CHAPTER 1

Setting the Scene

A microservices architecture allows you to design software systems as a collection of independent services that communicate through APIs. Each microservice performs a specific business function and is responsible for its data storage, processing, and delivery.

Microservices have become the de facto standard for building complex and scalable systems. This is not surprising, given the numerous benefits of this software architecture style. Microservices offer increased scalability, flexibility, and resilience, as well as the ability to develop and deploy services independently. Additionally, microservices provide the potential for easier maintenance and updates compared to other architecture styles. Mapping microservices into small teams in an organization also gives you a lot of efficiency in development. However, going on the adventure of microservices while knowing only the benefits is the wrong approach.

The microservices approach also leads to the distribution of the system's functionality across multiple microservices, each running independently and communicating with others through a network. As a result, the overall system becomes distributed, consisting of numerous smaller, independent services that work together to deliver the desired functionality. This architecture style allows for greater flexibility, scalability, and resilience, but it also introduces additional complexity, which you need to manage carefully. You can get a lot of knowledge from many books and articles on the Internet, but when you get hands-on with the code, the story changes.

This book covers some of the essential concepts of microservices practically, but only by explaining those concepts. In this book, you build an application where users can exercise their brains by performing mathematical calculations. Users will be presented with a new problem every time they access the application. They provide their alias (a short name) and submit answers to the problem. After they submit their input, the web page will indicate whether their answer is correct.

M. Macero García and T. Telang, *Learn Microservices with Spring Boot 3*,
https://doi.org/10.1007/978-1-4842-9757-5_1

First, you define a use case: an application to build. Then you start with a small monolith based on some sound reasoning. Once you have the minimal application in place, you evaluate whether it's worth moving to microservices and learn how to do that well, including building RESTful APIs, adding security features, and scaling services. The book also introduces essential microservices design principles, such as the single responsibility principle, loose coupling, service autonomy, and bounded contexts from domain-driven design. With the introduction of the second microservice, we analyze the options you have for their communication. Then, you can describe and implement the event-driven architecture pattern to reach loose coupling by informing other parts of the system about what happened, instead of explicitly calling others to action. Event-driven architecture is a powerful pattern for designing reactive and resilient microservices.

You will also explore commonly used communication patterns that facilitate seamless coordination and collaboration between microservices. These patterns can be classified into two categories—synchronous communication and asynchronous communication.

For *synchronous communication,* you will study patterns such as request-response, message broker, API gateway, choreography, and orchestration. These patterns enable real-time interactions between services, ensuring that requests and responses are handled in a coordinated manner.

On the other hand, for *asynchronous communication,* you will delve into event sourcing and publish-subscribe patterns. These patterns allow services to communicate asynchronously by leveraging events and messages, promoting loose coupling and enabling independent processing.

By examining these communication patterns, you'll gain a comprehensive understanding of how microservices can effectively interact and work together, both in synchronous and asynchronous scenarios. This knowledge empowers you to design robust and efficient communication mechanisms within your microservices architecture.

When you reach that point, you'll notice that a poorly designed distributed system has some flaws that you must fix with popular patterns: service discovery, routing, load balancing, traceability, and more.

You will also explore the circuit breaker pattern as another important communication pattern for microservices. The circuit breaker pattern acts as a safeguard against cascading failures and helps manage the resilience of communication between services.

By studying the circuit breaker pattern, you will understand how it detects failures in remote service calls and provides a fail-fast mechanism. It helps prevent further calls to a failing service, thus reducing the impact of failures on the overall system. The circuit breaker pattern also includes mechanisms for monitoring the health of services and adapting to changing conditions.

Incorporating the circuit breaker pattern into your communication strategies ensures that your microservices ecosystem remains stable, resilient, and responsive, even in the face of potential failures or degradation in service performance.

Adding them individually to your codebase instead of presenting all of them together helps you understand these patterns. You can also prepare these microservices for cloud deployment by understanding cloud-native architectures and containerization, learning about cloud-native architectures and containerization, using Docker for containerization and Kubernetes for orchestration, and comparing the different platform alternatives to run the applications. By following these steps, you can gain a practical understanding of microservices and learn how to design, implement, and deploy microservices-based systems.

Throughout this book you will build microservices from scratch using Spring Boot. The advantage of going step-by-step, pausing when needed to nail down the concepts, is that you will understand which problem each tool is trying to solve. That's why the evolving example is an essential part of this book. You can also grasp the concepts without coding a single line since the source code is presented and explained throughout the chapters.

All the code included in this book is available on GitHub in the organization `Book-Microservices-v3`. Multiple repositories are available and divided into chapters and sections, making it easier to see how the application evolves. The book includes notes with the version covered in each part. By following these steps, you can gain a practical understanding of microservices and learn how to design, implement, and deploy microservices-based systems.

Who Are You?

Let's start with this: how interesting will this book be to you? This book is practical, so let's play this game. If you identify with any of these statements, this book might be good for you:

- "I would like to learn how to build microservices with Spring Boot and how to use the related tools."
- "Everybody is talking about microservices, but I have no clue what a microservice is yet. I have read only theoretical explanations or just hype-enforcing articles. I can't understand the advantages, even though I work in IT…."
- "I would like to learn how to design and develop Spring Boot applications, but all I find are either quick-start guides with too simple examples or lengthy books that resemble the official documentation. I would like to learn the concepts following a more realistic, project-guided approach."
- "I got a new job, and they're using a microservices architecture. I've been working mainly in big, monolithic projects, so I'd like to have some knowledge and guidance to learn how everything works there, as well as the pros and cons of this architecture."
- "Every time I go to the cafeteria, developers talk about microservices, gateways, service discovery, containers, resilience patterns, and so on. I can't socialize anymore with my colleagues if I don't get what they're saying." (This one is a joke; don't read this book because of that, especially if you're not interested in programming.)

