Denis Kalinin

Modern Web Development with Kotlin

Modern Web Development with Kotlin

A concise and practical step-by-step guide

Denis Kalinin

This book is for sale at http://leanpub.com/modern-web-development-with-kotlin

This version was published on 2019-06-04

Contents

Preface

On 18 March 2014, Java 8 was finally released, and this event marked an important point in the history of programming languages. Java had finally got function literals! There were many other improvements, and all of this basically put to rest myths about Java being outdated. It was obvious to everyone that this release was going to be a tremendous success, and the language landscape was changing for good.

Interestingly, while presenting an upcoming version, Brian Goetz, the Java language architect, noted that even with all these shiny new features, the language still felt like Java. It was a relief for many developers who didn't want to feel at a loss with once a very familiar language. At the same time, it meant that for numerous Web developers, Java was going to stay something relatively heavyweight or even alien. As a result, even now, former Ruby or PHP programmers are much more likely to switch to Scala, another statically-typed JVM language, than Java 8.

So, what about Kotlin? It doesn't seem to be eager to introduce its own build tools or frameworks. Instead, it aims at reusing existing Java libraries and integrating into already established workflows. In particular, throughout this book, we will be using Gradle - a popular build tool initially developed for Java. Generally speaking, Kotlin is quite flexible when it comes to build tools, so you can use Maven or even Ant if you need to.

As a new language developed literally from scratch, Kotlin is also free from legacy problems. As a result, there are quite a few things that it does differently than Java even in supposedly similar situations.

For example, the `if` statement in Java allows to make the control flow depend on a certain condition:

```
1  List<Integer> ints = null;
2  if (checkCondition()) {
3    ints = generateList();
4  } else {
5    ints = Collections.emptyList();
6  }
```

Since `if` statements don't return any values, we have to define a variable and put the assignments in both branches. In Kotlin, `if` is an expression, so `ints` can be defined as a constant and initialized immediately:

```
val ints = if (checkCondition()) {
  generateList()
} else {
  emptyList()
}
```

As an additional bonus, another Kotlin feature called *type inference* liberates us from declaring `ints` as `List<Integer>`.

Similarly, `try/catch` blocks in Java don't return any values, so you have to declare necessary variables before these blocks. Add the requirement to always catch or throw checked exceptions, and you'll get a recipe for crafting complex and difficult to maintain code. As you will see later in this book, with Kotlin you don't actually need to use `try/catch` blocks at all! Instead, it's usually better to wrap dangerous logic and return an `Either` object:

```
val resultT = eitherTry {
  // invoke some dangerous code
}
```

This is only a fraction of features which enable Kotlin developers to write concise and easily readable code. Of course, we will be discussing these features in detail throughout the book.

Since Kotlin tries to fit into the existing Java ecosystem, I decided to take Vert.x - a primarily Java framework - and use it for developing our sample Web application. Basing our narrative on Vert.x[1] allows us to experience first-hand what it is like to use existing libraries in Kotlin.

Vert.x, however, is quite an unusual beast and differs from traditional frameworks in multiple ways.

First, it's completely asynchronous and in this respect similar to NodeJS. Instead of blocking the computation until the result becomes available, Vert.x uses callbacks:

```
router.get("/").handler { routingContext ->
  routingContext.response().end("Hello World")
}
```

As a modern programming language, Kotlin comes with a first-class support for lambdas (function literals) and therefore, allows to work with Vert.x in a pleasant, almost DSL-like way.

Second, Vert.x is not based on Servlets, which makes it a good choice for people without Java background, especially Web developers. The fact that Vert.x comes with an embedded Web server enables simple deployment and easy integration with frontend tools.

Writing a Web application usually involves using many additional tools. While Vert.x provides a lot out of the box, it doesn't provide everything we need. As a result, throughout the book, we will be

[1] http://vertx.io/

relying on popular third-party libraries. For instance, we will be using funKTionale for functional programming, Kovenant for writing asynchronous code, Flyway for migrations and so on.

Since Kotlin is primarily a backend language, most readers are probably more familiar with the server side of things. To make the book even more useful for them, I decided to include an entire chapter devoted to frontend integration. In this chapter, you will see how to build an effective workflow that includes Webpack, EcmaScript 6 and Sass.

I hope that this little introduction gave you several more reasons to learn Kotlin, so turn the page and let's get started!

Preparing the environment

In this section, we will set up everything we need to start writing Kotlin applications. Since Kotlin heavily relies on the Java ecosystem, you need to have at least a brief familiarity with Java tools. If you are already a JVM expert, you can skip this section, but I would still recommend flicking through it, because it provides a good overview and you may end up learning a couple of tricks.

Choosing the operating system

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/[2] 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, 16.04 definitely works in latest versions of VirtualBox without problems, earlier versions 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[3] about 20% of developers consistently choose Linux as their primary desktop operating system.

Installing Java Development Kit

Kotlin compiles into Java bytecode, so you will need to install the Java Development Kit (JDK) to run Kotlin programs. Fortunately, there are many ways to obtain JDK, and all of them are relatively straightforward.

If you're running a Linux distribution, then there's a good chance that it has a prebuilt OpenJDK package in its repository. For example, on Ubuntu you can get the latest update for OpenJDK 8 using the following command:

[2] https://www.virtualbox.org/wiki/Downloads
[3] http://stackoverflow.com/research/developer-survey-2016#technology-desktop-operating-system

```
$ sudo apt-get install openjdk-8-jdk
```

You can also download a prebuilt OpenJDK package from AdoptOpenJDK and then install it manually:

- Go to https://adoptopenjdk.net/releases.html[4] and download the latest JDK 8 (look for "OpenJDK 8 with HotSpot" and make sure that the full name looks something like "jdk8u172-b11", which means "version 8, update 172"). For Linux you'll probably end up getting a `tar.gz` file. Extract the contents of the archive anywhere (for example, to `~/DevTools/java8`);
- On Windows, just follow the instructions of the installer;
- Create an environment variable `JAVA_HOME` and point it to the JDK directory;
- Add the `bin` directory of JDK to your `PATH` so that the `java` command works from any directory on your machine;

On Linux, you can accomplish both goals if you add to your `~/.bashrc` the following:

```
JAVA_HOME=/home/user/DevTools/jdk8u172-b11
PATH=$JAVA_HOME/bin:$PATH
```

After that, you can start a new terminal window and check that Java is indeed installed:

```
$ java -version
openjdk version "1.8.0-adoptopenjdk"
OpenJDK Runtime Environment (build 1.8.0-adoptopenjdk-2018_05_19_01_00-b00)
OpenJDK 64-Bit Server VM (build 25.71-b00, mixed mode)
```

Note that Kotlin applications can run on Java 6, but Kotlin itself needs JDK 8 during development. Also many libraries that we will be using throughout book require Java 8, so we will stick to this version.

Installing Kotlin

Surprisingly, you don't need the Kotlin compiler in order to use Kotlin, but you may want to install it for educational purposes.

Depending on the operating system you're running, there are several ways to obtain Kotlin binaries. The most universal one is to use a release package from JetBrain's GitHub. Just follow these steps:

- Get the binaries from https://github.com/JetBrains/kotlin/releases/latest[5]
- Extract the contents of the archive anywhere (for example, to `~/DevTools/kotlinc`)
- Add the `bin` directory of the Kotlin distribution to your path:

[4] https://adoptopenjdk.net/releases.html
[5] https://github.com/JetBrains/kotlin/releases/latest

```
JAVA_HOME=/home/user/DevTools/jdk8u172-b11
KOTLIN_HOME=/home/user/DevTools/kotlinc
PATH=$JAVA_HOME/bin:$KOTLIN_HOME/bin:$PATH
```

Alternatively, you can install Kotlin using SDKMan[6]. First, install SDKMan itself, and then install Kotlin:

```
$ curl -s get.sdkman.io | bash
$ sdk install kotlin
```

If you're using Ubuntu 16.04 or later, you can get Kotlin binaries via Snap:

```
$ sudo snap install --classic kotlin
```

Snaps are updated automatically, so installing Kotlin as a snap is a good way to ensure that your compiler version is always up to date.

Installing IntelliJ IDEA

IntelliJ IDEA[7] is an IDE made by the same people who created Kotlin, and it is probably the most feature-rich solution for writing serious Kotlin applications. And the good news is that the Community edition, which is free and open-source, will work just fine. All latest versions come with the Kotlin plugin pre-installed, so you don't need to worry about it.

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; support\
 was removed in 8.0
```

You can safely ignore them for now. By the way, you can use the `idea.vmoptions` file that resides 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[8].

Sometimes IntelliJ goes crazy and starts highlighting valid parts of your code as errors. A common solution to this is to click "File -> Invalidate Caches / Restart…". IDEA will have to reindex everything after restart, but sometimes it helps in many cases.

[6] http://sdkman.io/
[7] https://www.jetbrains.com/idea/
[8] http://stackoverflow.com/questions/14763079/

Installing Atom Editor

Even though I strongly believe that using a full-blown IDE when working with Kotlin 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[9], 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 the 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 following along with the book narrative you may want to install the following:

- `language-kotlin` - adds syntax highlighting for Kotlin
- `linter-kotlin` - compiles your code behind the scenes using `kotlinc` to show compiler errors right inside the editor
- `terminal-plus` - allows to use the terminal session without leaving the editor

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++[10]. Moreover, it doesn't help you much when it comes to adding package imports or debugging code, and in this respect it is 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.

[9] http://atom.io
[10] 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 that when you're only starting with something new, learning the basics without using sophisticated tools provides a better understanding of technology and therefore gives you more confidence later. This is the main reason why we in the beginning of the book my examples are based on Atom.

Build tools

Strictly speaking, you don't need to download the Kotlin compiler to build Kotlin code. However, manually compiling source files could be a good exercise to get a better understanding of how things work, and what more sophisticated tools do behind the scenes. In this section, we're going to look at different approaches to building Kotlin projects, starting with the ubiquitous command line.

Command line

The command line tools provide basic functionality for building Kotlin applications. In order to check that they were installed correctly, simply invoke the following command:

```
$ kotlinc -version
info: kotlinc-jvm 1.3.31 (JRE 1.8.0-adoptopenjdk-jenkins_2018_05_19_01_00-b00)
```

kotlinc

`kotlinc` is the Kotlin Compiler, i.e. a tool that is used to compile source code into bytecode. In order to test the compiler, let's create a simple file called `Hello.kt` with the following contents:

```
fun main(args: Array<String>) {
  println("Hello World!")
}
```

The `main` method, just as in many other languages, is the starting point of the application. Unlike Java or Scala, however, in Kotlin you don't put this method into a class or object, but instead make it standalone. Once we have our file, we can compile it:

```
$ kotlinc Hello.kt
```

There will be no output, but `kotlinc` will create a new file in the current directory:

```
$ ls
Hello.kt  HelloKt.class
```

Note that a new file is named `HelloKt.class`, not `Hello.class`.

kotlin

`kotlin` is the Kotlin interpreter. If it is invoked without arguments, it will complain that it needs at least one file to run. If we want to execute the `main` method of the recently compiled example, we need to type the following:

```
$ kotlin HelloKt
Hello World!
```

Now that we understand how it works on the low level, let's look at something more interesting.

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. Not surprisingly, Gradle can also handle Kotlin projects via the "Gradle Script Kotlin" plugin.

You can download a binary distribution from from the official website[11]. The latest version at the time of this writing is 5.4 and it works perfectly well with Kotlin.

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=$JAVA_HOME/bin:$KOTLIN_HOME/bin:$GRADLE_HOME/bin:$PATH
```

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

```
$ mkdir gradle-kotlin
$ cd gradle-kotlin
$ gradle init --type java-library

BUILD SUCCESSFUL in 1s
2 actionable tasks: 2 executed
```

The `init` task initializes a skeleton project by creating several files and directories including `build.gradle`. Since we're not interested in learning Java, remove both `src/main/java` and `src/test` directories and copy our `Hello.kt` to `src/main/kotlin`:

[11] https://gradle.org/gradle-download/

```
$ rm -rf src/main/java
$ rm -rf src/test
$ mkdir -p src/main/kotlin
$ cp ../Hello.kt src/main/kotlin/
```

Finally, remove everything from the `gradle.build` file and add the following:

```
buildscript {
  ext.kotlin_version = '1.3.31'

  repositories {
    jcenter()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
  }
}

apply plugin: 'kotlin'

repositories {
  jcenter()
}

dependencies {
  compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

The `buildscript` part adds support for Kotlin projects via the `kotlin-gradle-plugin`. Here we're also specifying the language version for later reference. Note that we're also adding the Kotlin standard library to the list of compile dependencies.

Many projects use `mavenCental()` as the main repository for dependencies. Here, however, we're using `jcenter()` as a faster and more secure option.

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-kotlin.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 Kotlin interpreter allows you to modify the default classpath, which adds one more way to start the app:

```
$ kotlin -cp build/libs/gradle-kotlin.jar HelloKt
Hello World!
```

Again, the class files that were created during the compilation phase are regular Java bytecode files. Since they are no different than files produced by `javac` (the Java compiler), we should be able to run our Kotlin app using `java` (the Java interpreter). Let's try it:

```
$ java -cp build/libs/gradle-kotlin.jar HelloKt
```

Java will respond with the following error:

```
Exception in thread "main" java.lang.NoClassDefFoundError:
  kotlin/jvm/internal/Intrinsics
          at HelloKt.main(Hello.kt)
```

What happened? Well, Java loaded our JAR file and started running the code. The problem is that all Kotlin projects implicitly depend on the Kotlin standard library. We specified it as a compile-time dependency, but it didn't get to the final JAR file. 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 Kotlin library to the classpath:

```
$ java -cp $KOTLIN_HOME/lib/kotlin-runtime.jar:build/libs/gradle-kotlin.jar HelloKt
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 a typical solution usually involves building a so-called uber JAR[12] file. Building uber JAR files in Gradle is supported via the ShadowJar plugin[13]. In order to add it to our project, we need to make the following changes in our `build.gradle` file:

```
1  buildscript {
2    ext.kotlin_version = '1.3.31'
3    ext.shadow_version = '5.0.0'
4
5    repositories {
6      jcenter()
7    }
8
9    dependencies {
10     classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version",
11       "com.github.jengelman.gradle.plugins:shadow:$shadow_version"
```

[12] http://stackoverflow.com/questions/11947037/
[13] https://github.com/johnrengelman/shadow

```
12    }
13  }
14
15  apply plugin: 'kotlin'
16  apply plugin: 'com.github.johnrengelman.shadow'
17
18  repositories {
19    jcenter()
20  }
21
22  dependencies {
23    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
24  }
```

The ShadowJar plugin will analyze the `dependencies` section and copy class files from provided libraries into an uber JAR file. We can build this file by invoking the following command:

```
$ gradle shadowJar
```

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

```
$ java -cp build/libs/gradle-kotlin-all.jar HelloKt
Hello World!
```

All of this can bring us to the conclusion that we actually don't need to install Kotlin tools on production servers to "run Kotlin code". 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.

Editing source files

In the previous section, we learnt how to compile Kotlin files using command line tools and Gradle. Here we'll start using a full-blown IDE, but first let's look at a slightly more lightweight solution - Atom Editor.

Using Atom

When installed from a software repository, Atom adds its executable to the `PATH`, so you can invoke it from anywhere.

Let's start Atom from the directory containing the Gradle project using the following command:

```
$ atom .
```

The `language-kotlin` package adds syntax highlighting, while `linter-kotlin` shows compilation errors. Additionally, a very useful package called `terminal-plus` enhances Atom with a console so that you can compile and run your applications without leaving the editor. However, when experimenting with a simple prototype, constant switching to the console only to check the output may become boring. In this case, it makes sense to adjust the workflow a little bit.

Let's add the `application` plugin to our build.gradle file:

```
1  // buildscript part omitted
2
3  apply plugin: 'kotlin'
4  apply plugin: 'application'
5  apply plugin: 'com.github.johnrengelman.shadow'
6
7  mainClassName = "HelloKt"
8
9  repositories {
10   jcenter()
11 }
12
13 dependencies {
14   compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
15 }
```

The `application` plugin is responsible for running the app. In order to do that, it needs to know the name of the main class, and in our case it is `HelloKt`. Now we have one more way for running our "Hello World" example:

```
$ gradle run
```

Gradle will respond with the following:

```
> Task :run
Hello World!


BUILD SUCCESSFUL in 0s
3 actionable tasks: 1 executed, 2 up-to-date
```

Why all this hassle? Well, the important thing about the `:run` task (added by the `application` plugin) is that it becomes part of the Gradle build lifecycle. Gradle *knows* that `:run` depends on `:compileKotlin`, so if there are source code changes, it will recompile first. This comes in especially handy in the `watch` mode.

