Spring Security IN ACTION

SECOND EDITION

Laurențiu Spilcă

MANNING

Spring Security IN ACTION

SECOND EDITION

Laurențiu Spilcă

Spring Security in Action, Second Edition MEAP V08

1. Copyright_2023_Manning_Publications
2. welcome
3. 1_Security_today
4. 2_Hello_Spring_Security
5. 3_Managing_users
6. 4_Managing_passwords
7. 5_A_web_app’s_security_begins_with_filters
8. 6_Implementing_authentication
9. 7_Configuring_endpoint-level_authorization:_Restricting_access
10. 8_Configuring_endpoint-level_authorization:_Applying_restrictions
11. 9_Configuring_Cross-Site_Request_Forgery_(CSRF)_protection
12. 10_Configuring_Cross-Origin_Resource_Sharing_(CORS)
13. 11_Implement_authorization_at_the_method_level
14. 12_Implement_filtering_at_the_method_level

MEAP Edition Manning Early Access Program Spring Security in Action, Second Edition Version 8

Copyright 2023 Manning Publications
©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://livebook.manning.com/#!/book/spring-security-in-action-second-edition/discussion

For more information on this and other Manning titles go to

manning.com

welcome

Thank you for purchasing the MEAP of *Spring Security in Action*, *2nd edition*. With time, technologies evolve quickly. So does the Spring ecosystem and Spring Security together with it. Spring is an ecosystem of projects with a huge active community that continuously works to improve the framework.

More and more developers are aware that security is an essential aspect that needs to be considered from the very start of a software project. Since I wrote the first edition of Spring Security in Action, essential things have changed in how you write your Spring Security configurations. Clearly, you should know these changes and apply them properly in apps.

By the end of the book, you will have covered the following topics:

- Implementing authentication in a web app
- Implementing authorization at the endpoint level in a web app
- Implementing authorization at the method level in a web app
- Implementing an OAuth 2 authorization server using the Spring Security authorization server framework
- Implementing OAuth 2 resource server and client.
- Testing your Spring Security implementations

I have used Spring and Spring Security for many years with a great variety of systems, from the technical perspective to the business domain. However, teaching a complex subject such as Spring Security so it can be well understood is not an easy task. This is why your feedback is invaluable and will significantly help in the improvement of this book. Please let me know what comments and suggestions you have in the liveBook Discussion forum.

Thank you again for your interest and for purchasing the MEAP!

-Perry Lea

In this book

Copyright 2023 Manning Publications welcome brief contents 1 Security today 2 Hello Spring Security 3 Managing users 4 Managing passwords 5 A web app’s security begins with filters 6 Implementing authentication 7 Configuring endpoint-level authorization: Restricting access 8 Configuring endpoint-level authorization: Applying restrictions 9 Configuring Cross-Site Request Forgery (CSRF) protection 10 Configuring Cross-Origin Resource Sharing (CORS) 11 Implement authorization at the method level 12 Implement filtering at the method level

1 Security today

This chapter covers

- What Spring Security is and what you can solve by using it
- What security is for a software application
- Why software security is essential and why you should care

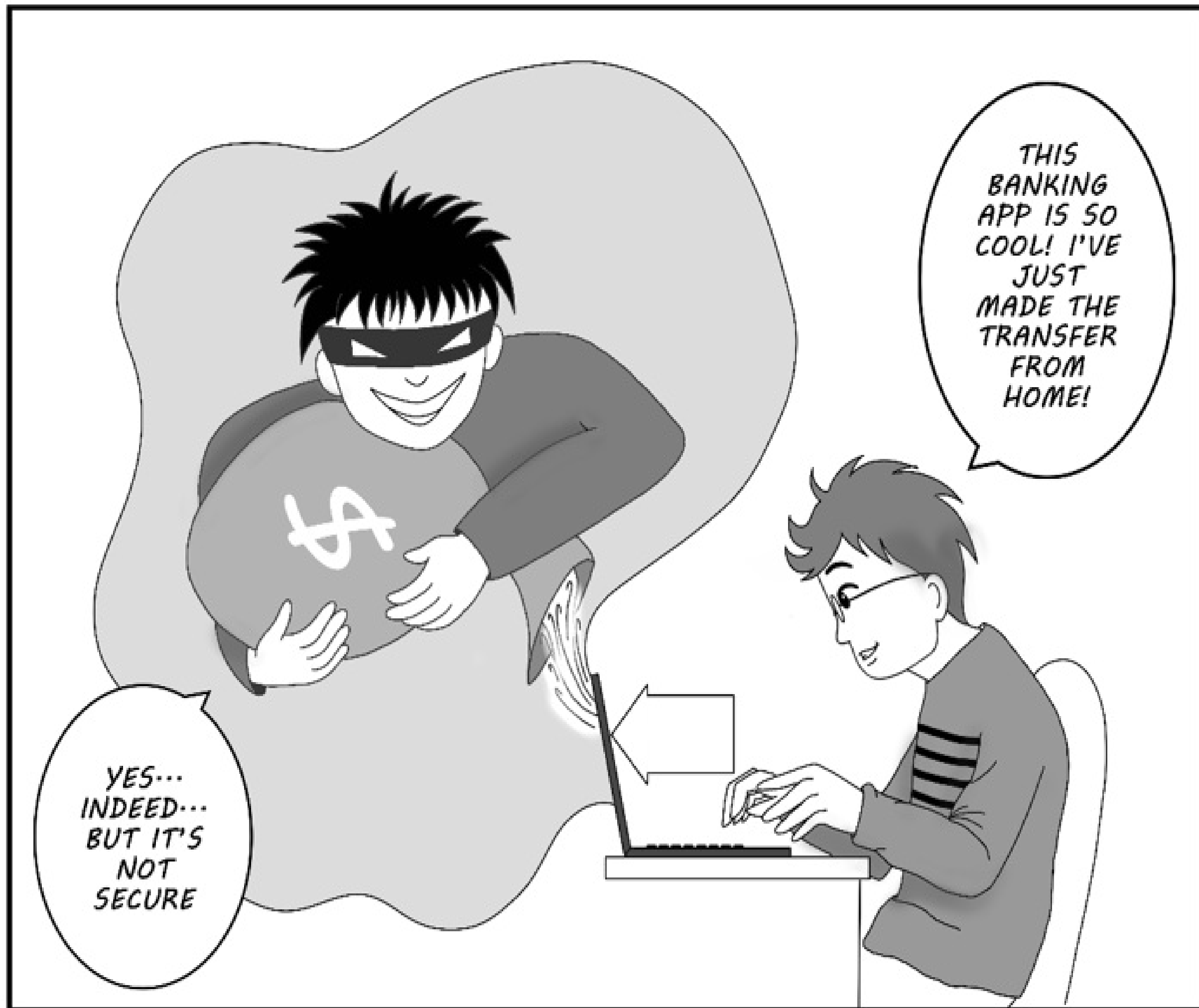
Developers have become increasingly more aware of the need for secure software, and are taking responsibility for security from the beginning of software development.

Generally, as developers, we begin by learning that the purpose of an application is to solve business issues. This purpose refers to something where data could be processed somehow, persisted, and eventually displayed to the user in a specific way as specified by some requirements. This overview of software development, which is somehow imposed from the early stages of learning these techniques, has the unfortunate disadvantage of hiding practices that are also part of the process. While the application works correctly from the user's perspective and, in the end, it does what the user expects in terms of functionality, there are lots of aspects hidden in the final result.

Nonfunctional software qualities such as performance, scalability, availability, and, of course, security, as well as others, can have an impact over time, from short to long term. If not taken into consideration early on, these qualities can dramatically affect the profitability of the application owners. Moreover, the neglect of these considerations can also trigger failures in other systems as well (for example, by the unwilling participation in a distributed denial of service (DDoS) attack). The hidden aspects of nonfunctional requirements (the fact that it's much more challenging to see if something's missing or incomplete) makes these, however, more dangerous.

Figure 1.1 A user mainly thinks about functional requirements. Sometimes, you might see them aware of performance, which is nonfunctional, but unfortunately, it's quite unusual that a user cares about security. Nonfunctional requirements tend to be more transparent than functional

ones.



There are multiple nonfunctional aspects to consider when working on a software system. In practice, all of these are important and need to be treated responsibly in the process of software development. In this book, we focus on one of these: security. You'll learn how to protect your application, step by step, using Spring Security.

In this chapter, I want to show you the big picture of security-related concepts. Throughout the book, we work on practical examples, and where appropriate, I'll refer back to the description I give in this chapter. Where applicable, I'll also provide you with more details. Here and there, you'll find references to other materials (books, articles, documentation) on specific

subjects that are useful for further reading.

1.1 Discovering Spring Security

In this section, we discuss the relationship between Spring Security and Spring. It is important, first of all, to understand the link between the two before starting to use those. If we go to the official website, https://spring.io/projects/spring-security, we see Spring Security described as a powerful and highly customizable framework for authentication and access control. I would simply say it is a framework that enormously simplifies applying (or "baking") security for Spring applications.

Spring Security is the primary choice for implementing application-level security in Spring applications. Generally, its purpose is to offer you a highly customizable way of implementing authentication, authorization, and protection against common attacks. Spring Security is an open-source software released under the Apache 2.0 license. You can access its source code on GitHub at https://github.com/spring-projects/spring-security/. I highly recommend that you contribute to the project as well.

Note

You can use Spring Security for both standard web servlets and reactive applications as well as non-web apps. In this book, we'll use Spring Security with the latest Java long-term supported, Spring, and Spring Boot versions (Java 17, Spring 6, and Spring Boot 3).

I can guess that if you opened this book, you work on Spring applications, and you are interested in securing those. Spring Security is most likely the best choice for you. It's the de facto solution for implementing application-level security for Spring applications. Spring Security, however, doesn't automatically secure your application. And it's not some kind of magic panacea that guarantees a vulnerability-free app. Developers need to understand how to configure and customize Spring Security around the needs of their applications. How to do this depends on many factors, from the functional requirements to the architecture.

Technically, applying security with Spring Security in Spring applications is simple. You’ve already implemented Spring applications, so you know that the framework’s philosophy starts with the management of the Spring context. You define beans in the Spring context to allow the framework to manage these based on the configurations you specify.

You use annotations to tell Spring what to do: expose endpoints, wrap methods in transactions, intercept methods in aspects, and so on. The same is true with Spring Security configurations, which is where Spring Security comes into play. What you want is to use annotations, beans, and in general, a Spring-fashioned configuration style comfortably when defining your application-level security. In a Spring application, the behavior that you need to protect is defined by methods.

To think about application-level security, you can consider your home and the way you allow access to it. Do you place the key under the entrance rug? Do you even have a key for your front door? The same concept applies to applications, and Spring Security helps you develop this functionality. It’s a puzzle that offers plenty of choices for building the exact image that describes your system. You can choose to leave your house completely unsecured, or you can decide not to allow everyone to enter your home.

The way you configure security can be straightforward like hiding your key under the rug, or it can be more complicated like choosing a variety of alarm systems, video cameras, and locks. In your applications, you have the same options, but as in real life, the more complexity you add, the more expensive it gets. In an application, this cost refers to the way security affects maintainability and performance.

But how do you use Spring Security with Spring applications? Generally, at the application level, one of the most encountered use cases is when you’re deciding whether someone is allowed to perform an action or use some piece of data. Based on configurations, you write Spring Security components that intercept the requests and that ensure whoever makes the requests has permission to access protected resources. The developer configures components to do precisely what’s desired. If you mount an alarm system, it’s you who should make sure it’s also set up for the windows as well as for the doors. If you forget to set it up for the windows, it’s not the fault of the

alarm system that it doesn’t trigger when someone forces a window.

Other responsibilities of Spring Security components relate to data storage as well as data transit between different parts of the systems. By intercepting calls to these different parts, the components can act on the data. For example, when data is stored, these components can apply encryption or hashing algorithms to the data. The data encodings keep the data accessible only to privileged entities. In a Spring application, the developer has to add and configure a component to do this part of the job wherever it’s needed. Spring Security provides us with a contract through which we know what the framework requires to be implemented, and we write the implementation according to the design of the application. We can say the same thing about transiting data.

In real-world implementations, you’ll find cases in which two communicating components don’t trust each other. How can the first know that the second one sent a specific message and it wasn’t someone else? Imagine you have a phone call with somebody to whom you have to give private information. How do you make sure that on the other end is indeed a valid individual with the right to get that data, and not somebody else? For your application, this situation applies as well. Spring Security provides components that allow you to solve these issues in several ways, but you have to know which part to configure and then set it up in your system. This way, Spring Security intercepts messages and makes sure to validate communication before the application uses any kind of data sent or received.

Like any framework, one of the primary purposes of Spring is to allow you to write less code to implement the desired functionality. And this is also what Spring Security does. It completes Spring as a framework by helping you write less code to perform one of the most critical aspects of an application—security. Spring Security provides predefined functionality to help you avoid writing boilerplate code or repeatedly writing the same logic from app to app. But it also allows you to configure any of its components, thus providing great flexibility. To briefly recap this discussion:

- You use Spring Security to bake application-level security into your applications in the “Spring” way. By this, I mean, you use annotations, beans, the Spring Expression Language (SpEL), and so on.

- Spring Security is a framework that lets you build application-level security. However, it is up to you, the developer, to understand and use Spring Security properly. Spring Security, by itself, does not secure an application or sensitive data at rest or in flight.
- This book provides you with the information you need to effectively use Spring Security.

Alternatives to Spring Security

This book is about Spring Security, but as with any solution, I always prefer to have a broad overview. Never forget to learn the alternatives that you have for any option. One of the things I've learned over time is that there's no general right or wrong. "Everything is relative" also applies here!

You won't find a lot of alternatives to Spring Security when it comes to securing a Spring application. One alternative you could consider is Apache Shiro (https://shiro.apache.org). It offers flexibility in configuration and is easy to integrate with Spring and Spring Boot applications. Apache Shiro sometimes makes a good alternative to the Spring Security approach.

If you've already worked with Spring Security, you'll find using Apache Shiro easy and comfortable to learn. It offers its own annotations and design for web applications based on HTTP filters, which greatly simplify working with web applications. Also, you can secure more than just web applications with Shiro, from smaller command-line and mobile applications to large-scale enterprise applications. And even if simple, it's powerful enough to use for a wide range of things from authentication and authorization to cryptography and session management.

However, Apache Shiro could be too "light" for the needs of your application. Spring Security is not just a hammer, but an entire set of tools. It offers a larger scale of possibilities and is designed specifically for Spring applications. Moreover, it benefits from a larger community of active developers, and it is continuously enhanced.

1.2 What is software security?

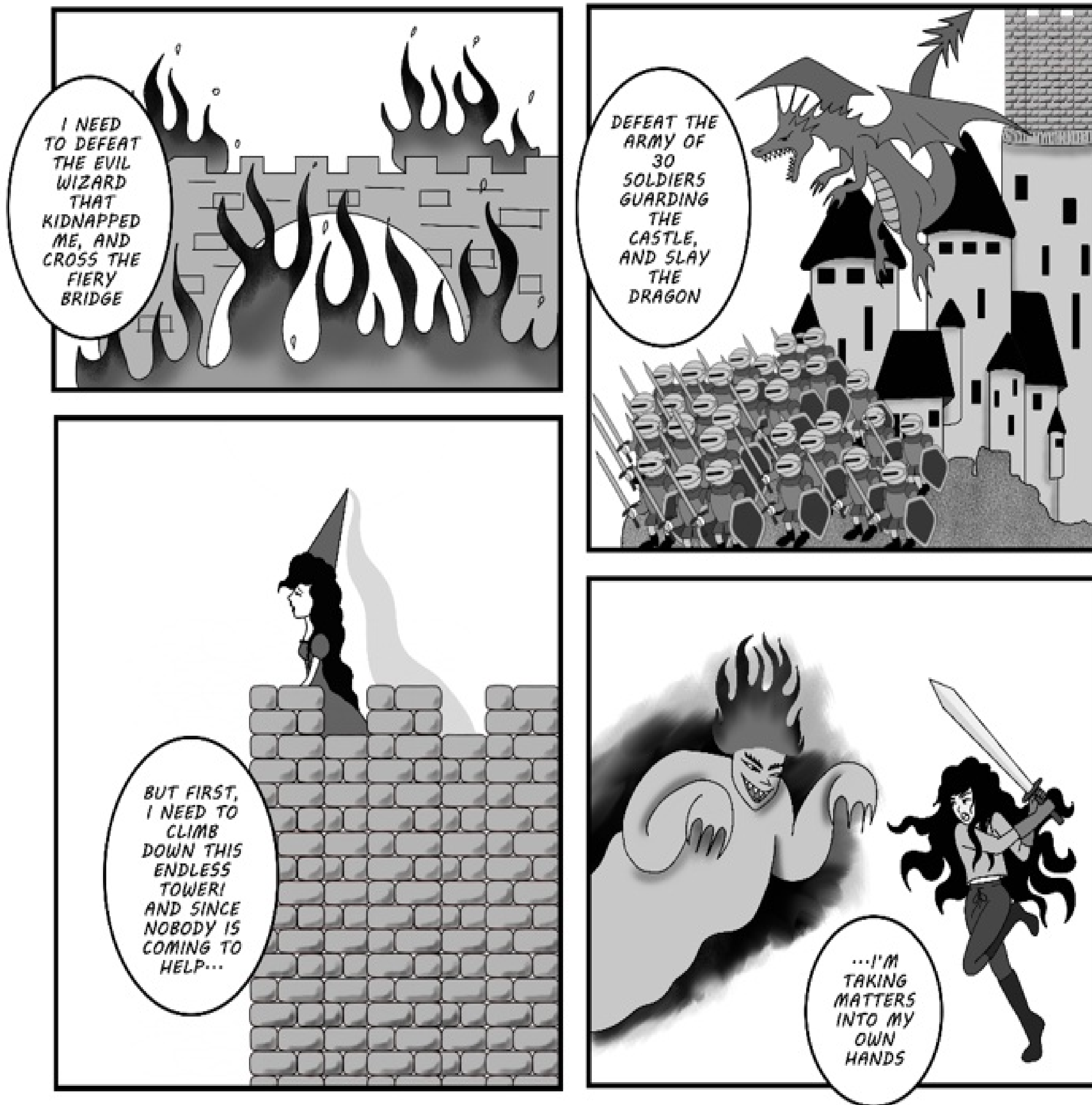
Software systems manage large amounts of data, of which a significant part can be considered sensitive, especially given the General Data Protection Regulations (GDPR) requirements. Any information that you, as a user, consider private is sensitive for your software application. Sensitive data can include harmless information like a phone number, email address, or identification number; although, we generally think more about data that is riskier to lose, like your credit card details. The application should ensure that there’s no chance for that information to be accessed, changed, or intercepted. No parties other than the users for whom this data is intended should be able to interact in any way with it. Broadly expressed, this is the meaning of security.

NOTE

GDPR created a lot of buzz globally after its introduction in 2018. It generally represents a set of European laws that refer to data protection and gives people more control over their private data. GDPR applies to the owners of systems that have users in Europe. The owners of such applications risk significant penalties if they don’t respect the regulations imposed.

We apply security in layers, with each layer requiring a different approach. Compare these layers to a protected castle (figure 1.2). A hacker needs to bypass several obstacles to obtain the resources managed by the app. The better you secure each layer, the lower the chance an individual with bad intentions manages to access data or perform unauthorized operations.

Figure 1.2 The Dark Wizard (a hacker) has to bypass multiple obstacles (security layers) to steal the Magic Sword (user resources) from the Princess (your application).

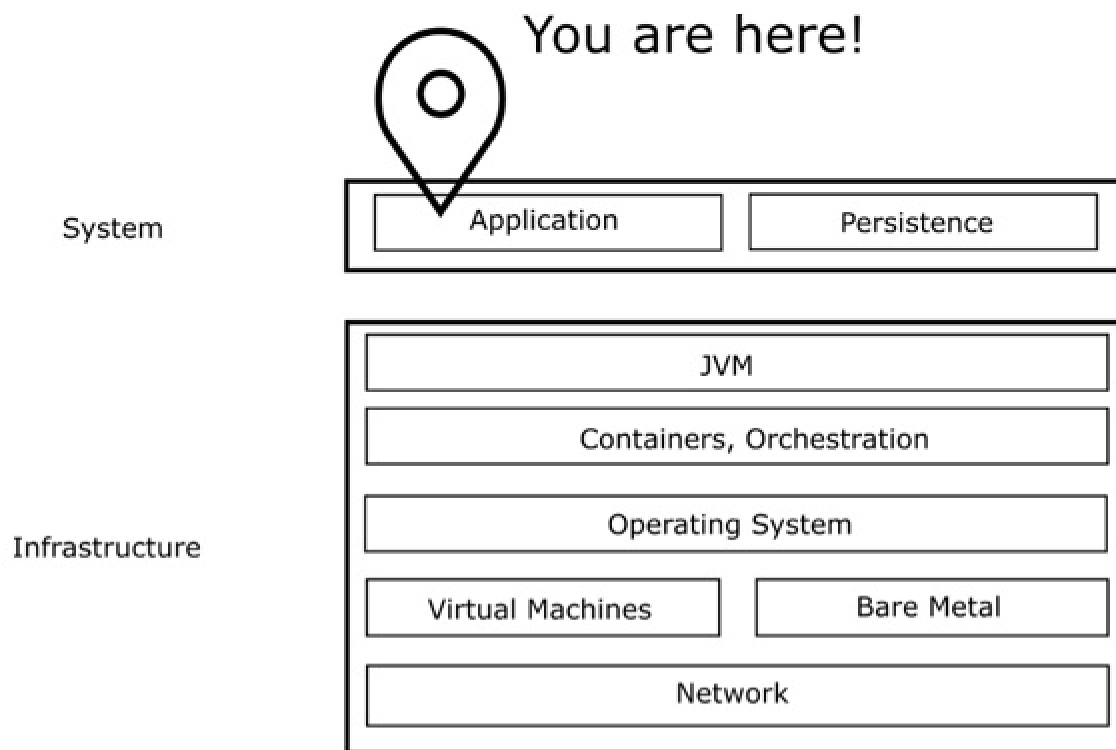


Security is a complex subject. In the case of a software system, security doesn't apply only at the application level. For example, for networking, there are issues to be taken into consideration and specific practices to be used, while for storage, it's another discussion altogether. Similarly, there's a different philosophy in terms of deployment, and so on. Spring Security is a framework that belongs to application-level security. In this section, you'll get a general picture of this security level and its implications.