Regarding the knowledge required to read this book, the following topics should be familiar to you:

- Java (we use Java 17)
- Spring (you don't need strong experience, but you should know at least how dependency injection works)
- Maven (if you know Gradle, you'll be fine as well)

How Is This Book Different from Other Books and Guides?

Software developers and architects read many technical books and guides because they're interested in learning new technologies, or their work requires it. They need to do that anyway since it's a constantly changing world. You can find all kinds of books and guides out there. You can learn quickly from the good ones, which teach you not only how to do stuff but also explain why you should do it that way.

Using the latest techniques just because they're new is the wrong way to go about it; you need to understand the reasoning behind them, so you use them in the best way possible.

This book uses that philosophy: it navigates through the code and design patterns, explaining the reasons to follow one way and not others.

Learning: An Incremental Process

If you look at the guides available on the Internet, you'll quickly notice that they are not real-life examples. Usually, when you apply those cases to more complex scenarios, they don't fit. Guides need to be deeper to help you build something real. Books, on the other hand, are much better at that. Plenty of good books explain concepts around an example; they are good because applying theoretical concepts to code is only sometimes possible if you don't see the code. The problem with some of these books is that they're less practical than guides. It helps when you read them first to understand the concepts and then code (or see) the example, which we have frequently given as a whole piece. It isn't easy to put concepts into practice when you view the final version directly. This book stays practical and starts with code. It evolves through the chapters so that you grasp the ideas one by one. We cover the problem before exposing the solution.

Because of this incremental way of presenting concepts, this book also allows you to code as you learn and to reflect on the challenges by yourself.

Is This a Guide or a Book?

The pages you have in front of you can't be called a guide: it will take you 15 or 30 minutes to finish them. Besides, each chapter introduces all the required topics to lay the foundation for the new code additions. But this is not the typical book either, in which

you go through isolated concepts, illustrated with scattered code fragments explicitly made for that situation. Instead, you start with a real-life application that is yet to be optimal, and you learn how to evolve it after learning about the benefits you can extract from that process.

That does not mean you can't just sit down and read it, but it's better if your code simultaneously plays with the options and alternatives presented. That's part of the book that makes it like a guide. To keep it simple, we call this a book from here onward.

From Basics to Advanced Topics

This book focuses on some essential concepts that build on the rest of the topics (Chapter 2)—Spring Boot, testing, logging, and so on. Then, it covers how to design and implement a production-ready Spring Boot application using a well-known layered design. It dives into how to implement a REST API, business logic, and database repositories (Chapters 3 and 5). While doing that, you see how Spring Boot works internally, so it's not magic to you anymore. You also learn how to build a basic frontend application with React (Chapter 4) because that will help you visualize how the backend architecture impacts the frontend. After that, the book enters the microservices world by introducing a second piece of functionality in a different Spring Boot app. The practical example helps you analyze the factors you should examine before deciding to move to microservices (Chapter 6). Then, you learn the differences between communicating microservices synchronously and asynchronously and how an event-driven architecture can help you keep your system components decoupled (Chapter 7). From there, the book takes you through the journey of tools and frameworks applicable to distributed systems to achieve necessary nonfunctional requirements: resilience, scalability, traceability, and deployment to the cloud, among others (Chapter 8).

If you are already familiar with Spring Boot applications and how they work, you can go quickly through the first chapters and focus more on the second part of the book. More advanced topics are covered in the second part, including event-driven design, service discovery, routing, distributed tracing, testing with Cucumber, and so on. However, pay attention to the foundations set up in the first part: test-driven development, the focus on the minimum viable product (MVP), and monolith-first.

Skeleton with Spring Boot, the Professional Way

First, the book guides you through creating an application using Spring Boot. The contents mainly focus on the backend side, but you will create a simple web page with React to demonstrate how to use the exposed functionality as a REST API.

It's important to point out that you don't create "shortcut code" to showcase Spring Boot features: that's not the objective of this book. It uses Spring Boot as a vehicle to teach concepts, but you could use any other framework, and the ideas of this book would still be valid.

You learn how to design and implement the application following the well-known three-tier, three-layer pattern. You do this supported by an incremental example with hands-on code. While writing the applications, you'll pause a few times to get into the details about how Spring Boot works with so little code (autoconfiguration, starters, etc.).

Test-Driven Development

In the first chapters, you use test-driven development (TDD) to map the prerequisites presented to technical features. This book tries to show this technique so you can see the benefits from the beginning: why it's always a good idea to think about the test cases before writing your code. JUnit 5, AssertJ, and Mockito will serve you to build useful tests efficiently.

The plan is the following: you'll learn how to create the tests first, then make them fail, and finally implement the logic to make them work.