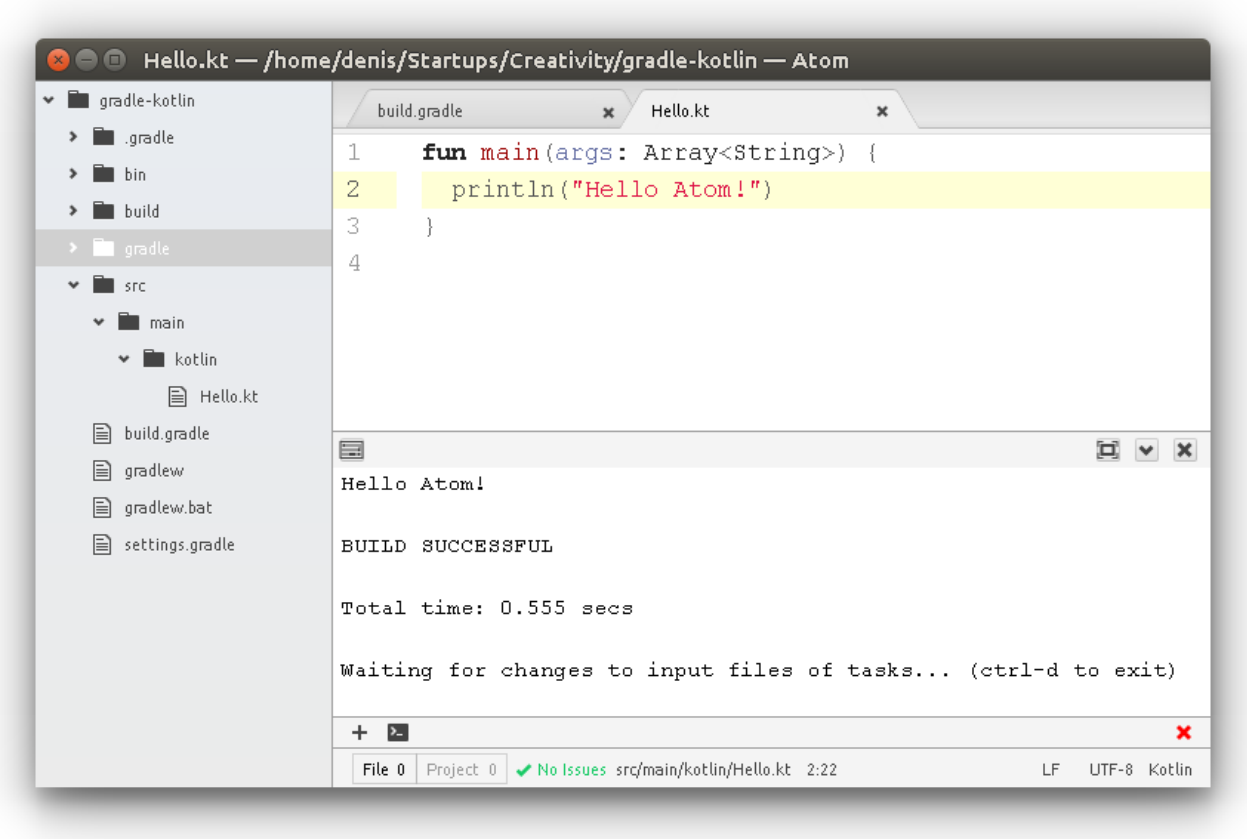
Try typing the following in the Terminal Plus window:

```
$ gradle -t run
```

You will see the already familiar "Hello World" greeting, but in addition to this Gradle will print the following:

```
Waiting for changes to input files of tasks... (ctrl-d to exit)
<-------------> 0% WAITING
> IDLE
```

Now if you make any changes to the `Hello.kt` file, Gradle will recompile it and run the app again. And the good news is that subsequent builds will be much faster.



Atom as a mini-IDE

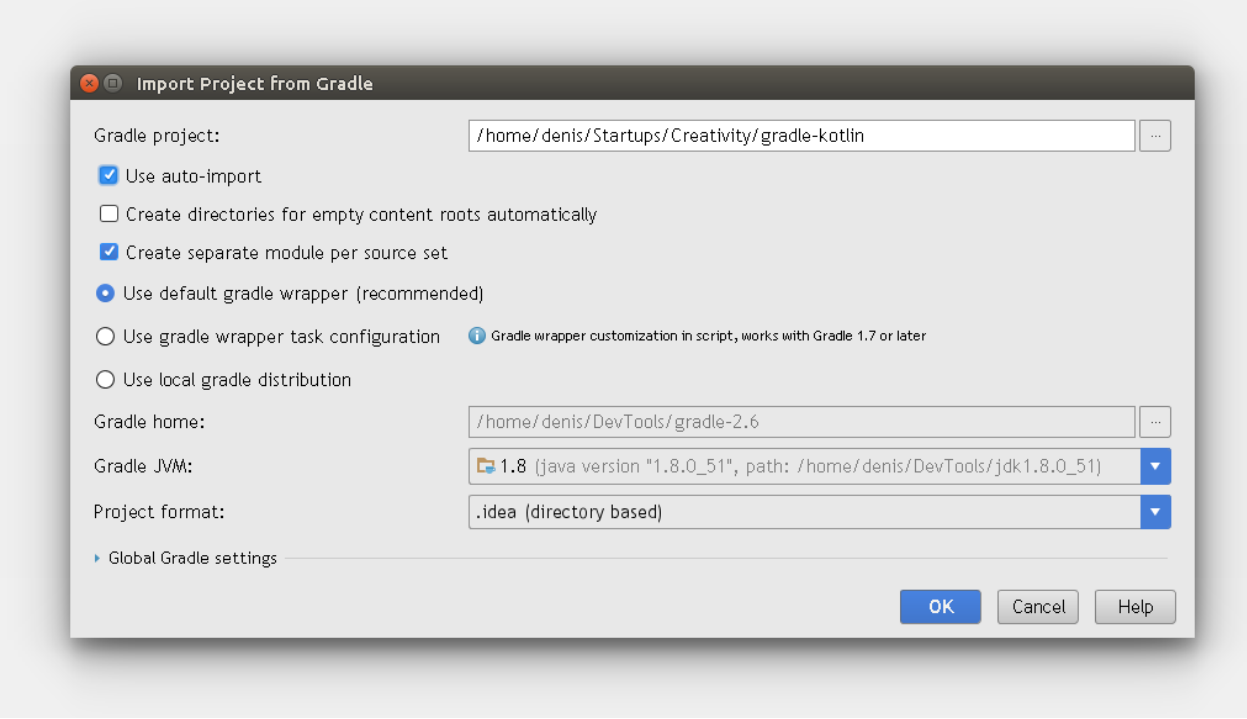
Using IntelliJ IDEA

If you installed IntelliJ IDEA by simply extracting the archive, you can start it by navigating to the `bin` subdirectory and invoking the `idea.sh` script from the command line:

```
$ cd ~/DevTools/idea/bin
$ ./idea.sh
```

When started for the first time, it will ask about importing old settings and show the Welcome window. I recommend using this window to create desktop entries. Simply select "Configure/Create Desktop Entry" and follow the instructions.

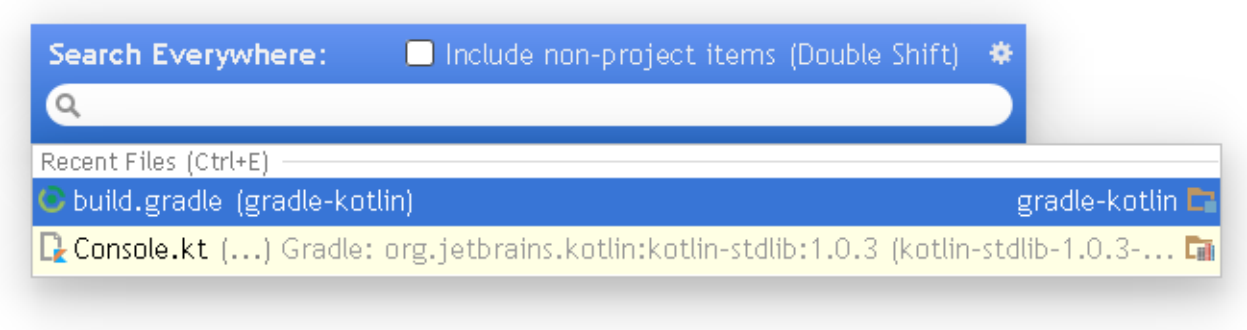
Now, since both Gradle and Kotlin are already integrated in IntelliJ IDEA, importing our project will be an extremely easy task. Simply click "Open" and then choose the directory of the project. The Gradle importing wizard will appear, but you can leave all settings as is. The only thing I recommend doing here is to check "Use auto-import". With auto-importing enabled, IDEA will download and add new libraries automatically once it detects a change in the `build.gradle` file.



Importing Gradle projects

I suspect that most developers are already familiar with at least one IDE from JetBrains, so I'm not going to explain how to use it in detail. However, here are a couple of tricks that you may find useful.

If you type <Shift> twice, IDEA will open the "Search Everywhere" window. This will allow you to quickly open any file, type or method.



Search Everywhere

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>. This is especially useful when you rely on the type inferrer but want to check that you're getting the type you want.

Language fundamentals

In this section, I will demonstrate the basics of the Kotlin programming language. This includes things that are common to most programming languages as well as several features which are specific to Kotlin.

Using the REPL

If we run the Kotlin compiler without parameters, it will start an interactive shell (also known as REPL or Read-Eval-Print-Loop). In this shell, every entered command will be interpreted, and its result will be printed to the console:

```
$ kotlinc
Welcome to Kotlin version 1.3.31 (JRE 1.8.0_144-jdk_2017_08_24_20_46-b00)
Type :help for help, :quit for quit
>>> "Hello World!"
Hello World!
```

In order to quit from the REPL, type `:quit`.

While playing with REPL, it may also be desirable to enable reflection capabilities. Simply start the REPL with the corresponding library:

```
$ kotlinc -cp ~/DevTools/kotlinc/lib/kotlin-reflect.jar
```

The interpreter accompanied with reflection is especially useful while learning the language, and while I'm going to show all output here, feel free to start your own REPL session and follow along.

Defining values

There are three main keywords for defining everything in Kotlin:

keyword	description
`val`	defines a constant (or value)
`var`	defines a variable, very rarely used
`fun`	defines a method or function

Defining a constant is usually as simple as typing

```
>>> val num = 42
>>>
```

Since the assignment operator in Kotlin doesn't return any value, there was no output in the console. However, if you type `num`, you will see its value:

```
>>> num
42
```

By default, the REPL doesn't print type information. If you want to know the actual Kotlin type, type the following:

```
>>> num::class
class kotlin.Int
```

OK, now we know that it's an integer (or `kotlin.Int` in the Kotlin world). Notice that Kotlin 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

```
>>> val num: Short = 42
>>> num::class
class kotlin.Short
```

Variables are not as necessary in Kotlin as they were in Java, but they are supported by the language and could be useful sometimes:

```
>>> var hello = "Hello"
>>> hello::class
class kotlin.String
```

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

As we will see later, most Java statements are expressions in Kotlin, and therefore they evaluate to some value. This feature essentially allows you to write code using `val`s and resort to `var`s only occasionally.

There are several types of functions in Kotlin, and they are defined using slightly different syntax. When defining so-called *single-expression* functions, it is required to specify argument types, but specifying the return type is optional:

```
>>> fun greet(name: String) = "Hello " + name
>>> greet("Joe")
Hello Joe
```

Notice that for single-expression functions the body is specified after the = symbol and curly braces are completely unnecessary. The regular syntax for defining functions in Kotlin is the following:

```
fun greet(name: String): String {
  return "Hello " + name
}
```

Here we're specifying `String` as the return type (after parentheses), defining the body inside curly braces and using the `return` keyword.

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 Kotlin replacement for `void` (which, by the way, is not even a keyword in Kotlin):

```
fun printGreeting(name: String): Unit {
  println("Hello " + name)
}
```

Declaring `Unit` for procedure-like functions is optional. If you don't specify any type, the `Unit` type is assumed.

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

```
>>> fun greet(name: String = "User") = "Hello " + name
>>> greet()
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.

Lambdas

Sometimes it is useful to declare a function literal (or *lambda*/*anonymous function*) and pass it as a parameter or assign it to a variable (or constant). For example, the `greet` function from the example above could also be defined the following way:

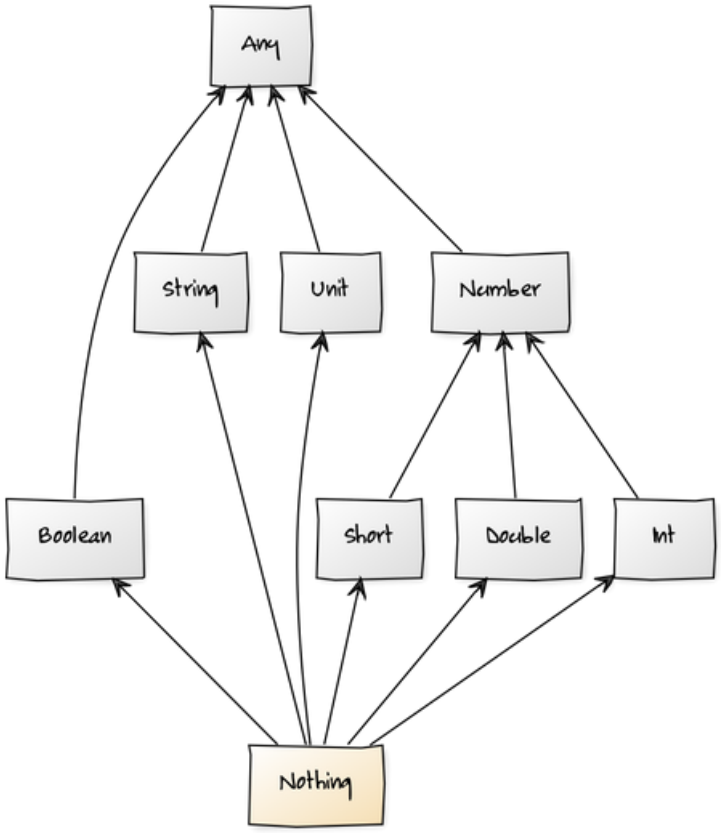
```
>>> var greetVar: (String) -> String = { name -> "Hello " + name }
```

You don't have to type the type of the variable, but if you want the type inferrer to do its work, you'll have to specify the type of parameters:

```
>>> var greetVar = { name: String -> "Hello " + name }
```

Type hierarchy

Since we are on the topic of types, let's take a look at a very simplified type hierarchy (with nullable types omitted, more on them later):



Type hierarchy

`Any` serves as the supertype of all types and resides at the top of the hierarchy. If you inspect its underlying class while running the JVM, you will see that it translates to `java.lang.Object`, but this correspondence is not always that strict. For example, `kotlin.Unit` may correspond to `void` in the Java world when it's used in the result type position.

The idiomatic way to check the type is to use `is`:

```
>>> val str = "Hello"
>>> str is String
true
>>> str is Any
true
```

Kotlin, just like Scala, has a concept of so-called *bottom types* which implicitly inherit from all other types. In the diagram above, `Nothing` implicitly inherits from all types (including user-defined ones, of course). You are unlikely to use bottom types directly in your programs, but they are useful to understand type inference. More on this later.

Nullable types

If you look at the diagram above, you will see that numeric types in Kotlin have a common supertype called `Number`. Unlike Java, though, `Number`s disallow `null`s:

```
>>> var num: Int = null
error: null can not be a value of a non-null type Int
```

If you need a type that allows `null`s, you should append `?` to the type name:

```
>>> var num: Int? = null
>>> num
null
```

After you defined your variable as nullable, Kotlin enforces certain restrictions on its use. Unconditional access to its member fields and methods will fail at compile time:

```
>>> var str: String? = "Hello"
>>> str.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable rec\
eiver of type String?
str.length
```

As the compiler suggests, you can either use a safe call with `?.`, or ignore the danger with `!!.`:

```
>>> str?.length
null
str!!.length
5
```

Obviously, with the latter there is a risk of getting `NullPointerException`:

```
>>> str = null
>>> str?.length
null
>>> str!!.length
kotlin.KotlinNullPointerException
```

Another useful thing when working with nullable types is the *elvis* operator written as `?:`:

```
<expression_1> ?: <expression_2>
```

As almost everything in Kotlin, the elvis operator returns a value. This result is determined by the `<expression_1>`. If it's not `null`, this non-null value is returned, otherwise the `<expression_2>` is returned. Consider:

```
>>> var str: String? = "Hello"
>>> str ?: "Hi"
Hello
>>> var str: String? = null
>>> str ?: "Hi"
Hi
```

The elvis operator is often used when it is necessary to convert a value of a nullable type into a value of a non-nullable type.

Collections

If you need to create a collection, there are several helper functions available:

```
>>> val lst = listOf(1, 2, 3)
>>> lst
[1, 2, 3]
>>> val arr = arrayOf(1, 2, 3)
```

In Kotlin, square brackets translate to `get` and `set` calls, so you can use either `[]` or `get` to access both array and list elements:

```
>>> arr[0]
1
>>> arr.get(0)
1
>>> lst[0]
1
>>> lst.get(0)
1
```

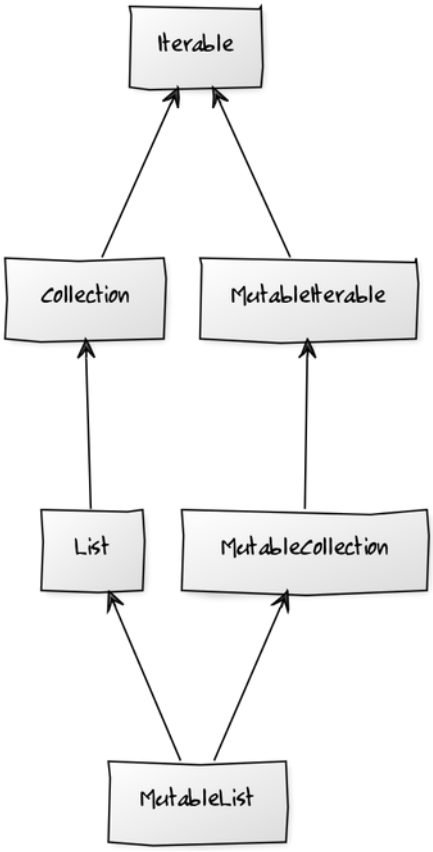
Kotlin distinguishes between mutable and immutable collections, and when you use `listOf`, you actually get an immutable list:

```
>>> lst.add(4)
error: unresolved reference: add
```

If you need a mutable one, use a corresponding function with the `mutable` prefix:

```
>>> val lst = mutableListOf(1,2,3)
>>> lst.add(4)
true
>>> lst
[1, 2, 3, 4]
```

A greatly simplified List hierarchy is shown below:



The List type hierarchy

As you see here, lists come in two flavours - mutable and immutable. Kotlin uses immutable collections by default, but mutable ones are also available.

Another important point is that Kotlin 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:

```
>>> val lst = listOf<Short>(1, 2, 3)
>>> lst[0]::class
class kotlin.Short
```

Defining classes

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

```
>>> 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 (note the absence of the `new` keyword):

```
>>> val person = Person()
>>> person
Line1$Person@4e04a765
```

Note that the REPL showed `Line1$Person@4e04a765` as a value of this instance. We {line-numbers=off}`kotlin // simplified signature fun 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 here. The base implementation of the `toString` method simply returns the name of the class followed by an object hashcode.

