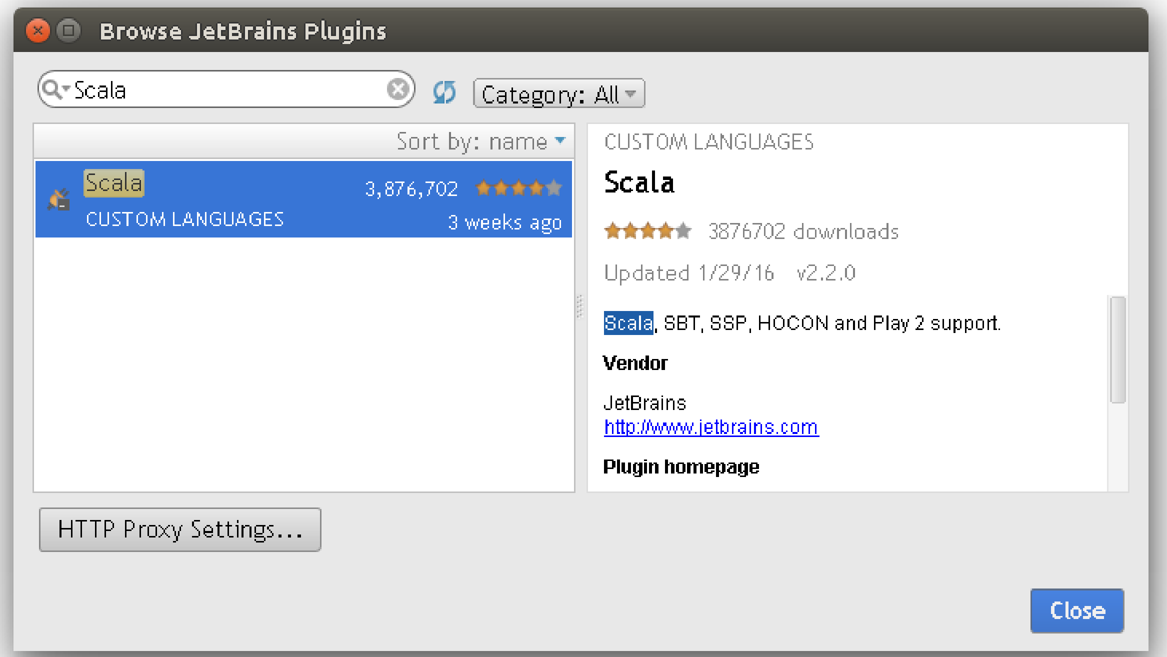
```
$ cd ~/DevTools/idea15u/bin
$ ./idea.sh
```

During its first run, IDEA may ask questions about installing launcher shortcuts and importing settings from the previously installed versions. After that IntelliJ will greet you with the Welcome window.



IntelliJ welcome window

Since we are going to use Scala, let's install the official Scala plugin, which is not bundled with IDEA. Simply choose "Configure -> Plugins -> Install JetBrains plugin..." and type "Scala" in the opened window. The Scala plugin will be the only unfiltered plugin, so you will only need to press the Install button to actually install it:



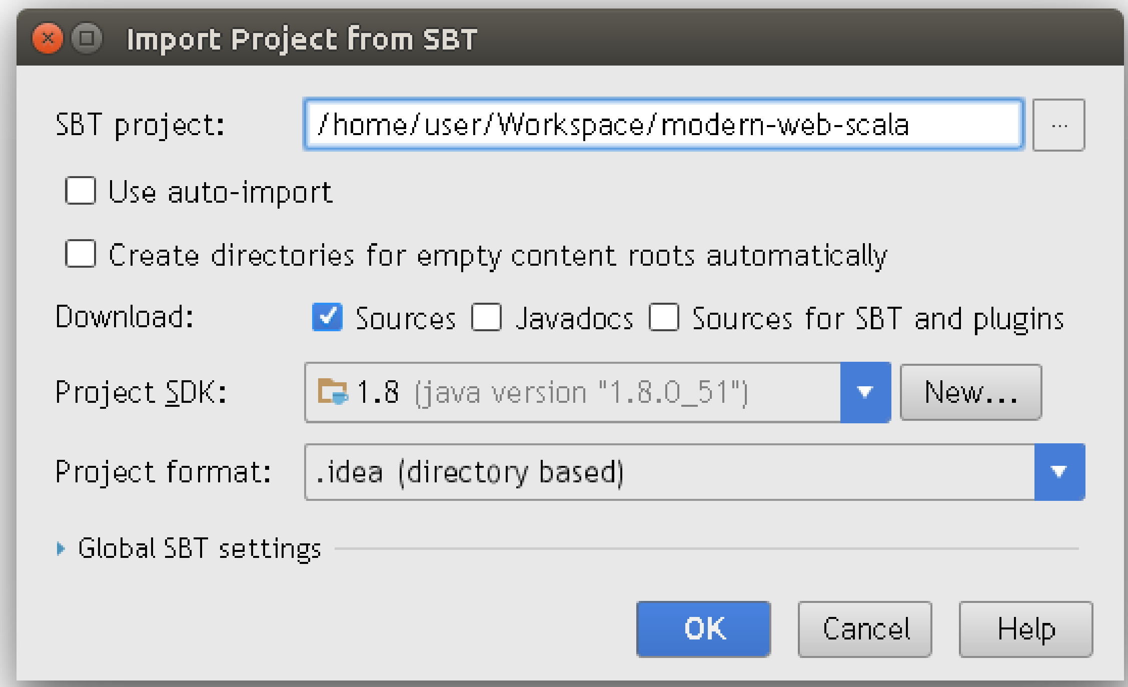
Scala 3,876,702
CUSTOM LANGUAGES 3 weeks ago