"charitalics"Application-level security (figure 1.3) refers to everything that an

application should do to protect the environment it executes in, as well as the data it processes and stores. Mind that this isn't only about the data affected and used by the application. An application might contain vulnerabilities that allow a malicious individual to affect the entire system!

Figure 1.3 We apply security in layers, and each layer depends on those below it. In this book, we discuss Spring Security, which is a framework used to implement application-level security at the top-most level.



To be more explicit, let's discuss using some practical cases. We'll consider a situation in which we deploy a system as in figure 1.4. This situation is common for a system designed using a microservices architecture, especially if you deploy it in multiple availability zones in the cloud.

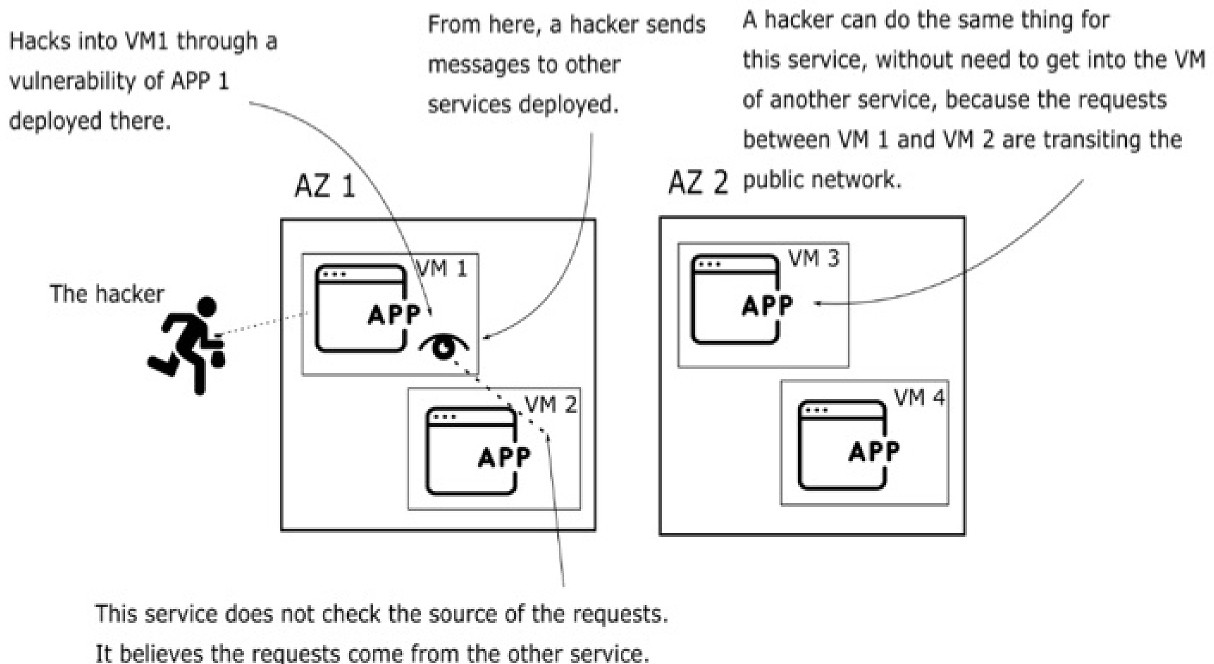
Note

If you're interested in implementing efficient cloud-oriented Spring apps, I strongly recommend you Cloud Native Spring in Action by Thomas Vitale (Manning, 2022). In this book, the author focuses on all the needed aspects a professional needs to know for implementing well-done Spring apps for

cloud deployments.

With such microservices architectures, we can encounter various vulnerabilities, so you should exercise caution. As mentioned earlier, security is a cross-cutting concern that we design on multiple layers. It’s a best practice when addressing the security concerns of one of the layers to assume as much as possible that the above layer doesn’t exist. Think about the analogy with the castle in figure 1.2. If you manage the “layer” with 30 soldiers, you want to prepare them to be as strong as possible. And you do this even knowing that before reaching them, one would need to cross the fiery bridge.

Figure 1.4 If a malicious user manages to get access to the virtual machine (VM) and there’s no applied application-level security, a hacker can gain control of the other applications in the system. If communication is done between two different availability zones (AZ), a malicious individual will find it easier to intercept the messages. This vulnerability allows them to steal data or to impersonate users.



With this in mind, let’s consider that an individual driven by bad intentions would be able to log in to the virtual machine (VM) that’s hosting the first application. Let’s also assume that the second application doesn’t validate the requests sent by the first application. The attacker can then exploit this

vulnerability and control the second application by impersonating the first one.

Also, consider that we deploy the two services to two different locations. Then the attacker doesn't need to log in to one of the VMs as they can directly act in the middle of communications between the two applications.

NOTE

An *availability zone* (AZ in figure 1.4) in terms of cloud deployment is a separate data center. This data center is situated far enough geographically (and has other dependencies) from other data centers of the same region that, if one availability zone fails, the probability that others are failing too is minimal. In terms of security, an important aspect is that traffic between two different data centers generally goes across a public network.

I referred earlier to authentication and authorization. These are present in most applications. Through authentication, an application identifies a user (a person or another application). The purpose of identifying these is to be able to decide afterward what they should be allowed to do—that's authorization. I provide quite a lot of details on authentication and authorization, starting with chapter 3 and continuing throughout the book.

In an application, you often find the need to implement authorization in different scenarios. Consider another situation: most applications have restrictions regarding the user accessing certain functionality. Achieving this implies first the need to identify who creates an access to request for a specific feature—that's authentication. As well, we need to know their privileges to allow the user to use that part of the system. As the system becomes more complex, you'll find different situations that require a specific implementation related to authentication and authorization.

For example, what if you'd like to authorize a particular component of the system against a subset of data or operations on behalf of the user? Let's say the printer needs access to read the user's documents. Should you simply share the credentials of the user with the printer? But that allows the printer more rights than needed! And it also exposes the credentials of the user. Is there a proper way to do this without impersonating the user? These are

essential questions, and the kind of questions you encounter when developing applications: questions that we not only want to answer, but for which you'll see applications with Spring Security in this book.

Depending on the chosen architecture for the system, you'll find authentication and authorization at the level of the entire system, as well as for any of the components. And as you'll see further along in this book, with Spring Security, you'll sometimes prefer to use authorization even for different tiers of the same component. In chapter 11, we'll discuss more on method security, which refers to this aspect. The design gets even more complicated when you have a predefined set of roles and authorities.

I would also like to bring to your attention data storage. Data at rest adds to the responsibility of the application. Your app shouldn't store all its data in a readable format. The application sometimes needs to keep the data either encrypted with a private key or hashed. Secrets like credentials and private keys can also be considered data at rest. These should be carefully stored, usually in a secrets vault.

NOTE

We classify data as "at rest" or "in transition." In this context, *data at rest refers* to data in computer storage or, in other words, persisted data. *Data in transition* applies to all the data that's exchanged from one point to another. Different security measures should, therefore, be enforced, depending on the type of data.

Finally, an executing application must manage its internal memory as well. It may sound strange, but data stored in the heap of the application can also present vulnerabilities. Sometimes the class design allows the app to store sensitive data like credentials or private keys for a long time. In such cases, someone who has the privilege to make a heap dump could find these details and then use them maliciously.

With a short description of these cases, I hope I've managed to provide you with an overview of what we mean by application security, as well as the complexity of this subject. Software security is a tangled subject. One who is willing to become an expert in this field would need to understand (as well as

to apply) and then test solutions for all the layers that collaborate within a system. In this book, however, we’ll focus only on presenting all the details of what you specifically need to understand in terms of Spring Security. You’ll find out where this framework applies and where it doesn’t, how it helps, and why you should use it. Of course, we’ll do this with practical examples that you should be able to adapt to your own unique use cases.

1.3 Why is security important?

The best way to start thinking about why security is important is from your point of view as a user. Like anyone else, you use applications, and these have access to your data. These can change your data, use it, or expose it. Think about all the apps you use, from your email to your online banking service accounts. How would you evaluate the sensitivity of the data that is managed by all these systems? How about the actions that you can perform using these systems? Similarly to data, some actions are more important than others. You don’t care very much about some of those, while others are more significant. Maybe for you, it’s not that important if someone would somehow manage to read some of your emails. But I bet you’d care if someone else could empty your bank accounts.

Once you’ve thought about security from your point of view, try to see a more objective picture. The same data or actions might have another degree of sensitivity to other people. Some might care a lot more than you if their email is accessed and someone could read their messages. Your application should make sure to protect everything to the desired degree of access. Any leak that allows the use of data and functionalities, as well as the application, to affect other systems is considered a vulnerability, and you need to solve it.

Not considering enough security comes with a price that I’m sure you aren’t willing to pay. In general, it’s about money. But the cost can differ, and there are multiple ways through which you can lose profitability. It isn’t only about losing money from a bank account or using a service without paying for it. These things indeed imply cost. The image of a brand or a company is also valuable, and losing a good image can be expensive—sometimes even more costly than the expenses directly resulting from the exploitation of a vulnerability in the system! The trust that users have in your application is

one of its most valuable assets, and it can make the difference between success or failure.

Here are a few fictitious examples. Think about how you would see these as a user. How can these affect the organization responsible for the software?

- A back-office application should manage the internal data of an organization but, somehow, some information leaks out.
- Users of a ride-sharing application observe that money is debited from their accounts on behalf of trips that aren't theirs.
- After an update, users of a mobile banking application are presented with transactions that belong to other users.

In the first situation, the organization using the software, as well as its employees, can be affected. In some instances, the company could be liable and could lose a significant amount of money. In this situation, users don't have the choice to change the application, but the organization can decide to change the provider of their software.

In the second case, users will probably choose to change the service provider. The image of the company developing the application would be dramatically affected. The cost lost in terms of money in this case is much less than the cost in terms of image. Even if payments are returned to the affected users, the application will still lose some customers. This affects profitability and can even lead to bankruptcy. And in the third case, the bank could see dramatic consequences in terms of trust, as well as legal repercussions.

In most of these scenarios, investing in security is safer than what happens if someone exploits a vulnerability in your system. For all of the examples, only a small weakness could cause each outcome. For the first example, it could be a broken authentication or a cross-site request forgery (CSRF). For the second and third examples, it could be a lack of method access control. And for all of these examples, it could be a combination of vulnerabilities.

Of course, from here we can go even further and discuss the security in defense-related systems. If you consider money important, add human lives to the cost! Can you even imagine what could be the result if a health care system was affected? What about systems that control nuclear power? You

can reduce any risk by investing early in the security of your application and by allocating enough time for security professionals to develop and test your security mechanisms.

Note

The lessons learned from those who failed before you are that the cost of an attack is usually higher than the investment cost of avoiding the vulnerability.

In the rest of this book, you’ll see examples of ways to apply Spring Security to avoid situations like the ones presented. I guess there will never be enough word written about how important security is. When you have to make a compromise on the security of your system, try to estimate your risks correctly.

1.4 What will you learn in this book?

This book offers a practical approach to learning Spring Security. Throughout the rest of the book, we’ll deep dive into Spring Security, step by step, proving concepts with simple to more complex examples. To get the most out of this book, you should be comfortable with Java programming, as well as with the basics of the Spring Framework. If you haven’t used the Spring Framework or you don’t feel comfortable yet using its basics, I recommend you first read "charitalics"Spring Start Here, another book I wrote (Manning, 2021). After reading that book, you can enhance your Spring knowledge with "charitalics"Spring In Action, 6th edition, by Craig Walls (Manning, 2022) as well as "charitalics"Spring Boot Up & Running by Mark Heckler (O’Reilly Media, 2021).

In this book, you’ll learn

- The architecture and basic components of Spring Security and how to use it to secure your application
- Authentication and authorization with Spring Security, including the OAuth 2 and OpenID Connect flows, and how these apply to a production-ready application
- How to implement security with Spring Security in different layers of

your application
- Different configuration styles and best practices for using those in your project
- Using Spring Security for reactive applications
- Testing your security implementations

To make the learning process smooth for each described concept, we’ll work on multiple simple examples.

When we finish, you’ll know how to apply Spring Security for the most practical scenarios and understand where to use it and its best practices. I also strongly recommend that you work on all the examples that accompany the explanations.

1.5 Summary

- Spring Security is the leading choice for securing Spring applications. It offers a significant number of alternatives that apply to different styles and architectures.
- You should apply security in layers for your system, and for each layer, you need to use different practices.
- Security is a cross-cutting concern you should consider from the beginning of a software project.
- Usually, the cost of an attack is higher than the cost of investment in avoiding vulnerabilities to begin with.
- Sometimes the smallest mistakes can cause significant harm. For example, exposing sensitive data through logs or error messages is a common way to introduce vulnerabilities in your application.

2 Hello Spring Security

This chapter covers

- Creating your first project with Spring Security
- Designing simple functionalities using the basic components for authentication and authorization
- Underlying concept and how to use it in a given project
- Applying the basic contracts and understanding how they relate to each other
- Writing custom implementations for the primary responsibilities
- Overriding Spring Boot’s default configurations for Spring Security

Spring Boot appeared as an evolutionary stage for application development with the Spring Framework. Instead of you needing to write all the configurations, Spring Boot comes with some preconfigured, so you can override only the configurations that don’t match your implementations. We also call this approach "charitalics"convention-over-configuration. Spring Boot is no longer a new concept and today we enjoy writing applications using its third version.

Before Spring Boot, developers used to write dozens of lines of code again and again for all the apps they had to create. This situation was less visible in the past when we developed most architectures monolithically. With a monolithic architecture, you only had to write such configurations once at the beginning, and you rarely needed to touch them afterward. When service-oriented software architectures evolved, we started to feel the pain of boilerplate code that we needed to write for configuring each service. If you find it amusing, you can check out chapter 3 from "charitalics"Spring in Practice by Willie Wheeler with Joshua White (Manning, 2013). This chapter of an older book describes writing a web application with Spring 3. In this way, you’ll understand how many configurations you had to write for one small one-page web application. Here’s the link for the chapter: https://livebook.manning.com/book/spring-in-practice/chapter-3/

For this reason, with the development of recent apps and especially those for microservices, Spring Boot became more and more popular. Spring Boot provides autoconfiguration for your project and shortens the time needed for the setup. I would say it comes with the appropriate philosophy for today’s software development.

In this chapter, we’ll start with our first application that uses Spring Security. For the apps that you develop with the Spring Framework, Spring Security is an excellent choice for implementing application-level security. We’ll use Spring Boot and discuss the defaults that are configured by convention, as well as a brief introduction to overriding these defaults. Considering the default configurations provides an excellent introduction to Spring Security, one that also illustrates the concept of authentication.

Once we get started with the first project, we’ll discuss various options for authentication in more detail. In chapters 3 through 6, we’ll continue with more specific configurations for each of the different responsibilities that you’ll see in this first example. You’ll also see different ways to apply those configurations, depending on architectural styles. The steps we’ll approach in the current chapter follow:

1. Create a project with only Spring Security and web dependencies to see how it behaves if you don’t add any configuration. This way, you’ll understand what you should expect from the default configuration for authentication and authorization.
2. Change the project to add functionality for user management by overriding the defaults to define custom users and passwords.
3. After observing that the application authenticates all the endpoints by default, learn that this can be customized as well.
4. Apply different styles for the same configurations to understand best practices.

2.1 Starting your first project

Let’s create the first project so that we have something to work on for the first example. This project is a small web application, exposing a REST endpoint. You’ll see how, without doing much, Spring Security secures this

endpoint using HTTP Basic authentication. HTTP Basic is a way a web app authenticates a user by means of a set of credentials (username and password) that the app gets in the header of the HTTP request.

Just by creating the project and adding the correct dependencies, Spring Boot applies default configurations, including a username and a password when you start the application.

Note

You have various alternatives to create Spring Boot projects. Some development environments offer the possibility of creating the project directly. If you need help with creating your Spring Boot projects, you can find several ways described in the appendix. For even more details, I recommend Mark Heckler's *Spring Boot Up & Running* (O'Reilly Media, 2021) and Spring Boot in Practice (Manning, 2022) by Somnath Musib or even Spring Start Here (Manning, 2021), another book I wrote.

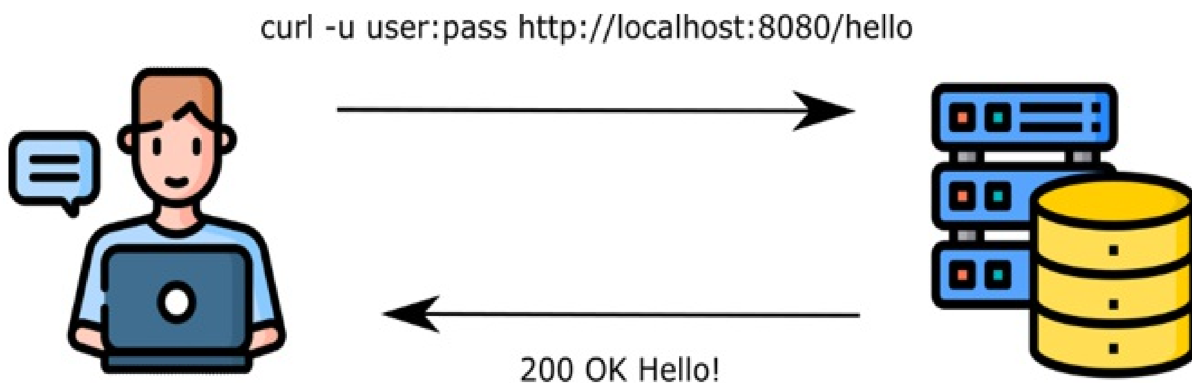
The examples in this book refer to the book's companion source code. With each example, I also specify the dependencies that you need to add to your pom.xml file. You can, and I recommend that you do, download the projects provided with the book and the available source code at https://www.manning.com/downloads/2105. The projects will help you if you get stuck with something. You can also use these to validate your final solutions.

Note

The examples in this book are not dependent on the build tool you choose. You can use either Maven or Gradle. But to be consistent, I built all the examples with Maven.

The first project is also the smallest one. As mentioned, it's a simple application exposing a REST endpoint that you can call and then receive a response as described in figure 2.1. This project is enough to learn the first steps when developing an application with Spring Security and Spring Boot. It presents the basics of the Spring Security architecture for authentication and authorization.

Figure 2.1 Our first application uses HTTP Basic to authenticate and authorize the user against an endpoint. The application exposes a REST endpoint at a defined path (/hello). For a successful call, the response returns an HTTP 200 status message and a body. This example demonstrates how the authentication and authorization configured by default with Spring Security works.



We begin learning Spring Security by creating an empty project and naming it-ssia-ch2-ex1. (You'll also find this example with the same name in the projects provided with the book.) The only dependencies you need to write for our first project are `spring-boot-starter-web` and `spring-boot-starter-security`, as shown in listing 2.1. After creating the project, make sure that you add these dependencies to your pom.xml file. The primary purpose of working on this project is to see the behavior of a default configured application with Spring Security. We also want to understand which components are part of this default configuration, as well as their purpose.

Listing 2.1 Spring Security dependencies for our first web app

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

We could directly start the application now. Spring Boot applies the default configuration of the Spring context for us based on which dependencies we add to the project. But we wouldn't be able to learn much about security if

we don’t have at least one endpoint that’s secured. Let’s create a simple endpoint and call it to see what happens. For this, we add a class to the empty project, and we name this class `HelloController`. To do that, we add the class in a package called `controllers` somewhere in the main namespace of the Spring Boot project.

NOTE

Spring Boot scans for components only in the package (and its sub-packages) that contains the class annotated with `@SpringBootApplication`. If you annotate classes with any of the stereotype components in Spring, outside of the main package, you must explicitly declare the location using the `@ComponentScan` annotation.

In the following listing, the `HelloController` class defines a REST controller and a REST endpoint for our example.

Listing 2.2 The `HelloController` class and a REST endpoint

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

The `@RestController` annotation registers the bean in the context and tells Spring that the application uses this instance as a web controller. Also, the annotation specifies that the application has to set the response body of the HTTP response from the method’s return value. The `@GetMapping` annotation maps the /hello path to the implemented method through a GET request. Once you run the application, besides the other lines in the console, you should see something that looks similar to this:

```
Using generated security password: 93a01cf0-794b-4b98-86ef-54860f
```

Each time you run the application, it generates a new password and prints this password in the console as presented in the previous code snippet. You must

use this password to call any of the application's endpoints with HTTP Basic authentication. First, let's try to call the endpoint without using the `Authorization` header:

```
curl http://localhost:8080/hello
```

NOTE

In this book, we use cURL to call the endpoints in all the examples. I consider cURL to be the most readable solution. But if you prefer, you can use a tool of your choice. For example, you might want to have a more comfortable graphical interface. In this case, Postman is an excellent choice. If the operating system you use does not have any of these tools installed, you probably need to install them yourself.