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

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

```
>>> class Person(name: String)
>>> val person = Person("Joe")
```

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:

```
>>> class Person(val name: String)
>>> val person = Person("Joe")
>>> person.name
Joe
```

Much better! Adding `val` in front of the parameter name defines a read-only property (similar to an immutable field in other languages). Similarly `var` defines a mutable property (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:

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

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

```
class Person(val name: String) {
  override fun toString() = "Person(" + name + ")"
  operator fun invoke(): String = name
  fun 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. Fortunately, you can easily paste multi-line snippets of code to the REPL, no special commands are necessary.

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

```
>>> val person = Person("Joe")
>>> person
Person(Joe)
>>> person.sayHello()
Hi! I'm Joe
```

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

```
>>> person.invoke()
Joe
>>> person()
Joe
```

Yep, using parentheses on the instance of a class actually calls the `invoke` method defined on this class. Note that in order to achieve this, we had to mark the method definition with the `operator` keyword.

Defining objects

Kotlin 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 {
  fun random100() = Math.round(Math.random() * 100)
}
```

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

```
>>> RandomUtils.random100()
35
```

You can put an object declaration inside of a class and mark it with the `companion` keyword:

```
class Person(val name: String) {
  fun score() = Person.random100()
  companion object Person {
    fun random100() = Math.round(Math.random() * 100)
  }
}
```

With `companion object`, you can even omit its name:

```
class Person(val name: String) {
  fun score() = random100()
  companion object {
    fun random100() = Math.round(Math.random() * 100)
  }
}
```

Finally, you can always define a standalone method just as we did with the `main` function in our "Hello World" example.

Type parametrization

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

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

```
class Cell<T>(val contents: T) {
  fun get(): T = contents
}
```

We defined a class called `Cell` and specified one read-only property `contents`. The type of this property is not yet known. Kotlin doesn't allow *raw types*, so you cannot simply create a new `Cell`, you must create a `Cell` *of something*:

```
>>> val cell = Cell<Int>(2)
>>>
```

Here we're defining a `Cell` passing the `Int` as a type parameter. Fortunately, in many cases it's not necessary as the type can be inferred (for example, from the values passed as constructor arguments):

```
>>> val cell = Cell(2)
>>>
```

Here we passed `2` as an argument, and it was enough for the type inferrer to decide on the type of this instance.

In addition to classes, we can also parametrize functions and methods. When we were discussing collections, we used the `listOf` method, which is parametrized. Of course, you can create your own parametrized methods as well:

```
>>> fun <T> concat(a: T, b: T): String = a.toString() + b.toString()
>>> concat(1, 2)
12
```

And again, usually it's not necessary to specify the type explicitly, but you can always do this to override type inference. This is exactly what we did to define a `List` of `Shorts`:

```
>>> val lst = listOf<Short>(1, 2, 3)
```

Extension functions

Sometimes it is necessary to add a new method to a type, which source code is not in your control. Kotlin (like C#) allows you to do this with *extension functions*.

For example, let's enhance the `String` type by adding a new function that checks whether the string contains only spaces or not:

```
>>> fun String.onlySpaces(): Boolean = this.trim().length == 0
```

Here we're prepending the function name with the name of a *receiver* type. Inside the function, we can refer to the receiver object using the `this` keyword. After our extension function is defined, it becomes available on all `String` instances:

```
>>> "   ".onlySpaces()
true
>>> " 3 ".onlySpaces()
false
```

Interestingly, we can define an extension method on `Any?`:

```
fun Any?.toStringAlt(): String {
  if (this == null)
    return "null"
  else
    return this.toString()
}
```

This, in turn, allows us to invoke the method even if the value is `null`:

```
>>> null.toStringAlt()
null
>>> "22".toStringAlt()
22
```

Packages and imports

Kotlin classes are organized into packages similarly to Java or C#. For example, we could move our `main` function from the "Hello World" example into a package called `hello`. In order to do this, we would put a package declaration on the top of the file:

```
package hello

fun main(args: Array<String>) {
  println("Hello World!")
}
```

Since packages follow the directory structure, we would need to create a new folder called `hello` and move our file there.

After this, the main function becomes a package-level function, and this must be also reflected in the `gradle.build` file:

```
mainClassName = "hello.HelloKt"
```

The `hello.HelloKt` is known as fully qualified name[15] (FQN for short).

More generally, when we need to use something from another package, we need to either reference it by its fully-qualified name or use *imports*.

For example, if we want to use the `File` class from the Java standard library (to which, by the way, all Kotlin programs have access), we need to import it first:

```
>>> import java.io.File
>>> val file = File("readme.txt")
```

Just like in Java or C#, we can import all classes from a particular package using a wildcard `*`:

```
>>> import java.io.*
```

Unlike Java or C#, though, Kotlin can also import standalone functions defined on the package level.

In addition to the imports defined by a developer, Kotlin automatically imports `java.lang.*`, `kotlin.io.*`, `kotlin.collections.*`, `kotlin.*` and so on. This is exactly why we could use methods like `println` or `listOf` in previous examples without importing anything.

When importing a type or function, you can choose how to refer to it in your code:

```
>>> import kotlin.collections.listOf as immutableListOf
>>> immutableListOf<Short>(1, 2, 3)
[1, 2, 3]
```

Loops and conditionals

Unlike Java with *if statements*, in Kotlin *if expressions* always result in a value. In this respect they are similar to Java's ternary operator `?:`.

Let's define a lambda that uses the `if` expression to determine whether the argument is an even number:

```
>>> val isEven = { num: Int -> if (num % 2 == 0) true else false }
>>> isEven
(kotlin.Int) -> kotlin.Boolean
```

When `isEven` is inspected in REPL, the interpreter responds with `(kotlin.Int) -> kotlin.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:

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

expression	type
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.

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 `Any`, so `Any` will be the type of the whole expression:

```
>>> val isEven = { num: Int -> if (num % 2 == 0) 1 else 1.2 }
>>> isEven
(kotlin.Int) -> kotlin.Any
```

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.

What if the *else branch* never returns and instead throws an exception? Well, it's time to recall `Nothing`, which sits at the bottom of the Kotlin type hierarchy. In this case, the type of the *else branch* is considered to be `Nothing`. Since `Nothing` is the bottom type, any other type is considered a supertype for it and therefore the whole expression will have the type of the *if branch*.

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

There's not much you can do with `Nothing`, but it is included in the type hierarchy, and it makes the rules that the type inferrer uses more clear.

while and for loops

The `while` loop is almost a carbon copy of its Java counterpart. Unlike *if*, *while* is not an expression, so it's called for a side effect. Moreover, it almost always utilizes a `var` for iterations:

```
1  var it = 5
2  while (it > 0) {
3    println(it)
4    it -= 1
5  }
```

In a modern language it looks rather *old school* and in fact, you will rarely use it in Kotlin. There are many alternative ways to achieve the same thing, including `for` loops:

```
>>> for (i in 5 downTo 1) println(i)
5
4
3
2
1
```

This syntax looks odd and certainly requires some explanation, so here you go. The `for` loop in Kotlin works with everything that has the `iterator` function (extension functions will work as well). The type of the value that `iterator` returns doesn't matter as long as it has two functions: `next()` and `hasNext()`. The `hasNext()` function is used by the `for` loop to determine whether there are more items or it's time to stop.

OK, what about `5 downTo 1`. It turns out that `downTo` is an extension function defined on all integer types (`Int`, `Long`, `Short` and so on). It accepts an argument and builds an `IntProgression`. Not surprisingly, the `IntProgression` has the `iterator` method (required by `for`).

One thing that is left to explain is the fact that we used the `downTo` function without dots or parentheses. Why is that? If you look at Kotlin source code, you will see that its signature looks like this:

```
// simplified signature
infix fun Int.downTo(to: Int): IntProgression { /* implementation code */ }
```

The magic word here is `infix` because it marks the function as suited for infix notation.

By the way, nothing prevents you from defining your own functions in this way!

String templates

Just as most modern languages, Kotlin allows to execute arbitrary code inside string literals. In order to use this feature, simply put `$` in front of any variable or value you want to interpolate:

```
>>> val name = "Joe"
>>> val greeting = "Hello $name"
>>> greeting
Hello Joe
```

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

```
>>> "Random number is ${Math.random()}"
Random number is 0.4884723200806935
```

If you need to spread your string literal across multiple lines or include a lot of special characters without escaping you can use triple-quote to create *raw strings*. Note that the usual backslash escaping doesn't work there:

```
>>> """First line\nStill first line"""
First line\nStill first line
```

Interfaces

Interfaces in Kotlin are quite similar to Java 8. They can contain both abstract (without implementation) methods as well as concrete ones. You can declare your class as implementing multiple interfaces, but if they contain abstract members, then you must either implement them or mark your class as `abstract`, so the Java rule 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.

Let's look at a rather simplistic example:

```
1  interface A { fun a(): Unit = println("a") }
2
3  interface B { fun b(): Unit }
```

We defined two interfaces so that interface `A` has one concrete method, and interface `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 interfaces, we will have to implement the `b` method:

```
class C: A, B { override fun 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 how Kotlin supports functional programming either via build-in language constructs or by means of available third-party libraries.

Higher-order functions

Functions that accept other functions as their arguments are called *higher-order functions*[16]. They are supported by virtually any modern programming language including JavaScript, Ruby, Python, PHP, Java, C# and so on. Not surprisingly, Kotlin supports them as well.

map, filter, foreach

Let's create a simple list that we will be using to demonstrate common higher-order functions supported by Kotlin collections:

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

Arguably, the most well-known higher-order function is `map`. It accepts a function literal that is used to transform collection elements one by one. For example:

```
list.map { el -> el * el }              // [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. The above example can also be written like this:

```
list.map { it * it }
```

Here we're using a special value called `it`, which is available in lambda expressions and refers to the first argument.

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

```
list.filter { el -> el % 2 == 0 }       // [2, 4]
```

Here we're creating a new collection containing only even numbers.

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

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

```
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 - `[1, 2, 3, 4]` - we have another list of strings:

```
val list2 = listOf("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:

```
1  list.map { el1 ->
2    list2.map { el2 ->
3      "$el1$el2"
4    }
5  }
```

The resulting collection will have the following elements:

```
[[1a, 1b], [2a, 2b], [3a, 3b], [4a, 4b]]
```

We ended up with a list of lists. Not quite what we expected, right? The problem here is the outer `map`. If you look at the Kotlin source code, you will see that the `map` method accepts a function that takes one element and returns another element.

```
// simplified map signature
fun <T, R> List<T>.map(transform: (T) -> R): List<R>
```

However, we want to pass a function that takes an element and returns a list. Is there such a function defined on `kotlin.collections.List`? It turns out that yes, and it's called `flatMap`:

```
// simplified flatMap signature
fun <T, R> List<T>.flatMap(transform: (T) -> List<R>): List<R>
```

Using `flatMap` the problem can be solved easily:

```
list.flatMap { el1 ->
  list2.map { el2 ->
    "$el1$el2"
  }
}
```

The result is exactly what we needed:

```
[1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b]
```

fold and foldRight

The `fold` method defined on `Iterable` takes two parameters. First parameter - the initial value - is used as the staring point of the computation, and the second parameter is a function (or lambda) that describes how to compute the next value using the previous one and the current accumulator value. It may sound terribly complicated, but in reality `fold` is absolutely straightforward to use.

For example, we can use `fold` to calculate the sum of elements in our list - `[1, 2, 3, 4]`:

```
list.fold(0, { acc, next ->
  acc + next
})
```

Note that the type of the initial value determines the type of the whole expression. There is also a similar method called `foldRight`, which does the same thing but iterates from right to left.

One interesting feature of Kotlin is that whenever a lambda happens to be the last arguments of the function, it can be moved out of parentheses:

```
list.fold(0) { acc, next ->
  acc + next
}
```

The latter version looks nicer and it is generally receommended to use this style when applicable.

Tuples and data classes

Quite often, classes are created only for storing data. In order to simplify the whole process, Kotlin introduces so called *data classes* (very similar to *case classes* in Scala). Remember our `Person` class from the section about fundamentals? Let's add several more fields:

```
data class Person(val firstName: String,
  val lastName: String,val birthYear: Int)
```

By making `Person` a data class we are getting the following:

- a correctly implemented `hashcode` method used by some collections from the standard library
- a correctly implemented `equals` method, which makes two objects with the same set of values equal, so `Person("Joe", "Black", 1990) == new Person("Joe", "Black", 1990)` returns `true`
- a new `toString` method, which prints property values instead of hashcode
- a `copy` method
- a number of `componentN` methods

The `copy` method allows to make modified copies of the object:

```
>>> val person = Person("Joe", "Black", 1990)
>>> person.copy(lastName = "Smith")
Person(firstName=Joe, lastName=Smith, birthYear=1990)
```

Note that specifying parameter names when calling the function is not specific to the `copy` function. In fact, using names when calling a function is encouraged when you pass `Boolean` arguments. Mixing two styles in one call is not allowed, though.

The `componentN` methods allow to access object properties without referring to their names:

```
>>> person.component1()
Joe
>>> person.component3()
1990
```

In addition, the `componentN` methods enable the *destructuring* of an object into variables or values:

```
>>> val person = Person("Joe", "Black", 1990)
>>> val (first, last, year) = person
>>> last
Black
```

This idea of accessing properties is somewhat related to *tuples*[17]. A tuple is a finite ordered list of elements and it's often used in functional programming. In Kotlin, when you need a tuple, it's usually better to define a data class. However, there are also two generic tuples defined in the standard library - `Pair` and `Triple`:

[17] https://en.wikipedia.org/wiki/Tuple

```
>>> val pair = Pair(1, "one")
>>> pair.first
1
>>> pair.component1()
1
```

Laziness

By default, all values in Kotlin are computed *eagerly* before they are passed to a method. To understand why this is important, let's write a simple logging function:

```
fun logEagerly(enabled: Boolean, message: String): Unit {
  if (enabled) {
    println(message)
  }
}
```

In theory, this method only prints the `message` if the logging is enabled. However, the string template needs to be fully resolved even if `isEnabled` is `false`:

```
logEagerly(isEnabled, "User ${userId()} logged out from the system")
```

More specifically, the `userId()` function will be called even if its output is not used.

Let's modify the `log` function so that instead of taking a `String`, it takes a function that returns a `String`:

```
fun logLazily(enabled: Boolean, message: () -> String): Unit {
  if (enabled) {
    println(message())
  }
}
```

Now, if you want to call this function, you will have to wrap your string in curly braces (Kotlin requires all lambdas to be wrapped in curly braces):

```
logLazily(isEnabled, {"User ${userId()} logged out from the system" })
```

The main difference, however, is that `userId()` is not called until needed. If `isEnabled` is `false`, it will not be called at all!

More generally, if you want to delay the initialization of a value, you can use the following syntax:

```
val lazyValue: String by lazy {
    println("initializing!")
    "Hello"
}
println("main")
println(lazyValue)
```

If you try running this code, you'll see that the word `main` is printed before `initializing`. This happens because the initilizing lambda expression is not executed until `lazyValue` is referenced.

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. The most well-known example of an ADT is `Option`, so we're going to examine this important concept using our own (rather minimalistic) implementation of this type.

Let's start by defining `Option` as an abstract class with two abstract methods:

```
abstract class Option<out T> {
  abstract fun get(): T
  abstract fun isEmpty(): Boolean
}
```

In essence, the `Option` class defines a container that may or may not contain a value. The absence of a value is represented by the `None` object:

```
object None : Option<Nothing>() {
  override fun get() = throw Exception()
  override fun isEmpty() = true
}
```

Since there is no value, we can only throw an exception on `get()`.

Similarly, `Some` serves as a storage for the existing value:

```
class Some<out T>(val value: T) : Option<T>() {
  override fun get() = value
  override fun isEmpty() = false
}
```

As a result, `Option` may either contain a value or be empty, i.e `None` and `Some` form disjoint sets. Therefore, `Option` is an ADT because it's formed by combining two other types - `None` and `Some`.

when expressions

You already know that it is possible to use the `is` operator to check the type of an expression:

```
>>> val maybeName: Option<*> = Some("Joe")
>>> maybeName is Some
true
>>> maybeName is Option
true
>>> maybeName is None
false
```

Kotlin, however, provides an alternative way of doing something similar - the *when expression*:

```
1  when(maybeName) {
2    is Some -> println("Some")
3    is None -> println("None")
4  }
```

Here we're passing `maybeName` to the `when` expression and it chooses one of the branches depending on the type. It may also contain the `else` block used when nothing else satisfies conditions.