Microservices

Once you have your first application ready, we introduce a second one that will interact with the existing functionality. From that moment on, you'll have a microservices architecture. It doesn't make any sense to try to understand the advantages of microservices if you have only one of them. The real-life scenarios are always distributed systems with functionality split into different services. As usual, to keep it practical, you'll analyze the specific situation for your case study, so you'll see if moving to microservices fits your needs.

The book covers not only the reasons to split the system but also the disadvantages that come with that choice. Once you make the decision to move to microservices, you'll learn which patterns you should use to build a good architecture for the distributed system: service discovery, routing, load balancing, distributed tracing, containerization, and some other supporting mechanisms.

Event-Driven Systems

An additional concept that does not always need to come with microservices is the *event-driven architecture*. This book uses it since it's a pattern that fits well into a microservice architecture, and you'll make your choice based on good examples. You learn about the differences between synchronous and asynchronous communication, as well as their main pros and cons.

This asynchronous way of thinking introduces new ways of designing code, with *eventual consistency* as one of the key changes to embrace. You'll look at it while coding your project, using RabbitMQ or Apache Kafka to send and receive messages between microservices. Both of these messaging systems allow for distributed and decoupled processing within your architecture.

Nonfunctional Requirements

When you build an application in the real world, you must take into account some requirements that are not directly related to functionalities, but that prepare your system to be more robust, to keep working in the event of failures, or to ensure data integrity.

Many of these *nonfunctional requirements* are related to things that can go wrong with your software: network failures that make part of your system unreachable, a high traffic volume that collapses your backend capacity, external services that don't respond, and so on.

In this book, you learn how to implement and verify patterns to make the system more resilient and scalable. In addition, it discusses the importance of data integrity and the tools that help you guarantee it.

The good part about learning how to design and solve all these nonfunctional requirements is that it's knowledge applicable to any system, no matter the programming language and frameworks you're using.

Online Content

For this second edition of the book, we decided to create an online space where you can keep learning new topics related to microservice architectures. On this web page, you'll find new guides that extend the practical use case covering other important aspects of distributed systems. Additionally, new versions of the repositories using up-to-date dependencies are published there.

The first guide that you'll find online is about testing a distributed system with Cucumber (https://cucumber.io/). This framework helps you build human-readable test scripts to make sure your functionalities work end-to-end.

Visit https://github.com/Book-Microservices-v3/book-extras for all the extra content and new updates about the book.

Summary

This chapter introduced the main goals of this book: to teach you the main aspects of a microservice architecture by starting simple and then growing your knowledge through the development of a sample project.

We also briefly covered the main content of the book: from monolith-first to microservices with Spring Boot, test-driven development, event-driven systems, common architecture patterns, nonfunctional requirements, and end-to-end testing with Cucumber (online).

The next chapter starts with the first step of your learning path: a review of some basic concepts.

CHAPTER 2

Basic Concepts

This book follows a practical approach, so most of the tools covered are introduced as you need them. However, we'll go over some core concepts separately because they're either the foundations of the evolving example or used extensively in the code examples, namely, Spring, Spring Boot, testing libraries, Lombok, and logging. These concepts deserve a separate introduction to avoid long interruptions in your learning path, which is why this chapter includes an overview of them.

Remember that these sections intend to provide a sound knowledge base of these frameworks and libraries. The primary objective of this chapter is to refresh the concepts in your mind (if you already learned them) or grasp the basics so that you don't need to consult external references before reading the rest of the chapters.

Spring

The Spring Framework (https://spring.io/projects/spring-framework) is a popular open-source application development framework consisting of a collection of libraries and tools that provide a comprehensive programming and configuration model for developing enterprise-level Java applications. It offers a wide range of features and modules—such as dependency injection, aspect-oriented programming, data access, transaction management, validation, internationalization, web development, and more—that can be used individually or in combination to develop robust and scalable applications. It is widely used by developers and organizations worldwide, and it has become a standard for building complex Java-based applications. At the time of writing the book, the latest version of the Spring framework is 6.0.9.

The Spring Framework primarily supports the Java programming language. However, it also provides support for other JVM-based programming languages such as Groovy and Kotlin.

M. Macero García and T. Telang, *Learn Microservices with Spring Boot 3*,
https://doi.org/10.1007/978-1-4842-9757-5_2

Spring Framework is popular for several reasons:

1. **Modular and flexible**: Spring Framework is highly modular and provides a wide range of features that can be used individually or in combination. It allows developers to use only the modules they need, making it flexible and lightweight.
2. **Dependency injection (DI)**: Spring Framework supports powerful and flexible dependency injection, allowing developers to easily manage dependencies between components of an application, thus reducing code coupling and improving maintainability.
3. **Aspect-Oriented Programming (AOP)**: Spring AOP provides a way to add cross-cutting concerns to an application, such as logging, caching, and security, without having to modify the core application logic.
4. **Data access**: Spring provides several modules for database access and object-relational mapping, including Spring Data JDBC (`https://spring.io/projects/spring-data-jdbc`) and Spring Data JPA.
5. **Transaction management**: Spring provides a powerful transaction management framework, which supports both declarative and programmatic transaction management.
6. **Web applications**: Spring provides robust support for building web applications. The Spring framework offers various features and modules that facilitate the development of web-based solutions with ease and efficiency. At the heart of web application development with Spring lies Spring MVC (Model-View-Controller). Spring MVC is a powerful framework that follows the MVC architectural pattern, enabling developers to build scalable and maintainable web applications. It provides a structured approach to handling HTTP requests and mapping them to appropriate controllers, processing business logic, and rendering views for the user.