And the response to the call:

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/hello"
}
```

The response status is HTTP 401 Unauthorized. We expected this result as we didn't use the proper credentials for authentication. By default, Spring Security expects the default username (user) with the provided password (in my case, the one starting with `93a01`). Let's try it again but now with the proper credentials:

```
curl -u user:93a01cf0-794b-4b98-86ef-54860f36f7f3 http://localhos
```

The response to the call now is

```
Hello!
```

NOTE

The HTTP 401 Unauthorized status code is a bit ambiguous. Usually, it's used to represent a failed authentication rather than authorization. Developers

use it in the design of the application for cases like missing or incorrect credentials. For a failed authorization, we’d probably use the 403 Forbidden status. Generally, an HTTP 403 means that the server identified the caller of the request, but they don’t have the needed privileges for the call that they are trying to make.

Once we send the correct credentials, you can see in the body of the response precisely what the `HelloController` method we defined earlier returns.

Calling the endpoint with HTTP Basic authentication

With cURL, you can set the HTTP basic username and password with the `-u` flag. Behind the scenes, cURL encodes the string `<username>:<password>` in Base64 and sends it as the value of the `Authorization` header prefixed with the string `Basic`. And with cURL, it’s probably easier for you to use the `-u` flag. But it’s also essential to know what the real request looks like. So, let’s give it a try and manually create the `Authorization` header.

In the first step, take the `<username>:<password>` string and encode it with `Base64`. When our application sends the call, we need to know how to form the correct value for the `Authorization` header. You do this using the Base64 tool in a Linux console. You could also find a web page that encodes strings in Base64, like https://www.base64encode.org. This snippet shows the command in a Linux or a Git Bash console (the -n parameter means no trailing new line should be added):

```
echo -n user:93a01cf0-794b-4b98-86ef-54860f36f7f3 | base64
```

Running this command returns this Base64-encoded string:


```
dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01NDg2MGYzNmY3ZjM=
```


You can now use the Base64-encoded value as the value of the `Authorization` header for the call. This call should generate the same result as the one using the `-u` option:

```
curl -H "Authorization: Basic dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZ
[CA]NDg2MGYzNmY3ZjM="  localhost:8080/hello
```

The result of the call is

```
Hello!
```

There’re no significant security configurations to discuss with a default project. We mainly use the default configurations to prove that the correct dependencies are in place. It does little for authentication and authorization. This implementation isn’t something we want to see in a production-ready application. But the default project is an excellent example that you can use for a start.

With this first example working, at least we know that Spring Security is in place. The next step is to change the configurations such that these apply to the requirements of our project. First, we’ll go deeper with what Spring Boot configures in terms of Spring Security, and then we see how we can override the configurations.

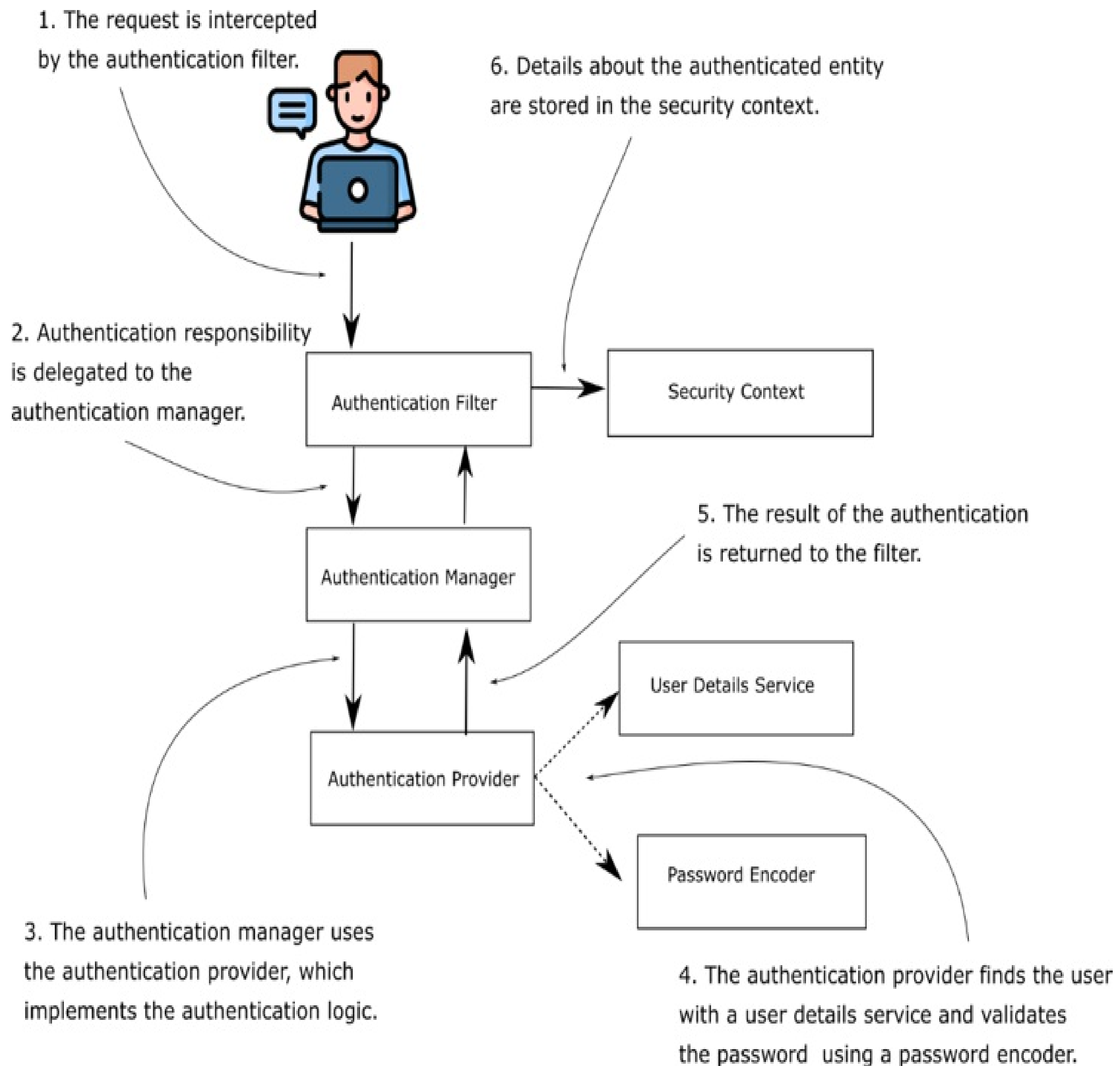
2.2 The big picture of Spring Security class design

In this section, we discuss the main actors in the overall architecture that take part in the process of authentication and authorization. You need to know this aspect because you’ll have to override these preconfigured components to fit the needs of your application. I’ll start by describing how Spring Security architecture for authentication and authorization works and then we’ll apply that to the projects in this chapter. It would be too much to discuss these all at once, so to minimize your learning efforts in this chapter, I’ll discuss the high-level picture for each component. You’ll learn details about each in the following chapters.

In section 2.1, you saw some logic executing for authentication and authorization. We had a default user, and we got a random password each time we started the application. We were able to use this default user and password to call an endpoint. But where is all of this logic implemented? As you probably know already, Spring Boot sets up some components for you, depending on what dependencies you use (the convention-over-configuration that we were discussing at the beginning of this chapter).

In figure 2.2, you can see the big picture of the main actors (components) in the Spring Security architecture and the relationships among these. These components have a preconfigured implementation in the first project. In this chapter, I make you aware of what Spring Boot configures in your application in terms of Spring Security. We'll also discuss the relationships among the entities that are part of the authentication flow presented.

Figure 2.2 The main components acting in the authentication process for Spring Security and the relationships among these. This architecture represents the backbone of implementing authentication with Spring Security. We'll refer to it often throughout the book when discussing different implementations for authentication and authorization.



In figure 2.2, you can see that

1. The authentication filter delegates the authentication request to the authentication manager and, based on the response, configures the security context.
2. The authentication manager uses the authentication provider to process authentication.
3. The authentication provider implements the authentication logic.
4. The user details service implements user management responsibility, which the authentication provider uses in the authentication logic.
5. The password encoder implements password management, which the authentication provider uses in the authentication logic.
6. The security context keeps the authentication data after the authentication process.

In the following paragraphs, I'll discuss these autoconfigured beans:

- `UserDetailsService`
- `PasswordEncoder`

You can see these in figure 2.2 as well. The authentication provider uses these beans to find users and to check their passwords. Let's start with the way you provide the needed credentials for authentication.

An object that implements a `UserDetailsService` interface with Spring Security manages the details about users. Until now, we used the default implementation provided by Spring Boot. This implementation only registers the default credentials in the internal memory of the application. These default credentials are "user" with a default password that's a universally unique identifier (UUID). This default password is randomly generated when the Spring context is loaded (at the app startup). At this time, the application writes the password to the console where you can see it. Thus, you can use it in the example we just worked on in this chapter.

This default implementation serves only as a proof of concept and allows us to see that the dependency is in place. The implementation stores the credentials in-memory—the application doesn't persist the credentials. This approach is suitable for examples or proof of concepts, but you should avoid

it in a production-ready application.

And then we have the `PasswordEncoder`. The `PasswordEncoder` does two things:

- Encodes a password (usually using an encryption or a hashing algorithm)
- Verifies if the password matches an existing encoding

Even if it's not as obvious as the `UserDetailsService` object, the `PasswordEncoder` is mandatory for the Basic authentication flow. The simplest implementation manages the passwords in plain text and doesn't encode these. We'll discuss more details about the implementation of this object in chapter 4. For now, you should be aware that a `PasswordEncoder` exists together with the default `UserDetailsService`. When we replace the default implementation of the `UserDetailsService`, we must also specify a `PasswordEncoder`.

Spring Boot also chooses an authentication method when configuring the defaults, HTTP Basic access authentication. It's the most straightforward access authentication method. Basic authentication only requires the client to send a username and a password through the HTTP `Authorization` header. In the value of the header, the client attaches the prefix `Basic`, followed by the Base64 encoding of the string that contains the username and password, separated by a colon (:).

NOTE

HTTP Basic authentication doesn't offer confidentiality of the credentials. Base64 is only an encoding method for the convenience of the transfer; it's not an encryption or hashing method. While in transit, if intercepted, anyone can see the credentials. Generally, we don't use HTTP Basic authentication without at least HTTPS for confidentiality. You can read the detailed definition of HTTP Basic in RFC 7617 (https://tools.ietf.org/html/rfc7617).

The `AuthenticationProvider` defines the authentication logic, delegating the user and password management. A default implementation of the `AuthenticationProvider` uses the default implementations provided for the

`UserDetailsService` and the `PasswordEncoder`. Implicitly, your application secures all the endpoints. Therefore, the only thing that we need to do for our example is to add the endpoint. Also, there's only one user who can access any of the endpoints, so we can say that there's not much to do about authorization in this case.

HTTP vs. HTTPS

You might have observed that in the examples I presented, I only use HTTP. In practice, however, your applications communicate only over HTTPS. For the examples we discuss in this book, the configurations related to Spring Security aren't different, whether we use HTTP or HTTPS. So that you can focus on the examples related to Spring Security, I won't configure HTTPS for the endpoints in the examples. But, if you want, you can enable HTTPS for any of the endpoints as presented in this sidebar.

There are several patterns to configure HTTPS in a system. In some cases, developers configure HTTPS at the application level; in others, they might use a service mesh or they could choose to set HTTPS at the infrastructure level. With Spring Boot, you can easily enable HTTPS at the application level, as you'll learn in the next example in this sidebar.

In any of these configuration scenarios, you need a certificate signed by a certification authority (CA). Using this certificate, the client that calls the endpoint knows whether the response comes from the authentication server and that nobody intercepted the communication. You can buy such a certificate, if you need it. If you only need to configure HTTPS to test your application, you can generate a self-signed certificate using a tool like OpenSSL (https://www.openssl.org/). Let's generate our self-signed certificate and then configure it in the project:

```
openssl req -newkey rsa:2048 -x509 -keyout key.pem -out cert.pem
```

After running the `openssl` command in a terminal, you'll be asked for a password and details about your CA. Because it is only a self-signed certificate for a test, you can input any data there; just make sure to remember the password. The command outputs two files: key.pem (the private key) and cert.pem (a public certificate). We'll use these files further to generate our

self-signed certificate for enabling HTTPS. In most cases, the certificate is the Public Key Cryptography Standards #12 (PKCS12). Less frequently, we use a Java KeyStore (JKS) format. Let's continue our example with a PKCS12 format. For an excellent discussion on cryptography, I recommend *Real-World Cryptography* by David Wong (Manning, 2020).

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out certifica
```

The second command we use receives as input the two files generated by the first command and outputs the self-signed certificate.

Mind that if you run these commands in a Bash shell on a Windows system, you might need to add `winpty` before it, as shown in the next code snippet:

```
winpty openssl req -newkey rsa:2048 -x509 -keyout key.pem -out ce
winpty openssl pkcs12 -export -in cert.pem -inkey key.pem -out ce
```

Finally, having the self-signed certificate, you can configure HTTPS for your endpoints. Copy the certificate.p12 file into the resources folder of the Spring Boot project and add the following lines to your application.properties file:

```
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:certificate.p12
server.ssl.key-store-password=12345       #A
```

The password (in my case, "12345") was requested in the prompt after running the command for generating the certificate. This is the reason why you don't see it in the command. Now, let's add a test endpoint to our application and then call it using HTTPS:

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

If you use a self-signed certificate, you should configure the tool you use to make the endpoint call so that it skips testing the authenticity of the

certificate. If the tool tests the authenticity of the certificate, it won’t recognize it as being authentic, and the call won’t work. With cURL, you can use the `-k` option to skip testing the authenticity of the certificate:

```
curl -k -u user:93a01cf0-794b-4b98-86ef-54860f36f7f3  https://loc
```

The response to the call is

```
Hello!
```

Remember that even if you use HTTPS, the communication between components of your system isn’t bulletproof. Many times, I’ve heard people say, “I’m not encrypting this anymore, I use HTTPS!” While helpful in protecting communication, HTTPS is just one of the bricks of the security wall of a system. Always treat the security of your system with responsibility and take care of all the layers involved in it.

2.3 Overriding default configurations

Now that you know the defaults of your first project, it’s time to see how you can replace these. You need to understand the options you have for overriding the default components because this is the way you plug in your custom implementations and apply security as it fits your application. And, as you’ll learn in this section, the development process is also about how you write configurations to keep your applications highly maintainable. With the projects we’ll work on, you’ll often find multiple ways to override a configuration. This flexibility can create confusion. I frequently see a mix of different styles of configuring different parts of Spring Security in the same application, which is undesirable. So this flexibility comes with a caution. You need to learn how to choose from these, so this section is also about knowing what your options are.

In some cases, developers choose to use beans in the Spring context for the configuration. In other cases, they override various methods for the same purpose. The speed with which the Spring ecosystem evolved is probably one of the main factors that generated these multiple approaches. Configuring a project with a mix of styles is not desirable as it makes the code difficult to understand and affects the maintainability of the application. Knowing your

options and how to use them is a valuable skill, and it helps you better understand how you should configure application-level security in a project.

In this section, you'll learn how to configure a `UserDetailsService` and a `PasswordEncoder`. These two components usually take part in authentication, and most applications customize them depending on their requirements. While we'll discuss details about customizing them in chapters 3 and 4, it's essential to see how to plug in a custom implementation. The implementations we use in this chapter are all provided by Spring Security.

2.3.1 Customizing user details management

The first component we talked about in this chapter was `UserDetailsService`. As you saw, the application uses this component in the process of authentication. In this section, you'll learn to define a custom bean of type `UserDetailsService`. We'll do this to override the default one configured by Spring Boot. As you'll see in more detail in chapter 3, you have the option to create your own implementation or to use a predefined one provided by Spring Security. In this chapter, we aren't going to detail the implementations provided by Spring Security or create our own implementation just yet. I'll use an implementation provided by Spring Security, named `InMemoryUserDetailsManager`. With this example, you'll learn how to plug this kind of object into your architecture.

NOTE

Interfaces in Java define contracts between objects. In the class design of the application, we use interfaces to decouple objects that use one another. To enforce this interface characteristic when discussing those in this book, I mainly refer to them as *contracts*.

To show you the way to override this component with an implementation that we choose, we'll change what we did in the first example. Doing so allows us to have our own managed credentials for authentication. For this example, we don't implement our class, but we use an implementation provided by Spring Security.

In this example, we use the `InMemoryUserDetailsManager` implementation. Even if this implementation is a bit more than just a `UserDetailsService`, for now, we only refer to it from the perspective of a `UserDetailsService`. This implementation stores credentials in memory, which can then be used by Spring Security to authenticate a request.

NOTE

An `InMemoryUserDetailsManager` implementation isn't meant for production-ready applications, but it's an excellent tool for examples or proof of concepts. In some cases, all you need is users. You don't need to spend the time implementing this part of the functionality. In our case, we use it to understand how to override the default `UserDetailsService` implementation.

We start by defining a configuration class. Generally, we declare configuration classes in a separate package named config. Listing 2.3 shows the definition for the configuration class. You can also find the example in the project ssia-ch2-ex2.

NOTE

The examples in this book are designed for Java 17, which is the latest long-term supported Java version. For this reason, I expect more and more applications in production to use Java 17. So it makes a lot of sense to use this version for the examples in this book. If you app uses Java 8, or Java 11 I recommend you read as well the first edition of this book which was designed for earlier Java, Spring Boot, and Spring Security versions.

Listing 2.3 The configuration class for the `UserDetailsService` bean

```
@Configuration       #A
public class ProjectConfig {

  @Bean      #B
  UserDetailsService userDetailsService() {
    return new InMemoryUserDetailsManager();
  }
}
```

We annotate the class with `@Configuration`. The `@Bean` annotation instructs Spring to add the instance returned by the method to the Spring context. If you execute the code exactly as it is now, you'll no longer see the autogenerated password in the console. The application now uses the instance of type `UserDetailsService` you added to the context instead of the default autoconfigured one. But, at the same time, you won't be able to access the endpoint anymore for two reasons:

- You don't have any users.
- You don't have a `PasswordEncoder`.

In figure 2.2, you can see that authentication depends on a `PasswordEncoder` as well. Let's solve these two issues step by step. We need to

1. Create at least one user who has a set of credentials (username and password)
2. Add the user to be managed by our implementation of `UserDetailsService`
3. Define a bean of the type `PasswordEncoder` that our application can use to verify a given password with the one stored and managed by `UserDetailsService`

First, we declare and add a set of credentials that we can use for authentication to the instance of `InMemoryUserDetailsManager`. In chapter 3, we'll discuss more about users and how to manage them. For the moment, let's use a predefined builder to create an object of the type `UserDetails`.

Note

You'll sometimes see that I use `var` in the code. Java 10 introduced the reserved type name `var`, and you can only use it for local declarations. In this book, I use it to make the syntax shorter, as well as to hide the variable type. We'll discuss the types hidden by `var` in later chapters, so you don't have to worry about that type until it's time to analyze it properly. But using var helps me take out details that are unnecessary for the moment to allow you focus on the discussed topic.

When building the instance, we have to provide the username, the password,

and at least one authority. The "charitalics"authority is an action allowed for that user, and we can use any string for this. In listing 2.4, I name the authority `read`, but because we won't use this authority for the moment, this name doesn't really matter.

Listing 2.4 Creating a user with the `User` builder class for `UserDetailsService`

```
@Configuration
public class ProjectConfig {

  @Bean
  UserDetailsService userDetailsService() {
    var user = User.withUsername("john")    #A
                 .password("12345")     #A
                 .authorities("read")    #A
                 .build();     #A

     return new InMemoryUserDetailsManager(user);    #B
  }
}
```

Note

You'll find the class `User` in the org.springframework.security.core.userdetails package. It's the builder implementation we use to create the object to represent the user. Also, as a general rule in this book, if I don't present how to write a class in a code listing, it means Spring Security provides it.

As presented in listing 2.4, we have to provide a value for the username, one for the password, and at least one authority. But this is still not enough to allow us to call the endpoint. We also need to declare a `PasswordEncoder`.

When using the default `UserDetailsService`, a `PasswordEncoder` is also auto-configured. Because we overrode `UserDetailsService,` we also have to declare a `PasswordEncoder`. Trying the example now, you'll see an exception when you call the endpoint. When trying to do the authentication, Spring Security realizes it doesn't know how to manage the password and fails. The exception looks like that in the next code snippet, and you should see it in your application's console. The client gets back an HTTP 401

Unauthorized message and an empty response body:

```
curl -u john:12345 http://localhost:8080/hello
```

The result of the call in the app's console is

```
java.lang.IllegalArgumentException:
There is no PasswordEncoder mapped for the id "null"
    at
org.springframework.security.crypto.password
[CA].DelegatingPasswordEncoder$
[CA]UnmappedIdPasswordEncoder.matches(DelegatingPasswordEncoder.j
    at org.springframework.security.crypto.password
[CA].DelegatingPasswordEncoder.matches(DelegatingPasswordEncoder.
```

To solve this problem, we can add a `PasswordEncoder` bean in the context, the same as we did with the `UserDetailsService`. For this bean, we use an existing implementation of `PasswordEncoder`:

```
@Bean
public PasswordEncoder passwordEncoder() {
  return NoOpPasswordEncoder.getInstance();
}
```

NOTE

The `NoOpPasswordEncoder` instance treats passwords as plain text. It doesn't encrypt or hash them. For matching, `NoOpPasswordEncoder` only compares the strings using the underlying `equals(Object o)` method of the `String` class. You shouldn't use this type of `PasswordEncoder` in a production-ready app. `NoOpPasswordEncoder` is a good option for examples where you don't want to focus on the hashing algorithm of the password. Therefore, the developers of the class marked it as `@Deprecated`, and your development environment will show its name with a strikethrough.

You can see the full code of the configuration class in the following listing.