Option

The abstract data type `Option` defined earlier is actually a very useful abstraction. In fact, it is so useful, that many programming languages (Scala, Java 8) have it as part of their standard libraries. There is no `Option` in the Kotlin standard library, but it can be found in a library called "funKTionale"[18] written by Mario Arias.

In order to include this library in your project, simply add the following to the `build.gradle` file:

```
1  buildscript {
2    ext.funktionale_version = '1.2'
3    // ...
4  }
5  dependencies {
6    compile "org.funktionale:funktionale-all:$funktionale_version"
7    // ...
8  }
```

[18] https://github.com/MarioAriasC/funKTionale

Just like our minimalistic version, the `Option` type from funKTionale is formed by two subtypes: `None` and `Some`. In addition to `isEmpty()` and `get()`, it comes with a great number of useful methods and extension functions.

The safest way to create a value of type `Option` is to use the `toOption()` extension function. This function becomes available on all values (including nulls) after importing the `option` package:

```
// import org.funktionale.option.*

val name = getName()
val maybeName = name.toOption()
```

If `getName()` returns `null`, `maybeName` will be initialized as `None`, otherwise it will become `Some` with a returned value inside.

Here are only several of the methods defined on the `Option` class:

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 is not
`get`	returns the value if it's there or throws an exception if it is not
`forEach`	allows to use existing value in a functional block without returning anything
`map`	allows to transform one option into another

Let's look at one concrete example to get the idea of how `Options` are used. Imagine that we have a method that returns `Int` values but occasionally returns `null`s:

```
1  fun generateNumber(): Int? {
2    val num = Math.round(Math.random() * 100)
3    return if (num <= 90) num.toInt() else null
4  }
```

Now assume that we need to generate two numbers and then calculate their sum:

```
1  val maybeNum1 = generateNumber().toOption()
2  val maybeNum2 = generateNumber().toOption()
```

Calculating the sum only makes sense if both values are present. Obviously, we can use the already familiar `isDefined()` and `get()` methods:

```
val maybeSum = if (maybeNum1.isDefined() && maybeNum2.isDefined())
  Option.Some(maybeNum1.get() + maybeNum2.get())
else Option.None
```

Since the `if` expression in Kotlin actually returns a value, this approach doesn't look that bad. However, it seems that we're performing to much "dancing" around these values.

Remember how we used the combination of `map` and `flatMap` to concatenate the values of two lists? From the point of view of a functional programmer, our current task is no different, so let's do it again:

```
val maybeSum = maybeNum1.flatMap { num1 ->
  maybeNum2.map { num2 -> num1 + num2 }
}
```

Much better! I suggest you memorize this construct because as we will see later, it is extremely common in functional programming and used for doing a great deal of seemingly unrelated things like working with dangerous code or asynchronous results.

By the way, in functional programming the presence of `flatMap` is a characteristic of a *monadic structure*. Unlike Haskell or Scala, Kotlin doesn't have a special syntactic sugar for dealing with monads, but this feature may be added in future versions.

What if we had not two, but several options? For this case, funKTionale provides an extension method called `sequential()`, which you can use to convert a `List` of `Options` to an `Option` of a `List`:

```
val seq = listOf(maybeNum1, maybeNum2, maybeNum3)
val maybeSum = seq.sequential().map { list ->
  list.fold(0) { acc, next -> acc + next }
}
```

Here we're forming a new list containing all of our `Option` values. The `sequential()` function builds a new `Some` instance with `List<Int>` if all values are present or `None` if at least one of them is absent. Once we have the list, calculating the sum using `fold()` is straightforward.

Either

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

```
object DangerousService {
  fun queryNextNumber(): Long {
    val source = Math.round(Math.random() * 100)
    if (source <= 60) return source
    else throw 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:

```
>>> DangerousService.queryNextNumber()
60
>>> DangerousService.queryNextNumber()
8
>>> DangerousService.queryNextNumber()
java.lang.Exception: The generated number is too big!
        at Line7$DangerousService.queryNextNumber(line7.kts:5)
```

To work with dangerous code in Kotlin you can use `try-catch-finally` blocks. The syntax mostly follows Java's , so you just wrap dangerous code in the `try` block and then if an exception occurs, you can do something about it in the `catch` block.

```
val number = try {
  DangerousService.queryNextNumber()
} catch(e: Exception) {
  e.printStackTrace()
  60
}
```

The good news about `try` blocks in Kotlin 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` cannot store information about the exception.

Another problem is that since there are no checked exceptions, it is not quite obvious how to tell the caller which error may happen during the method invocation.

A better alternative is to use `Either` from funKTionale. The `Either` type is an algebraic data type that is formed by two subtypes - `Left` and `Right`. Traditionally, `Right` is used for storing successfully calculated values and `Left` is used for storing errors.

The `Either` class comes with a lot of utility methods that make working with it very convenient:

method	description
`isRight`	checks whether it's `Right`
`isLeft`	checks whether it's `Left`
`left`	returns `LeftProjection`
`right`	returns `RightProjection`
`fold`	allows to specify what to do in each case

In addition to these methods, both projections have `map`, `flatMap`, `filter` and other familiar methods.

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

```
val numberE = eitherTry { DangerousService.queryNextNumber() }
```

Even better, we can make `Either` the return type of our function by wrapping its body in `eitherTry`:

```
fun queryNextNumber(): Either<Throwable, Long> = eitherTry {
  val source = Math.round(Math.random() * 100)
  if (source <= 60) source
  else throw Exception("The generated number is too big!")
}
```

Now the callers of our method will know that it may throw an exception and they should take it into account:

```
numberE.fold({
  exc -> reportError()
}, {
  res -> processNumber(res)
})
```

Let's now assume that we have two `Either` objects generated by `queryNextNumber()` and we want to calculate their sum if both numbers are present. If one of the numbers is absent, we still want to save the exception for future investigation. Here is how we can solve this problem with projections:

```
if (number1T.isRight() && number2T.isRight()) {
  Either.Right<Throwable, Long>(number1T.right().get() +
    number2T.right().get())
} else {
  val exception = if (number1T.isLeft())
    number1T.left().get()
  else number2T.left().get()
  Either.Left<Throwable, Long>(exception)
}
```

Even though, it achieves the goal, the code is extremely clumsy. The problem with the `Either` type from funKTionale is its unbiased nature: it doesn't make any assumptions about `Left` and `Right` and treats them equally. Consequently, in order to use `maps` or `flatMap` we have to switch to projections and it doesn't provide a great user experience.

In practice, we can always agree to associate `Right` with success and `Left` with failure, therefore making `Either` *right-biased*. And fortunately, Kotlin's extension functions allow us to do this without even touching funKTionale source code.

Since version 0.9, FunKTionale adds[19] a new type called `Disjunction`, which is basically a right biased `Either`. However, implementing something similar manually is a great exercise that will allow us to experience working with extension functions first-hand.

Let's define the `map` and `flatMap` methods on `Either` so that they transform the `RightProjection`, but leave `LeftProjection` as is:

```
fun <L, R, NR> Either<L, R>.map(f: (R) -> NR) =
  this.right().map(f)

fun <L, R, NR> Either<L, R>.flatMap(f: (R) -> Either<L, NR>) =
  this.right().flatMap(f)
```

That was easy, wasn't it? Here we're simply calling `right()` and delegating everything to `RightProjection`. This way, we can also implement other methods such as `filter`, `forEach` etc.

Now that we have `map` and `flatMap` available on `Either`, we can solve our task using an already familiar construct:

[19] https://github.com/MarioAriasC/funKTionale/issues/11

```
val maybeSum = number1T.flatMap { num1 ->
  number2T.map { num2 -> num1 + num2 }
}
```

Note that when using `maps` and `flatMaps` we always end up with the same container type that was used initially. This happens because `maps` and `flatMaps` can merely transform the value inside a container, but they don't change the type of a container itself.

Promise

The final thing we are going to examine here is the `Promise` class. `Promises` allow you to work with asynchronous code in a type-safe and straightforward manner without resorting to concurrent primitives like threads or semaphores.

The current version of Kotlin doesn't include its own build-in mechanisms for working with asynchronous code, so we're going to use an excellent library called Kovenant[20] written by Mark Platvoet. In order to include this library in your project, add the following lines to your build.gradle file:

```
buildscript {
  ext.kovenant_version = '3.3.0'
  // ...
}
dependencies {
  compile "nl.komponents.kovenant:kovenant:$kovenant_version"
  // ...
}
```

If you enabled the auto-importing feature, IDEA will download necessary files, and Kovenant will be ready to use.

In essence, a promise represents an eventual result of an asynchronous operation. The important thing here is that with promises you can work with asynchronous results even before they are computed. To understand how to use promises in Kotlin, let's make one simple adjustment to our fictitious service:

[20] https://github.com/mplatvoet/kovenant

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

Here we're adding a pause of 2 seconds before our service even starts doing its job.

Imagine that we need to calculate the sum of two numbers returned by this service. If you try to simply call it from the main thread, it will hang for at least 4 seconds.

To avoid it, we can wrap our calls in `task`:

```
// import nl.komponents.kovenant.task

val num1P = task { DangerousService.queryNextNumber() }
val num2P = task { DangerousService.queryNextNumber() }
```

The Kovenant API defines `task` as a function that takes a `Context` and a function literal:

```
fun <V> task(context: Context = Kovenant.context,
  body: () -> V): Promise<V, Exception>
```

For `Context` there is a default value, so we don't have to worry about it for now and instead we can simply put our code in curly braces and get a `Promise` back. Note that there will be no pauses here, and we'll get our `Promises` pretty much immediately. The next question is - what we can do with them? It turns out quite a lot of things, actually.

The `Promise` class provides a few methods that you can use:

method	description
`always`	applies the provided callback when the `Promise` completes, successfully or not
`fail`	applies the provided callback when the `Promise` completes with an exception
`success`	applies the provided callback when the `Promise` completes successfully
`isSuccess`	checks whether the `Promise` was completed successfully
`isFailure`	checks whether the `Promise` was completed with an error
`get`	waits until `Promise` completes and returns the value (blocking the thread)
`then`	allows to transform one `Promise` into another

So, in theory, it is possible to work with `Promises` relying solely on callbacks:

```
num1P.success { num1 ->
  num2P.success { num2 ->
    println(num1 + num2)
  }
}
```

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.

Fortunately, the `kovenant-functional` module introduces several additional methods including `map` (which is an alias for `then`) and `bind`. If you look at the `bind` signature, you will see that it's actually `flatMap` that we've seen earlier. Since we already know how to combine two values with `map` and `flatMap`, the solution becomes straightforward:

```
val sum = num1P.bind { num1 ->
  num2P.map { num2 -> num1 + num2 }
}
```

What if we have many promises and want to use their results? In this case, we can pass them (up to twenty elements) into `combine` and get back a `Promise` of a `Tuple`:

```
val sum = combine(num1P, num2P).map { tuple ->
  tuple.component1() + tuple.component1()
}
```

Since all tuples in Kovenant are defined as data classes, their components could be easily extracted if necessary:

```
val sum = combine(num1P, num2P).map { tuple ->
  val (num1, num2) = tuple
  num1 + num2
}
```

As you probably noticed, `Promise` comes with a built-in way for dealing with exceptions, so when your function already returns a `Promise`, usually there is no need for `Either`.

Working with Vert.x

In this section, we will finally start diving into the world of Web development. There are many ways to develop Web applications in Java (and therefore, in Kotlin), but we're going to concentrate on using Vert.x - a modern toolkit for crafting reactive applications.

Installing Vert.x

If you've ever heard about Web development in Java, then you probably know about Servlets. Servlets have been around since the late 90s, and they are a proven and established technology. However, we are going to choose another path and base our application on Vert.x[21].

Vert.x comes with its own Web server built on top of Netty - a high-performance and non-blocking framework originally developed by JBoss. And the good thing about it is that we will be able to start the server in the embedded mode from the `main` function and therefore get full control over our app.

Before starting to work on our Web application, let's check that we have everything in place. Our `build.gradle` at this stage should look like this:

```
buildscript {
  ext.kotlin_version = '1.3.31'
  ext.shadow_version = '5.0.0'
  ext.funktionale_version = '1.2'
  ext.kovenant_version = '3.3.0'

  repositories {
    jcenter()
  }

  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version",
      "com.github.jengelman.gradle.plugins:shadow:$shadow_version"
  }
}

apply plugin: 'kotlin'
apply plugin: 'application'
apply plugin: 'com.github.johnrengelman.shadow'
```

[21] http://vertx.io/

```
20
21 repositories {
22   jcenter()
23 }
24
25 dependencies {
26   compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
27   compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
28   compile "org.funktionale:funktionale-all:$funktionale_version"
29   compile "nl.komponents.kovenant:kovenant:$kovenant_version"
30 }
31
32 mainClassName = "HelloKt"
```

Let's start by adding the Vert.x Core Module packages to the dependencies section of `build.gradle` (the companion code is available on GitHub[22]):

```
1 buildscript {
2   ext.vertx_version = '3.7.1'
3   // ...
4 }
5 dependencies {
6   compile "io.vertx:vertx-core:$vertx_version"
7   // ...
8 }
```

One more thing that we need to add is the Logging Facade - SLF4J. Vert.x uses Java's standard logging facilities, so we also need to add the bridge - `slf4j-jdk14`. Let's do that:

```
1 buildscript {
2   ext.slf4j_version = '1.7.25'
3   // ...
4 }
5 dependencies {
6   runtime "org.slf4j:slf4j-jdk14:$slf4j_version"
7   compile "org.slf4j:slf4j-api:$slf4j_version"
8   // ...
9 }
```

We are going to leave logging settings untouched, so Vert.x will use its reasonable defaults. This should suffice for now, but we will revisit the issue later when we get to deployment.

[22] https://github.com/denisftw/modern-web-kotlin

Now you can refresh the Gradle project in the IDE. Once IDEA downloads necessary files (and there will be quite a few of them - remember, we are embedding an entire Web server), we can continue working.

Dispatching requests

To check that everything works, put the following into the `Hello.kt`:

```
import io.vertx.core.*

fun main(args: Array<String>) {
  val server = Vertx.vertx().createHttpServer()
  server.
    requestHandler({ req ->
      req.response().end("Hello World!") }).
    listen(8080) { handler ->
      if (!handler.succeeded()) {
        System.err.println("Failed to listen on port 8080")
      }
    }
}
```

Here we're creating an instance of `HttpServer` and assigning a primitive request handler to it. The easiest way to run the application from IntelliJ IDEA is to click on the "K" icon and choose "Run" from the menu.

If you look at the console output, you may spot the following warning:

```
$ gradle build
w: /home/user/Workspace/gradle-kotlin/src/main/kotlin/Hello.kt: (4, 22):
Calls to static methods in Java interfaces are deprecated in JVM target 1.6.
Recompile with '-jvm-target 1.8'
```

By default, Gradle produces Java 6-compatible code, but it doesn't make much sense since Vert.x 3 always requires Java 8. Let's get rid of this warning by adding the following lines to the `build.gradle` file:

```
compileKotlin {
  kotlinOptions.jvmTarget= "1.8"
}
```

Finally, re-run the app to see it in action:

```
$ gradle run
<==========---> 83% EXECUTING [1s]
> :run
```

If you open the browser and navigate to "http://localhost:8080/", you'll see a familiar "Hello World" rendered on the page.

OK, now we're serving requests on `/`, but what about other paths? Well, the best way to handle them is to introduce a router.

In order to use a router we will need another dependency - the `vertx-web` module - so let's add it:

```
dependencies {
  compile "io.vertx:vertx-web:$vertx_version"
  // ...
}
```

In Vert.x, a `Router` is an object that accepts requests and dispatches them to corresponding handlers. We can rewrite our previous example in the following way:

```
val vertx = Vertx.vertx()
val server = vertx.createHttpServer()
val router = Router.router(vertx)
val logger = LoggerFactory.getLogger("VertxServer")

router.route().handler { routingContext ->
  val response = routingContext.response()
  response.end("Hello World!")
}

server.requestHandler{ router.accept(it) }.listen(8080) { handler ->
  if (!handler.succeeded()) {
    logger.error("Failed to listen on port 8080")
  }
}
```

We have a couple of lines more, but I would argue that the code is actually more readable now. Instead of appending a handler to the server, we're appending it to a route, which makes more sense.

However, right now we don't make use of routing - whichever path you type in the browser address field, the result is always the same. This happens because the parameterless `route()` method catches everything. To make it catch only the root path, we should make the following change:

```
// ...
router.get("/").handler { routingContext ->
  val response = routingContext.response()
  response.end("Hello World!")
}
```

After this, our "Hello World!" message will be served only on the root path, and any attempts to load anything else will result in a big "Resource not found" message accompanied with the "404 Not Found" status.

Thymeleaf templates

Obviously, we're not going to serve our pages by manually rendering strings. After all, this book is about *modern* Web development and not about developing apps like it's 1995.

Vert.x has a decent support for rendering templates including Handlebars, Jade and Thymeleaf. We're going to use Thymeleaf[23], but feel free to evaluate other options taking into account your specific needs.

Support for templates is extracted into separate modules, so we need to add the Thymeleaf module to the list of dependencies:

```
dependencies {
  compile "io.vertx:vertx-web-templ-thymeleaf:$vertx_version"
  // ...
}
```

Then, in the root directory of our project we need to create a new folder for storing templates:

```
$ mkdir -p public/templates
```

The `index.html` template will have the following content:

[23] http://www.thymeleaf.org/

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home page</title>
</head>
<body>
  <h1>Hello Thymeleaf!</h1>
  Current time: <span th:text="${time}">01/01/2016 0:00 AM</span>
</body>
</html>
```

The only interesting thing here is `th:text`. When our template is rendered, the engine will take the value of `time` and replace the placeholder text. This way, we will see that variables are correctly resolved and everything works as expected.

