CHAPTER 3

A Basic Spring Boot Application

You could start writing code directly, but that, even while being pragmatic, would be far from a real case. Instead, you'll see how to define a product you want to build and split it into small chunks. This requirements-oriented approach is used throughout the book to make it more practical. In real life, you'll always have these business requirements.

The web application you build will encourage users to exercise their brains daily. To begin with, it will present users with two-digit multiplications, one every time they access the page. They will type their alias (a short name) and their guess for the result of the operation. The idea is that they should use only mental calculations. After they send the data, the web page will indicate if the guess was correct or not.

To keep user motivation as high as possible, you will use some gamification. For each correct guess, the users get points and they will see their score in a ranking so that they can compete with other people.

This is the main idea of the complete application. But you won't build it all at once. This book emulates an agile way of working in which you split requirements into *user stories,* small chunks of functionality that give value by themselves. You'll follow this methodology to keep this book as close as possible to real life since a vast majority of IT companies use agile.

Let's start simple and focus first on the multiplication solving logic. Consider the first user story here.

USER STORY 1 As a user of the application, I want to be presented with a random multiplication problem so that I can solve it using mental calculation and exercise my brain.

M. Macero García and T. Telang, *Learn Microservices with Spring Boot 3*,
https://doi.org/10.1007/978-1-4842-9757-5_3

To make this work, you need to build a minimal skeleton of the web application. Therefore, split the user story into several subtasks:

1. Create a basic service with business logic.
2. Create a basic API to access this service (REST API).
3. Create a basic web page to ask the users to solve that calculation.

This chapter focuses on Tasks 1 and 2. After creating the skeleton of your first Spring Boot application, you'll use test-driven development to build the main logic of this component: generating multiplication challenges and verifying attempts from the users to solve those challenges. Then, you'll add the controller layer that implements the REST API. You'll learn about the advantages of this layered design.

This learning path includes some reasoning about one of the most important features in Spring Boot: autoconfiguration. You'll use the practical case to see how, for example, the application includes its own embedded web server, only because you'll add a specific dependency to this project.

Setting Up the Development Environment

To set up the development environment, the following essential tools and components must be installed and configured.

Java Development Kit 17

We use Java 17 in this book. Make sure you get at least that version of the JDK from the official downloads page (`https://jdk.java.net/17/`). Install it following the instructions for your OS.

Integrated Development Environment (IDE)

A good IDE is also convenient for developing Java code. If you have a preferred one, just use it. Otherwise, you can download the community version of IntelliJ IDEA (`https://www.jetbrains.com/idea/download/`).

HTTPie

In this book, we also use HTTPie to quickly test the web applications. It's a command-line tool that allows you to interact with an HTTP server. It is a user-friendly alternative to cURL, another popular command-line tool for working with HTTP requests. You can follow the instructions at `https://httpie.io/cli` to download it for Linux, Mac, or Windows. In addition to downloading and installing HTTPie locally, you can also try it online without any installation. By visiting `https://httpie.io/cli`, you can access a web-based interface that allows you to run HTTPie commands directly from your browser. This is excellent for learning and experimentation and to explore the tool's functionality. Let's go over how you can explicitly define the JSON and XML format when fetching and sending data using the HTTP GET and POST methods.

Working with JSON

To request JSON data with HTTPie, you can add the `Accept: application/json` header to your request.

GET Request

The following is an example of using HTTPie to make a GET request to a given URL (`https://api.example.com/resource`) with the intention of receiving a response in JSON.

```
$ http --json GET https://api.example.com/resource Accept:application/json
```

The `--json` option signals that the request should be sent with the `Content-Type: application/json` header and it sets the `Accept: application/json` header.

GET specifies the HTTP method to be used (in this case, a GET request).

The `https://api.example.com/resource` URL is the server address where the request is being sent. The `Accept:application/json` header explicitly asks the server to respond with JSON-formatted data.

POST Request

To send JSON data with HTTPie, you can use the `--json, -j` option, followed by key-value pairs. For example:

```
$ http --json POST https://api.example.com/resource key1=value1 key2=value2
```

This sends the key-value pairs as a JSON object in the body of the request.

cURL

Alternatively, if you are a `cURL` user, you can easily map this book's `HTTP` commands to cURL commands.

Using cURL cURL (short for "Client URL") is a command-line tool used to transfer data with URLs. Here is an example of using cURL to make a simple HTTP GET request:

```
$ curl https://www.example.com
```

To make a POST request with JSON data, use this command:

```
$ curl -X POST -H "Content-Type: application/json" -d
'{"key": "value"}' https://www.example.com/api
```

You can visit cURL's official website (`https://curl.se/`) to learn more about this tool.

The Skeleton Web App

It's time to write some code! Spring offers a fantastic way to build the skeleton of an application: the Spring Initializr. This web-based tool allows you to select the components and libraries you want to include in your Spring Boot project, and it generates the structure and dependency configuration into a ZIP file that you can download. With Spring Initializr, developers can generate a new Spring Boot project with just a few clicks, and the resulting project can be easily imported into their preferred

Integrated Development Environment (IDE) for further development. We'll use the Spring Initializr a few times in the book, since it saves time over creating the project from scratch, but you can also create the project yourself if you prefer that option.

Source Code: chapter03 You can find all the source code for this chapter on GitHub, in the `chapter03` repository.

See https://github.com/Book-Microservices-v3/chapter03.

Navigate to https://start.spring.io/ and fill in some data, as shown in Figure 3-1.

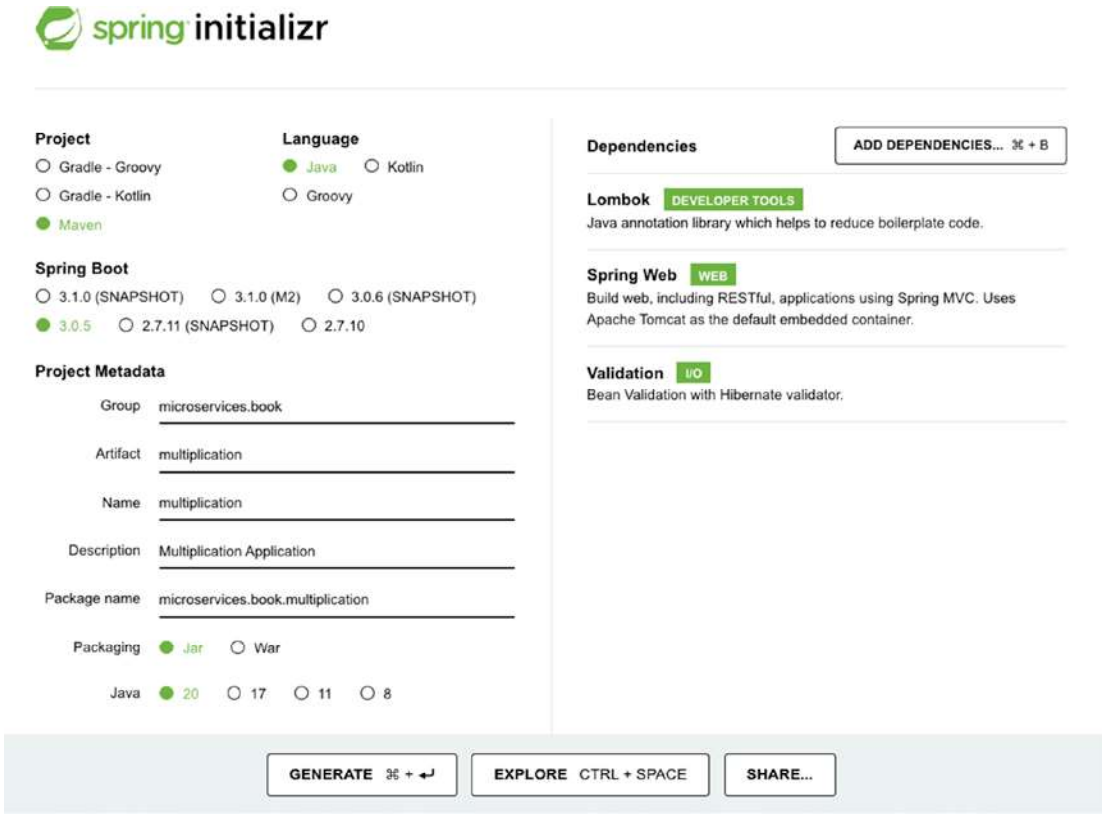

Figure 3-1. *Creating a Spring Boot project with the Spring Initializr*

All the code in this book uses Maven, Java, and the Spring Boot version 3.1.0, so let's stick to them. If that Spring Boot version is not available, you can select a more recent one. In that case, remember to change it later in the generated `pom.xml` file if you want to use the same version as in the book. You may also go ahead with other Java and Spring Boot versions, but then some of the code examples in this book might not work for you.

Give some values to the group (`microservices.book`) and the artifact (`multiplication`). Select Java 17. Do not forget to add the Spring Web, Validation, and Lombok dependencies from the list or search tool. You already know what Lombok is intended for, and you'll see what the other two dependencies do in this chapter. That's all you need for now.

Generate the project and extract the ZIP contents. The `multiplication` folder contains everything you need to run the application, you should see the project files as shown in the Figure 3-2. You can now open this with your favorite IDE, usually by selecting the `pom.xml` file.