Listing 2.5 The full definition of the configuration class

```
@Configuration
public class ProjectConfig {
```

```
  @Bean
  UserDetailsService userDetailsService() {
    var user = User.withUsername("john")
                 .password("12345")
                 .authorities("read")
                 .build();

    return new InMemoryUserDetailsManager(user);
  }

  @Bean    #A
  PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

Let’s try the endpoint with the new user having the username John and the password 12345:

```
curl -u john:12345 http://localhost:8080/hello
Hello!
```

NOTE

Knowing the importance of unit and integration tests, some of you might already wonder why we don’t also write tests for our examples. You will actually find the related Spring Security integration tests with all the examples provided with this book. However, to help you focus on the presented topics for each chapter, I have separated the discussion about testing Spring Security integrations and detail this in chapter 18.

2.3.2 Applying authorization at the endpoint level

With new management for the users in place, as described in section 2.3.1, we can now discuss the authentication method and configuration for endpoints. You’ll learn plenty of things regarding authorization configuration in chapters 7 through 12. But before diving into details, you must understand the big picture. And the best way to achieve this is with our first example. With default configuration, all the endpoints assume you have a valid user managed by the application. Also, by default, your app uses HTTP Basic

authentication, but you can easily override this configuration.

As you'll learn in the next chapters, HTTP Basic authentication doesn't fit into most application architectures. Sometimes we'd like to change it to match our application. Similarly, not all endpoints of an application need to be secured, and for those that do, we might need to choose different authentication methods and authorization rules. To customize authentication and authorization, we'll need to define a bean of type `SecurityFilterChain`. For this example, I'll continue writing the code in the project ssia-ch2-ex3.

Listing 2.6 Defining a `SecurityFilterChain` bean

```
@Configuration
public class ProjectConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {


    return http.build();
  }

  // Omitted code

}
```

We can then alter the configuration using different methods of the `HttpSecurity` object as shown in the next listing.

Listing 2.7 Using the `HttpSecurity` parameter to alter the configuration

```
@Configuration
public class ProjectConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());    #A

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()    #B
```

```
    );

    return http.build();
  }

  // Omitted code

}
```

The code in listing 2.7 configures endpoint authorization with the same behavior as the default one. You can call the endpoint again to see that it behaves the same as in the previous test from section 2.3.1. With a slight change, you can make all the endpoints accessible without the need for credentials. You'll see how to do this in the following listing.

Listing 2.8 Using `permitAll()` to change the authorization configuration

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.anyRequest().permitAll()    #A
    );

    return http.build();
  }

  // Omitted code
}
```

Now, we can call the /hello endpoint without the need for credentials. The `permitAll()` call in the configuration, together with the `anyRequest()` method, makes all the endpoints accessible without the need for credentials:

```
curl http://localhost:8080/hello
```

And the response body of the call is

```
Hello!
```

In this example you used two configuration methods:

1. `httpBasic()` – which helped you configure the authentication approach. Calling this method you instructed your app to accept HTTP Basic as an authentication method.
2. `authorizeHttpRequests()` – which helped you configure the authorization rules at the endpoint level. Calling this method you instructed the app how it should authorize the requests received on specific endpoints.

For both methods you had to use a `Customizer` object as a parameter. `Customizer` is a contract you implement to define the customization for either Spring Security element you configure: the authentication, the authorization, or particular protection mechanisms such as CSRF or CORS (which will discuss in chapters 9 and 10). The following snippet shows you the definition of the `Customizer` interface. Observe that `Customizer` is a functional interface (so we can use lambda expressions to implement it) and the `withDefaults()` method I used in listing 2.8 is in fact just a `Customizer` implementation that does nothing.

```
@FunctionalInterface
public interface Customizer<T> {
  void customize(T t);

  static <T> Customizer<T> withDefaults() {
    return (t) -> {
    };
  }
}
```

In earlier Spring Security versions you could apply configurations without a `Customizer` object by using a chaining syntax, as shows in the following code snippet. Observe that instead of providing a `Customizer` object to the `authorizeHttpRequests()` method, the configuration just follows the method's call.

```
http.authorizeHttpRequests()
       .anyRequest().authenticated()
```

The reason this approach has been left behind is because a `Customizer` object allows you more flexibility in moving the configuration elsewhere where that's needed. Sure, with simple examples, using lambda expressions is comfortable. But in real-world apps the configurations can grow a lot. In such cases, the ability to move these configurations in separate classes helps you keep these configurations easier to maintain and test.

The purpose of this example is to give you a feeling for how to override default configurations. We'll get into the details about authorization in chapters 7 through 10.

Note

In earlier versions of Spring Security, a security configuration class needed to extend a class named `WebSecurityConfigurerAdapter`. We don't use this practice anymore today. In case your app uses an older codebase or you need to upgrade an older codebase, I recommend you read also the first edition of Spring Security in Action.

2.3.3 Configuring in different ways

One of the confusing aspects of creating configurations with Spring Security is having multiple ways to configure the same thing. In this section, you'll learn alternatives for configuring `UserDetailsService` and `PasswordEncoder`. It's essential to know the options you have so that you can recognize these in the examples that you find in this book or other sources like blogs and articles. It's also important that you understand how and when to use these in your application. In further chapters, you'll see different examples that extend the information in this section.

Let's take the first project. After we created a default application, we managed to override `UserDetailsService` and `PasswordEncoder` by adding new implementations as beans in the Spring context. Let's find another way of doing the same configurations for `UserDetailsService` and `PasswordEncoder`.

We can directly use the `SecurityFilterChain` bean to set both the

`UserDetailsService` and the `PasswordEncoder` as shown in the following listing. You can find this example in the project ssia-ch2-ex3.

Listing 2.9 Setting `UserDetailsService` with the `SecurityFilterChain` bean

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
        c -> c.anyRequest().authenticated()
    );

    var user = User.withUsername("john")     #A
        .password("12345")
        .authorities("read")
        .build();

    var userDetailsService =      #B
      new InMemoryUserDetailsManager(user);

    http.userDetailsService(userDetailsService);     #C

    return http.build();
}

  // Omitted code

}
```

In listing 2.9, you can observe that we declare the `UserDetailsService` in the same way as in listing 2.5. The difference is that now this is done locally inside the bean method creating the `SecurityFilterChain`. We also call the `userDetailsService()` method from the `HttpSecurity` to register the `UserDetailsService` instance. Listing 2.10 shows the full contents of the configuration class.

Listing 2.10 Full definition of the configuration class

```
@Configuration
public class ProjectConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
        c -> c.anyRequest().authenticated()
    );

    var user = User.withUsername("john")    #A
        .password("12345")
        .authorities("read")
        .build();

    var userDetailsService =     #B
        new InMemoryUserDetailsManager(user);

    http.userDetailsService(userDetailsService);    #C

    return http.build();
  }

  @Bean
  PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

Any of these configuration options are correct. The first option, where we add the beans to the context, lets you inject the values in another class where you might potentially need them. But if you don't need that for your case, the second option would be equally good.

2.3.4 Defining custom authentication logic

As you've already observed, Spring Security components provide a lot of flexibility, which offers us a lot of options when adapting these to the architecture of our applications. Up to now, you've learned the purpose of `UserDetailsService` and `PasswordEncoder` in the Spring Security architecture. And you saw a few ways to configure them. It's time to learn

that you can also customize the component that delegates to these, the `AuthenticationProvider`.

Figure 2.3 The `AuthenticationProvider` implements the authentication logic. It receives the request from the `AuthenticationManager` and delegates finding the user to a `UserDetailsService`, and verifying the password to a `PasswordEncoder`.

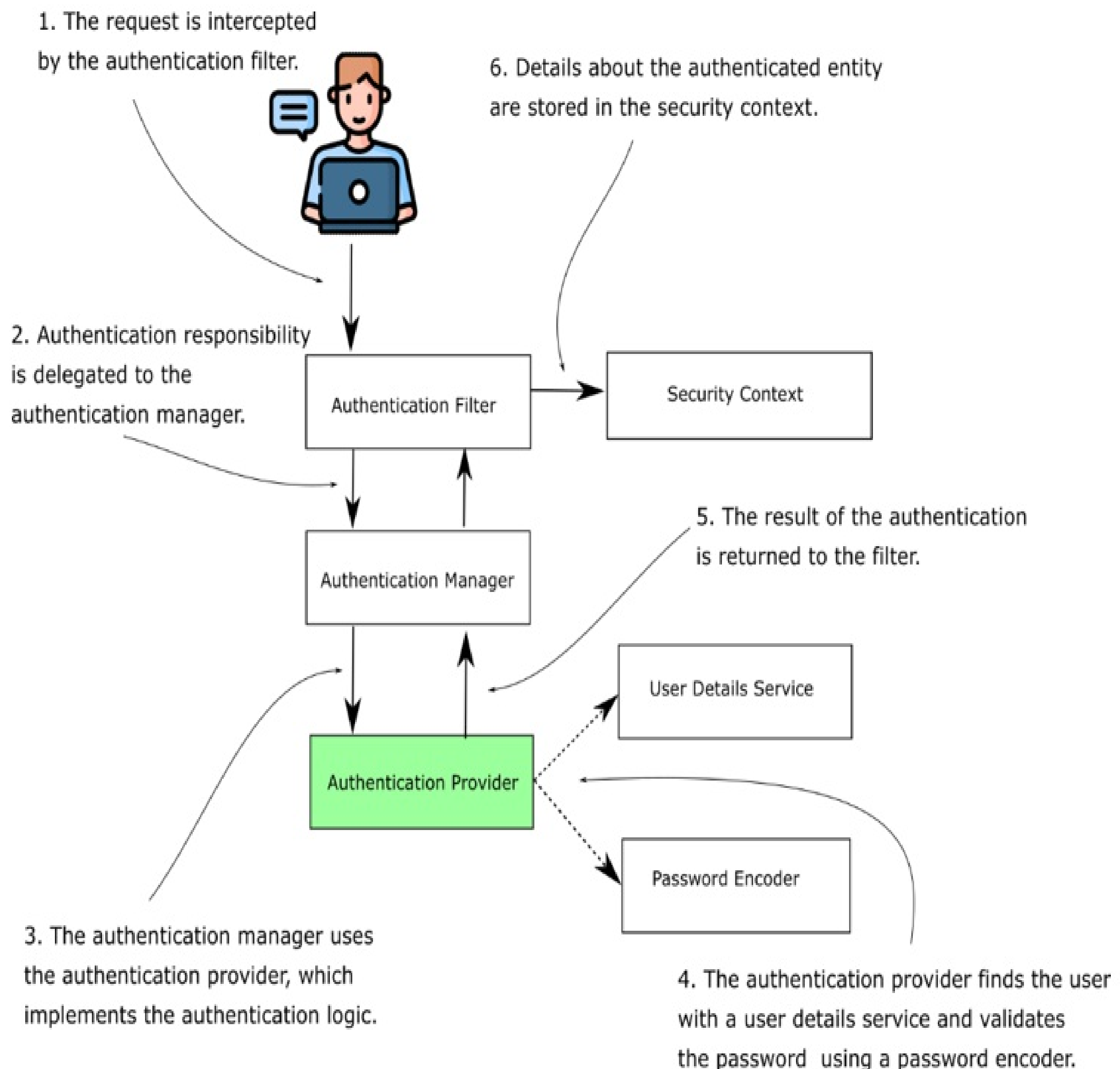


Figure 2.3 shows the `AuthenticationProvider`, which implements the authentication logic and delegates to the `UserDetailsService` and `PasswordEncoder` for user and password management. So we could say that with this section, we go one step deeper in the authentication architecture to

learn how to implement custom authentication logic with `AuthenticationProvider`.

Because this is a first example, I only show you the brief picture so that you better understand the relationship between the components in the architecture. But we'll detail more in chapters 3 through 6.

I recommend that you consider the responsibilities as designed in the Spring Security architecture. This architecture is loosely coupled with fine-grained responsibilities. That design is one of the things that makes Spring Security flexible and easy to integrate in your applications. But depending on how you make use of its flexibility, you could change the design as well. You have to be careful with these approaches as they can complicate your solution. For example, you could choose to override the default `AuthenticationProvider` in a way in which you no longer need a `UserDetailsService` or `PasswordEncoder`. With that in mind, the following listing shows how to create a custom authentication provider. You can find this example in the project ssia-ch2-ex4.

Listing 2.11 Implementing the `AuthenticationProvider` interface

```
@Component
public class CustomAuthenticationProvider
[CA]implements AuthenticationProvider {

  @Override
  public Authentication authenticate
  [CA](Authentication authentication) throws AuthenticationExcept

    // authentication logic here
  }

  @Override
  public boolean supports(Class<?> authenticationType) {

    // type of the Authentication implementation here
  }
}
```

The `authenticate(Authentication authentication)` method represents all the logic for authentication, so we'll add an implementation like that in listing

2.12. I’ll explain the usage of the `supports()` method in detail in chapter 6. For the moment, I recommend you take its implementation for granted. It’s not essential for the current example.

Listing 2.12 Implementing the authentication logic

```
@Override
public Authentication authenticate(
   Authentication authentication)
     throws AuthenticationException {

   String username = authentication.getName();    #A
   String password = String.valueOf(
      authentication.getCredentials());

   if ("john".equals(username) &&       #B
       "12345".equals(password)) {
     return new UsernamePasswordAuthenticationToken(
                  username,
                  password,
                  Arrays.asList());
   } else {
     throw new AuthenticationCredentialsNotFoundException("Error!
   }

}
```

As you can see, here the condition of the `if-else` clause is replacing the responsibilities of `UserDetailsService` and `PasswordEncoder`. Your are not required to use the two beans, but if you work with users and passwords for authentication, I strongly suggest you separate the logic of their management. Apply it as the Spring Security architecture designed it, even when you override the authentication implementation.

You might find it useful to replace the authentication logic by implementing your own `AuthenticationProvider`. If the default implementation doesn’t fit entirely into your application’s requirements, you can decide to implement custom authentication logic. The full `AuthenticationProvider` implementation looks like the one in the next listing.

Listing 2.13 The full implementation of the authentication provider

```
@Component
public class CustomAuthenticationProvider
  implements AuthenticationProvider {

  @Override
  public Authentication authenticate(
     Authentication authentication)
       throws AuthenticationException {

      String username = authentication.getName();
      String password = String.valueOf(authentication.getCredenti

      if ("john".equals(username) &&
          "12345".equals(password)) {
        return new UsernamePasswordAuthenticationToken(
             username, password, Arrays.asList());
      } else {
        throw new AuthenticationCredentialsNotFoundException("Err
      }
    }

    @Override
    public boolean supports(Class<?> authenticationType) {
        return UsernamePasswordAuthenticationToken
                   .class
                   .isAssignableFrom(authenticationType);
    }
}
```

In the configuration class, you can register the `AuthenticationProvider` in the `configure(AuthenticationManagerBuilder auth)` method shown in the following -listing.

Listing 2.14 Registering the new implementation of `AuthenticationProvider`

```
@Configuration
public class ProjectConfig {

  private final CustomAuthenticationProvider authenticationProvid

  public ProjectConfig(
    CustomAuthenticationProvider authenticationProvider) {

    this.authenticationProvider = authenticationProvider;
  }
```

```
  @Bean
  SecurityFilterChain configure(HttpSecurity http) throws Excepti
    http.httpBasic(Customizer.withDefaults());

    http.authenticationProvider(authenticationProvider);

    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );

    return http.build();
  }
}
```

You can now call the endpoint, which is accessible by the only user recognized, as defined by the authentication logic—John, with the password 12345:

```
curl -u john:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

In chapter 6, you'll learn more details about the `AuthenticationProvider` and how to override its behavior in the authentication process. In that chapter, we'll also discuss the `Authentication` interface and its implementations, such as the `UserPasswordAuthenticationToken`.

2.3.5 Using multiple configuration classes

In the previously implemented examples, we only used a configuration class. It is, however, good practice to separate the responsibilities even for the configuration classes. We need this separation because the configuration starts to become more complex. In a production-ready application, you probably have more declarations than in our first examples. You also might find it useful to have more than one configuration class to make the project readable.

It's always a good practice to have only one class per each responsibility. For this example, we can separate user management configuration from

authorization configuration. We do that by defining two configuration classes: `UserManagementConfig` (defined in listing 2.15) and `WebAuthorizationConfig` (defined in listing 2.16). You can find this example in the project ssia-ch2-ex5.

Listing 2.15 Defining the configuration class for user and password management

```
@Configuration
public class UserManagementConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var userDetailsService = new InMemoryUserDetailsManager();

    var user = User.withUsername("john")
                .password("12345")
                .authorities("read")
                .build();

    userDetailsService.createUser(user);
    return userDetailsService;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

In this case, the `UserManagementConfig` class only contains the two beans that are responsible for user management: `UserDetailsService` and `PasswordEncoder`. The next listing shows this definition.

Listing 2.16 Defining the configuration class for authorization management

```
@Configuration
public class WebAuthorizationConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());
```

```
    http.authorizeHttpRequests(
        c -> c.anyRequest().authenticated()
    );

    return http.build();
  }
}
```

Here the `WebAuthorizationConfig` class needs define a bean of type `SecurityFilterChain` to configure the authentication and authorization rules.

2.4 Summary

- Spring Boot provides some default configurations when you add Spring Security to the dependencies of the application.
- You implement the basic components for authentication and authorization: `UserDetailsService`, `PasswordEncoder`, and `AuthenticationProvider`.
- You can define users with the `User` class. A user should at least have a username, a password, and an authority. Authorities are actions that you allow a user to do in the context of the application.
- A simple implementation of a `UserDetailsService` that Spring Security provides is `InMemoryUserDetailsManager.` You can add users to such an instance of `UserDetailsService` to manage the user in the application's memory.
- The `NoOpPasswordEncoder` is an implementation of the `PasswordEncoder` contract that uses passwords in cleartext. This implementation is good for learning examples and (maybe) proof of concepts, but not for production-ready applications.
- You can use the `AuthenticationProvider` contract to implement custom authentication logic in the application.
- There are multiple ways to write configurations, but in a single application, you should choose and stick to one approach. This helps to make your code cleaner and easier to understand.

3 Managing users

This chapter covers

- Describing a user with the `UserDetails` interface
- Using the `UserDetailsService` in the authentication flow
- Creating a custom implementation of `UserDetailsService`
- Creating a custom implementation of `UserDetailsManager`
- Using the `JdbcUserDetailsManager` in the authentication flow

One of my colleagues from the university cooks pretty well. He’s not a chef in a fancy restaurant, but he’s quite passionate about cooking. One day, when sharing thoughts in a discussion, I asked him about how he manages to remember so many recipes. He told me that’s easy. “You don’t have to remember the whole recipe, but the way basic ingredients match with each other. It’s like some real-world contracts that tell you what you can mix or should not mix. Then for each recipe, you only remember some tricks.”

This analogy is similar to the way architectures work. With any robust framework, we use contracts to decouple the implementations of the framework from the application built upon it. With Java, we use interfaces to define the contracts. A programmer is similar to a chef, knowing how the ingredients “work” together to choose just the right “implementation.” The programmer knows the framework’s abstractions and uses those to integrate with it.

This chapter is about understanding in detail one of the fundamental roles you encountered in the first example we worked on in chapter 2—the `UserDetailsService`. Along with the `UserDetailsService`, we’ll discuss the following interfaces (contracts):

- `UserDetails,` which describes the user for Spring Security.
- `GrantedAuthority,` which allows us to define actions that the user can execute.
- `UserDetailsManager`, which extends the `UserDetailsService` contract.

Beyond the inherited behavior, it also describes actions like creating a user and modifying or deleting a user's password.

From chapter 2, you already have an idea of the roles of the `UserDetailsService` and the `PasswordEncoder` in the authentication process. But we only discussed how to plug in an instance defined by you instead of using the default one configured by Spring Boot. We have more details to discuss:

- What implementations are provided by Spring Security and how to use them
- How to define a custom implementation for contracts and when to do so
- Ways to implement interfaces that you find in real-world applications
- Best practices for using these interfaces

The plan is to start with how Spring Security understands the user definition. For this, we'll discuss the `UserDetails` and `GrantedAuthority` contracts. Then we'll detail the `UserDetailsService` and how `UserDetailsManager` extends this contract. You'll apply implementations for these interfaces (like `InMemoryUserDetailsManager`, `JdbcUserDetailsManager`, and `LdapUserDetailsManager`). When these implementations aren't a good fit for your system, you'll write a custom implementation.

3.1 Implementing authentication in Spring Security

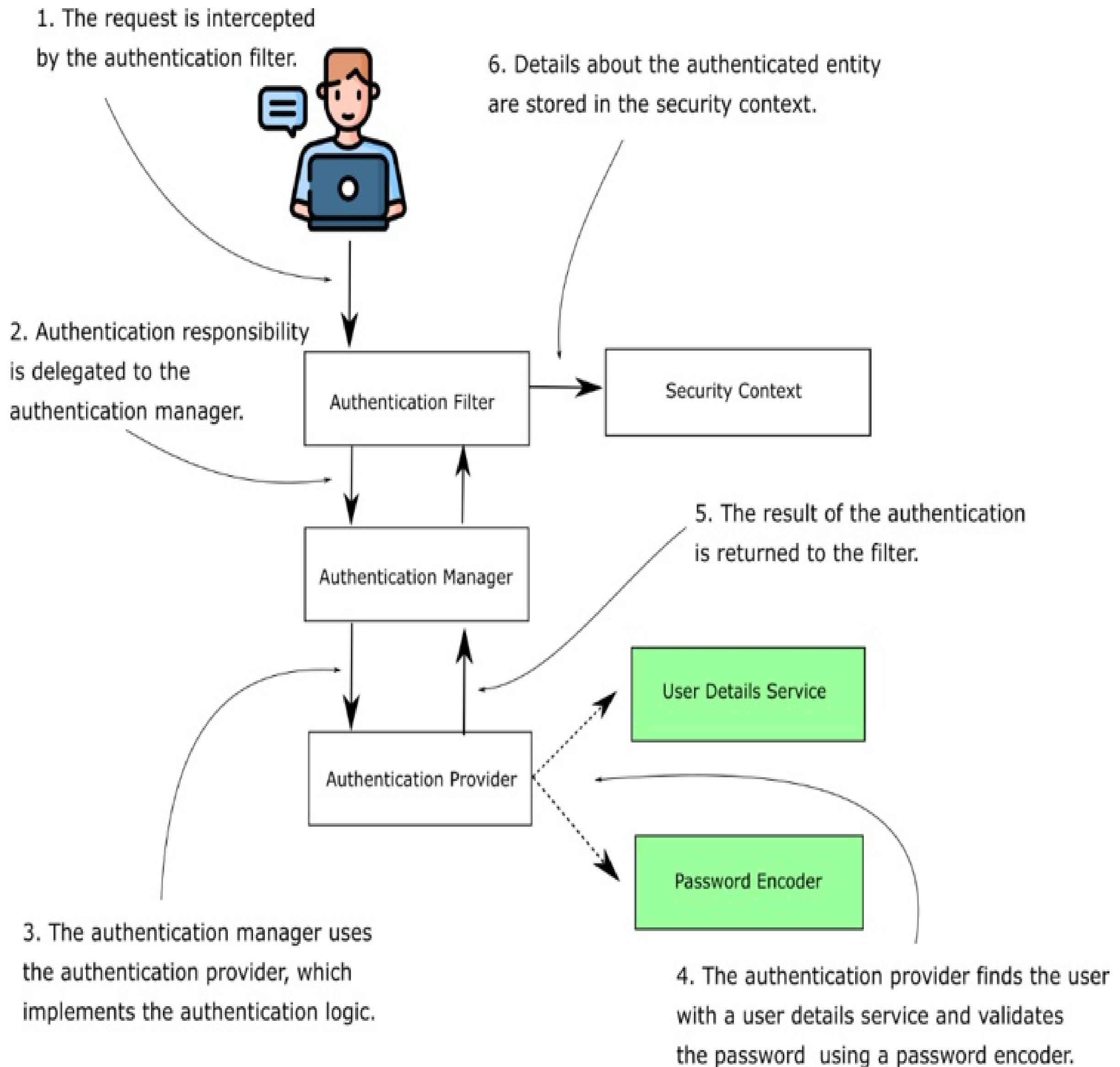
In the previous chapter, we got started with Spring Security. In the first example, we discussed how Spring Boot defines some defaults that define how a new application initially works. You have also learned how to override these defaults using various alternatives that we often find in apps. But we only considered the surface of these so that you have an idea of what we'll be doing. In this chapter, and chapters 4 and 5, we'll discuss these interfaces in more detail, together with different implementations and where you might find them in real-world applications.