In the application code, let's first initialize the template engine:

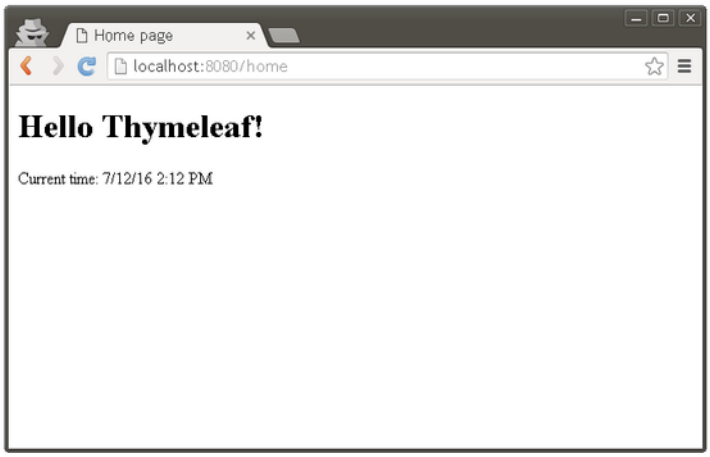
```
val templateEngine = ThymeleafTemplateEngine.create(vertx)
```

Once the engine is initialized, we can use it to render the `index.html` template for a specific route:

```
router.get("/home").handler { ctx ->
  val params = JsonObject(mapOf("time" to SimpleDateFormat().format(Date())))
  templateEngine.render(params, "public/templates/index.html") { buf ->
    if (buf.failed()) {
      logger.error("Template rendering failed", buf.cause())
    } else {
      val response = ctx.response()
      response.end(buf.result())
    }
  }
}
```

We're putting information about current time into the `params` object (thus making it available for the template engine) and then calling the `render` method. The method takes the JSON map of parameters, the template path and a lambda. Since all rendering happens asynchronously, this function literal works as a callback.

If all is well, you should see a similar page:



Hello Thymeleaf

Watching for source changes

When you are in the process of developing something, you constantly need to change source code and check results. That's why the `-t` flag in Gradle that we discovered earlier is so useful, and that's why we would like to have something similar now. Watching for source changes while there is a server running is quite different, though. Luckily, this scenario is handled by Vert.x itself, but in order to enable it, we will need to make some changes.

First, instead of starting the server from the `main` function, we need to delegate this task to the Vert.x Launcher. The Launcher will be able to start our server if we extend the `AbstractVerticle` class and put our code to its `start` method:

```
1  package verticles
2
3  class MainVerticle : AbstractVerticle() {
4    override fun start(startFuture: Future<Void>?) {
5      val server = vertx.createHttpServer()
6      // ...
7      server.requestHandler{ router.accept(it) }.listen(8080) { handler ->
8        if (!handler.succeeded()) {
9          System.err.println("Failed to listen on port 8080")
10       }
11     }
12   }
13 }
```

We're putting the new class in a new package called `verticles` and copying the body of our `main` function to the `start` method. The only difference is that here we already have a reference to the `Vertx` instance, so we don't need to create it manually.

Second, we need to make some changes to `build.gradle` so that the `:run` task knows how to start the application:

```
mainClassName = 'io.vertx.core.Launcher'

def mainVerticleName = 'verticles.MainVerticle'

def watchForChange = 'src/**/*.kt'

def doOnChange = 'gradle classes'

run {
  args = ['run', mainVerticleName, "--redeploy=$watchForChange",
          "--launcher-class=$mainClassName", "--on-redeploy=$doOnChange"]
}
```

Here, we're instructing Gradle to run our application using a specific set of command line arguments. The arguments such as `--redeploy` and `--on-redeploy` will be passed to the Vertx Launcher so that it knows what to do when source code changes. Here we only care about Kotlin files, thus the `src/**/*.kt` mask.

If you start the app using Gradle, you will see the following:

```
$ gradle run
Starting a Gradle Daemon (subsequent builds will be faster)

> Task :run
Jun 15, 2017 3:33:02 PM io.vertx.core.impl.launcher.commands.Watcher
INFO: Watched paths: [/home/denis/Startups/Creativity/gradle-kotlin/./src]
Jun 15, 2017 3:33:02 PM io.vertx.core.impl.launcher.commands.Watcher
INFO: Starting the vert.x application in redeploy mode
Starting vert.x application...
361c9516-7651-4333-81fb-bc859a2e3718-redeploy
Jun 15, 2017 3:33:03 PM verticles.MainVerticle
INFO: Starting the server
```

The server will start listening on port 8080 as usual, but if you change anything in the source code, it will rerun the `:classes` task and redeploy the server:

```
INFO: Redeploying!
Stopping vert.x application '361c9516-7651-4333-81fb-bc859a2e3718-redeploy'
Application '361c9516-7651-4333-81fb-bc859a2e3718-redeploy' terminated with status 0
:compileKotlin
INFO: Redeployment done
```

Experience suggests that this little trick usually dramatically increases productivity.

Serving asynchronous results

What if want wanted to query information from a remote server instead of generating it on a local machine? In theory, we could make a blocking request, and once we have the result, pass it further. This, however, conflicts with the non-blocking nature of Vert.x. Fortunately, Vert.x provides a family of `blockingHandler` methods, and if we were using Java, we would definitely utilize one of them.

However, in Kotlin, we can wrap our request into a `task` and get back a Kovenant `Promise`:

```
1  val timeP = task { SimpleDateFormat().format(Date()) }
2  timeP.success { time ->
3    val params = JsonObject(mapOf("time" to time))
4    templateEngine.render(params, "public/templates/index.html") { buf ->
5      // ...
6    }
7  }
```

However, there is one little problem. When Kovenant executes a request, it uses a thread pool. At the same time, Vert.x maintains its own pool of worker threads. Since we need only one thread pool, we should make Kovenant use the Vert.x one. In order to do that, let's add an integration library called `klutter-vertx3`:

```
1  buildscript {
2    ext.klutter_version = '2.5.3'
3    // ...
4  }
5  dependencies {
6    compile "uy.kohesive.klutter:klutter-vertx3:$klutter_version"
7    // ...
8  }
```

To make sure that Kovenant is correctly initialized, we need to call `VertxInit.ensure()` on server start:

```
override fun start(startFuture: Future<Void>?) {
  VertxInit.ensure()
  val server = vertx.createHttpServer()
  // ...
}
```

Obviously, there is no need to wrap date formatting into a `Promise`, so let's start making requests to *real* servers.

Adding sunrise and sunset times

We are going to improve 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 another location if you want. According to TravelMath.com[24] Sydney's latitude is roughly -33.8830 and the longitude is 151.2167.

A great service called "Sunrise-Sunset"[25] 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:

```
{"results":{"sunrise":"2016-07-11T20:58:47+00:00","sunset":"2016-07-12T07:02:54+00:0\
0","solar_noon":"2016-07-12T02:00:51+00:00","day_length":36247,"civil_twilight_begin\
":"2016-07-11T20:31:34+00:00","civil_twilight_end":"2016-07-12T07:30:08+00:00","naut\
ical_twilight_begin":"2016-07-11T20:00:43+00:00","nautical_twilight_end":"2016-07-12\
T08:00:58+00:00","astronomical_twilight_begin":"2016-07-11T19:30:34+00:00","astronom\
ical_twilight_end":"2016-07-12T08:31:08+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[26]. This is how the formatted result is supposed to look:

[24] http://www.travelmath.com/cities/Sydney,+Australia
[25] http://sunrise-sunset.org/
[26] http://codebeautify.org/jsonviewer

```
{
    "results": {
        "sunrise": "2016-07-11T20:58:47+00:00",
        "sunset": "2016-07-12T07:02:54+00:00",
        "solar_noon": "2016-07-12T02:00:51+00:00",
        "day_length": 36247,
        "civil_twilight_begin": "2016-07-11T20:31:34+00:00",
        "civil_twilight_end": "2016-07-12T07:30:08+00:00",
        "nautical_twilight_begin": "2016-07-11T20:00:43+00:00",
        "nautical_twilight_end": "2016-07-12T08:00:58+00:00",
        "astronomical_twilight_begin": "2016-07-11T19:30:34+00:00",
        "astronomical_twilight_end": "2016-07-12T08:31:08+00:00"
    },
    "status": "OK"
}
```

In order to obtain required information programmatically, we need two things - an HTTP client and a JSON Parser. Vert.x provides both, but since we're using Kotlin, we're going to utilize two Kotlin libraries - Fuel[27] and Kotson[28] - to solve both tasks. Let's add these libraries as dependencies:

```
buildscript {
  ext.fuel_version = '1.15.0'
  ext.kotson_version = '2.5.0'
  // ...
}
dependencies {
  compile "com.github.kittinunf.fuel:fuel:$fuel_version"
  compile "com.github.salomonbrys.kotson:kotson:$kotson_version"
  // ...
}
```

First, we need to send a request to Sunrise-Sunset and get back a string:

[27] https://github.com/kittinunf/Fuel
[28] https://github.com/SalomonBrys/Kotson

```
val timesP = task {
  val url = "http://api.sunrise-sunset.org/json?" +
    "lat=-33.8830&lng=151.2167&formatted=0"
  val (request, response) = url.httpGet().responseString()
  val jsonStr = String(response.data, Charset.forName("UTF-8"))
  // ...
}
```

Here we're using destructuring to extract necessary fields from a response string. Also, since we're not going to need the `request` field, we can utilize a new (Kotlin 1.1) feature and replace the unused reference with the underscore:

```
val (_, response) = url.httpGet().responseString()
```

Then we need to use Gson's `JsonParser` to extract required values and put them in a `Pair`:

```
val timesP = task {
  // ...
  val json = JsonParser().parse(jsonStr).obj
  val sunrise = json["results"]["sunrise"].string
  val sunset = json["results"]["sunset"].string
  Pair(sunrise, sunset)
}
```

To keep code more maintainable, it's better to create a data class for storing both values:

```
data class SunInfo(val sunrise: String, val sunset: String)

val sunInfoP = task {
  // ...
  SunInfo(sunrise, sunset)
}
```

Once the `Promise` is resolved, we need to pass both values to the template engine by putting them into the `RoutingContext`:

```
sunInfoP.success { sunInfo ->
  val params = JsonObject(mapOf(
    "sunrise" to sunInfo.sunrise,
    "sunset" to sunInfo.sunset
  ))
  // ...
}
```

Finally, we need to change the template so that it will be able to show sunset and sunrise information:

```
<h1>Hello Thymeleaf!</h1>
<div>Sunrise time: <span th:text="${sunrise}">01/01/2016 2:00 PM</span></div>
<div>Sunset time: <span th:text="${sunset}">01/01/2016 2:00 PM</span></div>
<!-- omitted -->
```

Try restarting the server and refreshing the page - it should work now. The page will show the following information:

```
Sunrise time: 2016-07-11T20:58:47+00:00
Sunset time: 2016-07-12T07:02:54+00:00
```

The sunrise at 20:58 PM and sunset at 7:02 AM certainly look strange. Besides, the information about the dates seems redundant here, so let's address these issues next.

Using Java 8 Date and Time API

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 complete disaster even by its creators. The good news is that Java 8 introduced a new Date and Time API written from scratch and inspired by a popular date manipulation library JodaTime[29]. Since this API is part of standard Java, we don't need to add anything to the list of dependencies.

Strings that we receive from Sunrise-Sunset are formatted using the ISO 8601 standard[30]. The new Time and Date API 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 processing part is shown below:

[29] http://www.joda.org/
[30] https://en.wikipedia.org/wiki/ISO_8601

```
val sunInfoP = task {
  // ...
  val sunriseTime = ZonedDateTime.parse(sunrise)
  val sunsetTime = ZonedDateTime.parse(sunset)
  val formatter = DateTimeFormatter.ofPattern("HH:mm:ss").
      withZone(ZoneId.of("Australia/Sydney"))
  SunInfo(sunriseTime.format(formatter), sunsetTime.format(formatter))
}
```

You can try refreshing the page and check that now the information is presented in the following way:

```
Sunrise time: 06:58:47
Sunset time: 17:02:54
```

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[31] 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[32] and then use a generated application identifier.

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

[31] http://openweathermap.org/
[32] http://openweathermap.org/appid

```
{
  "main": {
    "temp": 12.18,
    "pressure": 1013,
    "humidity": 54,
    "temp_min": 11.67,
    "temp_max": 12.78
  }
}
```

We already know how to query this information with Fuel and extract the temperature with Kotson, so let's do it also wrapping the whole processing in a `Promise`:

```
val temperatureP = task {
  val url = "http://api.openweathermap.org/data/2.5/" +
      "weather?lat=-33.8830&lon=151.2167&units=metric"
  val (_, response) = url.httpGet().responseString()
  val jsonStr = String(response.data, Charset.forName("UTF-8"))
  val json = JsonParser().parse(jsonStr).obj
  json["main"]["temp"].double
}
```

Let's also create a new data class for storing both `SunInfo` and weather:

```
data class SunWeatherInfo(val sunInfo: SunInfo, val temperature: Double)
```

As you probably remember from the section about functional programming, there are many ways to combine two `Promises`. For instance, we can do it by using `map` and `bind`:

```
val sunWeatherInfoP = sunInfoP.bind { sunInfo ->
  temperatureP.map { temp -> SunWeatherInfo(sunInfo, temp) }
}
```

Finally, we need to pass resolved values to the template:

```
sunWeatherInfoP.success { info ->
  val params = JsonObject(mapOf(
    "sunrise" to info.sunInfo.sunrise,
    "sunset" to info.sunInfo.sunset,
    "temperature" to info.temperature
  ))
  // ...
}
```

And make sure that the template takes into account new information:

```
<body>
<!-- omitted -->
<div>Current temperature: <span th:text="${temperature}">0</span></div>
</body>
```

If you try refreshing the page after restarting the server, you will see that information about current temperature is showing below sunrise and sunset times:

```
Sunrise time: 06:58:47
Sunset time: 17:02:54
Current temperature: 11.93
```

Excellent!

Moving logic into services

If you take a look at our `start` 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 shouldn't be there. Besides, what if we want to use this logic somewhere else? It's much better to put this logic into *services*, which handlers can use to obtain needed data.

We can start refactoring the `start` 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:

```
package services

class WeatherService {
  fun getTemperature(lat: Double, lon: Double):
    Promise<Double, Exception> = task {
    // ...
    json["main"]["temp"].double
  }
}
```

Using the same approach, we can also take logic related to sunrise and sunset times from the `start` 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:

```
package services

class SunService {
  fun getSunInfo(lat: Double, lon: Double):
    Promise<SunInfo, Exception> = task {
    // ...
    SunInfo(formatter.print(sunriseTime),
        formatter.print(sunsetTime))
  }
}
```

In order to use the services, we must have references to their instances in the `MainVerticle` class. Let's simply instantiate both services and make them regular constants:

```
class MainVerticle : AbstractVerticle() {
  override fun start(startFuture: Future<Void>?) {
    val weatherService = WeatherService()
    val sunService = SunService()
    // ...
  }
}
```

Finally, the handler for `/home`, which now can delegate most of its logic to the services, becomes quite simple:

```
router.get("/home").handler { ctx ->
  val lat = -33.8830
  val lon = 151.2167
  val sunInfoP = sunService.getSunInfo(lat, lon)
  val temperatureP = weatherService.getTemperature(lat, lon)
  // ...
}
```

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

Serving static assets

When it comes to Web development, usually there are several types of assets that need to be sent back to clients: script files, fonts, images, style sheets and so on. No surprisingly, Vert.x provides a special kind of handler called `StaticHandler` that does exactly that. Adding it is very easy:

```
val staticHandler = StaticHandler.create().setWebRoot("public").
  setCachingEnabled(false)
router.route("/public/*").handler(staticHandler)
```

Here we're simply specifying the webroot directory and instructing Vert.x to serve all requests that start with `/public/` with the `StaticHandler`. We're also disabling caching, which will later allow us to change frontend assets without restarting the server.

To ensure that it actually works let's create a file called `styles.css` and put it into the `public/compiled` directory:

```
body {
    color: #777;
}
```

Since "black" is considered a very unnatural color, making all text grey seems like a good idea. Finally, we need to reference `styles.css` from the template:

```
<head>
    <!-- omitted -->
    <link rel="stylesheet" href="/public/compiled/styles.css" media="screen">
</head>
```

After restarting the server and refreshing the page, you may notice that text became grayish. This means that our style was correctly loaded by the browser.

Frontend integration

Let's take a break from the backend development and see how we can integrate Vert.x with modern frontend tools. After all, using Thymeleaf templates to make a page dynamic 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 Gradle 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[33] 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

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, simply type:

[33] 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 v8.9.4 while working on this book, so you can do the same. Simply type:

```
$ nvm install v8.9.4
```

`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 v8.9.4
```

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 v8.9.4 > /dev/null
```

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