7. **Security**: Spring Security (https://spring.io/projects/spring-security) provides a comprehensive security framework, which supports authentication, authorization, and other security-related features.
8. **Large community and documentation**: Spring Framework has a large and active community of developers and users, who continuously contribute to the development and improvement of the framework, provide support, and share knowledge and resources.
9. **Integration with other frameworks and libraries**: Spring Framework provides seamless integration with other technologies and frameworks, such as Hibernate, JPA, REST, and more, making it easier to develop and deploy applications.
10. **Testability**: Spring Framework makes it easier to write unit and integration tests, which helps ensure the quality and reliability of the application.

Spring provides lots of built-in implementations for many aspects of software development, such as the following:

- Spring Data (https://spring.io/projects/spring-data) simplifies data access for relational and NoSQL databases. It provides support for several popular data access technologies, such as JDBC, Hibernate, JPA, and MongoDB. It makes it easy to work with databases and perform CRUD (Create, Read, Update and Delete) operations.
- Spring Batch (https://spring.io/projects/spring-batch) is a framework for building batch processing applications. It is a lightweight, comprehensive framework that provides developers with the tools to create batch jobs that can process large amounts of data efficiently and reliably.
- Spring Security (https://spring.io/projects/spring-security) is a security framework that abstracts security features to applications.

- Spring Cloud (https://spring.io/projects/spring-cloud) provides tools for developers to quickly build some of the common patterns in distributed systems.
- Spring Integration (https://spring.io/projects/spring-integration)is an implementation of enterprise integration patterns. It facilitates integration with other enterprise applications using lightweight messaging and declarative adapters.
- Spring MVC is a web framework for building Java web applications based on the Model-View-Controller design pattern. It provides a flexible and powerful platform for developing scalable and maintainable web applications.
- Spring Web Flow (https://spring.io/projects/spring-webflow) provides a powerful and comprehensive solution for managing complex flows in web applications. It simplifies the development process by abstracting away the low-level details of flow control and state management, allowing developers to focus on creating intuitive and efficient user experiences.
- Spring WebFlux is a reactive web framework for building non-blocking and event-driven applications on the JVM. It provides an alternative to Spring MVC and supports asynchronous and non-blocking API for handling HTTP requests and responses using reactive programming principles.

These built-in implementations make it easier for developers to build robust, scalable, and maintainable applications, while reducing the amount of boilerplate code that needs to be written.

As you can see, Spring is divided into different modules. All the modules are built on top of the core Spring Framework, which establishes a common programming and configuration model for software applications. This model itself is another important reason to choose the framework since it facilitates good programming techniques such as the use of interfaces instead of classes to decouple application layers via dependency injection.

A key topic in Spring is the Inversion of Control (IoC) container, which is supported by the `ApplicationContext` interface (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html). The ApplicationContext interface is a key component of the Spring Framework that provides a powerful and flexible mechanism for managing object dependencies and resources in an application. The IoC container creates this "space" in your application where you, and the framework itself, can put some object instances such as database connection pools, HTTP clients, and so on. These objects, called *beans*, can be later used in other parts of your application, commonly through their public interface to abstract your code from specific implementations. The mechanism to reference one of these beans from the application context in other classes is called *dependency injection*, and in Spring this is possible via various techniques like XML, YAML, Java-based annotations, and JavaConfig. This provides a high degree of flexibility in how the application is structured.

With XML configuration, you can define the various components of your web application, such as controllers, views, and dependencies, in XML files. These files typically have a suffix like `.xml` and follow a defined structure based on the Spring framework's XML schema.

Spring also supports configuration using YAML (YAML Ain't Markup Language). YAML is a human-readable data serialization format that is often used for configuration files in various applications, including Spring. With YAML configuration, you can define your Spring beans, properties, and other configuration elements in a structured and easy-to-read format. YAML provides a more concise and visually appealing alternative to XML for configuration purposes. To use YAML configuration in Spring, you typically create a YAML file (with a `.yml` or `.yaml` extension) and define your configuration elements using indentation and key-value pairs. For example, you can define beans, their properties, and their dependencies in a hierarchical structure within the YAML file. Spring Boot offers built-in support for YAML-based configuration files. Spring Boot automatically loads and parses YAML files to configure the application at startup.

With YAML configuration, you can define your Spring beans, properties, and other configuration elements in a structured and an easy-to-read format. YAML provides a more concise and visually appealing alternative to XML for configuration purposes.

Spring also offers the option to configure web applications using annotations, leveraging the power and simplicity of Java code. Annotations provide a more concise and intuitive way to define the components and their configurations.

Spring also offers the option to configure web applications using annotations, leveraging the power and simplicity of Java code. Annotations provide a more concise and intuitive way to define the components and their configurations.

By using annotations, you can mark your classes, methods, or fields with specific annotations provided by Spring. For example, you can annotate a class with `@Controller` to indicate it as a Spring MVC controller, use `@RequestMapping` to specify request mappings, and use `@Autowired` to inject dependencies.

Using annotations reduces the need for YAML or XML configuration files, as the configuration can be directly embedded in the source code. This approach promotes better code readability, simplifies configuration management, and reduces the overall configuration overhead.