Figure 3.1 presents the authentication flow in Spring Security. This architecture is the backbone of the authentication process as implemented by Spring Security. It's really important to understand it because you'll rely on it

in any Spring Security implementation. You’ll observe that we discuss parts of this architecture in almost all the chapters of this book. You’ll see it so often that you’ll probably learn it by heart, which is good. If you know this architecture, you’re like a chef who knows their ingredients and can put together any recipe.

In figure 3.1, the shaded boxes represent the components that we start with: the `UserDetailsService` and the `PasswordEncoder`. These two components focus on the part of the flow that I often refer to as “the user management part.” In this chapter, the `UserDetailsService` and the `PasswordEncoder` are the components that deal directly with user details and their credentials. We’ll discuss the `PasswordEncoder` in detail in chapter 4.

Figure 3.1 Spring Security’s authentication flow. The `AuthenticationFilter` intercepts the request and delegates the authentication responsibility to the `AuthenticationManager`. To implement the authentication logic, the `AuthenticationManager` uses an authentication provider. To check the username and the password, the `AuthenticationProvider` uses a `UserDetailsService` and a `PasswordEncoder`.

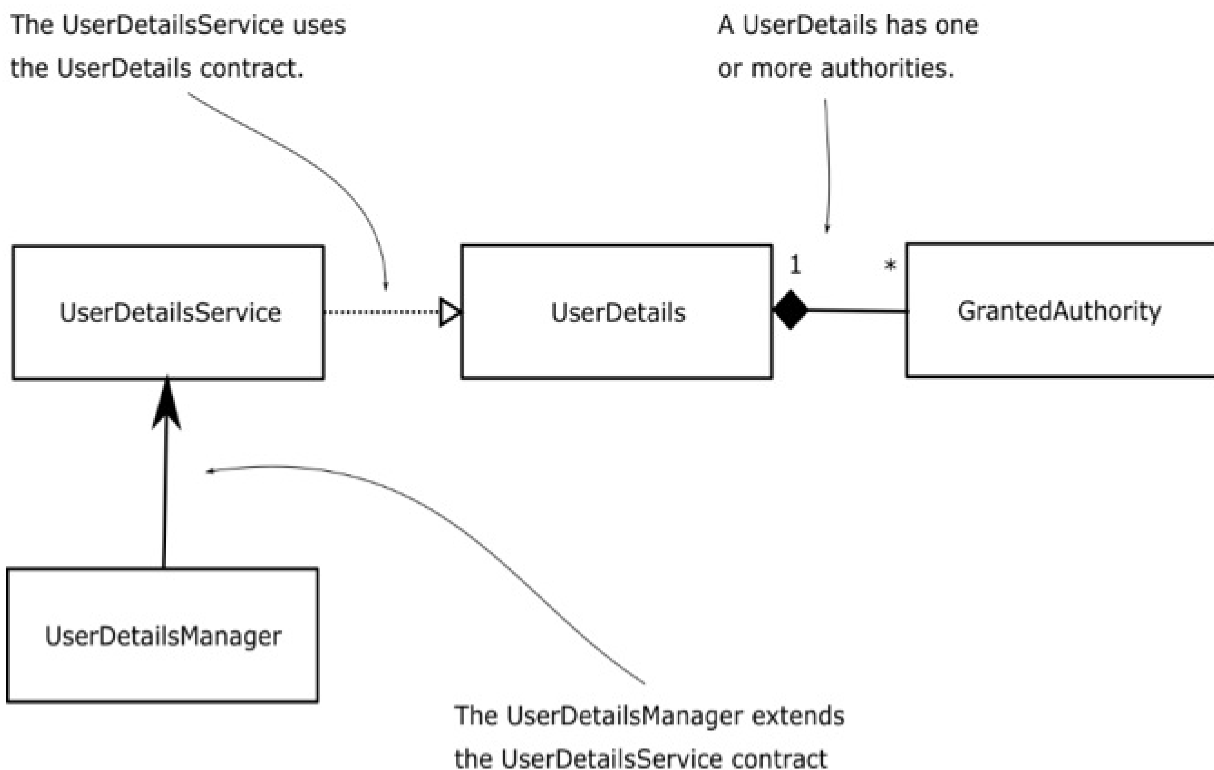


As part of user management, we use the `UserDetailsService` and `UserDetailsManager` interfaces. The `UserDetailsService` is only responsible for retrieving the user by username. This action is the only one needed by the framework to complete authentication. The `UserDetailsManager` adds behavior that refers to adding, modifying, or deleting the user, which is a required functionality in most applications. The separation between the two contracts is an excellent example of the "charitalics"interface segregation principle. Separating the interfaces allows for better flexibility because the framework doesn’t force you to implement behavior if your app doesn’t need it. If the app only needs to authenticate the

users, then implementing the `UserDetailsService` contract is enough to cover the desired functionality. To manage the users, `UserDetailsService` and the `UserDetailsManager` components need a way to represent them.

Spring Security offers the `UserDetails` contract, which you have to implement to describe a user in the way the framework understands. As you'll learn in this chapter, in Spring Security a user has a set of privileges, which are the actions the user is allowed to do. We'll work a lot with these privileges in chapters 7 through 12 when discussing authorization. But for now, Spring Security represents the actions that a user can do with the `GrantedAuthority` interface. We often call these "charitalics"authorities, and a user has one or more authorities. In figure 3.2, you find a representation of the relationship between the components of the user management part of the authentication flow.

Figure 3.2 Dependencies between the components involved in user management. The `UserDetailsService` returns the details of a user, finding the user by its name. The `UserDetails` contract describes the user. A user has one or more authorities, represented by the `GrantedAuthority` interface. To add operations such as create, delete, or change password to the user, the `UserDetailsManager` contract extends `UserDetailsService` to add operations.



Understanding the links between these objects in the Spring Security architecture and ways to implement them gives you a wide range of options to choose from when working on applications. Any of these options could be the right puzzle piece in the app that you are working on, and you need to make your choice wisely. But to be able to choose, you first need to know what you can choose from.

3.2 Describing the user

In this section, you'll learn how to describe the users of your application such that Spring Security understands them. Learning how to represent users and make the framework aware of them is an essential step in building an authentication flow. Based on the user, the application makes a decision—a call to a certain functionality is or isn't allowed. To work with users, you first need to understand how to define the prototype of the user in your application. In this section, I describe by example how to establish a blueprint for your users in a Spring Security application.

For Spring Security, a user definition should respect the `UserDetails` contract. The `UserDetails` contract represents the user as understood by Spring Security. The class of your application that describes the user has to implement this interface, and in this way, the framework understands it.

3.2.1 Describing users with the UserDetails contract

In this section, you'll learn how to implement the `UserDetails` interface to describe the users in your application. We'll discuss the methods declared by the `UserDetails` contract to understand how and why we implement each of them. Let's start first by looking at the interface as presented in the following listing.

Listing 3.1 The `UserDetails` interface

```
public interface UserDetails extends Serializable {
  String getUsername();     #A
  String getPassword();
  Collection<? extends GrantedAuthority>
  [CA]getAuthorities();      #B
```

```
  boolean isAccountNonExpired();     #C
  boolean isAccountNonLocked();
  boolean isCredentialsNonExpired();
  boolean isEnabled();
}
```

The `getUsername()` and `getPassword()` methods return, as you'd expect, the username and the password. The app uses these values in the process of authentication, and these are the only details related to authentication from this contract. The other five methods all relate to authorizing the user for accessing the application's resources.

Generally, the app should allow a user to do some actions that are meaningful in the application's context. For example, the user should be able to read data, write data, or delete data. We say a user has or hasn't the privilege to perform an action, and an authority represents the privilege a user has. We implement the `getAuthorities()` method to return the group of authorities granted for a user.

NOTE

As you'll learn in chapter 6, Spring Security uses authorities to refer either to fine-grained privileges or to roles, which are groups of privileges. To make your reading more effortless, in this book, I refer to the fine-grained privileges as authorities.

Furthermore, as seen in the `UserDetails` contract, a user can

- Let the account expire
- Lock the account
- Let the credentials expire
- Disable the account

If you choose to implement these user restrictions in your application's logic, you need to override the following methods: `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()`, `isEnabled()`, such that those needing to be enabled return true. Not all applications have accounts that expire or get locked with certain conditions. If you do not need to implement these functionalities in your application, you can simply make

these four methods return true.

NOTE

The names of the last four methods in the `UserDetails` interface may sound strange. One could argue that these are not wisely chosen in terms of clean coding and maintainability. For example, the name `isAccountNonExpired()` looks like a double negation, and at first sight, might create confusion. But analyze all four method names with attention. These are named such that they all return false for the case in which the authorization should fail and true otherwise. This is the right approach because the human mind tends to associate the word “false” with negativity and the word “true” with positive scenarios.

3.2.2 Detailing on the GrantedAuthority contract

As you observed in the definition of the `UserDetails` interface in section 3.2.1, the actions granted for a user are called authorities. In chapters 7 through 12, we’ll write authorization configurations based on these user authorities. So it’s essential to know how to define them.

The authorities represent what the user can do in your application. Without authorities, all users would be equal. While there are simple applications in which the users are equal, in most practical scenarios, an application defines multiple kinds of users. An application might have users that can only read specific information, while others also can modify the data. And you need to make your application differentiate between them, depending on the functional requirements of the application, which are the authorities a user needs. To describe the authorities in Spring Security, you use the `GrantedAuthority` interface.

Before we discuss implementing `UserDetails`, let’s understand the `GrantedAuthority` interface. We use this interface in the definition of the user details. It represents a privilege granted to the user. A user must have at least one authority. Here’s the implementation of the `GrantedAuthority` definition:

```
public interface GrantedAuthority extends Serializable {
```

```
    String getAuthority();
}
```

To create an authority, you only need to find a name for that privilege so you can refer to it later when writing the authorization rules. For example, a user can read the records managed by the application or delete them. You write the authorization rules based on the names you give to these actions.

In this chapter, we'll implement the `getAuthority()` method to return the authority's name as a `String`. The `GrantedAuthority` interface has only one abstract method, and in this book, you often find examples in which we use a lambda expression for its implementation. Another possibility is to use the `SimpleGrantedAuthority` class to create authority instances. The `SimpleGrantedAuthority` class offers a way to create immutable instances of the type `GrantedAuthority`. You provide the authority name when building the instance. In the next code snippet, you'll find two examples of implementing a `GrantedAuthority`. Here we make use of a lambda expression and then use the `SimpleGrantedAuthority` class:

```
GrantedAuthority g1 = () -> "READ";
GrantedAuthority g2 = new SimpleGrantedAuthority("READ");
```

3.2.3 Writing a minimal implementation of UserDetails

In this section, you'll write your first implementation of the `UserDetails` contract. We start with a basic implementation in which each method returns a static value. Then we change it to a version that you'll more likely find in a practical scenario, and one that allows you to have multiple and different instances of users. Now that you know how to implement the `UserDetails` and `GrantedAuthority` interfaces, we can write the simplest definition of a user for an application.

With a class named `DummyUser`, let's implement a minimal description of a user as in listing 3.2. I use this class mainly to demonstrate implementing the methods for the `UserDetails` contract. Instances of this class always refer to only one user, `"bill"`, who has the password `"12345"` and an authority named `"READ"`.

Listing 3.2 The `DummyUser` class

```
public class DummyUser implements UserDetails {

  @Override
  public String getUsername() {
    return "bill";
  }

  @Override
  public String getPassword() {
    return "12345";
  }

  // Omitted code

}
```

The class in the listing 3.2 implements the `UserDetails` interface and needs to implement all its methods. You find here the implementation of `getUsername()` and `getPassword()`. In this example, these methods only return a fixed value for each of the properties.

Next, we add a definition for the list of authorities. Listing 3.3 shows the implementation of the `getAuthorities()` method. This method returns a collection with only one implementation of the `GrantedAuthority` interface.

Listing 3.3 Implementation of the `getAuthorities()` method

```
public class DummyUser implements UserDetails {

  // Omitted code

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities()
    return List.of(() -> "READ");
  }

  // Omitted code

}
```

Finally, you have to add an implementation for the last four methods of the `UserDetails` interface. For the `DummyUser` class, these always return true,

which means that the user is forever active and usable. You find the examples in the following listing.

Listing 3.4 Implementation of the last four `UserDetails` interface methods

```
public class DummyUser implements UserDetails {

  // Omitted code

  @Override
  public boolean isAccountNonExpired() {
    return true;
  }

  @Override
  public boolean isAccountNonLocked() {
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }

  @Override
  public boolean isEnabled() {
    return true;
  }

  // Omitted code

}
```

Of course, this minimal implementation means that all instances of the class represent the same user. It’s a good start to understanding the contract, but not something you would do in a real application. For a real application, you should create a class that you can use to generate instances that can represent different users. In this case, your definition would at least have the username and the password as attributes in the class, as shown in the next listing.

Listing 3.5 A more practical implementation of the `UserDetails` interface

```
public class SimpleUser implements UserDetails {
```

```
  private final String username;
  private final String password;

  public SimpleUser(String username, String password) {
    this.username = username;
    this.password = password;
  }

  @Override
  public String getUsername() {
    return this.username;
  }

  @Override
  public String getPassword() {
    return this.password;
  }

  // Omitted code

}
```

3.2.4 Using a builder to create instances of the UserDetails type

Some applications are simple and don't need a custom implementation of the `UserDetails` interface. In this section, we take a look at using a builder class provided by Spring Security to create simple user instances. Instead of declaring one more class in your application, you quickly obtain an instance representing your user with the `User` builder class.

The `User` class from the org.springframework.security.core.userdetails package is a simple way to build instances of the `UserDetails` type. Using this class, you can create immutable instances of `UserDetails`. You need to provide at least a username and a password, and the username shouldn't be an empty string. Listing 3.6 demonstrates how to use this builder. Building the user in this way, you don't need to have a custom implementation of the `UserDetails` contract.

Listing 3.6 Constructing a user with the `User` builder class

```
UserDetails u = User.withUsername("bill")
                .password("12345")
```

```
                 .authorities("read", "write")
                 .accountExpired(false)
                 .disabled(true)
                 .build();
```

With the previous listing as an example, let’s go deeper into the anatomy of the `User` builder class. The `User.withUsername(String username)` method returns an instance of the builder class `UserBuilder` nested in the `User` class. Another way to create the builder is by starting from another instance of `UserDetails`. In listing 3.7, the first line constructs a `UserBuilder`, starting with the username given as a string. Afterward, we demonstrate how to create a builder beginning with an already existing instance of `UserDetails`.

Listing 3.7 Creating the User.UserBuilder instance

```
User.UserBuilder builder1 =
[CA]User.withUsername("bill");      #A

UserDetails u1 = builder1
                 .password("12345")
                 .authorities("read", "write")
                 .passwordEncoder(p -> encode(p))      #B
                 .accountExpired(false)
                 .disabled(true)
                 .build();      #C

User.UserBuilder builder2 = User.withUserDetails(u);    #D

UserDetails u2 = builder2.build();
```

You can see with any of the builders defined in listing 3.7 that you can use the builder to obtain a user represented by the `UserDetails` contract. At the end of the build pipeline, you call the `build()` method. It applies the function defined to encode the password if you provide one, constructs the instance of `UserDetails`, and returns it.

NOTE

Mind that the password encoder is given here as a `Function<String, String>` and not in the shape of the `PasswordEncoder` interface provided by Spring Security. This function’s only responsibility is to transform a

password in a given encoding. In the next section, we'll discuss in detail the `PasswordEncoder` contract from Spring Security that we used in chapter 2. We'll discuss the `PasswordEncoder` contract in more detail in chapter 4.

3.2.5 Combining multiple responsibilities related to the user

In the previous section, you learned how to implement the `UserDetails` interface. In real-world scenarios, it's often more complicated. In most cases, you find multiple responsibilities to which a user relates. And if you store users in a database, and then in the application, you would need a class to represent the persistence entity as well. Or, if you retrieve users through a web service from another system, then you would probably need a data transfer object to represent the user instances. Assuming the first, a simple but also typical case, let's consider we have a table in an SQL database in which we store the users. To make the example shorter, we give each user only one authority. The following listing shows the entity class that maps the table.

Listing 3.8 Defining the JPA `User` entity class

```
@Entity
public class User {

  @Id
  private Long id;
  private String username;
  private String password;
  private String authority;

  // Omitted getters and setters

}
```

If you make the same class also implement the Spring Security contract for user details, the class becomes more complicated. What do you think about how the code looks in the next listing? From my point of view, it is a mess. I would get lost in it.

Listing 3.9 The `User` class has two responsibilities

```
@Entity
public class User implements UserDetails {

  @Id
  private int id;
  private String username;
  private String password;
  private String authority;

  @Override
  public String getUsername() {
    return this.username;
  }

  @Override
  public String getPassword() {
    return this.password;
  }

  public String getAuthority() {
    return this.authority;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities()
    return List.of(() -> authority);
  }

  // Omitted code

}
```

The class contains JPA annotations, getters, and setters, of which both `getUsername()` and `getPassword()`override the methods in the `UserDetails` contract. It has a `getAuthority()` method that returns a `String`, as well as a `getAuthorities()` method that returns a `Collection`. The `getAuthority()` method is just a getter in the class, while `getAuthorities()` implements the method in the `UserDetails` interface. And things get even more complicated when adding relationships to other entities. Again, this code isn't friendly at all!

How can we write this code to be cleaner? The root of the muddy aspect of the previous code example is a mix of two responsibilities. While it's true that you need both in the application, in this case, nobody says that you have

to put these into the same class. Let’s try to separate those by defining a separate class called `SecurityUser`, which decorates the `User` class. As listing 3.11 shows, the `SecurityUser` class implements the `UserDetails` contract and uses that to plug our user into the Spring Security architecture. The `User` class has only its JPA entity responsibility remaining.

Listing 3.10 Implementing the `User` class only as a JPA entity

```
@Entity
public class User {

  @Id
  private int id;
  private String username;
  private String password;
  private String authority;

  // Omitted getters and setters

}
```

The `User` class in listing 3.10 has only its JPA entity responsibility remaining, and, thus, becomes more readable. If you read this code, you can now focus exclusively on details related to persistence, which are not important from the Spring Security perspective. In the next listing, we implement the `SecurityUser` class to wrap the `User` entity.

Listing 3.11 The `SecurityUser` class implements the `UserDetails` contract

```
public class SecurityUser implements UserDetails {

  private final User user;

  public SecurityUser(User user) {
    this.user = user;
  }

  @Override
  public String getUsername() {
    return user.getUsername();
  }

  @Override
```

```
  public String getPassword() {
    return user.getPassword();
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities()
    return List.of(() -> user.getAuthority());
  }

  // Omitted code

}
```

As you can observe, we use the `SecurityUser` class only to map the user details in the system to the `UserDetails` contract understood by Spring Security. To mark the fact that the `SecurityUser` makes no sense without a `User` entity, we make the field final. You have to provide the user through the constructor. The `SecurityUser` class decorates the `User` entity class and adds the needed code related to the Spring Security contract without mixing the code into a JPA entity, thereby implementing multiple different tasks.

NOTE

You can find different approaches to separate the two responsibilities. I don't want to say that the approach I present in this section is the best or the only one. Usually, the way you choose to implement the class design varies a lot from one case to another. But the main idea is the same: avoid mixing responsibilities and try to write your code as decoupled as possible to increase the maintainability of your app.

3.3 Instructing Spring Security on how to manage users

In the previous section, you implemented the `UserDetails` contract to describe users such that Spring Security understands them. But how does Spring Security manage users? Where are they taken from when comparing credentials, and how do you add new users or change existing ones? In chapter 2, you learned that the framework defines a specific component to which the authentication process delegates user management: the

`UserDetailsService` instance. We even defined a `UserDetailsService` to override the default implementation provided by Spring Boot.