```
$ npm --version
6.3.0
$ node --version
v8.9.4
```

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 the 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 the command line, you will need to either install it globally (which is not recommended) or add a script to the scripts section:

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

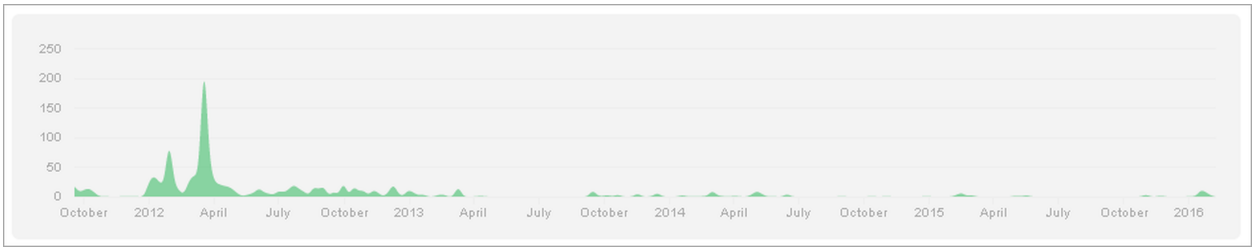
After doing this, you will be able to start Webpack by typing the following in the command line:

```
$ npm run watch
```

GruntJS and Gulp

Two popular build tools for JavaScript are GruntJS 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.

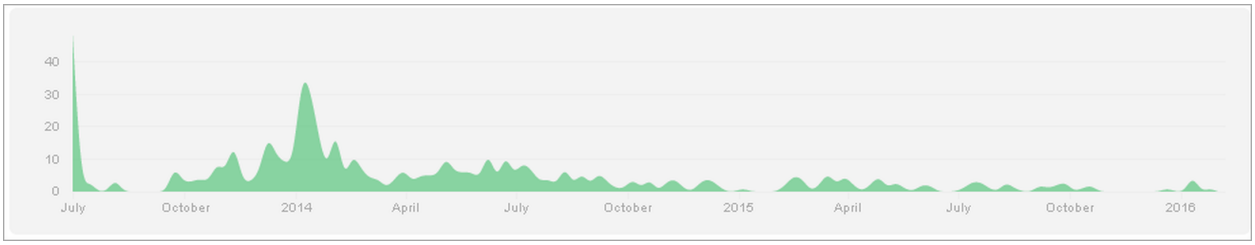


GruntJS commit chart

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 stylesheet 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 the 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 your 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 use either Browserify[34] with Gulp or Webpack[35]. It is possible to use Webpack as a Gulp plugin, but it's not necessary because 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.

[34] http://browserify.org/
[35] 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 current implementation.

One of the most popular projects that allow you to use new features of the next version JavaScript today is Babel[36]. 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 and specify them in a special file called `.babelrc`:

```
{
  "presets": [
    ["env", {
      "targets": {
        "browsers": ["last 2 versions"]
      }
    }],
    "stage-0",
    "react"
  ]
}
```

Here we're telling Babel to target the last 2 versions of popular browsers and also including `react` and `stage-0` presets.

Explaining React

React[37] 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[38] 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,

[36] https://babeljs.io/
[37] http://facebook.github.io/react/
[38] https://atom.io/packages/react

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[39] comes with a built-in service called `$http`, which allows to query the backend server 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!');
  });
```

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

React doesn't have have the `$http` service, but there are many HTTP client libraries available for JavaScript. We're going to use Axios[40] because it's widely used and comes with the Promise API. With Axios, the above example can be rewritten the following way:

```
axios.get('/api/data')
  .then(function (response) {
    console.log('Data received: ', response);
  })
  .catch(function (error) {
    console.log('Error occurred!');
  });
```

As you will see, the lack of built-in service like `$http` can be easily overcome by third-party tools.

Separating the frontend

Right now, we have the following directory structure:

[39] https://angularjs.org/
[40] https://github.com/mzabriskie/axios

```
$ tree -L 1
.
.
├── bin
├── build
├── build.gradle
├── gradle
├── gradlew
├── gradlew.bat
├── package.json
├── public
├── settings.gradle
└── src
```

Most of these files and directories are part of the backend. However, the `public` directory contains static frontend assets, which Vert.x serves using the `StaticHandler`. Basically, it means that in our setup, Vert.x *sees* already compiled, bundled and possibly minified JavaScript files as static resources. When the browser encounters a script tag like this one:

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

it simply sends a request to the server, i.e Vert.x. Since the path starts with `/public/`, the request is handled by the `StaticHandler`, and it simply sends back a file from the public directory. Vert.x *doesn't know* how this file was created - it could be typed by a developer in a text editor by hand 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`.

Initializing the frontend project

Even though we are going to store frontend sources in the `ui` directory, we can still initialize a new 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-react babel-preset-stage-0 exports-load\
er imports-loader webpack webpack-cli -D
```

After development dependencies are installed, we need to install React and Axios as application dependencies:

```
$ npm i react react-dom axios -S
```

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

```
{
  // ...
  "scripts": {
    "watch": "webpack --mode development --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:

```
const webpack = require('webpack');
const path = require('path');

module.exports = {
  entry: './ui/entry.js',
  output: {path: path.resolve(__dirname, 'public/compiled'), filename: 'bundle.js'},
  module: {
    rules: [
      {
        test: /\.jsx?$/,
        include: /ui/,
        use: {
          loader: 'babel-loader'
        }
      }
    ]
  }
}
```

First, we're importing Webpack definitions using the 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: {
  rules: [
    {
      test: /\.jsx?$/,
      include: /ui/,
      use: {
        loader: 'babel-loader'
      }
    }
  ]
}
 //...
```

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. During the build the `babel-loader` will read the instructions from `.babelrc` created earlier and act accordingly.

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';

class SunWeatherComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      sunrise: undefined,
      sunset: undefined,
      temperature: undefined
    };
  }
  render = () => {
```

```
13      return <div>
14        <div>Sunrise time: {this.state.sunrise}</div>
15        <div>Sunset time: {this.state.sunset}</div>
16        <div>Current temperature: {this.state.temperature}</div>
17      </div>
18    }
19  }
20
21  export default SunWeatherComponent;
```

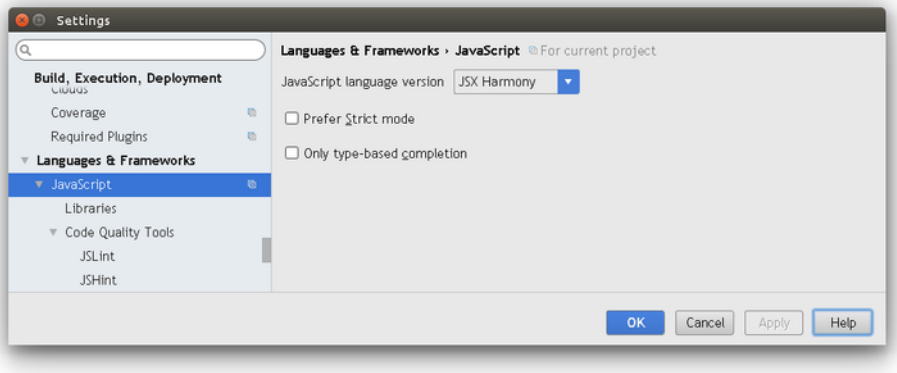
In `constructor`, we're defining the initial value of the `this.state` field. This field is used in the `render` function to render the component. The React template is partially copied from the `index.html`, in which we can replace the informational part with an empty div:

```
1  <html lang="en">
2  <!-- omitted -->
3  <body>
4      <h1>Hello Thymeleaf!</h1>
5      <div id="reactView"></div>
6  </body>
7  <script src="/public/compiled/bundle.js"></script>
8  </html>
```

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';
```

Note that if you're using IntelliJ, it will start complaining about the version of JavaScript. This is happening because by default it uses JavaScript version 5, which doesn't support `imports`. Just go to "Settings" and choose "JSX Harmony" as the version of JavaScript:



JavaScript language settings

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 our Thymeleaf template.

Try running Webpack via npm:

```
$ npm run watch

Hash: 27063854997a9e9b2159
Version: webpack 4.20.2
Time: 1242ms
    Asset    Size  Chunks             Chunk Names
bundle.js  726 kB       0  [emitted]  main
    + 174 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. We already have a data class containing all information:

```
data class SunWeatherInfo(val sunInfo: SunInfo, val temperature: Double)
```

If we want to be able to send objects of this class via JSON, we need to be able to serialize them. The easiest way to do it is to use Jackson Kotlin module[41]. Let's add it to the list of dependencies:

[41] https://github.com/FasterXML/jackson-module-kotlin

```
buildscript {
  ext.jackson_version = '2.9.4.1'
  // ...
}
dependencies {
  compile "com.fasterxml.jackson.module:jackson-module-kotlin:$jackson_version"
  // ...
}
```

Then we need to initialize an `ObjectMapper`:

```
val jsonMapper = jacksonObjectMapper()
```

After the `ObjectMapper` is initialized, we can use it to serialize our data object and send it back to a client:

```
router.get("/api/data").handler { ctx ->
  val lat = -33.8830
  val lon = 151.2167
  val sunInfoP = sunService.getSunInfo(lat, lon)
  val temperatureP = weatherService.getTemperature(lat, lon)
  val sunWeatherInfoP = sunInfoP.bind { sunInfo ->
    temperatureP.map { temp -> SunWeatherInfo(sunInfo, temp) }
  }
  sunWeatherInfoP.success { info ->
    val json = jsonMapper.writeValueAsString(info)
    val response = ctx.response()
    response.end(json)
  }
}
```

Since we're sending data via JSON, we don't need to pass it to the template anymore. The handler returning the page could be simplified as follows:

```
router.get("/home").handler { ctx ->
  templateEngine.render(ctx, "public/templates/", "index.html") { buf ->
    if (buf.failed()) {
      logger.error("Template rendering failed", buf.cause())
    } else {
      val response = ctx.response()
      response.end(buf.result())
    }
  }
}
```

Finally, we can load the data in our React component using Axios:

```
import axios from 'axios';

class SunWeatherComponent extends React.Component {
  componentDidMount = () => {
    axios.get('/api/data').then((response) => {
      const json = response.data;
      this.setState({
        sunrise: json.sunInfo.sunrise,
        sunset: json.sunInfo.sunset,
        temperature: json.temperature
      });
    })
  }
  // ...
}
```

We're using the `componentDidMount` callback function that is invoked after the initial rendering happened. We're sending a request to our backend, and after we receive a response, we're changing the state of the component. React will detect 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 without proper styling, so let's improve this part. We are going to use Sass[42] - 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

[42] http://sass-lang.com/

Sass, we will use the ConciseCSS[43] 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, whereas SASS uses indents. We are going to stick to SCSS because this style is more popular among developers.

First, we need to create a new directory in the `ui` folder:

```
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
```

After that, we need to create a new file called `style.scss`. This file will work as an aggregator by simply referencing 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:

```
1 import './scripts/main.js';
2 import './styles/style.scss';
```

This may look strange because we're referencing a SCSS file from a JavaScript file, but since we've specified the `entry.js` file as the entry point 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:

[43] http://concisecss.com/

```
$ npm i sass-loader node-sass css-loader style-loader -D
```

If you look at the Sass-loader GitHub page[44], you will see that it is recommended to apply three loaders - `style`, `css`, `sass` - sequentially. Let's follow their advice and add a new `loaders` entry to `webpack.config.js`:

```
1  const webpack = require('webpack');
2  const path = require('path');
3
4  module.exports = {
5    // ...
6    module: {
7      rules: [
8        // ...
9        {
10         test: /\.scss$/,
11         use: ['style-loader', 'css-loader', 'sass-loader']
12       }
13     ]
14   },
15   // ...
16 }
```

Note that loaders are applied from right to left. So, first Webpack will use the *sass-loader* to transform Sass to CSS, then it will use the *css-loader* to transform CSS to style. Finally, it will use the *style-loader* to apply the styles to your 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 and you should see our page uses different fonts now, which means that ConciseCSS styling 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 the *MiniCssExtractPlugin*. Let's start with installing this plugin via npm:

```
$ npm i mini-css-extract-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`:

[44] http://github.com/jtangelder/sass-loader

```
const webpack = require('webpack');
const path = require('path');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  // ...
  module: {
    rules: [
      // ...
      {
        test: /\.scss$/,
        use: [
          MiniCssExtractPlugin.loader,
          'css-loader',
          'sass-loader'
        ]
      }
    ]
  },
  plugins: [
    new MiniCssExtractPlugin({ filename: "styles.css" })
  ]
};
```

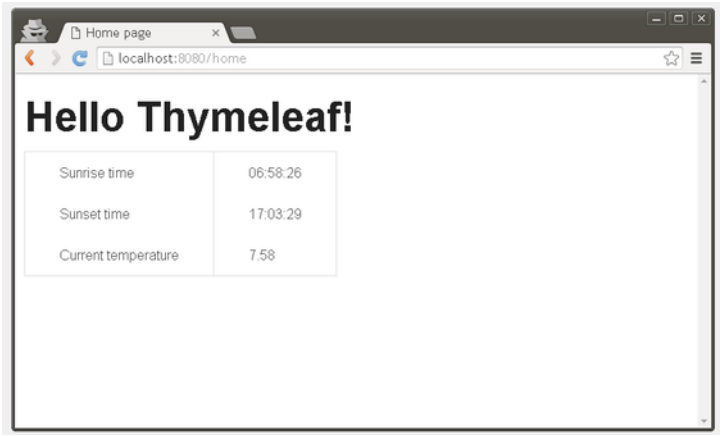
Here we're importing the `MiniCssExtractPlugin` and specifying the output file that will contain styling information. In the loaders section, we are replacing the `style-loader` with a loader from `MiniCssExtractPlugin`, but apart from that syntax is the same.

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 our template. We already did that in the previous section while we were playing with the `StaticHandler`, so we are fine here.

Let's also convert our `div` tags into a table. This will demonstrate that our styling is indeed working as ConciseCSS provides styles for the `table` tag which are noticeably different from default:

```
class SunWeatherComponent extends React.Component {
  // ...
  render = () => {
    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>
      </tbody>
    </table>
  }
}
```

After this change, the home page should look like following:



Presenting data in a table

And as I already pointed out previously, the font has also 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 implement a simple authentication system. Vert.x provides several handlers that will be very useful for us, but before diving into into this, we need to learn about configuring our application and integrating it with a database.

Vert.x configuration

Right now, all parameters our application needs are *hardcoded* in source files. This is obviously a bad practice and may present some challenges during deployment. Ideally, we should be able to change the port number, so it's better to extract it as a configuration parameter. Moreover, as you probably remember, during initialization of the `StaticHandler`, we disabled caching. This may be good for development, but it's absolutely unacceptable for production.

When it comes to configuration, Vert.x follows the current trend and allows to store application settings in JSON. Therefore, let's create a file called `development.json` in the `conf` directory with the following content:

```
{
  "server" : {
    "port": 8080,
    "caching": false
  }
}
```

The fact that we called this file `development.json` suggests that we may use a different one for production.

Then, in order to simplify working with configuration, let's create a simple data class containing corresponding fields:

```
data class ServerConfig(val port: Int, val caching: Boolean)
```

During the startup, Jackson's `ObjectMapper` will easily read this part of configuration and initialize `ServerConfig` with real values:

```
val jsonMapper = jacksonObjectMapper()
val serverConfig = jsonMapper.readValue(config().
        getJsonObject("server").encode(), ServerConfig::class.java)
val serverPort = serverConfig.port
val enableCaching = serverConfig.caching
```

After configuration values are read, we can use them to configure the app:

```
val staticHandler = StaticHandler.create().
        setWebRoot("public").setCachingEnabled(enableCaching)
server.requestHandler{ router.accept(it) }.listen(serverPort) { handler ->
  // ...
}
```

To tell Vert.x that `development.json` is indeed our configuration, we need to use the `-conf` command line option. If you're running app in IntelliJ, you need to specify the following as "Program arguments":

```
run verticles.MainVerticle -conf conf/development.json
```

In `build.gradle`, the change is similar:

```
// ...
def configurationFile = 'conf/development.json'
run {
  args = ['run', mainVerticleName, "--redeploy=$watchForChange",
          "--launcher-class=$mainClassName", "--on-redeploy=$doOnChange",
          "-conf $configurationFile"]
}
```

Since we haven't changed anything in the logic, the application should work as usual.

Installing PostgreSQL

Even though Vert.x authentication handlers will come in very handy, we still need to store user information somewhere. This brings us to the topic of databases.

Relational databases are probably the most popular tools when it comes to storing data. They are fast, reliable and supported by most programming languages and frameworks. Among existing open-source databases, PostgreSQL stands out as probably the most advanced.

If you are using Ubuntu 16.X, you can find PostgreSQL 9.5 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 9.5 from there. Relevant instructions can be found in the Ubuntu Downloads section of the official PostgreSQL website[45].

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 query results:

```
$ sudo apt-get install postgresql-9.5 postgresql-contrib-9.5 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 to perform the task:

```
$ sudo -u postgres psql --command "CREATE USER kotlinuser WITH SUPERUSER PASSWORD 'k\
otlinpass';"
$ sudo -u postgres createdb -O kotlinuser kotlindb
```

Here we're creating a new user called `kotlinuser` and a database called `kotlindb`.

Pooling connections with HikariCP

Since all database-related tools in Kotlin use JDBC (Java Database Connectivity), we need to add PostgreSQL JDBC drivers to the list of dependencies. We also need a library to perform connection pooling, so let's also add HikariCP[46] to the mix:

```
1  buildscript {
2    ext.hikari_version = '3.2.0'
3    ext.pgjdbc_version = '42.1.1'
4    // ...
5  }
6  dependencies {
7    compile "com.zaxxer:HikariCP:$hikari_version"
8    compile "org.postgresql:postgresql:$pgjdbc_version"
9    // ...
10 }
```

You can always find actual versions of drivers on MVN Repository[47].