Yet another Java-based configuration approach, known as "JavaConfig," has also gained significant popularity. It allows developers to define their application's configuration using plain Java classes annotated with Spring annotations. This provides a more programmatic and type-safe way of configuring Spring components.

You see all these configuration options in action as you work through the sample application.

Spring Boot

Spring Boot (`https://spring.io/projects/spring-boot`) is a framework that leverages Spring to quickly create stand-alone applications in Java-based languages. It has become a popular tool for building microservices.

Having so many available modules in Spring and other related third-party libraries that can be combined with the framework is powerful for software development. Yet, despite a lot of efforts to make Spring configuration easier, you still need to spend some time to set up everything you need for your application. And, sometimes, you just require the same configuration over and over again. *Bootstrapping* an application, meaning the process to configure your Spring application to have it up and running, is sometimes tedious. The advantage of Spring Boot is that it eliminates most of that process by providing default configurations and tools that are set up automatically for you. The main disadvantage is that if you rely too much on these defaults, you may lose control and awareness of what's happening. We unveil some of the Spring Boot implementations in the book to demonstrate how it works internally so that you can always be in control.

Spring Boot provides some predefined *starter packages* that are like collections of Spring modules and some third-party libraries and tools together. As an example, `spring-boot-starter-web` helps you build a stand-alone web application. It groups the Spring Core Web libraries with Jackson (JSON handling), validation, logging, autoconfiguration, and even an embedded Tomcat server, among other tools. Other starters include `spring-boot-starter-data-jpa`, which includes dependencies for Spring Data JPA, Hibernate, and a database driver, and `spring-boot-starter-test`, which includes dependencies for testing Spring Boot applications.

In addition to starters, *autoconfiguration* plays a key role in Spring Boot. This feature makes adding functionality to your application extremely easy. It also helps ensure that the application follows best practices and recommended design patterns, as the autoconfiguration logic is based on well-established Spring patterns and practices. Following the same example, just by including the web starter, you get an embedded Tomcat server. There's no need to configure anything. This is because the Spring Boot auto configuration classes scan your classpath, properties, components, and so on, and load some extra beans and behavior based on that.

To be able to manage different configuration options for your Spring Boot application, the framework introduces *profiles*. You can use profiles, for example, to set different values for the host to connect to when using a database in a development environment and a production environment. Additionally, you can use a different profile for tests, where you may need to expose additional functions or mock parts of your application. We cover profiles more in detail in Chapter 8.

You'll use the Spring Boot Web and Data starters to quickly build a web application with persistent storage. The Test starter will help you write tests, given that it includes some useful test libraries such as JUnit 5 (`https://junit.org/junit5/`) and AssertJ (`https://assertj.github.io/doc/`). Then, you'll see how to add messaging capabilities to your applications by adding the AMQP starter, which includes a message broker integration (RabbitMQ) that you'll use to implement an event-driven architecture. Chapter 8 includes a different type of starters, grouped within the Spring Cloud family. You'll use some of these tools to implement common patterns for distributed systems: routing with Spring Cloud Gateway (`https://cloud.spring.io`), service discovery with Consul (`https://www.consul.io/`), and load balancing with Spring Cloud Load Balancer (`https://spring.io/guides/gs/spring-cloud-loadbalancer/`), among others. Don't worry about all these new terms for now; they'll be explained in detail while you make progress on the practical example.

The next chapter covers in detail how these starters and Spring Boot autoconfiguration work, based on a practical example.

Lombok and Java

The code examples in this book use Project Lombok (https://projectlombok.org/), a library that generates Java code based on annotations. The current version of Lombok at the time of writing this book was 1.18.28. The main reason to include Lombok in the book is educational: it keeps the code samples concise, reducing the boilerplate so you can focus on what it matters.

Let's use one of the first simple classes as an example. Say you want to create an immutable multiplication challenge class with two factors. See Listing 2-1.

Listing 2-1. The Challenge Class in Plain Java

```
/**
 * Represents a challenge with two factors.
 */
public final class Challenge {
    // Both factors
    private final int factorA;
    private final int factorB;
    public Challenge(int factorA, int factorB) {
        this.factorA = factorA;
        this.factorB = factorB;
    }
    public int getFactorA() {
        return this.factorA;
    }
    public int getFactorB() {
        return this.factorB;
    }
   @Override
    public boolean equals(final Object o) {
        if (o == this) return true;
        if (!(o instanceof Challenge)) return false;
```

```
        final Challenge other = (Challenge) o;
        if (this.getFactorA() != other.getFactorA()) return false;
        if (this.getFactorB() != other.getFactorB()) return false;
        return true;
    }
    @Override
    public int hashCode() {
        final int PRIME = 59;
        int result = 1;
        result = result * PRIME + factorA;
        result = result * PRIME + factorB;
        return result;
    }
    @Override
    public String toString() {
        return "Challenge [factorA=" + factorA + ", factorB=" +
        factorB + "]";
    }
}
```

As you can see, the full class has some classic boilerplate code: constructors, getters, and the `equals`, `hashCode`, and `toString` methods. They don't add much to this book, yet you need them for the code to work.

The same class can be reduced with Lombok to its minimum expression. See Listing 2-2.