In this section, we experiment with various ways of implementing the `UserDetailsService` class. You'll understand how user management works by implementing the responsibility described by the `UserDetailsService` contract in our example. After that, you'll find out how the `UserDetailsManager` interface adds more behavior to the contract defined by the `UserDetailsService`. At the end of this section, we'll use the provided implementations of the `UserDetailsManager` interface offered by Spring Security. We'll write an example project where we'll use one of the best known implementations provided by Spring Security, the `JdbcUserDetailsManager` class. Learning this, you'll know how to tell Spring Security where to find users, which is essential in the authentication flow.

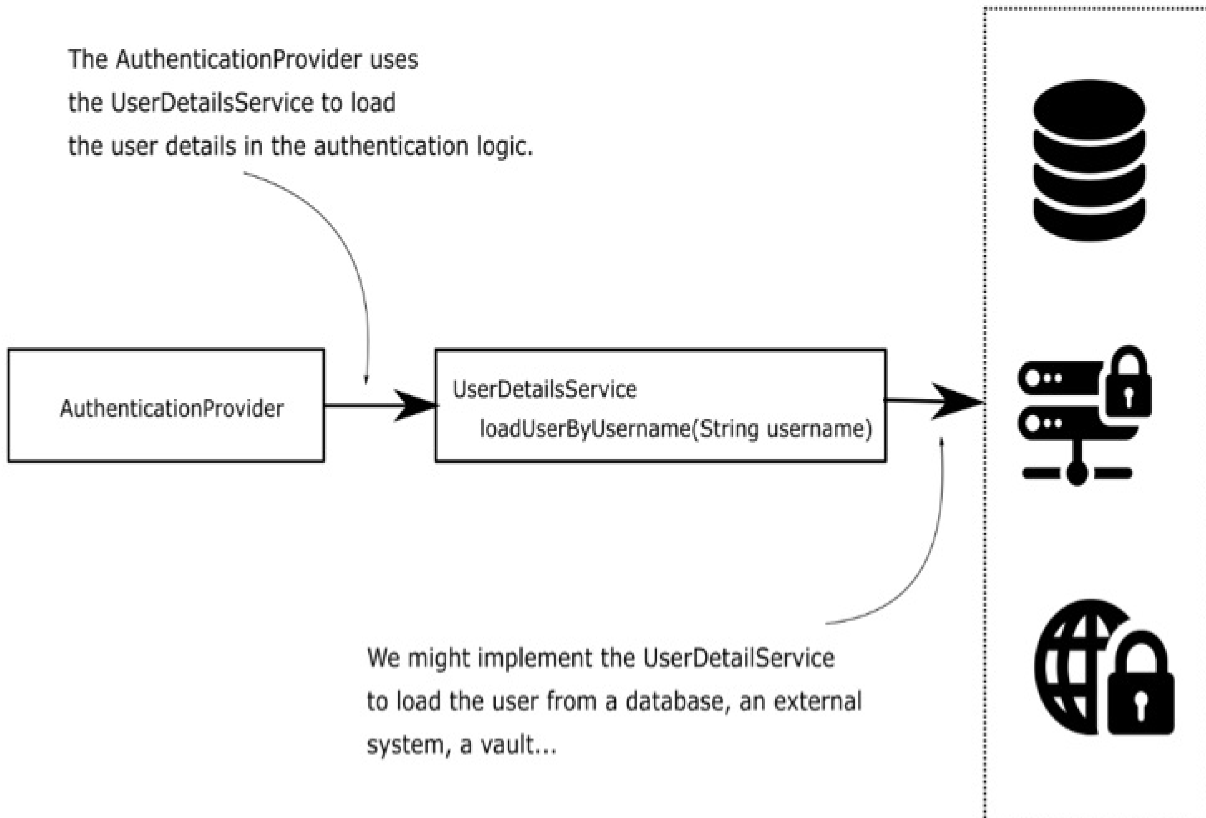
3.3.1 Understanding the UserDetailsService contract

In this section, you'll learn about the `UserDetailsService` interface definition. Before understanding how and why to implement it, you must first understand the contract. It is time to detail more on `UserDetailsService` and how to work with implementations of this component. The `UserDetailsService` interface contains only one method, as follows:

```
public interface UserDetailsService {

  UserDetails loadUserByUsername(String username)
      throws UsernameNotFoundException;
}
```

The authentication implementation calls the `loadUserByUsername(String username)` method to obtain the details of a user with a given username (figure 3.3). The username is, of course, considered unique. The user returned by this method is an implementation of the `UserDetails` contract. If the username doesn't exist, the method throws a `UsernameNotFoundException`.

Figure 3.3 The `AuthenticationProvider` is the component that implements the authentication logic and uses the `UserDetailsService` to load details about the user. To find the user by username, it calls the `loadUserByUsername(String username)` method.



NOTE

The `UsernameNotFoundException` is a `RuntimeException`. The `throws` clause in the `UserDetailsService` interface is only for documentation purposes. The `UsernameNotFoundException` inherits directly from the type `AuthenticationException`, which is the parent of all the exceptions related to the process of authentication. `AuthenticationException` inherits further the `RuntimeException` class.

3.3.2 Implementing the UserDetailsService contract

In this section, we work on a practical example to demonstrate the implementation of the `UserDetailsService`. Your application manages details about credentials and other user aspects. It could be that these are stored in a database or handled by another system that you access through a web service or by other means (figure 3.3). Regardless of how this happens in your system, the only thing Spring Security needs from you is an

implementation to retrieve the user by username.

In the next example, we write a `UserDetailsService` that has an in-memory list of users. In chapter 2, you used a provided implementation that does the same thing, the `InMemoryUserDetailsManager`. Because you are already familiar with how this implementation works, I have chosen a similar functionality, but this time to implement on our own. We provide a list of users when we create an instance of our `UserDetailsService` class. You can find this example in the project ssia-ch3-ex1. In the package named model, we define the `UserDetails` as presented by the following listing.

Listing 3.12 The implementation of the `UserDetails` interface

```
public class User implements UserDetails {

  private final String username;      #A
  private final String password;
  private final String authority;      #B

  public User(String username, String password, String authority)
    this.username = username;
    this.password = password;
    this.authority = authority;
  }

  @Override
  public Collection<? extends GrantedAuthority> getAuthorities()
    return List.of(() -> authority);      #C
  }

  @Override
  public String getPassword() {
    return password;
  }

  @Override
  public String getUsername() {
    return username;
  }

  @Override
  public boolean isAccountNonExpired() {     #D
    return true;
  }
```

```
  @Override
  public boolean isAccountNonLocked() {
    return true;
  }

  @Override
  public boolean isCredentialsNonExpired() {
    return true;
  }

  @Override
  public boolean isEnabled() {
    return true;
  }
}
```

In the package named services, we create a class called `InMemoryUserDetailsService`. The following listing shows how we implement this class.

Listing 3.13 The implementation of the `UserDetailsService` interface

```
public class InMemoryUserDetailsService implements UserDetailsSer

  private final List<UserDetails> users;     #A

  public InMemoryUserDetailsService(List<UserDetails> users) {
    this.users = users;
  }

  @Override
  public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {

    return users.stream()
      .filter(     #B
         u -> u.getUsername().equals(username)
      )
      .findFirst()      #C
      .orElseThrow(      #D
        () -> new UsernameNotFoundException("User not found")
      );
   }
}
```

The `loadUserByUsername(String username)` method searches the list of users for the given username and returns the desired `UserDetails` instance. If there is no instance with that username, it throws a `UsernameNotFoundException`. We can now use this implementation as our `UserDetailsService`. The next listing shows how we add it as a bean in the configuration class and register one user within it.

Listing 3.14 `UserDetailsService` registered as a bean in the configuration class

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    UserDetails u = new User("john", "12345", "read");
    List<UserDetails> users = List.of(u);
    return new InMemoryUserDetailsService(users);
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

Finally, we create a simple endpoint and test the implementation. The following listing defines the endpoint.

Listing 3.15 The definition of the endpoint used for testing the implementation

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

When calling the endpoint using cURL, we observe that for user John with password 12345, we get back an HTTP 200 OK. If we use something else, the application returns 401 Unauthorized.

```
curl -u john:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

3.3.3 Implementing the UserDetailsManager contract

In this section, we discuss using and implementing the `UserDetailsManager` interface. This interface extends and adds more methods to the `UserDetailsService` contract. Spring Security needs the `UserDetailsService` contract to do the authentication. But generally, in applications, there is also a need for managing users. Most of the time, an app should be able to add new users or delete existing ones. In this case, we implement a more particular interface defined by Spring Security, `UserDetailsManager`. It extends `UserDetailsService` and adds more operations that we need to implement.

```
public interface UserDetailsManager extends UserDetailsService {
  void createUser(UserDetails user);
  void updateUser(UserDetails user);
  void deleteUser(String username);
  void changePassword(String oldPassword, String newPassword);
  boolean userExists(String username);
}
```

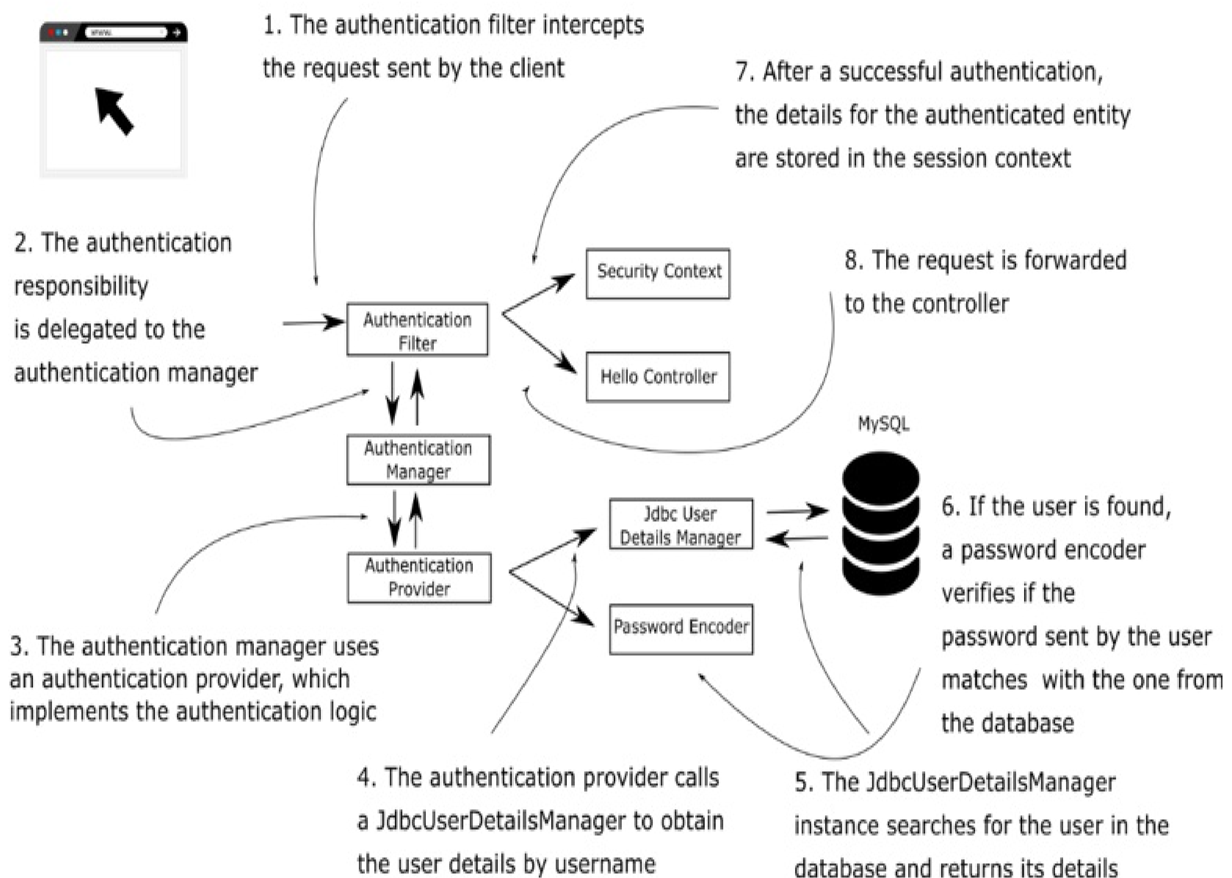
The `InMemoryUserDetailsManager` object that we used in chapter 2 is actually a `UserDetailsManager`. At that time, we only considered its `UserDetailsService` characteristics. Project ssia-ch3-ex2 accompanies the example in this section.

Using a JdbcUserDetailsManager for user management

Beside the `InMemoryUserDetailsManager`, we often use another `UserDetailManager` implementation, `JdbcUserDetailsManager`. The `JdbcUserDetailsManager` class manages users in an SQL database. It connects to the database directly through JDBC. This way, the `JdbcUserDetailsManager` is independent of any other framework or specification related to database connectivity.

To understand how the `JdbcUserDetailsManager` works, it's best if you put it into action with an example. In the following example, you implement an application that manages the users in a MySQL database using the `JdbcUserDetailsManager`. Figure 3.4 provides an overview of the place the `JdbcUserDetailsManager` implementation takes in the authentication flow.

Figure 3.4 The Spring Security authentication flow. Here we use a `JdbcUserDetailsManager` as our `UserDetailsService` component. The `JdbcUserDetailsManager` uses a database to manage users.



You'll start working on our demo application about how to use the `JdbcUserDetailsManager` by creating a database and two tables. In our case, we name the database `spring`, and we name one of the tables users and the other authorities. These names are the default table names known by the `JdbcUserDetailsManager`. As you'll learn at the end of this section, the `JdbcUserDetailsManager` implementation is flexible and lets you override

these default names if you want to do so. The purpose of the users table is to keep user records. The `JdbcUserDetailsManager` implementation expects three columns in the users table: a username, a password, and enabled, which you can use to deactivate the user.

You can choose to create the database and its structure yourself either by using the command-line tool for your database management system (DBMS) or a client application. For example, for MySQL, you can choose to use MySQL Workbench to do this. But the easiest would be to let Spring Boot itself run the scripts for you. To do this, just add two more files to your project in the resources folder: schema.sql and data.sql. In the schema.sql file, you add the queries related to the structure of the database, like creating, altering, or dropping tables. In the data.sql file, you add the queries that work with the data inside the tables, like `INSERT`, `UPDATE`, or `DELETE`. Spring Boot automatically runs these files for you when you start the application. A simpler solution for building examples that need databases is using an H2 in-memory database. This way, you don't need to install a separate DBMS solution.

Note

If you prefer, you could go with H2 as well (as I do in the ssia-ch3-ex2 project) when developing the applications presented in this book. But in most example of this book, I chose to implement the examples with an external DBMS to make it clear it's an external component of the system and, in this way, avoid confusion.

You use the code in the next listing to create the users table with a MySQL server. You can add this script to the schema.sql file in your Spring Boot project.

Listing 3.16 The SQL query for creating the users table

```
CREATE TABLE IF NOT EXISTS `spring`.`users` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(45) NOT NULL,
  `password` VARCHAR(45) NOT NULL,
  `enabled` INT NOT NULL,
  PRIMARY KEY (`id`));
```

The authorities table stores authorities per user. Each record stores a username and an authority granted for the user with that username.

Listing 3.17 The SQL query for creating the authorities table

```
CREATE TABLE IF NOT EXISTS `spring`.`authorities` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `username` VARCHAR(45) NOT NULL,
  `authority` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`));
```

NOTE

For simplicity, and to allow you focus on the Spring Security configurations we discuss, in the examples provided with this book, I skip the definitions of indexes or foreign keys.

To make sure you have a user for testing, insert a record in each of the tables. You can add these queries in the data.sql file in the resources folder of the Spring Boot project:

```
INSERT INTO `spring`.`authorities`
(username, authority)
VALUES
('john', 'write');

INSERT INTO `spring`.`users`
(username, password, enabled)
VALUES
('john', '12345', '1');
```

For your project, you need to add at least the dependencies stated in the following listing. Check your pom.xml file to make sure you added these dependencies.

Listing 3.18 Dependencies needed to develop the example project

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

NOTE

In your examples, you can use any SQL database technology as long as you add the correct JDBC driver to the dependencies.

Remember, you need to add the JDBC driver according to the database technology you use. For example, if you'd use MySQL, you need to add the MySQL driver dependency as presented in the next snippet.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

You can configure a data source in the application.properties file of the project or as a separate bean. If you choose to use the application.properties file, you need to add the following lines to that file:

```
spring.datasource.url=jdbc:h2:mem:ssia
spring.datasource.username=sa
spring.datasource.password=
spring.sql.init.mode=always
```

In the configuration class of the project, you define the `UserDetailsService` and the `PasswordEncoder`. The `JdbcUserDetailsManager` needs the `DataSource` to connect to the database. The data source can be autowired through a parameter of the method (as presented in the next listing) or through an attribute of the class.

Listing 3.19 Registering the `JdbcUserDetailsManager` in the configuration class

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService(DataSource dataSou
    return new JdbcUserDetailsManager(dataSource);
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

To access any endpoint of the application, you now need to use HTTP Basic authentication with one of the users stored in the database. To prove this, we create a new endpoint as shown in the following listing and then call it with cURL.

Listing 3.20 The test endpoint to check the implementation

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

In the next code snippet, you find the result when calling the endpoint with the correct username and password:

```
curl -u john:12345 http://localhost:8080/hello
```

The response to the call is

```
Hello!
```

The `JdbcUserDetailsManager` also allows you to configure the queries used. In the previous example, we made sure we used the exact names for the

tables and columns, as the `JdbcUserDetailsManager` implementation expects those. But it could be that for your application, these names are not the best choice. The next listing shows how to override the queries for the `JdbcUserDetailsManager`.

Listing 3.21 Changing `JdbcUserDetailsManager’s` queries to find the user

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSourc
  String usersByUsernameQuery =
     "select username, password, enabled
      [CA]from users where username = ?";
  String authsByUserQuery =
     "select username, authority
      [CA]from spring.authorities where username = ?";

  var userDetailsManager = new JdbcUserDetailsManager(dataSource)
  userDetailsManager.setUsersByUsernameQuery(usersByUsernameQuery
  userDetailsManager.setAuthoritiesByUsernameQuery(authsByUserQue
  return userDetailsManager;
}
```

In the same way, we can change all the queries used by the `JdbcUserDetailsManager` implementation.

Exercise

Write a similar application for which you name the tables and the columns differently in the database. Override the queries for the `JdbcUserDetailsManager` implementation (for example, the authentication works with a new table structure). The project ssia-ch3-ex2 features a possible solution.

Using an LdapUserDetailsManager for user management

Spring Security also offers an implementation of `UserDetailsManager` for LDAP. Even if it is less popular than the `JdbcUserDetailsManager`, you can count on it if you need to integrate with an LDAP system for user management. In the project ssia-ch3-ex3, you can find a simple demonstration of using the `LdapUserDetailsManager`. Because I can’t use a

real LDAP server for this demonstration, I have set up an embedded one in my Spring Boot application. To set up the embedded LDAP server, I defined a simple LDAP Data Interchange Format (LDIF) file. The following listing shows the content of my LDIF file.

Listing 3.22 The definition of the LDIF file

```
dn: dc=springframework,dc=org      #A
objectclass: top
objectclass: domain
objectclass: extensibleObject
dc: springframework

dn: ou=groups,dc=springframework,dc=org      #B
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: uid=john,ou=groups,dc=springframework,dc=org      #C
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John
sn: John
uid: john
userPassword: 12345
```

In the LDIF file, I add only one user for which we need to test the app's behavior at the end of this example. We can add the LDIF file directly to the resources folder. This way, it's automatically in the classpath, so we can easily refer to it later. I named the LDIF file server.ldif. To work with LDAP and to allow Spring Boot to start an embedded LDAP server, you need to add pom.xml to the dependencies as in the following code snippet:

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ldap</artifactId>
</dependency>
<dependency>
  <groupId>com.unboundid</groupId>
  <artifactId>unboundid-ldapsdk</artifactId>
</dependency>
```

In the application.properties file, you also need to add the configurations for the embedded LDAP server as presented in the following code snippet. The values the app needs to boot the embedded LDAP server include the location of the LDIF file, a port for the LDAP server, and the base domain component (DN) label values:

```
spring.ldap.embedded.ldif=classpath:server.ldif
spring.ldap.embedded.base-dn=dc=springframework,dc=org
spring.ldap.embedded.port=33389
```

Once you have an LDAP server for authentication, you can configure your application to use it. The next listing shows you how to configure the `LdapUserDetailsManager` to enable your app to authenticate users through the LDAP server.

Listing 3.23 The definition of the `LdapUserDetailsManager` in the configuration file

```
@Configuration
public class ProjectConfig {

  @Bean      #A
  public UserDetailsService userDetailsService() {
    var cs = new DefaultSpringSecurityContextSource(    #B
      [CA]"ldap://127.0.0.1:33389/dc=springframework,dc=org");
    cs.afterPropertiesSet();

    var manager = new LdapUserDetailsManager(cs);      #C

    manager.setUsernameMapper(      #D
      [CA]new DefaultLdapUsernameToDnMapper("ou=groups", "uid"));

    manager.setGroupSearchBase("ou=groups");    #E

    return manager;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

Let’s also create a simple endpoint to test the security configuration. I added

a controller class as presented in the next code snippet:

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

Now start the app and call the /hello endpoint. You need to authenticate with user John if you want the app to allow you to call the endpoint. The next code snippet shows you the result of calling the endpoint with cURL:

```
curl -u john:12345 http://localhost:8080/hello
```

The response to the call is

```
Hello!
```

3.4 Summary

- The `UserDetails` interface is the contract you use to describe a user in Spring Security.
- The `UserDetailsService` interface is the contract that Spring Security expects you to implement in the authentication architecture to describe the way the application obtains user details.
- The `UserDetailsManager` interface extends `UserDetailsService` and adds the behavior related to creating, changing, or deleting a user.
- Spring Security provides a few implementations of the `UserDetailsManager` contract. Among these are `InMemoryUserDetailsManager`, `JdbcUserDetailsManager`, and `LdapUserDetailsManager`.
- The `JdbcUserDetailsManager` class has the advantage of directly using JDBC and does not lock the application in to other frameworks.