HikariCP needs to know connection parameters, so let's add them to `development.json`:

[45] http://www.postgresql.org/download/linux/ubuntu/
[46] http://brettwooldridge.github.io/HikariCP/
[47] http://mvnrepository.com/artifact/org.postgresql/postgresql

```
{
  "dataSource": {
    "user": "kotlinuser",
    "password": "kotlinpass",
    "jdbcUrl": "jdbc:postgresql://localhost:5432/kotlindb"
  }
}
```

Again, for simplifying configuration we can create a data class:

```
data class DataSourceConfig(val user: String,
      val password: String, val jdbcUrl: String)
```

And then make Jackson parse it and initialize a proper object:

```
val dataSourceConfig = jsonMapper.readValue(config().
      getJsonObject("dataSource").encode(), DataSourceConfig::class.java)
```

Once we have connection parameters, we can initialize a data source. The only challenge is to store a reference to the data source somewhere and the simplest way to do it is to make `HikariDataSource` a nullable field of `MainVerticle`:

```
class MainVerticle : AbstractVerticle() {
  private var maybeDataSource: HikariDataSource? = null

  private fun initDataSource(config: DataSourceConfig): HikariDataSource {
    val hikariDS = HikariDataSource()
    hikariDS.username = config.user
    hikariDS.password = config.password
    hikariDS.jdbcUrl = config.jdbcUrl
    maybeDataSource = hikariDS
    return hikariDS
  }

  override fun stop(stopFuture: Future<Void>?) {
    maybeDataSource?.close()
  }
  // ...
}
```

Note that we're also returning a non-nullable `HikariDataSource` value from the `initDataSource` method.

Hashing passwords with BCrypt

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[48]) 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 hashes of passwords, 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.
- We store the hash in the database against the user identifier
- Next time the user tries to login, they supply their original password once again. We 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 the hash 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[49]. It's written in Java, so we can use it easily in Kotlin. Let's add the library's coordinates[50] to the list of dependencies:

[48] https://en.wikipedia.org/wiki/Universally_unique_identifier
[49] http://www.mindrot.org/projects/jBCrypt/
[50] http://mvnrepository.com/artifact/de.svenkubiak/jBCrypt

```
buildscript {
  ext.jbcrypt_version = '0.4.1'
  // ...
}
dependencies {
  compile "de.svenkubiak:jBCrypt:$jbcrypt_version"
  // ...
}
```

Before staring to write actual code that uses a new library, it is always good to try it in the Kotlin interpreter. We already used the interpreter with `kotlin-reflect` library. Similarly, nothing prevents us from adding `jBCrypt` to its classpath. Simply download the JAR file from the Maven central[51] and reference it via the command line:

```
$ kotlinc -cp ~/Downloads/jBCrypt-0.4.1.jar
Welcome to Kotlin version 1.3.31 (JRE 1.8.0_144-jdk_2017_08_24_20_46-b00)
Type :help for help, :quit for quit
>>>
```

Once we have jBCrypt on the classpath, we can use it:

```
>>> import org.mindrot.jbcrypt.BCrypt
>>> val hash = BCrypt.hashpw("password123", BCrypt.gensalt())
>>> hash
$2a$10$zaJPLukPodM.arBNMVONWenOb7SFACvAPm2eq2shRNr/UltUnv0pa
>>>
```

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:

```
>>> BCrypt.checkpw("password123", hash)
true
>>> BCrypt.checkpw("password321", hash)
false
```

Not surprisingly, supplying the `checkpw` with incorrect password makes this function return `false`.

[51] http://central.maven.org/maven2/de/svenkubiak/jBCrypt/0.4.1/

Migrating with Flyway

It is 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 a library called Flyway[52]. 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.

First, we need to add Flyway to the list of dependencies:

```
buildscript {
  ext.flyway_version = '4.2.0'
  // ...
}
dependencies {
  compile "org.flywaydb:flyway-core:$flyway_version"
  // ...
}
```

In order for Flyway evolutions to work, we need to write them as SQL and put our SQL files in a specific location. This specific location in our case will be `src/main/resources/db/migration`. By putting a file in the `src/main/resources` directory, we're making it part of the application classpath.

So, let's create a file called `V1__Users.sql` with the following content in `db/migration`:

```
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(), 'ktest',
  '$2a$10$zaJPLukPodM.arBNMVONWenOb7SFACvAPm2eq2shRNr/UltUnv0pa');
```

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 "ktest" and password "password123". Since jBCrypt appends salt to the hash, we don't need a separate column for storing salt.

Let's create a dedicated service that will be responsible for performing migrations:

[52] https://flywaydb.org/

```
class MigrationService(dataSource: DataSource?) {
  private val flyway: Flyway = Flyway()
  init {
    flyway.dataSource = dataSource
  }
  fun migrate(): Either<Throwable, Int> = eitherTry {
    flyway.migrate()
  }
}
```

Since primary constructors in Kotlin are not allowed to contain any logic, we're initializing `Flyway` in an `init` block. The `migrate` function attempts to perform a migration and returns `Either`. We should invoke it after we initialized the data source but before the rest of the application starts:

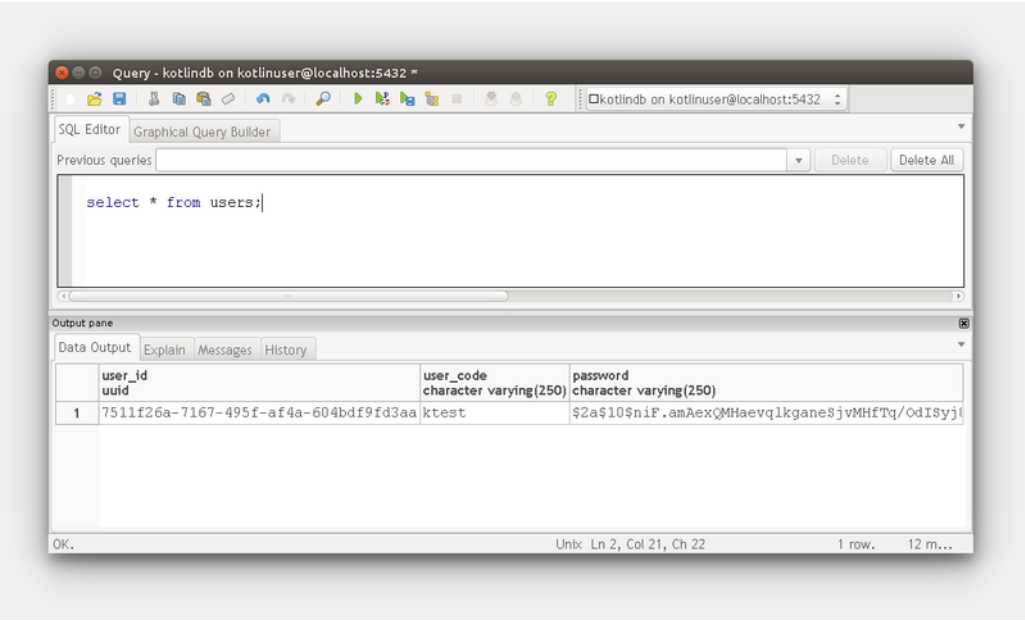
```
val dataSource = initDataSource(dataSourceConfig)
val migrationService = MigrationService(dataSource)
val migrationResult = migrationService.migrate()
migrationResult.fold({ exc ->
  logger.fatal("Exception occurred while performing migration", exc)
  vertx.close()
},{ _ ->
  logger.debug("Migration successful or not needed")
})
```

There is no point in starting the server if the migration has failed, so we're exiting the app. If everything is good, though, we're simply logging the debug message and moving on.

If you rerun the application, Flyway will apply the SQL script, and a new table will be created. To check this fact, you can try the following command in the console:

```
$ /usr/bin/sudo -u postgres psql -d kotlindb --command "SELECT user_code,password FR\
OM users;" | more
 user_code |                          password
-----------+------------------------------------------------------------
 stest     | $2a$10$zaJPLukPodM.arBNMVONWenOb7SFACvAPm2eq2shRNr/UltUnv0pa
(1 row)
```

Alternatively, you can use pgAdmin3 and see similar results:



pgAdmin showing the Users table

Implementing authentication logic

Now that we have a test user in the database, let's outline how we are going to proceed with implementing the authentication system. Usually, there are several ways to authenticate a user, but here we're going to use *form-based authentication*. In terms of Vert.x, it means that we will need to instantiate several built-in handlers, but most importantly we will need an `AuthProvider`.

The `AuthProvider` is an interface that has only one method - `authenticate`. This method will receive user credentials, and our task will be to perform password checking and send results to the sender. Since our password is stored in the database, we need the data source reference there. We can also make use of Jackson `ObjectMapper` because Vert.x will pass user credentials as JSON.

Let's start with adding a new data class for extracting JSON values:

```
1  data class AuthInfo(val username: String, val password: String)
```

And then start writing our implementation of `AuthProvider`:

```
class DatabaseAuthProvider(val dataSource: HikariDataSource,
    val jsonMapper: ObjectMapper) : AuthProvider {
  override fun authenticate(authInfoJson: JsonObject?,
      resultHandler: Handler<AsyncResult<User>>?) {
    val authInfo = jsonMapper.readValue(authInfoJson?.encode(),
      AuthInfo::class.java)
    // Query the database, check password and return the result
  }
}
```

Note that Vert.x expects us to return an instance of `User`. The recommended way to implement this interface is to extend the `AbstractUser` class:

```
class DatabaseUser(val id: UUID, val username: String,
    val passwordHash: String) : AbstractUser() {
  override fun doIsPermitted(permission: String?,
      resultHandler: Handler<AsyncResult<Boolean>>?) {
    val result = CompositeFuture.factory.succeededFuture(true)
    resultHandler?.handle(result)
  }
  override fun setAuthProvider(authProvider: AuthProvider?) {
  }
  override fun principal(): JsonObject {
    return JsonObject().put("username", username)
  }
}
```

The important part here is a set of properties (`id`, `username`, `passwordHash`) which correspond to a single database row. The `doIsPermitted` method is designed to check whether this particular user has a given permission. Since we're only interested in *authentication* (not *authorization*), we can make this method always return `true`. The `principal` method allows us to return a JSON object with information about the current user and the `setAuthProvider` is used for reattaching users to providers, which we don't need.

For querying databases, Vert.x provides an out of the box Java JDBC client. We, however, are going to use a Kotlin-based library called KotliQuery[53]. It may look quite minimalistic, but it has everything we need, so let's add it to dependencies:

[53] https://github.com/seratch/kotliquery

```
buildscript {
  ext.kotliquery_version = '1.2.1'
}
dependencies {
  compile "com.github.seratch:kotliquery:$kotliquery_version"
}
```

When working with KotliQuery, it is often convenient to create a helper method for converting a `kotliquery.Row` to our domain object. The best place for this method in our case is probably the `companion object` of `DatabaseUser`:

```
class DatabaseUser(val id: UUID, val username: String,
    val passwordHash: String) : AbstractUser() {
  companion object {
    fun fromDb(row: Row): DatabaseUser {
      return DatabaseUser(UUID.fromString(row.string("user_id")),
          row.string("user_code"), row.string("password"))
    }
  }
  // ...
}
```

After this, we will be able to use `fromDb` as a static factory method.

Now, let's get back to our `authenticate` method. We already have the `username`, so we can query the database and return a `DatabaseUser`. Working with databases is usually considered a dangerous business, so it's better to wrap our work in `eitherTry`:

```
override fun authenticate(authInfoJson: JsonObject?,
      resultHandler: Handler<AsyncResult<User>>?) {
    val authInfo = jsonMapper.readValue(authInfoJson?.encode(),
      AuthInfo::class.java)
    val userT = eitherTry { using(sessionOf(dataSource)) { session ->
      val query = queryOf("select * from users where user_code = ?",
        authInfo.username)
      val maybeUser = query.map { DatabaseUser.fromDb(it) }.asSingle.
          runWithSession(session)
      maybeUser ?: throw Exception("User ${authInfo.username} not found!")
    }}
    // ...
  }
```

There is quite a bit of code here and it's rather dense, so let me explain what we're doing here, step by step. The interesting part begins with the following:

```
// import kotliquery.using

using(sessionOf(dataSource)) { session ->
  // ...
}
```

Here we're creating a session from the `datasource` and the `session` reference becomes available in a lambda block. Inside this block, we're preparing a query and executing it. Interestingly, in KotliQuery we need to specify whether we're interested in only one result (`asSingle`) or several (`asList`). Of course, sometimes query doesn't return any rows, and in this case we're throwing an exception. All exceptions are absorbed by the `eitherTry` block, so we're fine.

Regardless of what `userT` holds (remember `Either` may contain *either* an exception or result), we need to make sure we handle all possibilities:

```
override fun authenticate(authInfoJson: JsonObject?,
    resultHandler: Handler<AsyncResult<User>>?) {
  // ...
  userT.fold({ exc ->
    val result = CompositeFuture.factory.failedFuture<User>(exc)
    resultHandler?.handle(result)
  }, { dbUser ->
    val isValid = BCrypt.checkpw(authInfo.password, dbUser.passwordHash)
    val result = if (isValid) {
      CompositeFuture.factory.succeededFuture(dbUser as User)
    } else {
      CompositeFuture.factory.failureFuture("Password is not valid!")
    }
    resultHandler?.handle(result)
  })
}
```

Here we're using the `CompositeFuture` methods to generate instances of `AsyncResult`. If the password supplied by the user matches the one from the database, we can put a `User` object into the `Future` and send it back. In case of a failure, we're wrapping an exception.

Back in `MainVerticle`, we need to instantiate our `AuthProvider` and assign generic handlers to certain routes:

```
val authProvider = DatabaseAuthProvider(dataSource, jsonMapper)
router.route("/hidden/*").handler(RedirectAuthHandler.create(authProvider))
router.route("/login").handler(BodyHandler.create())
router.route("/login").handler(FormLoginHandler.create(authProvider))
```

We're specifying that every URL that starts with `/hidden` will require a user to be logged in. By default, not logged users trying to view restricted resources will be redirected to the `/login`. After they enter their credentials, their `username` and `password` are sent to the `login` route via `POST` and the `FormLoginHandler` sends the credentials to the `AuthProvider`. The `FormLoginHandler` requires a `BodyHandler`, so we're adding it here.

In addition, several handlers must be assigned to `route()`, so they are applied to all requests:

```
router.route().handler(CookieHandler.create())
val sessionHandler = SessionHandler.create(LocalSessionStore.create(vertx))
sessionHandler.setAuthProvider(authProvider)
router.route().handler(sessionHandler)
```

These two handlers basically maintain a session store and therefore, allow logged users to use the website without having to enter their credentials all the time.

Page templates

Implementing the login page is probably the easiest part of implementing authentication. Let's create a file called `login.html` with the following content:

```
<html lang="en">
<head>
    <title>Login page</title>
    <link rel="stylesheet" href="/public/compiled/styles.css" media="screen">
</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>
```

Since we're using default parameters of the `FormLoginHandler`, we should stick to the default names for the form fields and action url.