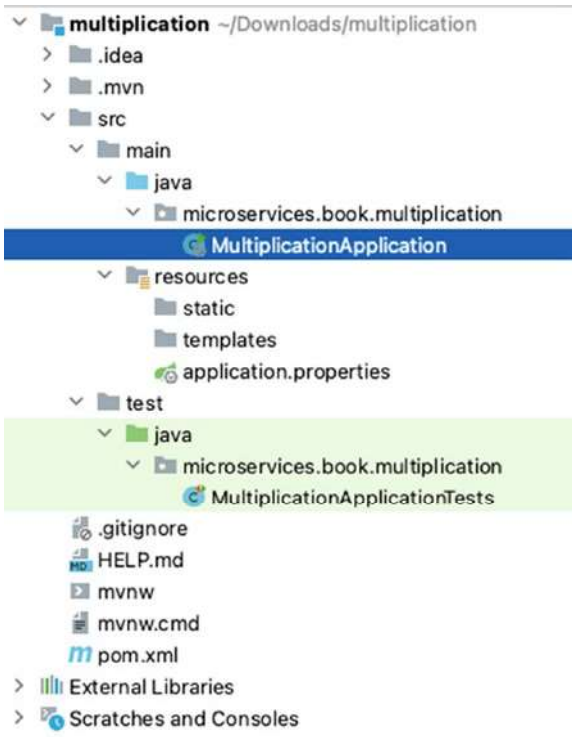

Figure 3-2. *Spring Boot project structure*

These are the main elements you'll find in the auto-generated package:

- There is Maven's `pom.xml` file, an XML file that serves as the Project Object Model (POM) for Maven-based Spring Boot applications. It defines the project's configuration, dependencies, and build settings. This is the main file used by Maven to build the application. When a developer adds a new dependency to the `pom.xml` file, Maven

automatically downloads and includes the required libraries in the project's build path. Maven also manages the project's build process, including compiling source code, running tests, and packaging the application. Inside this file, you can also find the configuration to build the application using Spring Boot's Maven plugin, which also knows how to package all its dependencies in a stand-alone `.jar` file and how to run these applications from the command line. You'll separately examine some of the dependencies added by Spring Boot.

- There is the Maven wrapper. This is a stand-alone version of Maven, so you don't need to install it to build your app. There is a `.mvn` folder and the `mvnw` executables for Windows and UNIX-based systems.
- The `HELP.md` file contains links to the Spring Boot's documentation. This file is a helpful resource for developers who are new to Spring Boot, as it provides information on how to get started with a new project and how to configure and customize the project to suit their needs.
- Assuming you'll use Git as a version control system, the included `.gitignore` has some predefined exclusions, so you don't commit the compiled classes or any IDE-generated files to the repository. The file contains a list of patterns that Git uses to match files and directories that should be ignored. When Git tracks changes in a project's source code, it checks the `.gitignore` file to determine which files and directories should be excluded from version control. This can include log files, temporary files, compiled code, and other files that are not essential to the project's source code.
- The `src` folder follows the standard Maven structure that divides the code into the subfolders, `main` and `test`. Both folders contain their respective `java` and `resources` children. In this case, there is a source folder for the main code and tests, and a resource folder for the main code.
  - There is a class created by default in the main sources, called `MultiplicationApplication`. It's already annotated with `@SpringBootApplication` and contains the `main` method that boots up the application. This is the standard way of defining the main class for a

Spring Boot application, as detailed in the reference documentation (https://docs.spring.io/spring-boot/docs/current/reference/html/using.html#using.using-the-springbootapplication-annotation). You'll learn about this class later.

- Within the `resources` folder are two empty subfolders: `static` and `templates`. You can safely delete them since they're intended to include static resources and HTML templates, which you won't use in this book.
- The `application.properties` file is where you can configure your Spring Boot application. It can include configuration like server port, database connection information, and logging settings. You'll add some configuration parameters later.

Now that you understand the different pieces of this skeleton, let's try to make it walk. To run this app, you can either use your IDE interface or run this command from the project's root folder:

```
multiplication $ ./mvnw spring-boot:run
```

Running Commands from the Terminal In this book, we use the $ character to represent the command prompt. Everything after that character is the command itself. Sometimes, it's important to highlight that you must run the command within a given folder in the workspace. In that case, you'll find the folder name before the $ character (e.g., `multiplication $`). Of course, the specific location of your workspace may be different.

Note also that some commands may vary depending on whether you're using a UNIX-based operating system (like Linux or Mac) or Windows. All the commands shown in this book use a UNIX-based system version.

When you run that command, you're using the Maven wrapper included in the project's main folder (`mvnw`) with the goal (what's next to the Maven executable) `spring-boot:run`. This goal is provided by Spring Boot's Maven plugin, included also in the `pom.xml` file generated by the Initializr web page. The Spring Boot application should start successfully. The last line in the logs should look like this:

```
INFO 4139 --- [main] m.b.m.MultiplicationApplication: Started
MultiplicationApplication in 6.599 seconds (JVM running for 6.912)
```

Great! You have your first Spring Boot app running without writing a single line of code! However, there isn't much you can do with it yet. What's this application doing? You'll figure that out soon.

Spring Boot Autoconfiguration

In the logs of the skeleton app, you can also find this log line:

```
INFO 74642 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  :
Tomcat started on port(s): 8080 (http) with context
```

What you get when you add the web dependency is an independently deployable web application that uses Tomcat, thanks to the autoconfiguration feature in Spring.

As introduced in the previous chapter, Spring Boot sets up libraries and the default configuration automatically. This saves you a lot of time when you rely on all these defaults. One of those conventions is to add a ready-to-use Tomcat server when you add the web starter to your project.

To learn more about Spring Boot autoconfiguration, let's go through how this specific case works, step-by-step—Figure 3-3 can some useful visual help.

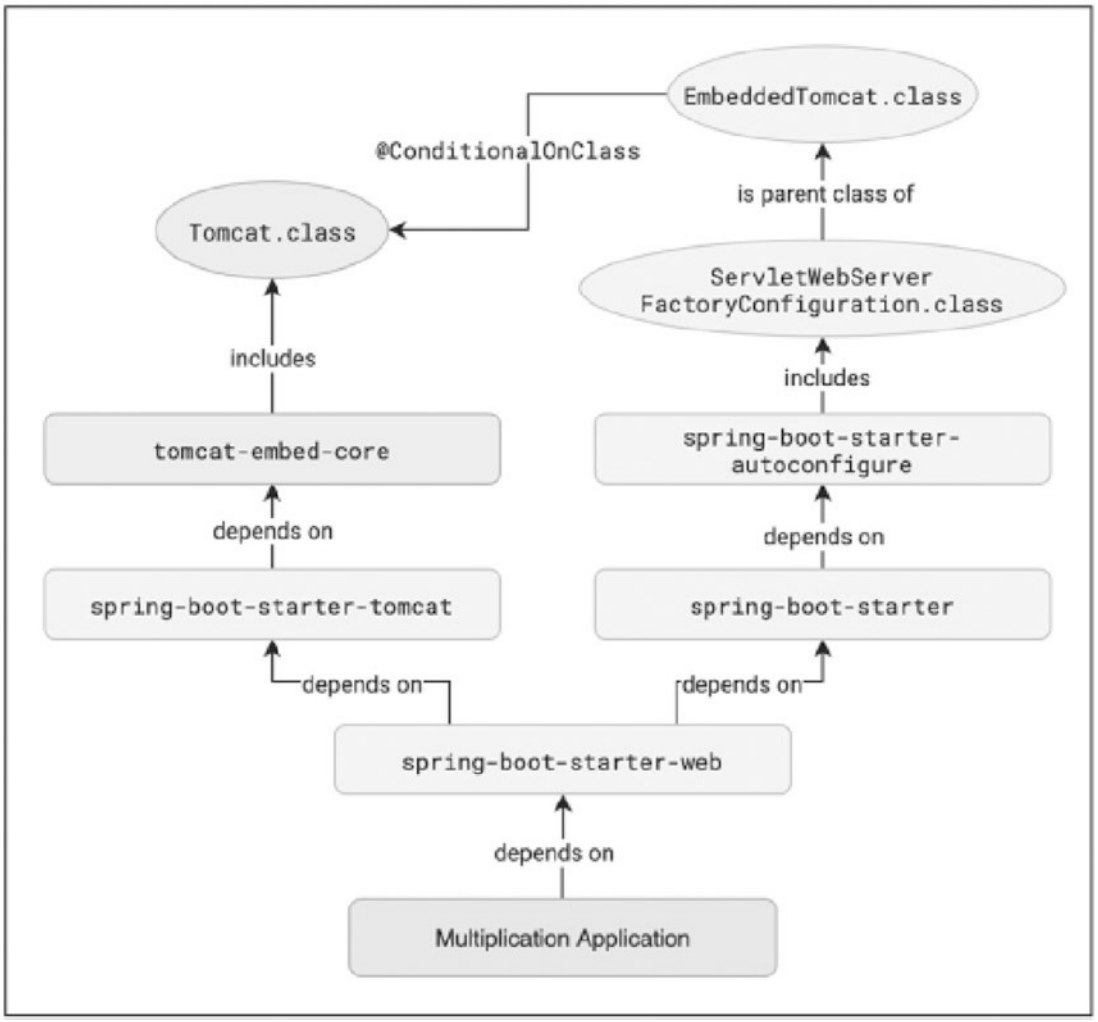

Figure 3-3. *Autoconfiguration example: embedded Tomcat*

The Spring Boot application you generated automatically has a main class annotated with `@SpringBootApplication`. This is a *shortcut* annotation because it groups several others, among them `@EnableAutoConfiguration`. As its name suggests, with this one you're enabling the autoconfiguration feature. Therefore, Spring activates this smart mechanism and finds and processes classes annotated with the `@Configuration` annotation, from your own code but also from your dependencies.

This project includes the `spring-boot-starter-web` dependency. This is one of the main Spring Boot components, which has all the tooling to build a web application, including the embedded Tomcat server. Inside this artifact's dependencies, the Spring

Boot developers added another starter, called `spring-boot-starter-tomcat`. See Listing 3-1 or the online sources (https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-starters/spring-boot-starter-web/build.gradle).

Listing 3-1. Web Starter Dependencies

```
plugins {
    id "org.springframework.boot.starter"
}
description = "Starter for building web, including RESTful, applications
using Spring MVC. Uses Tomcat as the default embedded container"
dependencies {
    api(project(":spring-boot-project:spring-boot-starters:spring-boot-
    starter"))
    api(project(":spring-boot-project:spring-boot-starters:spring-boot-
    starter-json"))
    api(project(":spring-boot-project:spring-boot-starters:spring-boot-
    starter-tomcat"))
    api("org.springframework:spring-web")
    api("org.springframework:spring-webmvc")
}
```

As you can see, Spring Boot artifacts use Gradle (since version 2.3), but you don't need to know the specific syntax to understand what the dependencies are. If you now check the dependencies of the `spring-boot-starter-tomcat` artifact (in Listing 3-2 or the online sources at https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-starters/spring-boot-starter-tomcat/build.gradle), you can see that it contains a library that doesn't belong to the Spring family, `tomcat-embed-core`. This is an Apache library that you can use to start a Tomcat embedded server. Its main logic is included in a class named `Tomcat`.