4 Managing passwords

This chapter covers

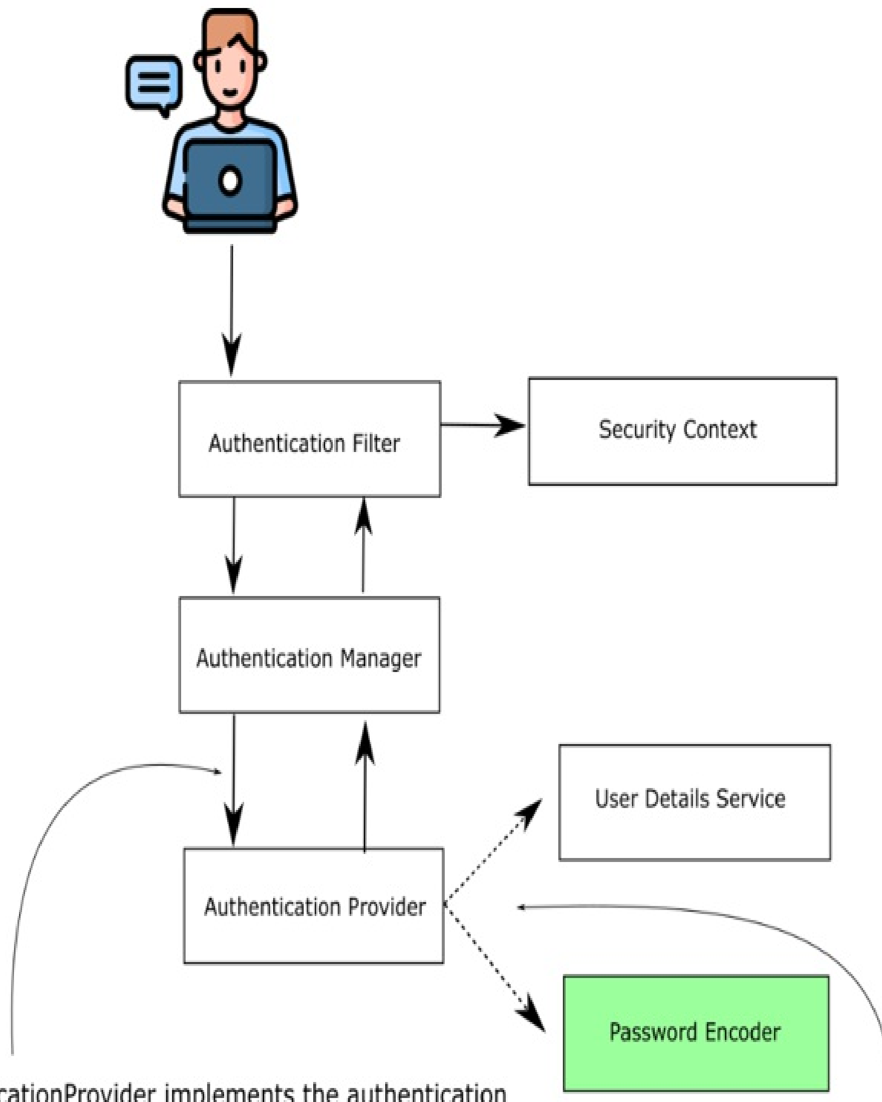
- Implementing and working with the `PasswordEncoder`
- Using the tools offered by the Spring Security Crypto module

In chapter 3, we discussed managing users in an application implemented with Spring Security. But what about passwords? They're certainly an essential piece in the authentication flow. In this chapter, you'll learn how to manage passwords and secrets in an application implemented with Spring Security. We'll discuss the `PasswordEncoder` contract and the tools offered by the Spring Security Crypto module (SSCM) for the management of passwords.

4.1 Using password encoders

From chapter 3, you should now have a clear image of what the `UserDetails` interface is as well as multiple ways to use its implementation. But as you learned in chapter 2, different actors manage user representation during the authentication and authorization processes. You also learned that some of these have defaults, like `UserDetailsService` and `PasswordEncoder`. You now know that you can override the defaults. We continue with a deep understanding of these beans and ways to implement them, so in this section, we analyze the `PasswordEncoder`. Figure 4.1 reminds you of where the `PasswordEncoder` fits into the authentication process.

Figure 4.1 The Spring Security authentication process. The `AuthenticationProvider` uses the `PasswordEncoder` to validate the user's password in the authentication process.



Because, in general, a system doesn't manage passwords in plain text, these usually undergo a sort of transformation that makes it more challenging to read and steal them. For this responsibility, Spring Security defines a separate contract. To explain it easily in this section, I provide plenty of code examples related to the `PasswordEncoder` implementation. We'll start with understanding the contract, and then we'll write our implementation within a project. Then in section 4.1.3, I'll provide you with a list of the most well-known and widely used implementations of `PasswordEncoder` provided by

Spring Security.

4.1.1 The PasswordEncoder contract

In this section, we discuss the definition of the `PasswordEncoder` contract. You implement this contract to tell Spring Security how to validate a user's password. In the authentication process, the `PasswordEncoder` decides if a password is valid or not. Every system stores passwords encoded in some way. You preferably store them hashed so that there's no chance someone can read the passwords. The `PasswordEncoder` can also encode passwords. The methods `encode()` and `matches()`, which the contract declares, are actually the definition of its responsibility. Both of these are parts of the same contract because these are strongly linked, one to the other. The way the application encodes a password is related to the way the password is validated. Let's first review the content of the `PasswordEncoder` interface:

```
public interface PasswordEncoder {

  String encode(CharSequence rawPassword);
  boolean matches(CharSequence rawPassword, String encodedPasswor

  default boolean upgradeEncoding(String encodedPassword) {
    return false;
  }
}
The interface defines two abstract methods and one with a default
```

The purpose of the `encode(CharSequence rawPassword)` method is to return a transformation of a provided string. In terms of Spring Security functionality, it's used to provide encryption or a hash for a given password. You can use the `matches(CharSequence rawPassword, String encodedPassword)` method afterward to check if an encoded string matches a raw password. You use the `matches()` method in the authentication process to test a provided password against a set of known credentials. The third method, called `upgradeEncoding(CharSequence encodedPassword)`, defaults to false in the contract. If you override it to return true, then the encoded password is encoded again for better security.

In some cases, encoding the encoded password can make it more challenging to obtain the cleartext password from the result. In general, this is some kind

of obscurity that I, personally, don't like. But the framework offers you this possibility if you think it applies to your case.

4.1.2 Implementing your PasswordEncoder

As you observed, the two methods `matches()` and `encode()` have a strong relationship. If you override them, they should always correspond in terms of functionality: a string returned by the `encode()` method should always be verifiable with the `matches()` method of the same `PasswordEncoder`. In this section, you'll implement the `PasswordEncoder` contract and define the two abstract methods declared by the interface. Knowing how to implement the `PasswordEncoder`, you can choose how the application manages passwords for the authentication process. The most straightforward implementation is a password encoder that considers passwords in plain text: that is, it doesn't do any encoding on the password.

Managing passwords in cleartext is what the instance of `NoOpPasswordEncoder` does precisely. We used this class in our first example in chapter 2. If you were to write your own, it would look something like the following listing.

Listing 4.1 The simplest implementation of a `PasswordEncoder`

```
public class PlainTextPasswordEncoder
  implements PasswordEncoder {

  @Override
  public String encode(CharSequence rawPassword) {
    return rawPassword.toString();      #A
  }

  @Override
  public boolean matches(
    CharSequence rawPassword, String encodedPassword) {
      return rawPassword.equals(encodedPassword);    #B
  }
}
```

The result of the encoding is always the same as the password. So to check if these match, you only need to compare the strings with `equals()`. A simple

implementation of `PasswordEncoder` that uses the hashing algorithm SHA-512 looks like the next listing.

Listing 4.2 Implementing a `PasswordEncoder` that uses SHA-512

```
public class Sha512PasswordEncoder
  implements PasswordEncoder {

  @Override
  public String encode(CharSequence rawPassword) {
    return hashWithSHA512(rawPassword.toString());
  }

  @Override
  public boolean matches(
    CharSequence rawPassword, String encodedPassword) {
    String hashedPassword = encode(rawPassword);
    return encodedPassword.equals(hashedPassword);
  }

  // Omitted code

}
```

In listing 4.2, we use a method to hash the string value provided with SHA-512. I omit the implementation of this method in listing 4.2, but you can find it in listing 4.3. We call this method from the `encode()` method, which now returns the hash value for its input. To validate a hash against an input, the `matches()` method hashes the raw password in its input and compares it for equality with the hash against which it does the validation.

Listing 4.3 The implementation of the method to hash the input with SHA-512

```
private String hashWithSHA512(String input) {
  StringBuilder result = new StringBuilder();
  try {
    MessageDigest md = MessageDigest.getInstance("SHA-512");
    byte [] digested = md.digest(input.getBytes());
    for (int i = 0; i < digested.length; i++) {
       result.append(Integer.toHexString(0xFF & digested[i]));
    }
  } catch (NoSuchAlgorithmException e) {
    throw new RuntimeException("Bad algorithm");
  }
```

```
  return result.toString();
}
```

You’ll learn better options to do this in the next section, so don’t bother too much with this code for now.

4.1.3 Choosing from the provided PasswordEncoder implementations

While knowing how to implement your `PasswordEncoder` is powerful, you must also be aware that Spring Security already provides you with some advantageous implementations. If one of these matches your application, you don’t need to rewrite it. In this section, we discuss the `PasswordEncoder` implementation options that Spring Security provides. These are

- `NoOpPasswordEncoder`—Doesn’t encode the password but keeps it in clear-text. We use this implementation only for examples. Because it doesn’t hash the password, "charitalics"you should never use it in a real-world scenario.
- `StandardPasswordEncoder`—Uses SHA-256 to hash the password. This implementation is now deprecated, and "charitalics"you shouldn’t use it for your new implementations. The reason why it’s deprecated is that it uses a hashing algorithm that we don’t consider strong enough anymore, but you might still find this implementation used in existing applications. Preferably, if you find it in existing apps, you should change it with some other, more powerful password encoder.
- `Pbkdf2PasswordEncoder`—Uses the password-based key derivation function 2 (PBKDF2).
- `BCryptPasswordEncoder`—Uses a bcrypt strong hashing function to encode the password.
- `SCryptPasswordEncoder`—Uses an scrypt hashing function to encode the password.

For more about hashing and these algorithms, you can find a good discussion in chapter 2 of "charitalics"Real-World Cryptography by David Wong (Manning, 2021). Here’s the link: https://livebook.manning.com/book/real-world-cryptography/chapter-2/.

Let’s take a look at some examples of how to create instances of these types of `PasswordEncoder` implementations. The `NoOpPasswordEncoder` doesn’t encode the password. It has an implementation similar to the `PlainTextPasswordEncoder` from our example in listing 4.1. For this reason, we only use this password encoder with theoretical examples. Also, the `NoOpPasswordEncoder` class is designed as a singleton. You can’t call its constructor directly from outside the class, but you can use the `NoOpPasswordEncoder.getInstance()` method to obtain the instance of the class like this:

```
PasswordEncoder p = NoOpPasswordEncoder.getInstance();
```

The `StandardPasswordEncoder` implementation provided by Spring Security uses SHA-256 to hash the password. For the `StandardPasswordEncoder`, you can provide a secret used in the hashing process. You set the value of this secret by the constructor’s parameter. If you choose to call the no-arguments constructor, the implementation uses the empty string as a value for the key. However, the `StandardPasswordEncoder` is deprecated now, and I don’t recommend that you use it with your new implementations. You could find older applications or legacy code that still uses it, so this is why you should be aware of it. The next code snippet shows you how to create instances of this password encoder:

```
PasswordEncoder p = new StandardPasswordEncoder();
PasswordEncoder p = new StandardPasswordEncoder("secret");
```

Another option offered by Spring Security is the `Pbkdf2PasswordEncoder` implementation that uses the PBKDF2 for password encoding. To create instances of the `Pbkdf2PasswordEncoder`, you have the following option:

```
PasswordEncoder p =
   New Pbkdf2PasswordEncoder(“secret”, 16, 310000,
    [CA]Pbkdf2PasswordEncoder.
    [CA]SecretKeyFactoryAlgorithm.PBKDF2WithHmacSHA256);
```

The PBKDF2 is a pretty easy, slow-hashing function that performs an HMAC as many times as specified by an iterations argument. The first three parameters received by the last call are the value of a key used for the encoding process, the number of iterations used to encode the password, and

the size of the hash. The second and third parameters can influence the strength of the result. The forth parameter gives the hash width. You can choose out of the following options:

- PBKDF2WithHmacSHA1
- PBKDF2WithHmacSHA256
- PBKDF2WithHmacSHA512

You can choose more or fewer iterations, as well as the length of the result. The longer the hash, the more powerful the password (same for the hash width). However, be aware that performance is affected by these values: the more iterations, the more resources your application consumes. You should make a wise compromise between the resources consumed for generating the hash and the needed strength of the encoding.

Note

In this book, I refer to several cryptography concepts that you might like to know more about. For relevant information on HMACs and other cryptography details, I recommend *Real-World Cryptography* by David Wong (Manning, 2020). Chapter 3 of that book provides detailed information about HMAC. You can find the book at https://livebook.manning.com/book/real-world-cryptography/chapter-3/ .

Another excellent option offered by Spring Security is the `BCryptPasswordEncoder`, which uses a bcrypt strong hashing function to encode the password. You can instantiate the `BCryptPasswordEncoder` by calling the no-arguments constructor. But you also have the option to specify a strength coefficient representing the log rounds (logarithmic rounds) used in the encoding process. Moreover, you can also alter the `SecureRandom` instance used for encoding:

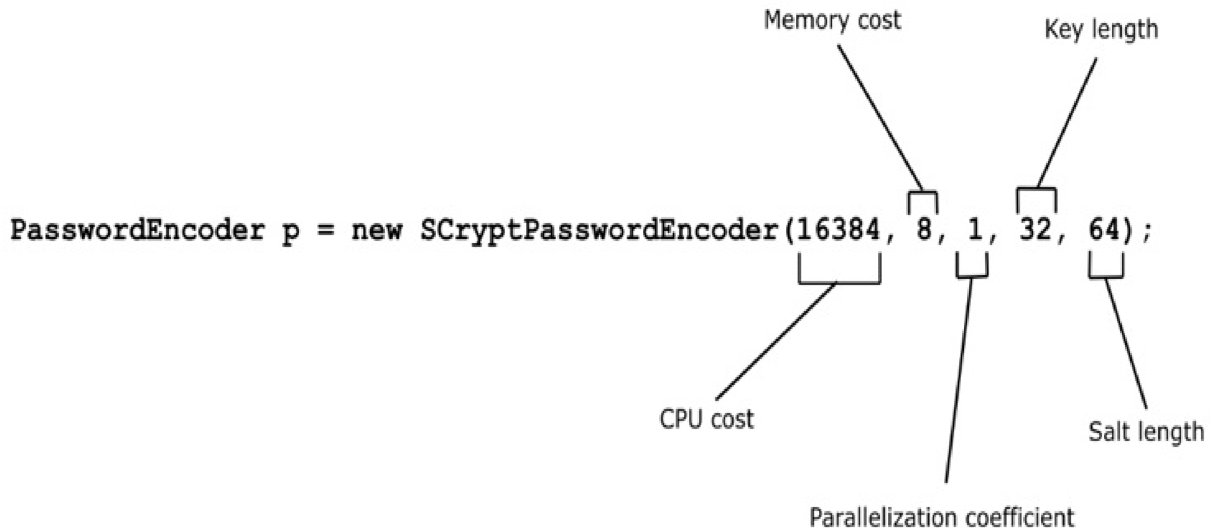
```
PasswordEncoder p = new BCryptPasswordEncoder();
PasswordEncoder p = new BCryptPasswordEncoder(4);

SecureRandom s = SecureRandom.getInstanceStrong();
PasswordEncoder p = new BCryptPasswordEncoder(4, s);
```

The log rounds value that you provide affects the number of iterations the

hashing operation uses. The number of iterations used is $2^{\text{log rounds}}$. For the iteration number computation, the value for the log rounds can only be between 4 and 31. You can specify this by calling one of the second or third overloaded constructors, as shown in the previous code snippet.

The last option I present to you is `SCryptPasswordEncoder` (figure 4.2). This password encoder uses an scrypt hashing function. For the `ScryptPasswordEncoder`, you have the following option to create its instances as presented in figure 4.2.

Figure 4.2 The `SCryptPasswordEncoder` constructor takes five parameters and allows you to configure CPU cost, memory cost, key length, and salt length.



4.1.4 Multiple encoding strategies with DelegatingPasswordEncoder

In this section, we discuss the cases in which an authentication flow must apply various implementations for matching the passwords. You'll also learn how to apply a useful tool that acts as a `PasswordEncoder` in your application. Instead of having its own implementation, this tool delegates to other objects that implement the `PasswordEncoder` interface.

In some applications, you might find it useful to have various password encoders and choose from these depending on some specific configuration. A

common scenario in which I find the `DelegatingPasswordEncoder` in production applications is when the encoding algorithm is changed, starting with a particular version of the application. Imagine somebody finds a vulnerability in the currently used algorithm, and you want to change it for newly registered users, but you do not want to change it for existing credentials. So you end up having multiple kinds of hashes. How do you manage this case? While it isn't the only approach for this scenario, a good choice is to use a `DelegatingPasswordEncoder` object.

The `DelegatingPasswordEncoder` is an implementation of the `PasswordEncoder` interface that, instead of implementing its encoding algorithm, delegates to another instance of an implementation of the same contract. The hash starts with a prefix naming the algorithm used to define that hash. The `DelegatingPasswordEncoder` delegates to the correct implementation of the `PasswordEncoder` based on the prefix of the password.

It sounds complicated, but with an example, you can observe that it is pretty easy. Figure 4.3 presents the relationship among the `PasswordEncoder` instances. The `DelegatingPasswordEncoder` has a list of `PasswordEncoder` implementations to which it delegates. The `DelegatingPasswordEncoder` stores each of the instances in a map. The `NoOpPasswordEncoder` is assigned to the key `noop`, while the `BCryptPasswordEncoder` implementation is assigned the key `bcrypt`. When the password has the prefix `{noop}`, the `DelegatingPasswordEncoder` delegates the operation to the `NoOpPasswordEncoder` implementation. If the prefix is `{bcrypt}`, then the action is delegated to the `BCryptPasswordEncoder` implementation as presented in figure 4.4.

Figure 4.3 In this case, the `DelegatingPasswordEncoder` registers a `NoOpPasswordEncoder` for the prefix `{noop}`, a `BCryptPasswordEncoder` for the prefix `{bcrypt}`, and an `SCryptPasswordEncoder` for the prefix `{scrypt}`. If the password has the prefix `{noop}`, the `DelegatingPasswordEncoder` forwards the operation to the `NoOpPasswordEncoder` implementation.

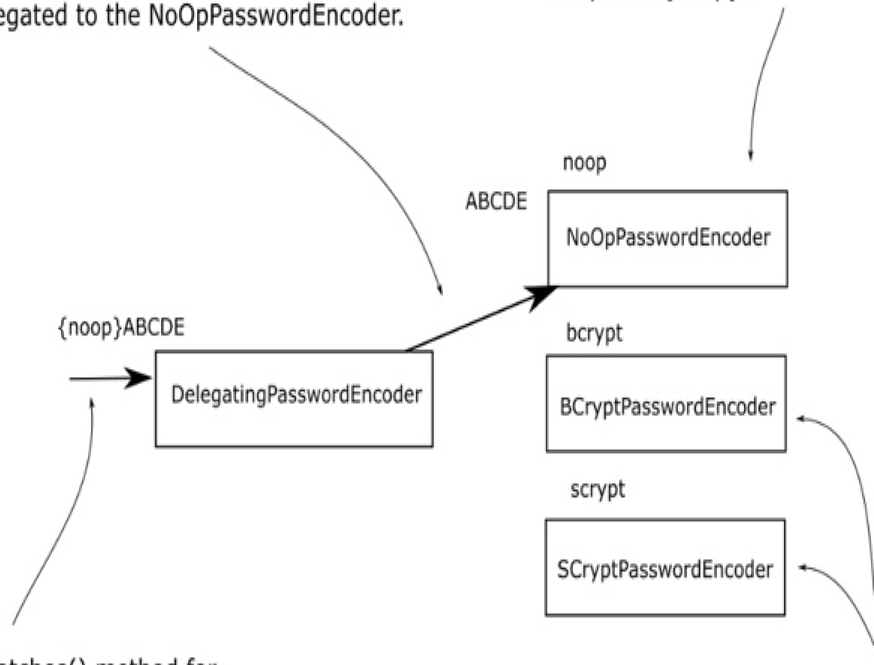
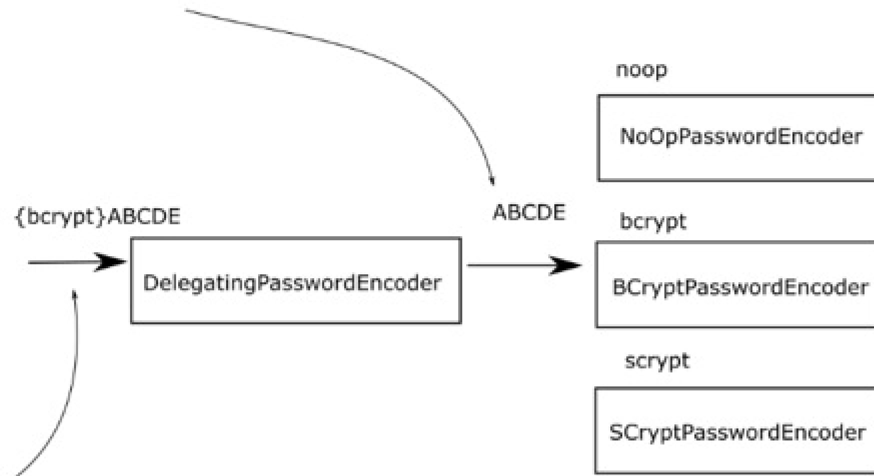


Figure 4.4 In this case, the `DelegatingPasswordEncoder` registers a `NoOpPasswordEncoder` for the prefix `{noop}`, a `BCryptPasswordEncoder` for the prefix `{bcrypt}`, and an `SCryptPasswordEncoder` for the prefix `{scrypt}`. When the password has the prefix `{bcrypt}`, the `DelegatingPasswordEncoder` forwards the operation to the `BCryptPasswordEncoder` implementation.