Before applying a handler to the `/login` route, we can also create a helper function which takes a `RoutingContext` and template name, renders the page and sends it back to the client:

```
fun renderTemplate(ctx: RoutingContext, template: String,
    params: JsonObject = JsonObject()) {
  templateEngine.render(params, "public/templates/$template") { buf ->
    val response = ctx.response()
    if (buf.failed()) {
      logger.error("Template rendering failed", buf.cause())
      response.setStatusCode(500).end()
    } else {
      response.end(buf.result())
    }
  }
}
```

By extracting rendering logic into a separate function, we can avoid unnecessary code duplication. With the `renderTemplate` method in place, implementing both `/login` and `/home` routes is trivial:

```
router.get("/login").handler { ctx ->
  renderTemplate(ctx,"login.html" ) }
router.get("/home").handler { ctx ->
  renderTemplate(ctx, "index.html") }
```

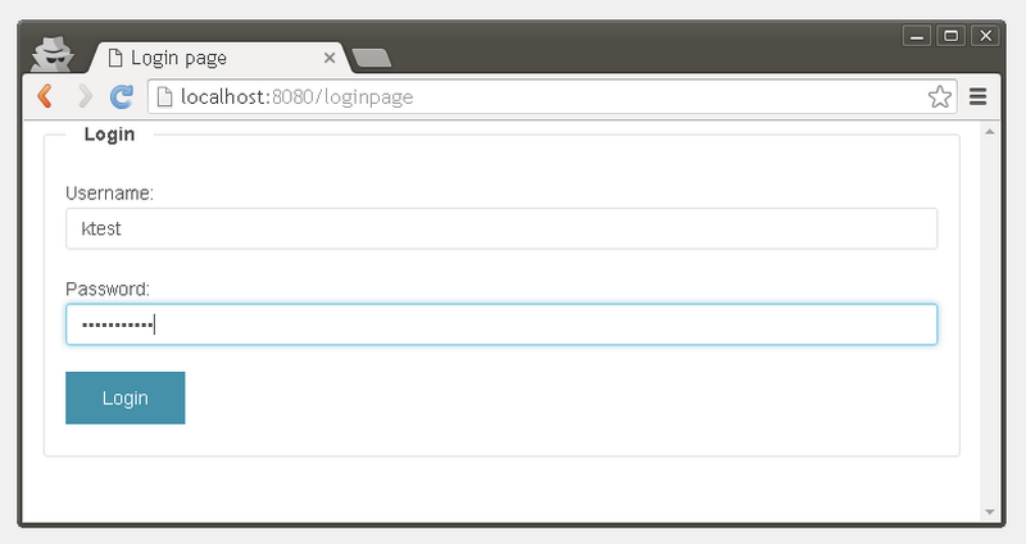
As for the hidden page, let's create its template in `admin.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home page</title>
    <link rel="stylesheet" href="/public/compiled/styles.css" media="screen">
</head>
<body>
<h1>Welcome, <span th:text="${username}">Username</span></h1>
<div>This a hidden page that only logged users can see</div>
</body>
</html>
```

Here we're making use of the principal information available to this route. Obviously, we need to put `username` into the context but other than that, its rendering is a piece of cake:

```
router.get("/hidden/admin").handler { ctx ->
  val params = JsonObject(mapOf(
              "username" to ctx.user().principal().getString("username")
  ))
  renderTemplate(ctx, "admin.html", params)
}
```

If you start the server and try accessing the `/hidden/admin` page, you will be redirected to login:



Login page

After entering user credentials (ktest/password123), you will be redirected to `hidden/admin`, and you will be able to see a personal greeting.

Testing and logging

In this section, we will look at two other aspects of Web development (and programming in general) - logging and testing. While these aspects are often considered less important than, say, implementing business logic, they are absolutely crucial when your application enters the production phase.

Testing with KotlinTest

Automated tests are a great way to quickly check the correctness of code. When it comes to testing in Kotlin, there are several solutions available, but the most popular one is probably KotlinTest[54].

Let's start by adding KotlinTest dependency to `build.gradle`:

```
buildscript {
  // ...
  ext.kotlintest_version = '3.1.10'
}
dependencies {
  // ...
  testCompile "io.kotlintest:kotlintest-runner-junit5:$kotlintest_version"
}

test {
  useJUnitPlatform()
}
```

All tests should be placed in the `src/test/kotlin` directory. Just to illustrate the idea, let's create a simple test and put it in a file called `ApplicationSpec.kt`:

[54] https://github.com/kotlintest/kotlintest

```
import io.kotlintest.shouldBe

class ApplicationSpec : StringSpec() {
  init {
    "DateTimeFormat must return 1970 as the beginning of epoch" {
      val beginning = ZonedDateTime.ofInstant(Instant.ofEpochSecond(0),
          ZoneId.systemDefault())
      val formattedYear = beginning.format(DateTimeFormatter.ofPattern("YYYY"))
      formattedYear shouldBe "1970"
    }
  }
}
```

This is a really silly test, but it shows a number of important points. First, we're extending the `StringSpec` class that provides DSL-like methods useful for writing tests. Then we're writing a description in plain English (this is why it's called `StringSpec`). Finally, we're adding an assertion that verifies a certain condition. In this example, `formattedYear` must contain string "1970", otherwise the test will fail.

We can run tests via the command line using Gradle:

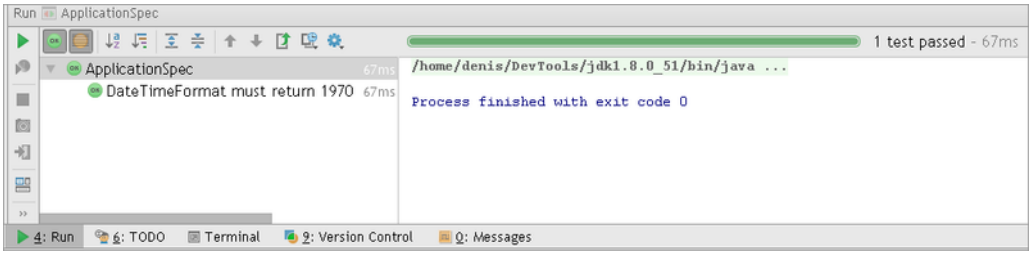
```
$ gradle test
:compileKotlin UP-TO-DATE
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestKotlin
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

BUILD SUCCESSFUL

Total time: 1.588 secs
```

Note that as long as all tests pass, Gradle doesn't print anything.

In addition to using the command line, we can also utilize IntelliJ IDEA features. In order to run our test from IntelliJ, right-click on the `ApplicationSpec` file and choose "Run ApplicationSpec" from the context menu. The green bar means that all is good, whereas the red one signals that something is wrong:



Testing in IntelliJ

Now that we're familiar with the basics, let's write something more useful.

Our `SunService` contains a lot of logic, which makes it a good candidate for testing. However, once we start writing a test, we will immediately see a problem. The thing is, `SunService` uses Fuel as an HTTP client and queries a remote host. This not only slows the test down, but makes it non-deterministic. How can we check that our service works properly if the remote host responds differently every single day?

Fortunately, Fuel was developed with testing in mind, so it is possible to customize the client before use. If you look at the docs, you will see that the `FuelManager` allows to set a custom `Client` that will be used by Fuel instead of the default one. This means that we can create our own `Client` implementation that always returns the same response without querying remote servers.

Since there is only one method in the `Client` interface, its implementation is surprisingly simple:

```
class FuelTestClient(val testResponse: Response) : Client {
  override fun executeRequest(request: Request): Response {
    return testResponse
  }
}
```

Following the same logic, we can create a custom `Response` and pass it to the constructor of `FuelTestClient`:

```
"SunService must retrieve correct sunset and sunrise information" {
  val json = """{
        "results":{
          "sunrise":"2016-04-14T20:18:12+00:00",
          "sunset":"2016-04-15T07:31:52+00:00"
        }
    }"""
  val testResponse = Response(URL("http://localhost/api/data"))
  testResponse.data = json.toByteArray()
  val testClient = FuelTestClient(testResponse)
  FuelManager.instance.client = testClient
  // ...
}
```

Here we're creating a fake JSON data using a raw string. Then we're passing the fake response to a `FuelTestClient` instance and instructing Fuel to use it. Once the preparations are finished, we can use `SunService` as usual:

```
"SunService must retrieve correct sunset and sunrise information" {
  // ...
  val lat = -33.8830
  val lon = 151.2167
  val sunService = SunService()
  val resultP = sunService.getSunInfo(lat, lon)
}
```

As expected, the service returns a `Promise`, which presents a new challenge. How can we test it if we don't know when it is resolved? Again, using asynchronous computations is becoming very common in modern programming, and most testing frameworks provide some way to deal with them. In KotlinTest, we can use the `eventually` method:

```
import io.kotlintest.eventually

class ApplicationSpec : StringSpec(), Eventually {
  init {
    "SunService must retrieve correct sunset and sunrise information" {
      // ...
      eventually(Duration.ofSeconds(5)) {
        val sunInfo = resultP.get()
        sunInfo.sunrise shouldBe "06:18:12"
      }
    }
  }
}
```

The `eventually` method will repeatedly test the code until it either passes, or the timeout of 5 seconds is reached. Inside the lambda block, we can use standard `KotlinTest` matchers against the computed result to verify expected behavior.

Logging in Vert.x

We already used logging to print simple diagnostic messages but didn't give much thought to its configuration. This could be fine for small apps, but once the application passes a certain threshold, this state of affairs becomes unacceptable.

Internally, Vert.x uses a simple but reliable logging framework commonly known as JUL (or Java Util Logging). JUL is a part of the Java standard library, and we can use it without putting any additional

libraries on the classpath or even specifying any additional command line flags. The only thing we need to do is to put a file called `vertx-default-jul-logging.properties` in the `src/resources` directory.

Here is a simple example of this file:

```
handlers=java.util.logging.ConsoleHandler,java.util.logging.FileHandler
java.util.logging.SimpleFormatter.format=%1$tY.%1$tm.%1$td %1$tk:%1$tM:%1$tS [%4$s] \
[%3$s] %5$s %6$s\n
java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter
java.util.logging.ConsoleHandler.level=FINEST
java.util.logging.FileHandler.level=INFO
java.util.logging.FileHandler.formatter=java.util.logging.SimpleFormatter
java.util.logging.FileHandler.pattern=%t/vertx.log
.level=INFO
io.vertx.ext.web.level=INFO
io.vertx.level=INFO
com.hazelcast.level=INFO
io.netty.util.internal.PlatformDependent.level=SEVERE
```

The settings in this file are mostly self-explanatory, but there are a couple of things worth highlighting. First, we're specifying the log format as follows:

```
%1$tY.%1$tm.%1$td %1$tk:%1$tM:%1$tS [%4$s] [%3$s] %5$s %6$s\n
```

In SimpleFormatter[55], all parts of a log record have a specific number, and we can reference them using this number. For example, `%1$` refers to the second argument (indexes are zero-based), which is a `Date` object representing time. Using different suffixes, such as `Y` (year), we can extract certain parts of the date. Other arguments are shown in documentation for `SimpleFormatter` (see above).

The `%t` in the pattern string of `FileHandler` points to the system's temporary directory. On Ubuntu, this directory is `/tmp`, so if needed, you will be able to find your log file there.

The log level for `io.vertx.ext.web.level` is set to `INFO`. If you *really* want to know what Vert.x does behind the scenes, set it to `FINEST`.

Even though Java Util Logging works, I personally prefer a slightly more sophisticated solution called Logback[56]. Logback also implements SLF4J API[57], which we're using already, so the only thing that we need to do is `slf4j-jdk14` with `logback-classic`:

[55] https://docs.oracle.com/javase/7/docs/api/java/util/logging/SimpleFormatter.html
[56] http://logback.qos.ch/
[57] http://www.slf4j.org/

```
buildscript {
  ext.logback_version = '1.2.3'
}
dependencies {
  //  "org.slf4j:slf4j-jdk14:$slf4j_version"
  runtime "ch.qos.logback:logback-classic:$logback_version"
  compile "org.slf4j:slf4j-api:$slf4j_version"
}
```

Finally, we need to put a Logback configuration file to the `src/resources`. Let's create a new file called `logback.xml` there:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration scan="true">
  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <charset>UTF-8</charset>
      <Pattern>%d %-4relative [%thread] %-5level %logger{35} - %msg%n</Pattern>
    </encoder>
  </appender>

  <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>logs/application.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
      <fileNamePattern>logs/application_%d{yyyy-MM-dd}.%i.log</fileNamePattern>
      <timeBasedFileNamingAndTriggeringPolicy
          class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
        <maxFileSize>5MB</maxFileSize>
      </timeBasedFileNamingAndTriggeringPolicy>
      <maxHistory>30</maxHistory>
    </rollingPolicy>

    <encoder>
      <charset>UTF-8</charset>
      <pattern>%d %-4relative [%thread] %-5level %logger{35} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="DEBUG">
    <appender-ref ref="CONSOLE"/>
    <appender-ref ref="FILE"/>
  </root>
</configuration>
```

This type of configuration may look rather old-fashioned, but it does everything we need, and besides, we usually don't need to read it every day. Basically, we're creating two appenders - `FILE` and `CONSOLE` - just as we did for JUL. Here, however, we're instructing Logback to backup log files once they reach a certain size. As a result, old files will be kept in the `logs` directory for quite some time, and if something bad happens, we will be able to get log information and investigate. Before trying to run the app, don't forget to create the `logs` directory because `RollingFileAppender` won't do it for you.

One last thing that's worth mentioning is log levels. Certain packages can be very annoying because they put too much information in the log. In this case, we can always increase their log levels or disable them completely. For example, if we want to make Thymeleaf and HikariCP print less details about what they do, we can add the following to `logback.xml`:

```
1  <logger name="org.thymeleaf.TemplateEngine" level="INFO"/>
2  <logger name="com.zaxxer.hikari.HikariConfig" level="INFO"/>
```

In addition to `INFO`, we can also set other levels up to `OFF`. The `OFF` level effectively silences the particularly annoying logger while leaving the rest untouched.

Hosting and deployment

We are finally here! Our application is built and ready to start serving first visitors. There are, however, several things that you need to take care of before you can run the server in production. Let's start with the hosting.

Hosting considerations

Many hosting providers offer virtual private servers (VPS) for as little as $5 a month. This usually includes 1Gb of RAM, which is hardly enough for modern JVM-based applications, so I would recommend starting with 2Gb servers. My experience suggests that 2Gb is enough for a Java-based website with not many visitors. As for the virtualization technology 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 OpenJDK. There is no real need to use a JDK from Oracle, so the binaries from the Ubuntu repository will work just fine. Alternatively, you can get compiled OpenJDK from other vendors, i.e, AdoptOpenJDK.

If your configuration becomes non-trivial, I would recommend you look at Ansible[58] and try to automate it. Ansible allows you to write a single file with instructions and then run it on as many servers as you need. Moreover, third-party Ansible roles may even simplify configuration.

Preparing the distribution

We already discussed how to package the entire app in a single JAR file, but before building the production distribution, it's better to minify our frontend assets. Since we're using Webpack, we need to add a new script to the `package.json` file:

[58] https://www.ansible.com/

```
{
  // ...
  "scripts": {
    "watch": "webpack --mode development --watch",
    "prod": "webpack --mode production"
  },
  // ...
}
```

You may also want to obfuscate your frontend assets, and if this is the case, the `UglifyJS` plugin comes in very handy. Just install it as a development dependency:

```
$ npm i uglifyjs-webpack-plugin -D
```

And then add the following lines to your `webpack.config.js`:

```
const UglifyJsPlugin = require("uglifyjs-webpack-plugin");

module.exports = {
  // ...
  optimization: {
    minimizer: [
      new UglifyJsPlugin({
        cache: true,
        parallel: true
      })
    ]
  }
};
```

After that, try running `npm run prod`, which will result in building minified files inside the same `public/compiled` directory:

```
$ npm run prod
Hash: ba9495e3c9d4fc41c0f7
Version: webpack 4.20.2
Time: 7645ms
Built at: 2018-10-13 19:39:07
     Asset      Size  Chunks             Chunk Names
styles.css  42.3 KiB       0  [emitted]  main
 bundle.js   115 KiB       0  [emitted]  main
```

Note that the minified `bundle.js` occupies only 115 kB.

The final step is to build the uber JAR using the `:shadowJar` task:

```
$ gradle shadowJar
:compileKotlin UP-TO-DATE
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:shadowJar UP-TO-DATE

BUILD SUCCESSFUL

Total time: 0.978 secs
```

Note that Gradle doesn't put frontend assets from the `public` directory into the final JAR. Therefore, they must be copied to a remote server manually, which is not always convenient. On the other hand, if you need to fix a typo in an HTML template, you don't need to redeploy the server.

As for the production configuration, it usually differs from the development one. For example, if we need to use another port for production (say, 9000) and enable caching for static assets, we could write the following in `production.json`:

```
{
  "server" : {
    "port": 9000,
    "caching": true
  },
  "dataSource": {
    "user": "kotlinuser",
    "password": "kotlinpass",
    "jdbcUrl": "jdbc:postgresql://localhost:5432/kotlindb"
  }
}
```

When everything is ready, starting the server is actually simple, and as I promised before, you don't need Kotlin for that. Just use `java` executable from the JDK and Vert.x command line arguments:

```
$ java -cp build/libs/gradle-kotlin-all.jar io.vertx.core.Launcher
  run verticles.MainVerticle -conf conf/production.json
```

That's it! The server will start, and after a couple of seconds it will be accepting client requests on port 9000.

Closing remarks

And we're finally here!

In this book, I tried to approach teaching a programming language in a rather unusual way. Instead of concentrating on numerous syntax constructs and unimportant nuances, we dived into the Kotlin ecosystem and even managed to build a Web app.

Now you should be very well-equipped to adopt Kotlin in your organization or start using it for building your own programs. I hope that you liked reading this book as much as I did writing it. If you have any questions or feedback, you can always reach me via email: `denis` AT `appliedscala.com`.

Good luck with your future projects.

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
AbstractVerticle	io.vertx.core.AbstractVerticle
AsyncResult	io.vertx.core.AsyncResult
AuthProvider	io.vertx.ext.auth.AuthProvider
BCrypt	org.mindrot.jbcrypt.BCrypt
BodyHandler	io.vertx.ext.web.handler.BodyHandler
Charset	java.nio.charset.Charset
CompositeFuture	io.vertx.core.CompositeFuture
CookieHandler	io.vertx.ext.web.handler.CookieHandler
DataSource	javax.sql.DataSource
DateTimeFormatter	java.time.format.DateTimeFormatter
Duration	java.time.Duration
Either	org.funktionale.either.Either
Flyway	org.flywaydb.core.Flyway
FormLoginHandler	io.vertx.ext.web.handler.FormLoginHandler
Future	io.vertx.core.Future
Handler	io.vertx.core.Handler
HikariDataSource	com.zaxxer.hikari.HikariDataSource
Instant	java.time.Instant
JsonObject	io.vertx.core.json.JsonObject
JsonParser	com.google.gson.JsonParser
LocalSessionStore	io.vertx.ext.web.sstore.LocalSessionStore
LoggerFactory	io.vertx.core.logging.LoggerFactory
ObjectMapper	com.fasterxml.jackson.databind.ObjectMapper
Option	org.funktionale.option.Option
Promise	nl.komponents.kovenant.Promise
RedirectAuthHandler	io.vertx.ext.web.handler.RedirectAuthHandler
Router	io.vertx.ext.web.Router
RoutingContext	io.vertx.ext.web.RoutingContext
SessionHandler	io.vertx.ext.web.handler.SessionHandler
StaticHandler	io.vertx.ext.web.handler.StaticHandler
ThymeleafTemplateEngine	io.vertx.ext.web.templ.thymeleaf.ThymeleafTemplateEngine
User	io.vertx.ext.auth.User
UserSessionHandler	io.vertx.ext.web.handler.UserSessionHandler
VertxInit	uy.klutter.vertx.VertxInit

type/object	fully qualified name
ZoneId	java.time.ZoneId
ZonedDateTime	java.time.ZonedDateTime