Listing 3-2. Tomcat Starter Dependencies

```
plugins {
    id "org.springframework.boot.starter"
}
```

```
description = "Starter for using Tomcat as the embedded servlet container.
Default servlet container starter used by spring-boot-starter-web"
dependencies {
    api("jakarta.annotation:jakarta.annotation-api")
    api("org.apache.tomcat.embed:tomcat-embed-core") {
        exclude group: "org.apache.tomcat", module: "tomcat-
        annotations-api"
    }
    api("org.glassfish:jakarta.el")
    api("org.apache.tomcat.embed:tomcat-embed-websocket") {
        exclude group: "org.apache.tomcat", module: "tomcat-
        annotations-api"
    }
}
```

Coming back to the hierarchy of dependencies, the `spring-boot-starter-web` also depends on `spring-boot-starter` (see Listing 3-1 and Figure 3-3 for some contextual help). That's the *core* Spring Boot starter, which includes the artifact `spring-boot-autoconfigure` (see Listing 3-3 or the online sources, at https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-starters/spring-boot-starter/build.gradle). That Spring Boot artifact has a whole set of classes annotated with `@Configuration`, which are responsible for a big part of the whole Spring Boot magic. There classes are intended to configure web servers, message brokers, error handlers, databases, and many more.

Listing 3-3. Spring Boot's Main Starter

```
plugins {
    id "org.springframework.boot.starter"
}
description = "Core starter, including auto-configuration support, logging
and YAML"
dependencies {
    api(project(":spring-boot-project:spring-boot"))
    api(project(":spring-boot-project:spring-boot-autoconfigure"))
    api(project(":spring-boot-project:spring-boot-starters:spring-boot-
    starter-logging"))
```

```
    api("jakarta.annotation:jakarta.annotation-api")
    api("org.springframework:spring-core")
    api("org.yaml:snakeyaml")
}
```

For this project, the relevant class that takes care of the embedded Tomcat server autoconfiguration is `ServletWebServerFactoryConfiguration`. Listing 3-4 shows its most relevant code fragment. The complete source code is available online (https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/servlet/ServletWebServerFactoryConfiguration.java).

Listing 3-4. ServletWebServerFactoryConfiguration Fragment

```
@Configuration(proxyBeanMethods = false)
class ServletWebServerFactoryConfiguration {
    @Configuration(proxyBeanMethods = false)
    @ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.
    class })
    @ConditionalOnMissingBean(value = ServletWebServerFactory.class,
    search = SearchStrategy.CURRENT)
    static class EmbeddedTomcat {
        @Bean
        TomcatServletWebServerFactory tomcatServletWebServerFactory(
                ObjectProvider<TomcatConnectorCustomizer>
                connectorCustomizers,
                ObjectProvider<TomcatContextCustomizer> contextCustomizers,
                ObjectProvider<TomcatProtocolHandlerCustomizer<?>>
                protocolHandlerCustomizers) {
            TomcatServletWebServerFactory factory = new
            TomcatServletWebServerFactory();
            factory.getTomcatConnectorCustomizers()
                    .addAll(connectorCustomizers.orderedStream().
                     collect(Collectors.toList()));
            factory.getTomcatContextCustomizers()
                    .addAll(contextCustomizers.orderedStream().
                    collect(Collectors.toList()));
```

```
            factory.getTomcatProtocolHandlerCustomizers()
                    .addAll(protocolHandlerCustomizers.orderedStream().
                    collect(Collectors.toList()));
            return factory;
        }
    }
    // ...
}
```

This class defines some inner classes, one of them being `EmbeddedTomcat`. As you can see, that one is annotated with this annotation:

```
@ConditionalOnClass({ Servlet.class, Tomcat.class, UpgradeProtocol.class })
```

Spring processes the `@ConditionalOnClass` annotation, which is used to load beans in the context if the linked class can be found in the classpath. In this case, the condition matches, since you already saw how the `Tomcat` class got into the classpath via the starter hierarchy. Therefore, Spring loads the bean declared in `EmbeddedTomcat`, which turns out to be a `TomcatServletWebServerFactory`.

That factory is contained inside Spring Boot's core artifact (`spring-boot`, a dependency included in `spring-boot-starter`). It sets up a Tomcat embedded server with some default configuration. This is where the logic to create an embedded web server finally lives.

Once more, just to recap, the `spring-boot-starter-web` simplifies dependency management and includes everything needed for developing web applications and RESTful web services, including the embedded Tomcat server, validations, and the Jackson library to serialize-deserialize Java objects to JSON and logging. The `spring-boot-starter` is a core starter in Spring Boot and serves as a parent starter for dependencies and autoconfiguration. The `spring-boot-starter-autoconfigure` includes all the autoconfiguration classes responsible for configuring different parts of the Spring application based on certain conditions, such as the presence or absence of certain Java classes in the classpath or beans in the context. The `spring-boot-starter-web` includes the `spring-boot-starter-tomcat` dependency, which provides the embedded Tomcat servlet container. Embedded Tomcat is the trimmed-down version of Tomcat optimized for programmatic use, consisting of the classes to start and manage a Tomcat instance within your application. The `ServletWebServerFactoryConfiguration` class is part of the autoconfiguration that's responsible for setting up the embedded

servlet web server, such as Tomcat (https://tomcat.apache.org/), Jetty (https://eclipse.dev/jetty/), or Undertow (https://undertow.io/). This configuration class plays a critical role in defining and customizing the behavior of the embedded web server. They often use conditional annotations like `@ConditionalOnClass`. The `@ConditionalOnClass` annotation is used to define a conditional situation, which allows you to specify that a particular bean should be created only if a specific class is present in the classpath. The `tomcat-embed-core` provides the core functionality required to embed Tomcat within a Java application. The Spring scans all the classes and, given that the condition stated in the `EmbeddedTomcat` is fulfilled (the Tomcat library is an included dependency), it loads a `TomcatServletWebServerFactory` bean in the context. This `factoryclass` starts an embedded Tomcat server with the default configuration, exposing an HTTP interface on the port 8080.

As you can imagine, this same mechanism applies to many other libraries for databases, web servers, message brokers, cloud-native patterns, security, and so on. In Spring Boot, you can find multiple starters that you can add as dependencies. When you do that, the autoconfiguration mechanism comes into play, and you get additional behavior out of the box. Many configuration classes are conditional on the presence of other classes like the ones analyzed here, but there are other condition types, for example, parameter values in the `application.properties` file.

Autoconfiguration is a key concept in Spring Boot. Once you understand it, the features that many developers consider magic are no longer a secret for you. This chapter navigated through the details because it's important that you know this mechanism so you can configure it according to your needs and avoid getting a lot of behavior that you don't want or simply don't need. As a good practice, carefully read the documentation of the Spring Boot modules you're using and familiarize yourself with the configuration options they allow.

Don't worry if you didn't understand this concept fully; we come back to the autoconfiguration mechanism a few times in this book. The reason is that you'll add extra features to your application, and, for that, you need to add extra dependencies to your project and analyze the new behavior they introduce.

Three-Tier, Three-Layer Architecture

The next step in this practical journey is designing how to structure the application and model the business logic in different classes.

A multitier architecture will provide the application with a more production-ready look. Most of the real-world applications follow this architecture pattern. Among web applications, the *three-tier design* is the most popular one and widely extended. These three tiers are as follows:

- *Client tier*: This tier is responsible for the user interface. Typically, this is called the *frontend*.
- *Application tier*: This contains all the business logic together with the interfaces to interact with it and the data interfaces for persistence. This maps to is called the *backend*.
- *Data store tier*: It's the database, file system, and so on, that persists the application's data.

This book mainly focuses on the application tier, although you'll use the other two as well. If you zoom in, the application tier is commonly designed using three layers:

- *Business layer*: This includes the classes that model the domain and the business specifics. It's where the intelligence of the application resides. Sometimes this layer is divided into two parts: domains (entities) and applications (services) providing business logic.
- *Presentation layer*: In this case, it will be represented by the `Controller` classes, which provide the functionality to the web client. The REST API implementation resides here.
- *Data layer*: This layer is responsible for persisting the entities in a data storage, usually a database. It can typically include *data access object* (DAO) classes, which work with objects that map directly to rows in a database, or *repository* classes, which are domain-centric, so they may need to translate from domain representations to the database structure.

The goal is now to apply this pattern to the Multiplication web application, as shown in Figure 3-4.

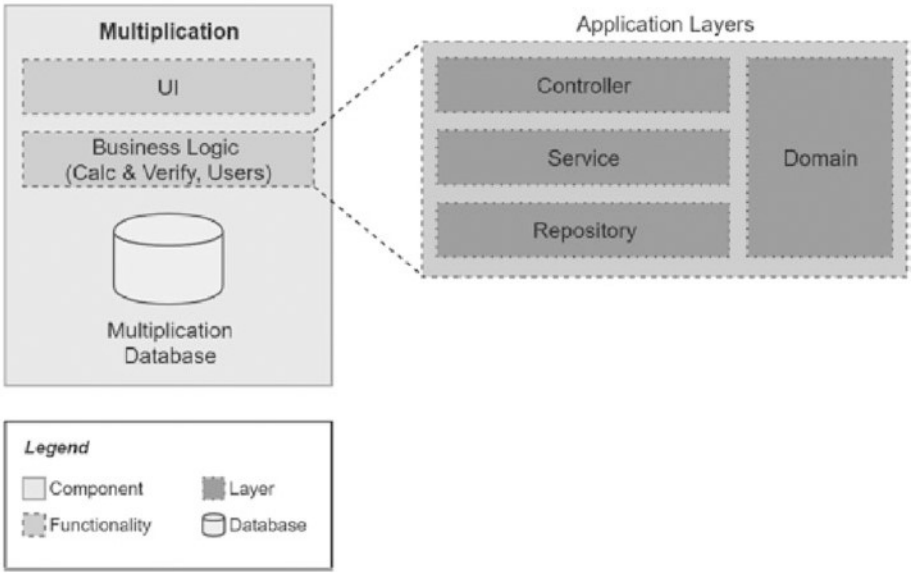

Figure 3-4. *Three-tier, three-layer architecture applied to the Spring Boot project*

The advantages of using this software architecture are all related to achieving loose coupling.