Listing 2-2. The Challenge Class Using Lombok

```
import lombok.Value;
@Value
public class Challenge {
    // Both factors
    int factorA;
    int factorB;
}
```

The `@Value` annotation provided by Lombok groups some other annotations in this library that you can also use separately. Each of the following annotations instructs Lombok to generate code blocks before the Java build phase:

- `@AllArgsConstructor` creates a constructor with all the existing fields.
- `@FieldDefaults` makes your fields `private` and `final`.
- `@Getter` generates getters for `factorA` and `factorB`.
- `@ToString` includes a simple implementation of concatenating fields.
- `@EqualsAndHashCode` generates basic `equals()` and `hashCode()` methods using all fields by default, but you can also customize it.

Not only does Lombok reduce the code to the minimum, but it also helps when you need to modify these classes. Adding a new field to the `Challenge` class in Lombok means adding one line (excluding usages of the class). If you would use the plain Java version, you would need to add the new argument to the constructor, add the `equals` and `hashCode` methods, and add a new getter. Not only does that mean extra work, but it's also error-prone: if you forgot the extra field in the `equals` method, for example, you would introduce a bug into your application.

Like many tools, Lombok also has detractors. The main reason not to like Lombok is that, since it's easy to add code to your classes, you might end up adding code that you don't really need (e.g., setters or extra constructors). Besides, you could argue that having a good IDE with code generation and a refactoring assistant can help more or less at the same level.

The coming chapters mainly use these Lombok features:

- We annotate with `@Value` the immutable classes.
- For the data entities, we use separately some of the annotations described earlier.
- We add the `@Slf4j` annotation for Lombok to create a logger using the standard Simple Logging Facade for Java API (SLF4J). SLF4J (`https://www.slf4j.org/`) is a logging facade or abstraction layer that provides a common logging API for various logging frameworks in the Java ecosystem. It allows developers to write code that uses

a single API for logging while being able to use different logging frameworks under the hood. The section entitled "Logging" in this chapter gives more background about these concepts.

In any case, you learn what these annotations do when you look at the code examples, so you don't need to dive into more details on how they work.

If you prefer plain Java code, just use the Lombok's code annotations in this book as a reference to know what extra code you need to include in your classes.

Java Records Starting with JDK 14, the Java records feature was introduced in preview mode and became a standard feature in JDK 16. If you use this feature, you could write the `Challenge` class in pure Java in a concise way as well.

```
public record Challenge(int factorA, int factorB) {}
```

Testing Basics

In this section, we go through some important testing approaches and libraries. We put them into practice in the next chapters, so it's good to learn (or review) the basic concepts first.

Test-Driven Development

The first practical chapters in this book encourage you to use *test-driven development* (TDD). TDD is a software development practice that involves writing automated tests before writing the actual code. The idea behind TDD is to write small, incremental tests that verify the functionality of the code at each step of development. This technique helps you by putting the focus first on what you need and what your expectations are and later move to the implementation. It makes you, as a developer, think about what the code should do under certain situations or use cases. In real life, TDD also helps you clarify vague requirements and discard invalid ones. TDD has become increasingly popular in recent years, and it is widely used in agile software development.

Given that this book is driven by a practical case, you'll find that TDD fits quite well within the main purpose.

Behavior-Driven Development

As an addition to the idea of writing your tests before your logic, *behavior-driven development* (BDD) brings some better structure and readability to your tests. BDD is a software development methodology that emphasizes collaboration between developers, testers, and business stakeholders. BDD is a way to design, build, and test software based on the behavior of the system being developed. In BDD, the focus is on defining the behavior of the system from the perspective of the end-users or stakeholders. This is done by creating executable specifications, also known as "feature files," which describe the expected behavior of the system in a language that all stakeholders can understand.

In BDD, we write the tests according to a *Given-When-Then* structure. This removes the gap between developers and business analysts when mapping use cases to tests. Analysts can just read the code and identify what is being tested there.

Keep in mind that BDD, like TDD, is a development process by itself and not simply a way of writing tests. Its main goal is to facilitate conversations to improve the definition of requirements and their test cases. In this book, the focus regarding BDD is on the test structure. See Listing 2-3 for an example of how these tests look.

Listing 2-3. An Example of a BDD Test Case Using a Given-When-Then Structure

```
@Test
public void getRandomMultiplicationTest() throws Exception {
    // given
    given(challengeGeneratorService.randomChallenge())
            .willReturn(new Challenge(70, 20));
    // when
    MockHttpServletResponse response = mvc.perform(
            get("/multiplications/random")
                    .accept(MediaType.APPLICATION_JSON))
            .andReturn().getResponse();
    // then
    then(response.getStatus()).isEqualTo(HttpStatus.OK.value());
    then(response.getContentAsString())
            .isEqualTo(json.write(new Challenge(70, 20)).getJson());
}
```

JUnit 5

JUnit 5 (https://junit.org/junit5/) is a popular open-source testing framework for Java-based applications. It provides a set of annotations, assertions, and test runners to help developers write and run repeatable automated tests for their code. The code in this book uses JUnit 5 for the unit tests. The Spring Boot Test starter includes these libraries, so you don't need to include it in your dependencies.

In general, the idea behind unit tests is that you can verify the behavior of your classes (units) separately. In this book, you'll write unit tests for every class where you put logic.

Among all the features in JUnit 5, we use mainly the basic ones, listed here:

- `@BeforeEach` and `@AfterEach` for code that should be executed before and after each test, respectively.
- `@Test` for every method that represents a test we want to execute.
- `@ExtendsWith` at the class level to add JUnit 5 extensions. We use this to add the Mockito extension and the Spring extension to our tests.