CUSTOM LANGUAGES
Scala
3876702 downloads
Updated 1/29/16 v2.2.0
Scala, SBT, SSP, HOCON and Play 2 support.
Vendor
JetBrains
http://www.jetbrains.com
Plugin homepage

Installing Scala plugin

The Scala plugin comes with SBT and Play support, so you don't need to install anything else. After restart IntelliJ will be ready to import Scala projects.

In order to import the project, click the Open button on the Welcome screen and then choose the directory with our Play project. The SBT import window will appear on the screen:


Import Project from SBT

SBT project: /home/user/Workspace/modern-web-scala

☐ Use auto-import

☐ Create directories for empty content roots automatically

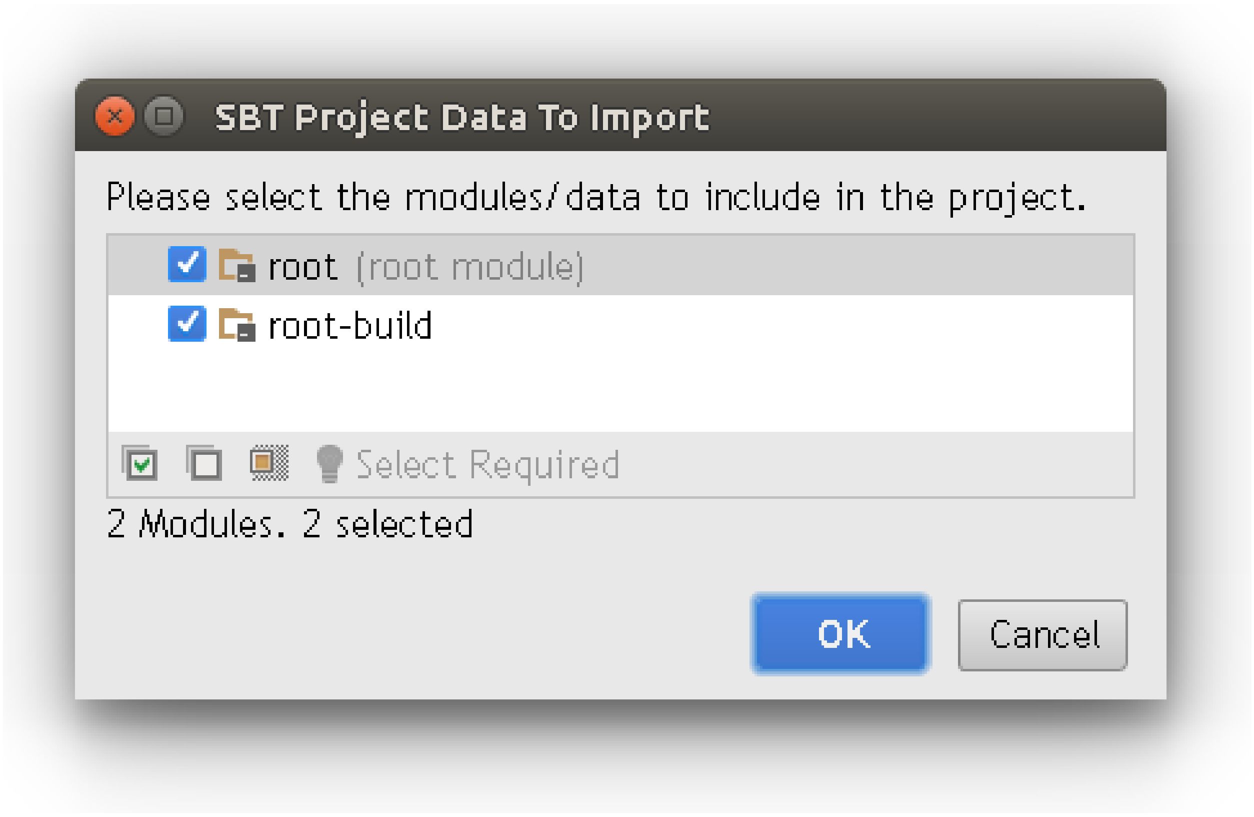
Download: ✓ Sources ☐ Javadocs ☐ Sources for SBT and plugins