- All layers are interchangeable (such as, for instance, changing the NoSQL database for a SQL database or file storage solution or changing from a REST API to any other interface). This is a key asset because it makes it easier to evolve the codebase. Additionally, you can replace complete layers by test mocks, which keeps your tests simple, as you'll see later in this chapter.
- The domain part is isolated and independent of everything else. It's not mixed with interface or database specifics.
- There is a clear separation of responsibilities: a class to handle database storage of the objects, a separate class for the REST API implementation, and another class for the business logic.

Spring is an excellent option to build this type of architecture, with many out-of-the-box features that will help you easily create a production-ready three-tier application. It provides three *stereotype* annotations for your classes that map to each of this design's layers, so you can use them to implement your architecture.

- The `@Controller` annotation is for the presentation layer. In this case, you'll implement a REST interface using controllers. You can use `@Controller` while building a traditional web application, but for building RESTful APIs, you must use a more specialized `@RestController` annotation.
- The `@Service` annotation is for classes implementing business logic.
- The `@Repository` annotation is for the data layer, namely, the classes that interact with the database.

When you annotate classes with these variants, they become Spring-managed *components*. When initializing the web context, Spring scans your packages, finds these classes, and loads them as beans in the context. Then, you can use dependency injection to wire (or *inject*) these beans and, for example, use services from the presentation layer (controllers). You'll see this in practice soon.

Modeling the Domain

Let's start by modeling the business domain because this will help you structure the project.

Domain Definition and Domain-Driven Design

This first web application takes care of generating multiplication challenges and verifying the subsequent attempts from the users. Let's define these three business entities.

- *Challenge*: Contains the two factors of a multiplication challenge.
- *User*: Identifies the person who will try to solve a Challenge.
- *Challenge Attempt*: Represents the attempt from a User to solve the operation from a Challenge.

You can model these domain objects and their relationship as shown in Figure 3-5.

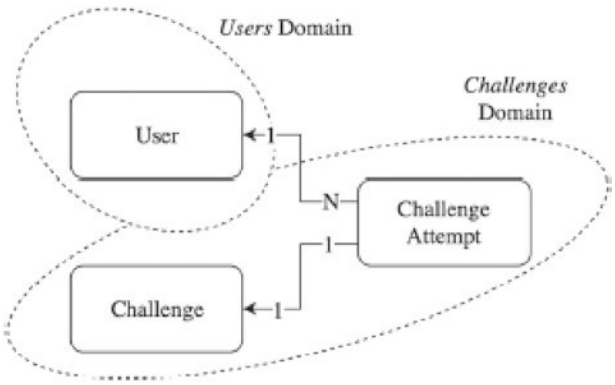

Figure 3-5. *Business model*

The relationships between these objects are as follows:

- Users and Challenges are independent entities. They don't keep any references.
- *Challenge Attempts* are always for a given User and a given Challenge. Conceptually, there could be many attempts for the same Challenge, given that there is a limited number of generated Challenges. Also, the same User may create many attempts since they can use the web application as many times as they want.

In Figure 3-5, you can also see how these three objects are split into two different domains: Users and Challenges. Finding domain boundaries (also known as bounded contexts; see `https://martinfowler.com/bliki/BoundedContext.html`) and defining relationships between your objects are essential tasks of designing software. This design approach based on domains is called *domain-driven design* (DDD). It helps you build an architecture that is modular, scalable, and loosely coupled. In this example, Users and Challenges are completely different concepts. Challenges, and their attempts, are related to Users, but they together have enough relevance to belong to their own domain.

To make DDD clearer, you can think of an evolved version of this small system where other domains relate to Users or Challenges. For instance, you could introduce social network features by creating the domain *Friends* and modeling relationships and interactions between users. If you mixed up the domains Users and Challenges, this evolution would be much harder to accomplish since the new domain has nothing to do with challenges.

For extra reading about DDD, you can get Eric Evans' book (https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/) or download the free InfoQ minibook (https://www.infoq.com/minibooks/domain-driven-design-quickly/).

Microservices and Domain-Driven Design A common mistake when designing microservices is thinking that each domain must be immediately split into a different microservice. This, however, may lead to premature optimization and an exponential complexity increase from the beginning of a software project. Deciding whether a use case should be a microservice requires a careful analysis of the specific context and needs of the system. It involves considering factors like domain boundaries, independence, reusability, complexity, data cohesion, and team organization.

You'll learn more about microservices and the monolith-first approach later. For now, the important takeaway is that modeling domains is a crucial task, but splitting domains doesn't require splitting the code into microservices. In this first application, you include both domains together, but not mixed up. You use a simple strategy for the split: root-level packages.

Domain Classes

It's time to create the `Challenge`, `ChallengeAttempt`, and `User` classes. First, you divide the root package (`microservices.book.multiplication`) in two—`Users` and `Challenges`—following the domains that you identified for the Multiplication application. Then, you create three empty classes with the chosen names in these two packages. See Listing 3-5.

Listing 3-5. Splitting Domains by Creating Different Root Packages

```
+- microservices.book.multiplication.user
|  \- User.java
+- microservices.book.multiplication.challenge
|  \- Challenge.java
|  \- ChallengeAttempt.java
```

Since you added Lombok as a dependency when you created the skeleton app, you can use it to keep the domain classes very small, as described in the previous chapter.

The `Challenge` class holds both factors of the multiplication. You add getters, a constructor with all fields, and the `toString()`, `equals()`, and `hashCode()` methods. See Listing 3-6.

Listing 3-6. The Challenge Class

```
package microservices.book.multiplication.challenge;

import lombok.*;
/**
 * This class represents a Challenge to solve a Multiplication (a * b).
 */
@Getter
@ToString
@EqualsAndHashCode
@AllArgsConstructor
public class Challenge {
    private int factorA;
    private int factorB;
}
```

The `User` class has the same Lombok annotations, an identifier for the user, and a friendly alias (e.g., the user's first name). See Listing 3-7.

Listing 3-7. The User Class

```
package microservices.book.multiplication.user;

import lombok.*;

/**
 * Stores information to identify the user.
 */
@Getter
@ToString
@EqualsAndHashCode
@AllArgsConstructor
```

```
public class User {
    private Long id;
    private String alias;
}
```

Attempts also have an ID, the value input by the user (`resultAttempt`), and whether it's correct or not. See Listing 3-8. You link it to the user via `userId`. Note that you also have here both challenge factors. You do this to avoid having a reference to a challenge because you can simply generate new challenges "on the fly" and copy them here to keep your data structures simple. Therefore, as you can see, you have multiple options to implement the business model depicted in Figure 3-5. To model the relationship with users, you use a reference; to model challenges, you embed the data inside the attempt. You will analyze this decision in more detail in Chapter 5, when you learn about data persistence.

Listing 3-8. The ChallengeAttempt Class

```
package microservices.book.multiplication.challenge;

import lombok.*;
import microservices.book.multiplication.user.User;
/**
 * Identifies the attempt from a {@link User} to solve a challenge.
 */
@Getter
@ToString
@EqualsAndHashCode
@AllArgsConstructor
public class ChallengeAttempt {
    private Long id;
    private Long userId;
    private int factorA;
    private int factorB;
    private int resultAttempt;
    private boolean correct;
}
```

Business Logic

Once you have defined the domain model, it's time to think about the other part of the business logic: the *application services*.

What You Need

Having looked at the requirements, you need the following:

- A way of generating a mid-complexity multiplication problem. Let's make all factors between 11 and 99.
- Some functionality to check whether an attempt is correct or not.

Random Challenges

Let's put test-driven development into practice for the business logic. First, you write a basic interface to generate random challenges. See Listing 3-9.

Listing 3-9. The ChallengeGeneratorService Interface

```
package microservices.book.multiplication.challenge;

public interface ChallengeGeneratorService {
    /**
     * @return a randomly generated challenge with factors between
     11 and 99
    */
    Challenge randomChallenge();
}
```

You place this interface in the `Challenge` package. Now, you write an empty implementation of this interface that wraps Java's `Random class`. See Listing 3-10. Besides the `no-args` constructor, you make the class testable by having a second constructor that accepts the random object.

Listing 3-10. An Empty Implementation of the ChallengeGeneratorService Interface

```
package microservices.book.multiplication.challenge;

import org.springframework.stereotype.Service;

import java.util.Random;
@Service
public class ChallengeGeneratorServiceImpl implements
ChallengeGeneratorService {
    private final Random random;
    ChallengeGeneratorServiceImpl() {
        this.random = new Random();
    }
    protected ChallengeGeneratorServiceImpl(final Random random) {
        this.random = random;
    }
    @Override
    public Challenge randomChallenge() {
        return null;
    }
}
```

To instruct Spring to load this service implementation in the context, you annotate the class with `@Service`. You can later inject this service into other layers by using the interface and not the implementation. This way, you keep loose coupling since you could swap the implementation without needing to change anything in other layers. You'll put dependency injection into practice soon. For now, focus on TDD and leave the `randomChallenge()` implementation non-functional (empty).

The next step is to write a test for this. You create a class in the same package but this time inside the `test` source folder. See Listing 3-11.

Listing 3-11. Creating the Unit Test Before the Real Implementation

```
package microservices.book.multiplication.challenge;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
```

```
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Spy;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Random;

import static org.assertj.core.api.BDDAssertions.then;
import static org.mockito.BDDMockito.given;
@ExtendWith(MockitoExtension.class)
public class ChallengeGeneratorServiceTest {
    private ChallengeGeneratorService challengeGeneratorService;
    @Spy
    private Random random;
    @BeforeEach
    public void setUp() {
        challengeGeneratorService = new ChallengeGeneratorServiceImp
        l(random);
    }
    @Test
    public void generateRandomFactorIsBetweenExpectedLimits() {
        // 89 is max - min range
        given(random.nextInt(89)).willReturn(20, 30);
        // when we generate a challenge
        Challenge challenge = challengeGeneratorService.randomChallenge();
        // then the challenge contains factors as expected
        then(challenge).isEqualTo(new Challenge(31, 41));
    }
}
```

In the previous chapter, you learned how you can use Mockito to replace the behavior of a given class with the `@Mock` annotation and the `MockitoExtension` class for JUnit 5. In this test, you need to replace the behavior of an object, not a class. You can use `@Spy` to stub an object. The Mockito extension will help create a `Random` instance using the empty constructor and stubbing it for you to override the behavior. This is the simplest way to get the test to work, since the basic Java classes implementing random generators do not work on interfaces (which you could then simply *mock* instead of *spy*).