Mockito

Mockito (https://site.mockito.org/) is a mocking framework for unit tests in Java. When you *mock* a class, you are overriding the real behavior of that class with some predefined instructions of what their methods should return or do for their arguments. This is an important requirement to write unit tests since you want to validate the behavior of one class only and simulate all its interactions.

The easiest way to mock a class with Mockito is to use the `@Mock` annotation in a field combined with `MockitoExtension` for JUnit 5. In Mockito, you can annotate a field with `@Mock` to create a mock of the corresponding class or interface. See Listing 2-4.

Listing 2-4. MockitoExtension and Mock Annotation Usage

```
@ExtendWith(MockitoExtension.class)
public class MultiplicationServiceImplTest {
    @Mock
    private ChallengeAttemptRepository attemptRepository;
    // [...] -> tests
}
```

Once the mock is created, you can define its behavior. For example, you might specify that when a certain method is called on the mock object with certain parameters, it should return a specific value or throw an exception. You could use the static method `Mockito.when()` to define custom behavior. See Listing 2-5.

Listing 2-5. Defining Custom Behavior with the Mockito.when() Function

```
import static org.mockito.Mockito.when;
// ...
when(attemptRepository.methodThatReturnsSomething())
    .thenReturn(predefinedResponse);
```

However, we will use the alternative methods from `BDDMockito` (https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/BDDMockito.html), also included in the Mockito dependency. This gives you a more readable, BDD-style way of writing unit tests. See Listing 2-6.

Listing 2-6. Using given to Define Custom Behavior

```
import static org.mockito.BDDMockito.given;
// ...
given(attemptRepository.methodThatReturnsSomething())
    .willReturn(predefinedResponse);
```

After executing the code under test, you can verify that the mock objects were called in the correct way, with the correct parameters, and so on. For example, you can check that an expected call to a mocked class was invoked. With Mockito, you use the `verify()` method for that. See Listing 2-7.

Listing 2-7. Verifying an Expected Call

```
import static org.mockito.Mockito.verify;
// ...
verify(attemptRepository).save(attempt);
```

As some extra background, it's good to know that there is also a BDD variation of `verify()`, called `then()`. Unfortunately, this replacement may be confusing when you combine `BDDMockito` with `BDDAssertions` (https://bit.ly/assertj-core-bdd-assertions-documentation) from AssertJ (covered in the next section). Since this book uses assertions more extensively than verifications, we will opt for `verifications` to better distinguish them.

Listing 2-8 shows a full example of a test using JUnit 5 and Mockito based on a class that you'll implement later in the book. For now, you can ignore the assertion; you'll get there soon.

Listing 2-8. A Complete Unit Test with JUnit5 and Mockito

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

import java.util.Optional;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import microservices.book.multiplication.event.ChallengeSolvedEvent;
import microservices.book.multiplication.event.EventDispatcher;
import microservices.book.multiplication.user.User;
import microservices.book.multiplication.user.UserRepository;

import static org.assertj.core.api.BDDAssertions.then;
import static org.mockito.BDDMockito.*;

@ExtendWith(MockitoExtension.class)
public class ChallengeServiceImplTest {
  private ChallengeServiceImpl challengeServiceImpl;

  @Mock
  private ChallengeAttemptRepository attemptRepository;

  @Mock
  private UserRepository userRepository;
```

```
  @Mock
  private EventDispatcher eventDispatcher;

  @BeforeEach
  public void setUp() {
    challengeServiceImpl = new ChallengeServiceImpl(attemptRepository,
        userRepository, eventDispatcher);
  }
  @Test
  public void checkCorrectAttemptTest() {
    // given
    long userId = 9L, attemptId = 1L;
    User user = new User("john_doe");
    User savedUser = new User(userId, "john_doe");
    ChallengeAttemptDTO attemptDTO =
        new ChallengeAttemptDTO(50, 60, "john_doe", 3000);
    ChallengeAttempt attempt =
        new ChallengeAttempt(null, savedUser, 50, 60, 3000, true);
    ChallengeAttempt storedAttempt =
        new ChallengeAttempt(attemptId, savedUser, 50, 60, 3000, true);
    ChallengeSolvedEvent event = new ChallengeSolvedEvent(attemptId, true,
        attempt.getFactorA(), attempt.getFactorB(), userId,
        attempt.getUser().getAlias());
    // user does not exist, should be created
    given(userRepository.findByAlias("john_doe"))
        .willReturn(Optional.empty());
    given(userRepository.save(user))
        .willReturn(savedUser);
    given(attemptRepository.save(attempt))
        .willReturn(storedAttempt);
    // when
    ChallengeAttempt resultAttempt =
        challengeServiceImpl.checkAttempt(attemptDTO);
    // then
    then(resultAttempt.isCorrect()).isTrue();
    verify(userRepository).save(user);
```

```
    verify(attemptRepository).save(attempt);
    verify(eventDispatcher).send(event);
  }
}
```

The test method `checkCorrectAttemptTest()` is for a scenario where the challenge attempt is correct. The necessary mock objects are initialized and passed to the `ChallengeServiceImpl` instance in the `setup()` method. Next, the `john_doe` username is defined and an instance named `attemptDTO` of `ChallengeAttemptDTO` is defined with specific values from a correct response. When the `checkAttempt()` method of `challengeServiceImpl` is called with the `attemptDTO`, assertions are made that the result of `checkAttempt` is marked as correct. The `verify()` methods are called to ensure that the methods were called as expected, including saving the user, saving the attempt, and sending the event.

AssertJ

The standard way to verify expected results with JUnit 5 is using assertions.

