Creating a Starter Project

We'll cover the following

Initializing the application

If you want to set up the application environment on your system then follow along with the upcoming instructions.

The easiest way to get started is using the Spring Initializr website. We'll create a Spring Boot RESTful web service that stores data in the in-memory H2 database.

Once you visit the Spring Initializr website, choose either Maven Project or Gradle Project. For language, select Kotlin, of course. Then choose the desired version of Spring Boot—for the examples in this chapter, we use version 2.1.2. For the Group text box, type a desired top-level package name—for example, `com.agiledeveloper`. For the Artifact, type `todo`. In the text box next to Search for Dependencies, type "Web" and select it from the drop-down list that appears. Also, type "H2" and select it from the drop-down. As a final dependency, type "JPA" and select it from the drop-down. Finally, click the Generate Project button and save the generated zip file to your system.

Unzip the `todo.zip` file on your system and, using the command-line tool, `cd` to the `todo` directory.

Examine the `pom.xml` file if you selected a Maven project, or the `build.gradle` file if you selected Gradle instead. Take note of the dependencies. You'll see dependencies on the H2 library, the Jackson-kotlin library, which will be used to create a JSON response, the JPA library, and the Kotlin-stdlib library compatible with JDK 8.

Spring Boot uses a special Kotlin-Spring compiler plugin to deal with a few things that conflict with the default Kotlin way and the Spring way. In Kotlin, classes are `final` by default. But Spring expects classes to be open. Without the Kotlin-Spring compiler plugin integration, each class written using Kotlin, like a controller, for example, will have to be explicitly marked `open`. Thanks to the plugin, we can write the classes without the open keyword. The plugin will inspect a class to see if it has some Spring-related meta-annotations, like `@Component`, `@Async`, `@Transactional`, and so on, and if it does, it'll automatically open those classes during compilation. Since these are meta-annotations, in addition to opening classes that are decorated with these annotations, the plugin also opens classes that are decorated with derived annotations like `@Component`.

Kotlin vs. Java

The entry point for a Spring Boot application is the class containing the `main()` method. If we were to create the Spring Boot application using Java, we'd have something like this:

```
//Java code only for comparison purpose
package com.agiledeveloper.todo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class TodoApplication {
    public static void main(String[] args) {
        SpringApplication.run(TodoApplication.class, args);
    }
}
```

You know that Kotlin is a language of low ceremony, and we don't need a class to create the `main()` function in Kotlin. The Spring Initializr tool created a much simpler file for our Kotlin version of the application:

```
package com.agiledeveloper.todo

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@SpringBootApplication
class TodoApplication

fun main(args: Array<String>) {
        runApplication<TodoApplication>(*args)
}
```

TodoApplication.kt

The Java version and the Kotlin version of `TodoApplication` have three differences between them: First, the `main()` function in the Kotlin version is a standalone top-level function, instead of being a member of the class. Second, the bootstrap call to `SpringApplication.run()` in the Java version has been replaced with a more concise, less cluttered call to `runApplication()`—thanks to the powerful generics facility of Kotlin, we're able to infer the class details from the parametric type provided. The third difference is the lack of semicolon—let's not forget that.

Running the application

We've not written any code yet, but before we make any changes, let's compile the downloaded starter project code and make sure it builds successfully.

If you choose to use Maven, run the following commands to build the code and to start the application:

```
./mvnw clean install
java -jar target/todo-0.0.1-SNAPSHOT.jar
```

If you choose Gradle instead, then run the following commands to build the code and to start the application:

```
./gradlew build
java -jar build/libs/todo-0.0.1-SNAPSHOT.jar
```

Refer back to the above commands anytime you want to build the application.

We have the starter code in place. Our application will manage tasks where each task has a description. We'll have three routes, all through the endpoint `task`, but with three different HTTP methods:

- GET method to list all available tasks.
- POST method to add a new task.
- DELETE method to remove an existing task.

Let's now move on and write some code for the application at hand in the next lesson.