Normally, you initialize what you need for all the tests in a method annotated with `@BeforeEach` so this happens before each test starts. Here, you construct the service implementation passing this stub object.

The only test method sets up the preconditions with `given()` following a BDD style. The way to generate random numbers between 11 and 99 is to get a random number between 0 and 89 and add 11 to it. Therefore, you know that the random object should be called with 89 to generate numbers in the range `11, 100`, so you override that call to return `20` when it's called the first time and `30` the second time. Then, when you call `randomChallenge()`, you expect it to get 20 and 30 as random numbers from `Random` (the stubbed object) and therefore return a `Challenge` object with some random numbers, say `31` and `41`.

So, you made a test that obviously fails when you run it. Let's try it; you can use your IDE or a Maven command from the project's root folder.

```
multiplication$ ./mvnw test
```

As expected, the test will fail. See the result in Listing 3-12.

Listing 3-12. Error Output After Running the Test for the First Time

```
Expecting:
 <null>
to be equal to:
 <Challenge(factorA=20, factorB=30)>
but was not.
Expected :Challenge(factorA=20, factorB=30)
Actual   :null
```

Now, you only need to make the test pass. In this case, the solution is quite simple, and you needed to figure it out while implementing the test. Later, you'll see more valuable cases of TDD, but this one helps to get started with this way of working. See Listing 3-13.

Listing 3-13. Implementing a Valid Logic to Generate Challenges

```
@Service
public class ChallengeGeneratorServiceImpl implements
ChallengeGeneratorService {
```

```
    private final static int MINIMUM_FACTOR = 11;
    private final static int MAXIMUM_FACTOR = 100;
    // ...
    private int next() {
        return random.nextInt(MAXIMUM_FACTOR - MINIMUM_FACTOR) +
        MINIMUM_FACTOR;
    }
    @Override
    public Challenge randomChallenge() {
        return new Challenge(next(), next());
    }
}
```

Now, run the test again. It passes this time:

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Test-driven development is just this simple. First, you design the tests, which will fail in the beginning. Then, you implement your logic to make them pass. In real life, you get the most of it when you get help to build the test cases from the people who define the requirements. You can write better tests and, therefore, a better implementation of what you really want to build.

Attempt Verification

To cover the second part of the business requirements, you implement an interface to verify attempts from users. See Listing 3-14.

Listing 3-14. The ChallengeService Interface

```
package microservices.book.multiplication.challenge;

public interface ChallengeService {
    /**
     * Verifies if an attempt coming from the presentation layer is
     correct or not.
     *
     * @return the resulting ChallengeAttempt object
```

```
     */
    ChallengeAttempt verifyAttempt(ChallengeAttemptDTO resultAttempt);
}
```

As you can see, the code is passing a `ChallengeAttemptDTO` object to the `verifyAttempt` method. This class doesn't exist yet. *Data transfer objects* (DTOs) carry data between different parts of the system. In this case, you use a DTO to model the data needed from the presentation layer to create an attempt. See Listing 3-15. An attempt from the user doesn't have the field `correct` and does not need to know about the user's ID. You can also use DTOs to validate data, as you'll see when you build the controllers.

Listing 3-15. The ChallengeAttemptDTO Class

```
package microservices.book.multiplication.challenge;

import lombok.Value;
/**
 * Attempt coming from the user
 */
@Value
public class ChallengeAttemptDTO {
    int factorA, factorB;
    String userAlias;
    int guess;
}
```

Continuing with a TDD approach, you create do-nothing logic in the `ChallengeServiceImpl` interface implementation. This time you use Lombok's `@Value`, a shortcut annotation to create an immutable class with an `all-args-constructor` and `toString`, `equals`, and `hashCode` methods. It'll also set your fields to be `private final`; that's why you didn't need to add that. See Listing 3-16.

Listing 3-16. An Empty ChallengeService Interface Implementation

```
package microservices.book.multiplication.challenge;

import org.springframework.stereotype.Service;

@Service
```

```
public class ChallengeServiceImpl implements ChallengeService {
    @Override
    public ChallengeAttempt verifyAttempt(ChallengeAttemptDTO attemptDTO) {
        return null;
    }
}
```

And now, you write a unit test for this class, verifying that it works for both correct and wrong attempts. See Listing 3-17.

Listing 3-17. Writing the Test to Verify Challenge Attempts

```
package microservices.book.multiplication.challenge;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.BDDAssertions.then;

public class ChallengeServiceTest {
    private ChallengeService challengeService;
    @BeforeEach
    public void setUp() {
        challengeService = new ChallengeServiceImpl();
    }
    @Test
    public void checkCorrectAttemptTest() {
        // given
        ChallengeAttemptDTO attemptDTO =
                new ChallengeAttemptDTO(50, 60, "john_doe", 3000);
        // when
        ChallengeAttempt resultAttempt =
                challengeService.verifyAttempt(attemptDTO);
        // then
        then(resultAttempt.isCorrect()).isTrue();
    }
```

```
    @Test
    public void checkWrongAttemptTest() {
        // given
        ChallengeAttemptDTO attemptDTO =
                new ChallengeAttemptDTO(50, 60, "john_doe", 5000);
        // when
        ChallengeAttempt resultAttempt =
                challengeService.verifyAttempt(attemptDTO);
        // then
        then(resultAttempt.isCorrect()).isFalse();
    }
}
```

The result of multiplying 50 and 60 is 3,000, so the first test case's assertion expects the `correct` field to be true, whereas the second test expects false for a wrong guess (5,000).

Let's now execute the tests. You can use your IDE or a Maven command and specify the name of the test to run.

```
multiplication$ ./mvnw -Dtest=ChallengeServiceTest test
```

The value after `-Dtest=` specifies the name of the test class that you want to run. In this case, the test class is `ChallengeServiceTest`. The test at the end of the command is to actually execute the tests. You'll see output similar to this:

```
[INFO] Results:
[INFO]
[ERROR] Errors:
[ERROR]   ChallengeServiceTest.checkCorrectAttemptTest:28 NullPointer
[ERROR]   ChallengeServiceTest.checkWrongAttemptTest:42 NullPointer
[INFO]
[ERROR] Tests run: 2, Failures: 0, Errors: 2, Skipped: 0
```

As foreseen, both tests will throw a null pointer exception for now because your `verifyAttempt` method in the `ChallengeServiceImpl` class is returning null.

Go back to the service implementation and make it work. See Listing 3-18.

Listing 3-18. Implementing the Logic to Verify Attempts

```
@Override
public ChallengeAttempt verifyAttempt(ChallengeAttemptDTO attemptDTO) {
    // Check if the attempt is correct
    boolean isCorrect = attemptDTO.getGuess() ==
            attemptDTO.getFactorA() * attemptDTO.getFactorB();
    // We don't use identifiers for now
    User user = new User(null, attemptDTO.getUserAlias());
    // Builds the domain object. Null id for now.
    ChallengeAttempt checkedAttempt = new ChallengeAttempt(null,
            user,
            attemptDTO.getFactorA(),
            attemptDTO.getFactorB(),
            attemptDTO.getGuess(),
            isCorrect
    );
    return checkedAttempt;
}
```

You need to create a user or find an existing one, connect that user to the new attempt, and store it in a database. We keep it simple for now. Later, this implementation should take care of more tasks.

Run the test again to verify that it's passing:

```
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time
elapsed: 0.083 s - in microservices.book.multiplication.challenge.
ChallengeServiceTest
```

Again, you used TDD successfully to build the logic to verify the challenge attempts.

The `Users` domain doesn't need any business logic within the scope of the first user story, so let's move to the next layer.

Presentation Layer

This section covers the presentation layer.

REST

Instead of building HTML from the server, we decided to approach the presentation layer as it's normally done in real software projects: with an API layer in between. By doing so, not only can you expose your functionality to other backend services, but you can also keep the backend and the frontend completely isolated. This way, you can start, for example, with a simple HTML page and plain JavaScript and later move to a full frontend framework without changing the backend code.

Among all the possible API alternatives, the most popular is REpresentational State Transfer (REST). It's normally built on top of HTTP, so it uses HTTP verbs to perform the API operations: GET, POST, PUT, DELETE, and so on. You'll build RESTful web services in this book, which are simply web services that conform to the REST architectural style. Therefore, you'll follow some conventions for URLs and HTTP verbs that have become the *de facto* standard. See Table 3-1.

Table 3-1. *Conventions for the REST APIs*

HTTP Verb	Operation on Collection, e.g., /challenges	Operation on Item, e.g., challenges/3
GET	Gets the full list of items	Gets the item
POST	Creates a new item	Not applicable
PUT	Not applicable	Updates the item
DELETE	Deletes the full collection	Deletes the item

There are a few different styles for writing REST APIs. Table 3-1 shows the most basic operations with some convention choices made for this book. There are also multiple aspects of the contents transferred via the API: pagination, null handling, format (e.g., JSON), security, versioning, and so on. If you are curious about how detailed these conventions can become for a real organization, you can look at Zalando's API Guidelines (`https://opensource.zalando.com/restful-api-guidelines/`).

REST APIs with Spring Boot

Building a REST API with Spring is a simple task. There is a specialization of the `@Controller` stereotype intended for building REST controllers called, unsurprisingly, `@RestController`.

To model resources and mappings for different HTTP verbs, you use the `@RequestMapping` annotation. It applies to the class and method levels, so you can simply build your API contexts. To simplify it, Spring provides variants like `@PostMapping`, `@GetMapping`, and so on, so you don't need to specify the HTTP verb.

Whenever you want to pass the body of a request to the method, you use the `@RequestBody` annotation. If you use a custom class, Spring Boot will try to deserialize it, using the type passed to the method. Spring Boot uses a JSON serialization format by default, although it also supports other formats when specified via the `Accept` HTTP header. In the web applications, you'll use all the Spring Boot defaults.

You can also customize the API with request parameters and read values from the request path. Consider this request as an example:

```
GET http://ourhost.com/challenges/5?factorA=40
```

These are its different parts:

- `GET` is the HTTP verb.
- `http://ourhost.com/` is the host where the web server is running. In this example, the application serves from the *root context*, `/`.
- `/challenges/` is an API context created by the application, to provide functionalities around this domain.
- `/5` is called a *path variable*. In this case, it represents the `Challenge` object with identifier `5`.
- `factorA=40` is a request parameter and its value.