Project SDK: 1.8 (java version "1.8.0_51")

Project format: .idea (directory based)

The SBT import window

It is important to specify `1.8` as Project SDK, but for other settings default values should work just fine. Finally, IDEA will ask which modules to import:

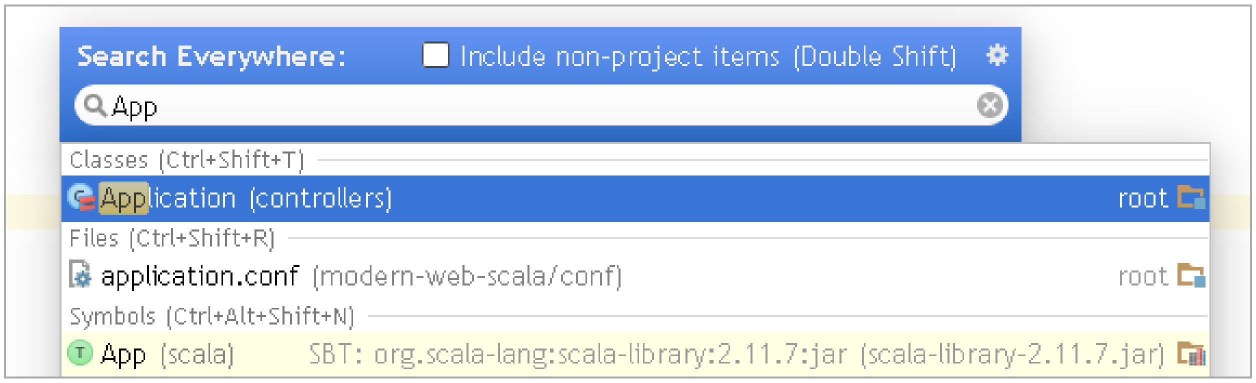

SBT Project Data To Import
Please select the modules/data to include in the project.
root (root module)
root-build
Select Required
2 Modules. 2 selected
OK
Cancel

Selecting which modules to import

By default, both `root` and `root-build` are selected and this is fine.

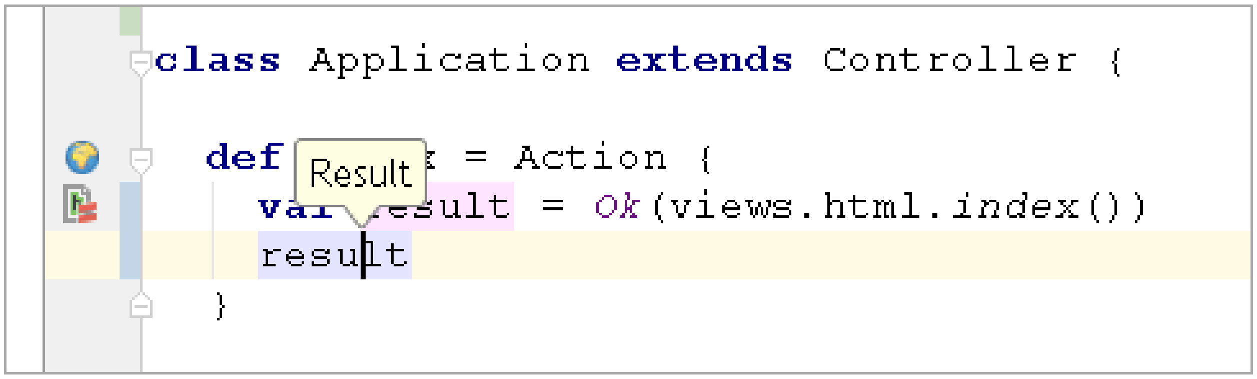
After IntelliJ opens and indexes all files and libraries, it will be ready to use. While it's an extremely powerful tool and covering all of its features in one small section is impossible, here is a couple of neat tricks that will make development more productive and possibly enjoyable.

To invoke the "Search Everywhere" window, simply press the <Shift> key twice. This will allow you to quickly open any file, type or method:


Search Everywhere:
Include non-project items (Double Shift)
App
Classes (Ctrl+Shift+T)
Application (controllers) root
Files (Ctrl+Shift+R)
application.conf (modern-web-scala/conf) root
Symbols (Ctrl+Alt+Shift+N)
App (scala) SBT: org.scala-lang:scala-library:2.11.7:jar (scala-library-2.11.7.jar)

Search Everywhere window

To see the type of any variable or value, simply move the caret to the part of code you are interested in and press <Alt> + <Equals>:



```
class Application extends Controller {

  def index = Action {
    val result = Ok(views.html.index())
    result
  }
```

Type inspection

This is especially useful when you rely on the type inferrer but want to check that you're getting the type you want.