Getting the Project Files

We'll cover the following

- Setting up Gradle
- Setting up Maven

We'll write code in the `unittests/airportapp` directory—our project location for this example. Since setting up the build files often takes effort, we can save significant time by using a pre-created project structure. Download the source code zip file from the book's website, unzip the file, and `cd` to the `unittests/airportapp` directory. Take a few minutes to study the project structure.

You'll find the following empty directories: `src/main/kotlin/com/agiledeveloper/airportstatus` and `src/test/kotlin/com/agiledeveloper/airportstatus`. These are the default directory structures, for source files and test files, used by both Gradle and Maven build tools. So we can use this structure irrespective of which build tool we pick from those two.

In addition to the two empty directories, in the project directory you'll also find files related to Gradle build and Maven build.

First, we have to pick either Gradle or Maven for the build tool. Maven is the more popular tool, while Gradle is the more pragmatic and lightweight tool. Depending on what you're comfortable with or what's used in projects you work on, feel free to make the appropriate choice.

If you choose to use Maven, skip the next subsection and proceed to Setting Up Maven. If you opt to use Gradle, read along.

Setting up Gradle

With Gradle you may either use Groovy DSL to create a `build.gradle` build file or the Kotlin DSL to create a `build.gradle.kts` build file. As you may guess, we'll use

the Kotlin DSL for Gradle in this chapter. You can use the readily provided build file from the `airportapp` project directory:

```
plugins {
  kotlin("jvm") version "1.3.41"
  application
  jacoco
}

repositories {
  mavenCentral()
  jcenter()
}
val test by tasks.getting(Test::class) {
  useJUnitPlatform {}

  testLogging.showStandardStreams = true
}

dependencies {
  implementation(kotlin("stdlib"))
  implementation(kotlin("reflect:1.3.41"))
  implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.2.2")
  implementation("com.beust:klaxon:5.0.2")

  testImplementation("io.kotlintest:kotlintest-runner-junit5:3.3.1")
  testImplementation("io.mockk:mockk:1.9")
}

tasks {
  getByName<JacocoReport>("jacocoTestReport") {
      afterEvaluate {
          setClassDirectories(files(classDirectories.files.map {
              fileTree(it) { exclude("**/ui/**") }
          }))
      }
  }
}

jacoco {
  toolVersion = "0.8.3"
}

application {
  mainClassName = "com.agiledeveloper.ui.AirportAppKt"
}

defaultTasks("clean", "test", "jacocoTestReport")
```

build.gradle.kts

We use the kotlin plugin to bring in the dependencies for Kotlin-related Gradle tasks to compile and test, the application plugin to be able to easily run the `main()` function through Gradle, and finally the `jacoco` plugin for the code coverage tool. The build file specifies both the `mavenCentral` repository, from which most necessary dependencies will be downloaded, and also the `jcenter` repository from which the JSON parser will be downloaded by Gradle.

which the JSON parser will be downloaded by Gradle.

The `useJunitPlatform {}` function is used to configure KotlinTest to run on top of the JUnit execution platform, a step needed by the KotlinTest tool. In the dependencies section, we have listed all the dependencies needed for this project: the Kotlin standard library, the reflection API, the coroutines library, the Klaxon JSON parser, the KotlinTest library, and the Mockk library. The `application {}` function is used to run the `main()` function from the `AirportAppKt` class, which we'll eventually write. Finally, the `defaultTask` specifies the build steps Gradle will follow by default, to clean up the project build directory, to compile and run the tests, and to measure code coverage.

To run the build file `build.gradle.kts`, we need the Gradle tool. The easiest way to download the desired version of Gradle is using the Gradle wrapper. It's already provided in the `airportapp` project directory for your convenience. To install Gradle and to run the tests, type the following command:

```
gradlew
```

On Windows, the command will execute the `gradlew.bat file`. On some Unix-like systems you may have to prefix the command with a dot-slash, like so, `./gradlew` to run the command tool from the current directory.

Even though we don't have any tests or code yet, the build command should download the specified dependencies and print a message on the console that the build was successful.

It's time to write our first test. Since you're using Gradle, skip the next subsection and proceed to Starting with a Canary Test.

Setting up Maven

You can use the readily provided Maven build file from the `airportapp` project directory. The `pom.xml` file contains the configurations for the necessary dependencies:

- The repositories include `mavenCentral` by default, but we've also added `jcenter` repository to download the JSON parser. The build file also shows configuration for the `kotlin-maven-plugin` and the `maven-compiler-plugin`, among other things.

- The build file also brings in the following dependencies: the Kotlin standard library, the reflection API, the coroutines library, the Klaxon JSON parser, the KotlinTest library, and the Mockk library.
- The build file is configured to create an archive, a jar file, with the necessary manifest entry to easily run the `main()` function from `AirportAppKt`, which we'll write eventually.

To use the build file, first download Maven if you don't already have it on your system. Once you have Maven installed, run the build and test using the command:

```
mvn package
```

Even though we don't have any tests or code yet, the build command should download the specified dependencies and print a message on the console that the build was successful.

It's time to write our first test.