To process this request, you could create a controller with `5` as a path variable called `challengeId` and `40` as a request parameter called `factorA`. See Listing 3-19.

Listing 3-19. An Example of Using Annotations to Map REST API URLs

```
@RestController
@RequestMapping("/challenges")
class ChallengeAttemptController {
    @GetMapping("/{challengeId}")
    public Challenge getChallengeWithParam(@PathVariable("challengeId")
    Long challengeId,
                                           @RequestParam("factorA") int
                                           factorA) {...}
}
```

The offered functionality doesn't stop there. You can also validate the requests given that REST controllers integrate with the `jakarta.validation` API. This means you can annotate the classes used during deserialization to avoid empty values or force numbers to be within a given range when you get requests from the client, just as examples.

Don't worry about the number of new concepts introduced. You learn about them, with practical examples, over the following sections.

Designing the APIs

You can use the requirements to design the functionalities you need to expose in the REST API.

- An interface to get a random, medium complexity multiplication
- An endpoint to send a guess for a given multiplication from a given user's alias

These are a read operation for challenges and an action to create attempts. Keeping in mind that multiplication challenges and attempts are different resources, you can split the API in two and assign the corresponding verbs to these actions:

- `GET /challenges/random` will return a randomly generated challenge.
- `POST /attempts/` will be the endpoint to send an attempt to solve a challenge.

Both resources belong to the `Challenges` domain. Eventually, you will also need a `/users` mapping to perform operations with the users, but we're leaving that for later since you don't need it to complete the first requirements (user story).

API-First Approach It's normally a good practice to define and discuss the API contract in your organization before implementing it. You should include the endpoints, HTTP verbs, allowed parameters, and request and response body examples. This way, other developers and clients can verify if the exposed functionality is what they need and give you feedback before you waste time implementing the wrong solution. This strategy is known as *API First*, and there are industry standards to write the API specifications, like OpenAPI.

If you want to know more about *API First* and *OpenAPI*, see `https://swagger.io/resources/articles/adopting-an-api-first-approach/`, from Swagger, the original creators of the specification.

Your First Controller

Now you'll create a controller that generates a random challenge. You already have that operation in the service layer, so you only need to use that method from the controller. That's what you should do in the presentation layer: keep it isolated from any business logic. You'll use it only to model the API and validate the data. See Listing 3-20.

Listing 3-20. The ChallengeController Class

```
package microservices.book.multiplication.challenge;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;
/**
 * This class implements a REST API to get random challenges
 */
@Slf4j
@RequiredArgsConstructor
```

```
@RestController
@RequestMapping("/challenges")
class ChallengeController {
    private final ChallengeGeneratorService challengeGeneratorService;
    @GetMapping("/random")
    Challenge getRandomChallenge() {
        Challenge challenge = challengeGeneratorService.randomChallenge();
        log.info("Generating a random challenge: {}", challenge);
        return challenge;
    }
}
```

The `@RestController` annotation tells Spring that this is specialized component modeling a REST controller. It's a combination of `@Controller` and `@ResponseBody`, which instructs Spring to put the result of this method as the HTTP response body. As a default in Spring Boot and if not instructed otherwise, the response will be serialized as JSON and included in the response body.

Note also that a `@RequestMapping("/challenges")` was added at the class level, so all mapping methods will have this added as a prefix.

There are also two Lombok annotations in this controller:

- `@RequiredArgsConstructor` creates a constructor with a `ChallengeGeneratorService` as the argument since the field is private and final, which Lombok understands as required. Spring uses dependency injection, so it'll try to find a bean implementing this interface and wire it to the controller. In this case, it'll take the only candidate, the service `ChallengeGeneratorServiceImpl`.
- `Slf4j` creates a logger named `log`. You use it to print a message to console with the generated challenge.

The `getRandomChallenge()` method has the `@GetMapping("/random")` annotation. It means that this method will handle `GET` requests to the context `/challenges/random`, the first part coming from the class-level annotation. It simply returns a `Challenge` object.

Let's now run the web application again and do a quick API test. From your IDE, run the `MultiplicationApplication` class or, from the console, use `mvnw spring-boot:run`.

Using HTTPie (see Chapter 2), you can try your new endpoint by doing a simple GET request to `localhost` (your machine) on port 8080 (Spring Boot's default). See Listing 3-21.

Listing 3-21. Making a Request to the Newly Created API

```
$ http localhost:8080/challenges/random
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Fri, 26 May 2023 11:21:40 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked
{
    "factorA": 74,
    "factorB": 92
}
```

You got an HTTP response with its header and body, a nicely serialized JSON representation of a challenge object. You did it! The application is finally doing something.

How Automatic Serialization Works

When covering how automatic configuration works in Spring Boot, you had a look at the example of the Tomcat embedded server, and we mentioned that there are many more autoconfiguration classes included as part of the `spring-boot-autoconfigure` dependency. Therefore, this other piece of *magic* involved in taking care of serializing a `Challenge` into a proper JSON HTTP response should no longer be a mystery to you. In any case, let's look at how this works since it's a core concept of the web module in Spring Boot. Also, it's quite common to customize this configuration in real life.

A lot of important logic and defaults for the Spring Boot Web module live in the `WebMvcAutoConfiguration` class (https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/servlet/WebMvcAutoConfiguration.java). This class collects all available HTTP message converters in the context together for later use. You can see a fragment of this class in Listing 3-22.

Listing 3-22. A Fragment of WebMvcAutoConfiguration Class Provided by Spring Web

```
@Override
public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
    this.messageConvertersProvider
            .ifAvailable((customConverters) -> converters.
addAll(customConverters.getConverters()));
}
```

The `HttpMessageConverter` interface (https://github.com/spring-projects/spring-framework/blob/main/spring-web/src/main/java/org/springframework/http/converter/HttpMessageConverter.java) is included in the core `spring-web` artifact. It defines the media types supported by the converter, which classes can convert to and from, and the `read` and `write` methods to do conversions.

Where are these converters coming from? More autoconfiguration classes. Spring Boot includes a `JacksonHttpMessageConvertersConfiguration` class (https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/http/JacksonHttpMessageConvertersConfiguration.java) that has some logic to load a bean of type `MappingJackson2HttpMessageConverter`. This logic is conditional on the presence of the `ObjectMapper` class in the classpath. That one is a core class of the Jackson libraries, the most popular implementation of JSON serialization for Java. The `ObjectMapper` class is included in the `jackson-databind` dependency. The class is in the classpath because its artifact is a dependency included in `spring-boot-starter-json`, which is itself included in the `spring-boot-starter-web`.

Again, it's easier to understand all this with a diagram. See Figure 3-6.

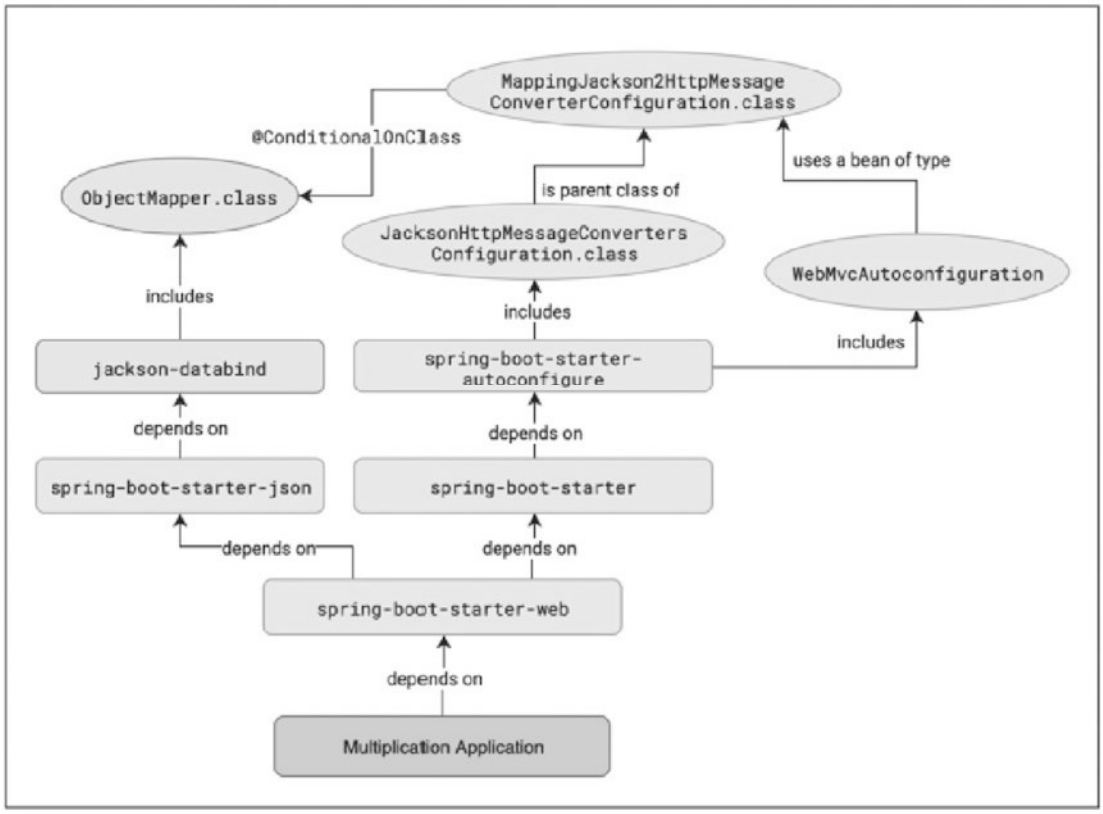

Figure 3-6. *Spring Boot Web JSON autoconfiguration*

The default `ObjectMapper` bean is configured in the `JacksonAutoConfiguration` class (https://github.com/spring-projects/spring-boot/blob/main/spring-boot-project/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jackson/JacksonAutoConfiguration.java). Everything there is set up in a flexible way. If you want to customize a specific feature, you don't need to consider this whole hierarchy. Normally, it's just a matter of overriding default beans.

For instance, if you wanted to change the JSON property naming to be snake-case instead of camel-case, you could declare a custom `ObjectMapper` in the app configuration that will be loaded instead of the default one. That's what Listing 3-23 does.