```
assertEquals("Hello, World!", actualGreeting);
```

There are not only assertions for equality of all kinds of objects but also to verify true/false, null, execution before a timeout, throwing an exception, and so on. You can find them all in the Assertions Javadoc (https://bit.ly/junit5-assertions-documentation).

Even though JUnit assertions are enough in most cases, they are not as easy to use and as readable as the ones provided by AssertJ (https://assertj.github.io/doc/). This library implements a fluent way of writing assertions and provides extra functionality so you can write more concise tests.

In its standard form, the previous example looks like this:

```
assertThat(actualGreeting).isEqualTo("Hello, World!");
```

However, as mentioned in previous sections, we want to use a BDD language approach. Therefore, we use the `BDDAssertions` class included in AssertJ. This class contains method equivalencies for all the `assertThat` cases, renamed as `then`.

```
then(actualGreeting).isEqualTo("Hello, World!");
```

In the book, we mostly have some basic assertions from AssertJ. If you're interested in extending your knowledge about AssertJ, you can start with the official documentation page (https://assertj.github.io/doc/).

Testing in Spring Boot

Both JUnit 5 and AssertJ are included in `spring-boot-starter-test`, so you simply need to include this dependency in your Spring Boot application to use them. Then, you can use different testing strategies.

One of the most popular ways to write tests in Spring Boot is to use the `@SpringBootTest` annotation. It will start a Spring context and make all your configured beans available for the test. If you're running integration tests and want to verify how different parts of your application work together, this approach is convenient.

When testing specific slices or individual classes of your application, it's better to use plain unit tests (without Spring at all) or more fine-grained annotations like `@WebMvcTest`, focused on controller-layer tests. This is the approach we use in the book, and we explain it more in detail when you get there.

For now, let's focus only on the integration between the libraries and frameworks described in this chapter.

- The Spring Test libraries (included via Spring Boot Test starter) come with a `SpringExtension` so you can integrate Spring in your JUnit 5 tests via the `@ExtendWith` annotation.
- The Spring Boot Test package introduces the `@MockBean` annotation that you can use to replace or add a bean in the Spring context, in a similar way to how Mockito's `@Mock` annotation can replace the behavior of a given class. This is helpful to test the application layers separately, so you don't need to bring all your real class behaviors in your Spring context together. You'll see a practical example when testing the application controllers.

Logging

In Java, you can log messages to the console just by using the `System.out` and `System.err` print streams.

```
System.out.println("Hello, standard output stream!");
```

This is considered simply good enough for a 12-factor app (https://12factor.net/logs), a popular set of best practices for writing cloud-native applications. The reason is that, eventually, some other tool will collect them from the standard outputs at the system level and aggregate them in an external framework.

Therefore, we'll write our logs to the standard and the error output. But that doesn't mean that you must stick to the plain, ugly `System.out` variants in Java.

Most professional Java apps use a logger implementation such as LogBack. And, given that there are multiple logging frameworks for Java, it's even better to choose a generic abstraction such as SLF4J.

The good news is that Spring Boot comes with all this logging configuration already set up for you. The default implementation is LogBack, and Spring Boot's preconfigured message format is as follows:

```
2020-03-22 10:19:59.556  INFO 93532 --- [main] o.s.b.w.embedded.
tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with
context path ''
```

SLF4J loggers are also supported. To use a logger, you create it via the `LoggerFactory`. The only argument it needs is a name. By default, it is common to use the factory method that takes the class itself and gets the logger name from it. See Listing 2-9.

Listing 2-9. Creating and Using a Logger with SLF4J

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
class ChallengeServiceImpl {
  private static final Logger log = LoggerFactory.
getLogger(ChallengeServiceImpl.class);
  public void dummyMethod() {
    var name = "John";
```

```
    log.info("Hello, {}!", name);
  }
}
```

As you see in the example, loggers support parameter replacement via the curly-braces placeholder.

Given that this book uses Lombok, you can replace that line to create a logger in your class with a simple annotation: `@Slf4j`. This helps keep your code concise. By default, Lombok creates a static variable named `log`. See Listing 2-10.

Listing 2-10. Using a Logger with Lombok

```
import lombok.extern.slf4j.Slf4j;
@Slf4j
class ChallengeServiceImpl {
  public void dummyMethod() {
    var name = "John";
    log.info("Hello, {}!", name);
  }
}
```

Summary and Achievements

This chapter reviewed some basic libraries and concepts that we use in the book: Spring Boot, Lombok, tests with JUnit and AssertJ, and logging. These are only a few of what you'll learn during the journey, but they were introduced separately to avoid long pauses in the main learning path. All the other topics, more related to the evolving architecture, are explained in detail as you navigate through the book pages.

Do not worry if you still feel like you have some knowledge gaps. The practical code examples in the next chapters will help you understand these concepts by providing extra context.

Chapter's Achievements:

- You reviewed the core ideas about Spring and Spring Boot.
- You understood how we'll use Lombok in the book to reduce boilerplate code.