Listing 3-23. Injecting Beans in the Context to Override Defaults in Spring Boot

```
@SpringBootApplication
public class MultiplicationApplication {
    public static void main(String[] args) {
        SpringApplication.run(MultiplicationApplication.class, args);
    }
    @Bean
    public ObjectMapper objectMapper() {
        var om = new ObjectMapper();
        om.setPropertyNamingStrategy(PropertyNamingStrategy.SNAKE_CASE);
        return om;
    }
}
```

Normally, you would add this bean declaration in a separated class annotated with `@Configuration`, but this piece of code is good enough for this quick example. If you run the app again and call the endpoint, you'll get the factor properties in snake-case. See Listing 3-24.

Listing 3-24. Verifying Spring Boot Configuration Changes with a New Request

```
$ http  localhost:8080/challenges/random
```

As you see, it's really easy to customize Spring Boot configuration by overriding beans. This specific case works because the default ObjectMapper is annotated with `@ConditionalOnMissingBean`, which makes Spring Boot load the bean only if there is no other bean of the same type defined in the context. Remember to remove this custom `ObjectMapper` since you use just Spring Boot defaults for now.

You might be missing the TDD approach for these controllers. The reason that we introduced a simple controller implementation is that it's easier for you to grasp the concepts about how controllers work in Spring Boot before diving into the testing strategies.

Testing Controllers with Spring Boot

This second controller will implement the REST API to receive attempts to solve challenges from the frontend. For this one, it's time to go back to a test-driven approach. First, create an empty shell of the new controller. See Listing 3-25.

Listing 3-25. An Empty Implementation of the ChallengeAttemptController

```
package microservices.book.multiplication.challenge;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
/**
 * This class provides a REST API to POST the attempts from users.
 */
@Slf4j
@RequiredArgsConstructor
@RestController
@RequestMapping("/attempts")
class ChallengeAttemptController {
    private final ChallengeService challengeService;
}
```

Similar to the previous implementation, Lombok adds a constructor with the service interface by annotating it with `@RequiredArgsConstructor`. Spring will inject the corresponding bean `ChallengeServiceImpl`. The `ChallengeAttemptController` class is a Spring `@RestController` component that handles incoming HTTP requests. Spring's core feature is the ability to manage and wire up beans in a Spring application context. When your Spring Boot application starts, it performs a component scan to find classes annotated with various stereotype annotations, like `@Controller`, `@Service`, `@Repository`, and in this case, `@RestController`. This phase is known as the *component scan*.

Now let's write a test with the expected logic. Keep in mind that testing a controller requires a slightly different approach since there is a web layer in between. Sometimes you'll want to verify features such as validation, request mapping, or error handling, which are configured by you but provided by Spring Boot. Therefore, you'll normally want a unit test that covers not only the class itself but also all these features around it.

In Spring Boot, there are multiple ways of implementing a controller test:

- *Without running the embedded server*—You can use `@SpringBootTest` without parameters or, even better, `@WebMvcTest` to instruct Spring to selectively load only the required configuration instead of the whole application context. Then, you simulate requests with a dedicated tool included in the Spring Test module, `MockMvc`.
- *Running the embedded server*—In this case, you use `@SpringBootTest` with its `webEnvironment` parameter set to `RANDOM_PORT` or `DEFINED_PORT`. Then, you must make real HTTP calls to the server. Spring Boot includes a `TestRestTemplate` class with some useful features to perform these test requests. This option is good when you want to test some web server configuration you may have customized (e.g., custom Tomcat configuration).

The best option is usually the first one and choosing a fine-grained configuration with `@WebMvcTest`. You get all the configuration surrounding your controller without taking extra time to boot up the server for each test. If you want to learn more about all these different options, check out this blog about how to test a controller in Spring Boot (https://thepracticaldeveloper.com/guide-spring-boot-controller-tests/).

You could write a test for a valid request and an invalid one, as shown in Listing 3-26.

Listing 3-26. Testing the Expected ChallengeAttemptController Logic

```
package microservices.book.multiplication.challenge;

import microservices.book.multiplication.user.User;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.json.
AutoConfigureJsonTesters;
```

```
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.json.JacksonTester;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.mock.web.MockHttpServletResponse;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;

import static org.assertj.core.api.BDDAssertions.then;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.post;

@ExtendWith(SpringExtension.class)
@AutoConfigureJsonTesters
@WebMvcTest(ChallengeAttemptController.class)
class ChallengeAttemptControllerTest {

    @MockBean
    private ChallengeService challengeService;

    @Autowired
    private MockMvc mvc;

    @Autowired
    private JacksonTester<ChallengeAttemptDTO> jsonRequestAttempt;

    @Autowired
    private JacksonTester<ChallengeAttempt> jsonResultAttempt;

    @Test
    void postValidResult() throws Exception {

        // given
        User user = new User(1L, "john");
        long attemptId = 5L;
```

```
        ChallengeAttemptDTO attemptDTO = new ChallengeAttemptDTO(50, 70,
        "john", 3500);
        ChallengeAttempt expectedResponse = new ChallengeAttempt(attemptId,
        user, 50, 70, 3500, true);
        given(challengeService
                .verifyAttempt(eq(attemptDTO)))
                .willReturn(expectedResponse);

        // when
        MockHttpServletResponse response = mvc.perform(
                post("/attempts").contentType(MediaType.APPLICATION_JSON)
                        .content(jsonRequestAttempt.write(attemptDTO).
                        getJson()))
                .andReturn().getResponse();

        // then
        then(response.getStatus()).isEqualTo(HttpStatus.OK.value());
        then(response.getContentAsString()).isEqualTo(
                jsonResultAttempt.write(
                        expectedResponse
                ).getJson());
    }

@Test
    void postInvalidResult() throws Exception {
        // given an attempt with invalid input data
        ChallengeAttemptDTO attemptDTO = new ChallengeAttemptDTO(2000, -70,
        "john", 1);

        // when
        MockHttpServletResponse response = mvc.perform(
                post("/attempts").contentType(MediaType.APPLICATION_JSON)
                        .content(jsonRequestAttempt.write(attemptDTO).
                        getJson()))
                .andReturn().getResponse();

        // then
```

```
        then(response.getStatus()).isEqualTo(HttpStatus.BAD_REQUEST.
        value());
    }
}
```

There are a few new annotations and helper classes in this code. Let's review them one by one:

- `@ExtendWith(SpringExtension.class)` makes sure that the JUnit 5 test loads the extensions for Spring so you can use a test context.
- `JacksonTester` may be used to serialize and deserialize objects using the same configuration (i.e., ObjectMapper) as the app would do at runtime. `@AutoConfigureJsonTesters` tells Spring to configure beans of type `JacksonTester` for some fields you declare in the test. In this case, `@Autowired` injects two `JacksonTester` beans from the test context. Spring Boot, when instructed via this annotation, builds these utility classes.
- `@WebMvcTest`, with the controller class as a parameter, makes Spring treat this as a presentation layer test. Thus, it'll load only the relevant configuration around the controller: validation, serializers, security, error handlers, and so on (see https://docs.spring.io/spring-boot/docs/current/reference/html/test-auto-configuration.html for a full list of included autoconfiguration classes).
- `@MockBean` comes with the Spring Boot Test module and helps you develop proper unit tests by allowing you to mock other layers and beans you're not testing. In this case, you replace the service bean in the context by a mock. You set the expected return values within the test methods, using `BDDMockito`'s `given()`.
- `@Autowired` might be familiar to you. It's a basic annotation in Spring to make it inject (or wire) a bean in the context to the field. It used to be common in all classes using Spring, but since version 4.3, it can be omitted from fields if they are initialized in a constructor and the class has only one constructor.

- The `MockMvc` class is used in Spring to simulate requests to the presentation layer when you make a test that doesn't load a real server. It's provided by the test context so you can just inject it in your test.

Valid Attempt Test

Now you can focus on the test cases and how to make them pass. The first test sets up the scenario for a valid attempt. It creates the DTO that acts as the data sent from the API client with a valid result. It uses BDDMockito's `given()` to specify that, when the service (a mocked bean) is called with an argument equal to (Mockito's `eq`) the DTO, it should return the expected `ChallengeAttempt` response.

You build the POST request with the static method `post` included in the helper class `MockMvcRequestBuilders`. The target is the expected path `/attempts`. The content type is set to `application/json`, and its body is the serialized DTO in JSON. You use the wired `JacksonTester` for serialization. Then, `mvc` does the request via `perform()`, and you get the response calling to `.andReturn()`. You could also call the `andExpect()` method instead if you use `MockMvc` for assertions, but it's better to do them separately with a dedicated assertions library like AssertJ.

In the last part of the test, you verify that the HTTP status code should be `200 OK` and that the result must be a serialized version of the expected response. Again, you use a `JacksonTester` object for this.

This test fails with a `404 NOT FOUND` when you execute it. See Listing 3-27. There is no implementation for that request, so the server can't simply find a logic to map that POST mapping.

Listing 3-27. The ChallengeAttemptControllerTest Fails

```
org.opentest4j.AssertionFailedError:
expected: 200
 but was: 404
Expected :200
Actual   :404
```

Now, go back to the `ChallengeAttemptController` and implement this mapping. See Listing 3-28.

Listing 3-28. Adding the Working Implementation to ChallengeAttemptController

```
@Slf4j
@RequiredArgsConstructor
@RestController
@RequestMapping("/attempts")
class ChallengeAttemptController {
    private final ChallengeService challengeService;
    @PostMapping
    ResponseEntity<ChallengeAttempt> postResult(@RequestBody
    ChallengeAttemptDTO challengeAttemptDTO) {
        return ResponseEntity.ok(challengeService.verifyAttempt(
        challengeAttemptDTO));
    }
}
```

We're using this new way here to show that there are ways to build different types of responses with the `ResponseEntity` static builder. It's a simple logic that just calls the service layer. The method is annotated with `@PostMapping` without parameters so it will handle a POST request to the context path already set at the class level. Note that here we're using a `ResponseEntity` as the return type instead of using the `ChallengeAttempt` directly. That other option would also work.

That's it! The first test case will pass now.

Validating Data in Controllers

The second test case, `postInvalidResult()`, does not allow the application to accept an attempt with negative or out-of-range numbers. It expects the logic to return a `400 BAD REQUEST`, which is a good practice when the error is on the client side, like this one. See Listing 3-29.

Listing 3-29. Verifying That the Client Gets a BAD REQUEST Status Code

```
// then
then(response.getStatus()).isEqualTo(HttpStatus.BAD_REQUEST.value());
```

If you run it before implementing the POST mapping in the controller, it fails with a `NOT FOUND` status code. With the implementation in place, it also fails. However, in this case, the result is even worse. See Listing 3-30.

Listing 3-30. Posting an Invalid Request Returns a 200 OK Status Code

```
org.opentest4j.AssertionFailedError:
expected: 400
 but was: 200
Expected :400
Actual   :200
```

The application is just accepting the invalid attempt and returning an `OK` status. This is wrong; you should not pass this attempt to the service layer but reject it in the presentation layer. To accomplish this, you can use the Java Bean Validation API (https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#validation-beanvalidation) integrated with Spring).

In the DTO class, you add some Java Validation annotations to indicate the valid inputs. See Listing 3-31. All these annotations are implemented in the `jakarta.validation-api` library, available in the classpath via `spring-boot-starter-validation`. This starter is included as part of the Spring Boot Web starter (`spring-boot-starter-web`).

Listing 3-31. Adding Validation Constraints to the DTO Class

```
package microservices.book.multiplication.challenge;
import lombok.Value;
import jakarta.validation.constraints.*;
/**
 * Attempt coming from the user
 */
@Value
public class ChallengeAttemptDTO {
    @Min(1) @Max(99)
    int factorA, factorB;
    @NotBlank
```

```
    String userAlias;
    @Positive
    int guess;
}
```

There are a lot of available constraints in that package (https://docs.jboss.org/hibernate/beanvalidation/spec/2.0/api/javax/validation/constraints/package-summary.html). You use `@Min` and `@Max` to define the range of allowed values for the multiplication factors, `@NotBlank` to make sure you always get an alias, and `@Positive` for the guess since you know you're handling only positive results (you can also use a predefined range here).

An important step to make these constraints work is to integrate them with Spring via the `@Valid` annotation in the controller's method argument. See Listing 3-32. If you add this, Spring Boot will analyze the constraints and throw an exception if they don't match.

Listing 3-32. Using the @Valid Annotation to Validate Requests

```
@PostMapping
ResponseEntity<ChallengeAttempt> postResult(
        @RequestBody @Valid ChallengeAttemptDTO challengeAttemptDTO) {
    return ResponseEntity.ok(challengeService.verifyAttempt(challengeAtt
    emptDTO));
}
```

As you might have guessed, there is autoconfiguration to handle the errors and build a predefined response when the object is invalid. By default, the error handler constructs a response with a `400 BAD_REQUEST` status code.

Starting with Spring Boot version 2.3, the validation messages are no longer included in the error response by default. This might be confusing for the callers since they don't know exactly what's wrong with the request. The reason to not include them is that these messages could potentially expose information to a malicious API client.

We want to enable validation messages for our educational goal, so we'll add two settings to the `application.properties` file. See Listing 3-33. These properties are listed in the official Spring Boot docs (https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix.application-properties.server), and you'll see what they do soon.

Listing 3-33. Adding Validation Logging Configuration to the application.properties File

```
server.error.include-message=always
server.error.include-binding-errors=always
```

To verify all the validation configuration, run the test again. This time it'll pass, and you'll see some extra logs, as shown in Listing 3-34.

Listing 3-34. An Invalid Request Causes Now the Expected Result

```
[Field error in object 'challengeAttemptDTO' on field 'factorB': rejected
value [-70];
[...]
[Field error in object 'challengeAttemptDTO' on field 'factorA': rejected
value [2000];
[...]
```

The controller handling REST API calls for users to send attempts is working now. If you start the application again, you can play with this new endpoint via the HTTPie command. First, you ask for a random challenge, as before. Then, you post an attempt to solve it. See Listing 3-35.

Listing 3-35. Running a Standard Use Case for the Application Using HTTPie Commands

```
$  http -b :8080/challenges/random
{
    "factorA": 52,
    "factorB": 59
}

$ http POST :8080/attempts factorA=58 factorB=92 userAlias=moises
guess=5400
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Sat, 27 May 2023 09:09:38 GMT
Keep-Alive: timeout=60
```

```
Transfer-Encoding: chunked

{
    "correct": false,
    "factorA": 58,
    "factorB": 92,
    "id": null,
    "resultAttempt": 5400,
    "user": {
        "alias": "moises",
        "id": null
    }
}
```

The first command uses the `-b` parameter to print only the body of the response. As you see, you can also omit `localhost`, and HTTPie will use it as default.

As expected, you get a serialized `ChallengeAttempt` object indicating that the result is incorrect. To send the attempt, you use the `POST` argument before the URL. JSON is the default content type in HTTPie, so you can simply pass `key-value` parameters, and this tool will convert it to proper JSON.

You can also try an invalid request to see how Spring Boot handles the validation errors. See Listing 3-36.

Listing 3-36. Error Response Including Validation Messages

```
$ http POST :8080/attempts factorA=58 factorB=92 userAlias=moises
guess=-400
HTTP/1.1 400
Connection: close
Content-Type: application/json
Date: Sat, 27 May 2023 09:09:39 GMT
Transfer-Encoding: chunked
{
    "error": "Bad Request",
    "errors": [
        {
```

```
            "arguments": [
                {
                    "arguments": null,
                    "code": "guess",
                    "codes": [
                        "challengeAttemptDTO.guess",
                        "guess"
                    ],
                    "defaultMessage": "guess"
                }
            ],
            "bindingFailure": false,
            "code": "Positive",
            "codes": [
                "Positive.challengeAttemptDTO.guess",
                "Positive.guess",
                "Positive.int",
                "Positive"
            ],
            "defaultMessage": "must be greater than 0",
            "field": "guess",
            "objectName": "challengeAttemptDTO",
            "rejectedValue": -400
        }
    ],
    "message": "Validation failed for object='challengeAttemptDTO'. Error
    count: 1",
    "path": "/attempts",
    "status": 400,
    "timestamp": "2023-05-27T09:09:39.212+00:00"
}
```

It's quite a verbose response. The main reason is that all the *binding errors* (those caused by the validation constraints) are added to the error response. This was switched on with `server.error.include-binding-errors=always`. Besides, the root `message` field also gives the client an overall description of what went wrong. This description is omitted by default, but you enabled it with the `server.error.include-message=always` property.

If this response goes to a user interface, you need to parse that JSON response in the frontend, get the fields that are invalid, and maybe display the `defaultMessage` fields. Changing this default message is simple since you can override it with the constraint annotations. Let's modify this annotation in `ChallengeAttemptDTO` and try this again with the same invalid request. See Listing 3-37.

Listing 3-37. Changing the Validation Message

```
@Positive(message = "How could you possibly get a negative result here? Try
again.")
int guess;
```

What Spring Boot does in this case to handle the errors is to sneakily add a `@Controller` to your context: the `BasicErrorController` (see https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/web/servlet/error/BasicErrorController.html). This one uses the `DefaultErrorAttributes` class (https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/web/servlet/error/DefaultErrorAttributes.html) to compose the error response.

In internationalized applications, it is crucial to provide users with meaningful error messages, especially for validation errors. Different languages and cultures might require different phrasing and wording for error messages to be easily understandable. To include validation messages in error responses for i18n purposes, you can customize the configuration of validation messages in Spring Boot by defining and managing them through message source properties files, which can be easily internationalized.

Summary and Achievements

You started this chapter by learning about the requirements of the application you'll build in this book. Then, we sliced the scope and took the first item for development: the functionality to generate a random challenge and allow users to guess the result.

You learned how to create the skeleton of a Spring Boot application and some best practices regarding software design and architecture: three-tier and three-layer architecture, domain-driven design, test-driven/behavior-driven development, basic unit tests with JUnit 5, and REST API design. In this chapter, you focused on the application tier and implemented the domain objects, the business layer, and the presentation layer as a REST API. See Figure 3-7.

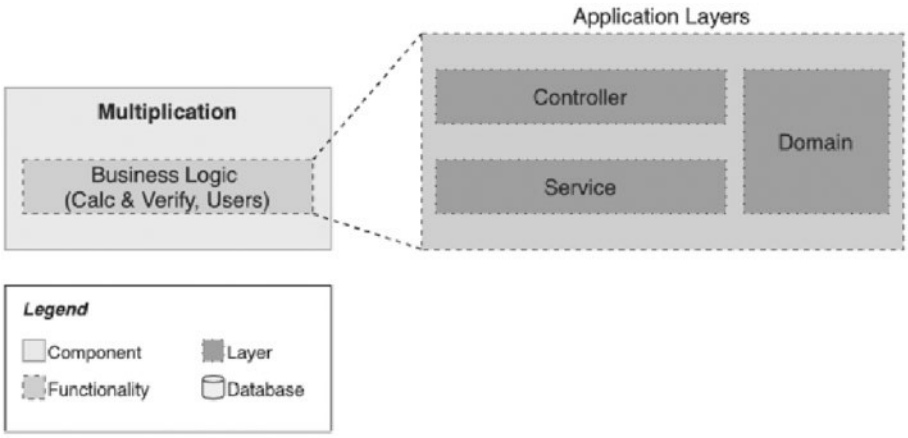

Figure 3-7. *Application status after Chapter 3*

A core concept in Spring Boot was also covered in this chapter: autoconfiguration. Now you know where a big part of the Spring Boot magic lives. In the future, you should be able to find your way through the reference documentation to override other default behaviors in any other configuration class.

The chapter also went through other features in Spring Boot, such as implementing `@Service` and `@Controller` components, testing controllers with `MockMvc`, and validating input via the Java Bean Validation API.

To complete this first web application, you need to build a user interface. Later, you'll also learn about the data layer to make sure you can persist users and attempts.

Chapter's Achievements:

- You learned how to build a properly structured Spring Boot application, following a three-layered design.
- You learned how Spring Boot's autoconfiguration works and the key to unveil its magic, based on two practical examples with supporting diagrams: the Tomcat embedded server and the JSON serialization defaults.
- You modeled an example business case following domain-driven design techniques.
- You developed two of the three layers of the first application (*service*, *controller*) using a test-driven development approach.
- You used the most important Spring MVC annotations to implement a REST API with Spring Boot.
- You learned how to test the controller layer in Spring using `MockMVC`.
- You added validation constraints to your API to protect it against invalid input.