Next, let’s find out how to define a `DelegatingPasswordEncoder`. You start by creating a collection of instances of your desired `PasswordEncoder` implementations, and you put these together in a `DelegatingPasswordEncoder` as in the following listing.

Listing 4.4 Creating an instance of `DelegatingPasswordEncoder`

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public PasswordEncoder passwordEncoder() {
    Map<String, PasswordEncoder> encoders = new HashMap<>();

    encoders.put("noop", NoOpPasswordEncoder.getInstance());
    encoders.put("bcrypt", new BCryptPasswordEncoder());
    encoders.put("scrypt", new SCryptPasswordEncoder());

    return new DelegatingPasswordEncoder("bcrypt", encoders);
  }
}
```

The `DelegatingPasswordEncoder` is just a tool that acts as a `PasswordEncoder` so you can use it when you have to choose from a collection of implementations. In listing 4.4, the declared instance of `DelegatingPasswordEncoder` contains references to a `NoOpPasswordEncoder`, a `BCryptPasswordEncoder`, and an `SCryptPasswordEncoder`, and delegates the default to the `BCryptPasswordEncoder` implementation. Based on the prefix of the hash, the `DelegatingPasswordEncoder` uses the right `PasswordEncoder` implementation for matching the password. This prefix has the key that identifies the password encoder to be used from the map of encoders. If there is no prefix, the `DelegatingPasswordEncoder` uses the default encoder. The default `PasswordEncoder` is the one given as the first parameter when constructing the `DelegatingPasswordEncoder` instance. For the code in listing 4.4, the default `PasswordEncoder` is bcrypt.

NOTE

The curly braces are part of the hash prefix, and those should surround the name of the key. For example, if the provided hash is `{noop}12345`, the `DelegatingPasswordEncoder` delegates to the `NoOpPasswordEncoder` that we registered for the prefix `noop`. Again, remember that the curly braces are mandatory in the prefix.

If the hash looks like the next code snippet, the password encoder is the one we assign to the prefix `{bcrypt}`, which is the `BCryptPasswordEncoder`. This is also the one to which the application will delegate if there is no prefix at all because we defined it as the default implementation:

```
{bcrypt}$2a$10$xn3LI/AjqicFYZFruSwve.681477XaVNaUQbr1gioaWPn4t1Ks
```

For convenience, Spring Security offers a way to create a `DelegatingPasswordEncoder` that has a map to all the standard provided implementations of `PasswordEncoder`. The `PasswordEncoderFactories` class provides a `createDelegatingPasswordEncoder()` static method that returns an implementation of `DelegatingPasswordEncoder` with a full set of `PasswordEncoder` mappings and bcrypt as a default encoder:

```
PasswordEncoder passwordEncoder = PasswordEncoderFactories.create
```

Encoding vs. encrypting vs. hashing

In the previous sections, I often used the terms encoding, encrypting, and hashing. I want to briefly clarify these terms and the way we use them throughout the book.

Encoding refers to any transformation of a given input. For example, if we have a function `x` that reverses a string, function `x -> y` applied to `ABCD` produces `DCBA`.

Encryption is a particular type of encoding where, to obtain the output, you provide both the input value and a key. The key makes it possible for choosing afterward who should be able to reverse the function (obtain the input from the output). The simplest form of representing encryption as a function looks like this:

```
(x, k) -> y
```

where `x` is the input, `k` is the key, and `y` is the result of the encryption. This way, an individual who knows the key can use a known function to obtain the input from the output `(y, k) -> x`. We call this *reverse function decryption*. If the key used for encryption is the same as the one used for decryption, we usually call it a *symmetric* key.

If we have two different keys for encryption ($(x, k_1) \rightarrow y$) and decryption ($(y, k_2) \rightarrow x$), then we say that the encryption is done with *asymmetric* keys. Then (k_1, k_2) is called a *key pair*. The key used for encryption, k_1, is also referred to as the *public* key, while k_2 is known as the *private* one. This way, only the owner of the private key can decrypt the data.

Hashing is a particular type of encoding, except the function is only one way. That is, from an output `y` of the hashing function, you cannot get back the input `x`. However, there should always be a way to check if an output `y` corresponds to an input `x`, so we can understand the hashing as a pair of functions for encoding and matching. If hashing is `x -> y`, then we should also have a matching function `(x,y) -> boolean`.

Sometimes the hashing function could also use a random value added to the

input: `(x, k) -> y`. We refer to this value as the *salt*. The salt makes the function stronger, enforcing the difficulty of applying a reverse function to obtain the input from the result.

To summarize the contracts we have discussed and applied up to now in this book, table 4.1 briefly describes each of the components.

Table 4.1 The interfaces that represent the main contracts for authentication flow in Spring Security

Contract	Description
`UserDetails`	Represents the user as seen by Spring Security.
`GrantedAuthority`	Defines an action within the purpose of the application that is allowable to the user (for example, read, write, delete, etc.).
`UserDetailsService`	Represents the object used to retrieve user details by username.
`UserDetailsManager`	A more particular contract for `UserDetailsService`. Besides retrieving the user by username, it can also be used to mutate a collection of users or a specific user.
`PasswordEncoder`	Specifies how the password is encrypted or hashed and how to check if a given encoded string matches a plaintext password.

4.2 Take advantage of the Spring Security Crypto

module

In this section, we discuss the Spring Security Crypto module (SSCM), which is the part of Spring Security that deals with cryptography. Using encryption and decryption functions and generating keys isn't offered out of the box with the Java language. And this constrains developers when adding dependencies that provide a more accessible approach to these features.

To make our lives easier, Spring Security also provides its own solution, which enables you to reduce the dependencies of your projects by eliminating the need to use a separate library. The password encoders are also part of the SSCM, even if we have treated them separately in previous sections. In this section, we discuss what other options the SSCM offers that are related to cryptography. You'll see examples of how to use two essential features from the SSCM:

- "charitalics"Key generators—Objects used to generate keys for hashing and encryption algorithms
- "charitalics"Encryptors—Objects used to encrypt and decrypt data

4.2.1 Using key generators

In this section, we discuss key generators. A "charitalics"key generator is an object used to generate a specific kind of key, generally needed for an encryption or hashing algorithm. The implementations of key generators that Spring Security offers are great utility tools. You'll prefer to use these implementations rather than adding another dependency for your application, and this is why I recommend that you become aware of them. Let's see some code examples of how to create and apply the key generators.

Two interfaces represent the two main types of key generators: `BytesKeyGenerator` and `StringKeyGenerator`. We can build them directly by making use of the factory class `KeyGenerators`. You can use a string key generator, represented by the `StringKeyGenerator` contract, to obtain a key as a string. Usually, we use this key as a salt value for a hashing or encryption algorithm. You can find the definition of the `StringKeyGenerator` contract in this code snippet:

```
public interface StringKeyGenerator {

    String generateKey();

}
```

The generator has only a `generateKey()` method that returns a string representing the key value. The next code snippet presents an example of how to obtain a `StringKeyGenerator` instance and how to use it to get a salt value:

```
StringKeyGenerator keyGenerator = KeyGenerators.string();
String salt = keyGenerator.generateKey();
```

The generator creates an 8-byte key, and it encodes that as a hexadecimal string. The method returns the result of these operations as a string. The second interface describing a key generator is the `BytesKeyGenerator`, which is defined as follows:

```
public interface BytesKeyGenerator {

  int getKeyLength();
  byte[] generateKey();

}
```

In addition to the `generateKey()` method that returns the key as a `byte[]`, the interface defines another method that returns the key length in number of bytes. A default `ByteKeyGenerator` generates keys of 8-byte length:

```
BytesKeyGenerator keyGenerator = KeyGenerators.secureRandom();
byte [] key = keyGenerator.generateKey();
int keyLength = keyGenerator.getKeyLength();
```

In the previous code snippet, the key generator generates keys of 8-byte length. If you want to specify a different key length, you can do this when obtaining the key generator instance by providing the desired value to the `KeyGenerators.secureRandom()` method:

```
BytesKeyGenerator keyGenerator = KeyGenerators.secureRandom(16);
```

The keys generated by the `BytesKeyGenerator` created with the

`KeyGenerators.secureRandom()` method are unique for each call of the `generateKey()` method. In some cases, we prefer an implementation that returns the same key value for each call of the same key generator. In this case, we can create a `BytesKeyGenerator` with the `KeyGenerators.shared(int length)` method. In this code snippet, `key1` and `key2` have the same value:

```
BytesKeyGenerator keyGenerator = KeyGenerators.shared(16);
byte [] key1 = keyGenerator.generateKey();
byte [] key2 = keyGenerator.generateKey();
```

4.2.2 Encrypt and decrypt secrets using encryptors

In this section, we apply the implementations of encryptors that Spring Security offers with code examples. An "charitalics"encryptor is an object that implements an encryption algorithm. When talking about security, encryption and decryption are common operations, so expect to need these within your application.

We often need to encrypt data either when sending it between components of the system or when persisting it. The operations provided by an encryptor are encryption and decryption. There are two types of encryptors defined by the SSCM: `BytesEncryptor` and `TextEncryptor`. While they have similar responsibilities, they treat different data types. `TextEncryptor` manages data as a string. Its methods receive strings as inputs and return strings as outputs, as you can see from the definition of its interface:

```
public interface TextEncryptor {

  String encrypt(String text);
  String decrypt(String encryptedText);

}
```

The `BytesEncryptor` is more generic. You provide its input data as a byte array:

```
public interface BytesEncryptor {

  byte[] encrypt(byte[] byteArray);
```

```
  byte[] decrypt(byte[] encryptedByteArray);

}
```

Let’s find out what options we have to build and use an encryptor. The factory class `Encryptors` offers us multiple possibilities. For `BytesEncryptor`, we could use the `Encryptors.standard()` or the `Encryptors.stronger()` methods like this:

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";

BytesEncryptor e = Encryptors.standard(password, salt);
byte [] encrypted = e.encrypt(valueToEncrypt.getBytes());
byte [] decrypted = e.decrypt(encrypted);
```

Behind the scenes, the standard byte encryptor uses 256-byte AES encryption to encrypt input. To build a stronger instance of the byte encryptor, you can call the `Encryptors.stronger()` method:

```
BytesEncryptor e = Encryptors.stronger(password, salt);
```

The difference is small and happens behind the scenes, where the AES encryption on 256-bit uses Galois/Counter Mode (GCM) as the mode of operation. The standard mode uses cipher block chaining (CBC), which is considered a weaker method.

`TextEncryptors` come in three main types. You create these three types by calling the `Encryptors.text()`, `Encryptors.delux()`, or `Encryptors.queryableText()` methods. Besides these methods to create encryptors, there is also a method that returns a dummy `TextEncryptor`, which doesn’t encrypt the value. You can use the dummy `TextEncryptor` for demo examples or cases in which you want to test the performance of your application without spending time spent on encryption. The method that returns this no-op encryptor is `Encryptors.noOpText()`. In the following code snippet, you’ll find an example of using a `TextEncryptor`. Even if it is a call to an encryptor, in the example, `encrypted` and `valueToEncrypt` are the same:

```
String valueToEncrypt = "HELLO";
```

```
TextEncryptor e = Encryptors.noOpText();
String encrypted = e.encrypt(valueToEncrypt);
```

The `Encryptors.text()` encryptor uses the `Encryptors.standard()` method to manage the encryption operation, while the `Encryptors.delux()` method uses an `Encryptors.stronger()` instance like this:

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";

TextEncryptor e = Encryptors.text(password, salt);     #A
String encrypted = e.encrypt(valueToEncrypt);
String decrypted = e.decrypt(encrypted);
```

4.3 Summary

- The `PasswordEncoder` has one of the most critical responsibilities in authentication logic—dealing with passwords.
- Spring Security offers several alternatives in terms of hashing algorithms, which makes the implementation only a matter of choice.
- Spring Security Crypto module (SSCM) offers various alternatives for implementations of key generators and encryptors.
- Key generators are utility objects that help you generate keys used with cryptographic algorithms.
- Encryptors are utility objects that help you apply encryption and decryption of data.

5 A web app's security begins with filters

This chapter covers

- Working with the filter chain
- Defining custom filters
- Using Spring Security classes that implement the `Filter` interface

In Spring Security, HTTP filters delegate the different responsibilities that apply to an HTTP request. In Spring Security, in general, HTTP filters manage each responsibility that must be applied to the request. The filters form a chain of responsibilities. A filter receives a request, executes its logic, and eventually delegates the request to the next filter in the chain (figure 5.1).

Figure 5.1 The filter chain receives the request. Each filter uses a manager to apply specific logic to the request and, eventually, delegates the request further along the chain to the next filter.

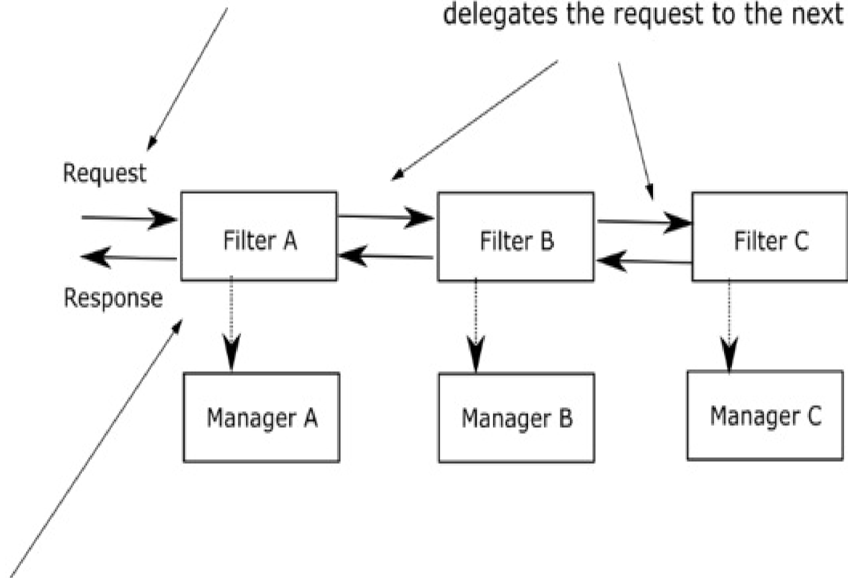
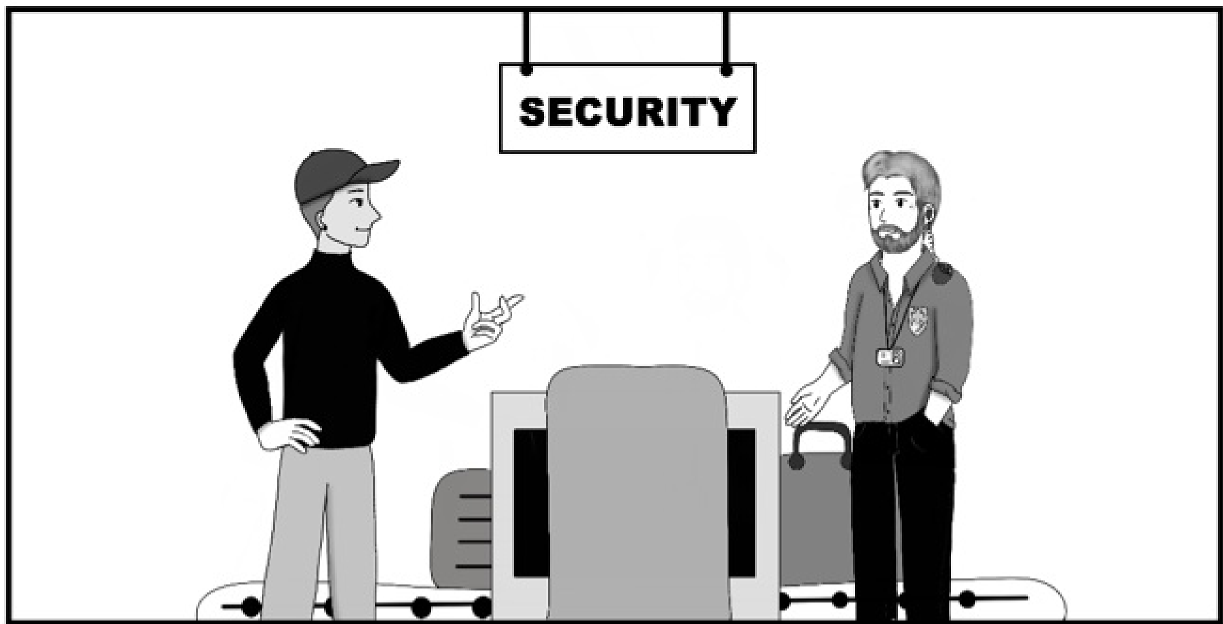


Figure 5.2 At the airport, you go through a filter chain to eventually board the aircraft. In the same way, Spring Security has a filter chain that acts on the HTTP requests received by the application.

FILTER 1

FILTER 2



FILTER 3

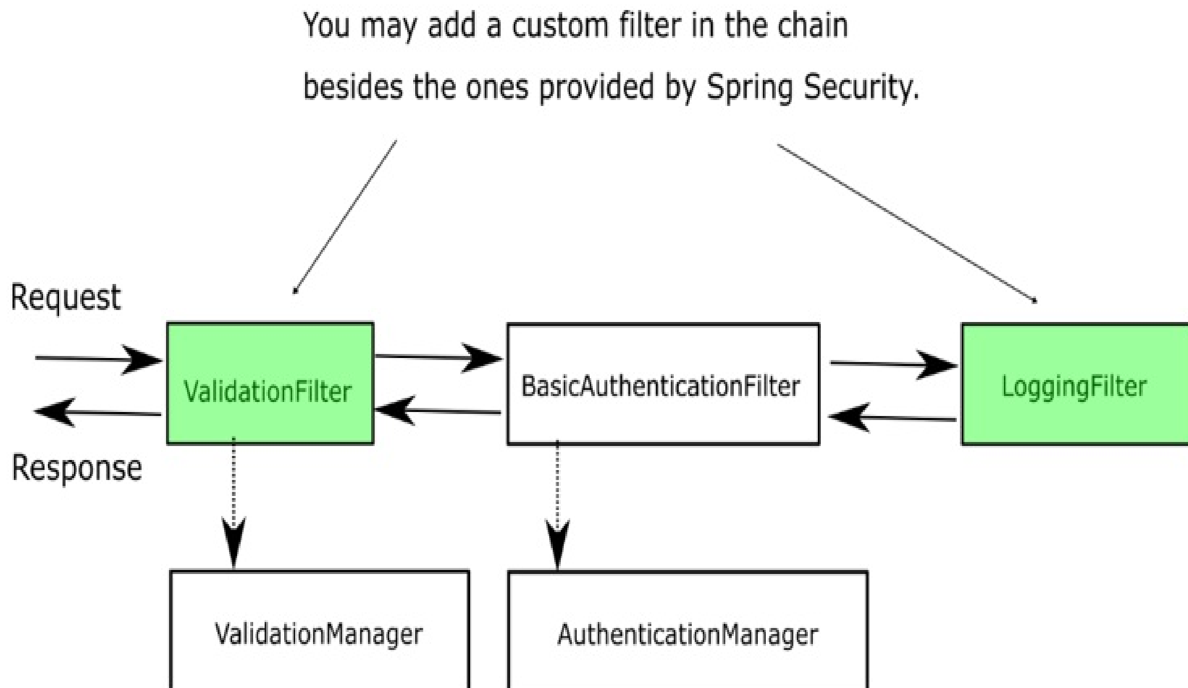
Let's take an analogy as an example. When you go to the airport, from entering the terminal to boarding the aircraft, you go through multiple filters (figure 5.2). You first present your ticket, then your passport is verified, and afterward, you go through security. At the airport gate, more "filters" might be applied. For example, in some cases, right before boarding, your passport and visa are validated once more. This is an excellent analogy to the filter

chain in Spring Security. In the same way, you customize filters in a filter chain with Spring Security that act on HTTP requests. Spring Security provides filter implementations that you add to the filter chain through customization, but you can also define custom filters.

In this chapter, we’ll discuss how you can customize filters that are part of the authentication and authorization architecture in Spring Security of a web app. For example, you might want to augment authentication by adding one more step for the user, like checking their email address or using a one-time password. You can, as well, add functionality referring to auditing authentication events. You’ll find various scenarios where applications use auditing authentication: from debugging purposes to identifying a user’s behavior. Using today’s technology and machine learning algorithms can improve applications, for example, by learning the user’s behavior and knowing if somebody hacked their account or impersonated the user.

Knowing how to customize the HTTP filter chain of responsibilities is a valuable skill. In practice, applications come with various requirements, where using default configurations doesn’t work anymore. You’ll need to add or replace existing components of the chain. With the default implementation, you use the HTTP Basic authentication method, which allows you to rely on a username and password. But in practical scenarios, there are plenty of situations in which you’ll need more than this. Maybe you need to implement a different strategy for authentication, notify an external system about an authorization event, or simply log a successful or failed authentication that’s later used in tracing and auditing (figure 5.3). Whatever your scenario is, Spring Security offers you the flexibility of modeling the filter chain precisely as you need it.

Figure 5.3 You can customize the filter chain by adding new filters before, after, or at the position of existing ones. This way, you can customize authentication as well as the entire process applied to request and response.



5.1 Implementing filters in the Spring Security architecture

In this section, we discuss the way filters and the filter chain work in Spring Security architecture. You need this general overview first to understand the implementation examples we work on in the next sections of this chapter. You learned in chapters 2 and 3 that the authentication filter intercepts the request and delegates authentication responsibility further to the authorization manager. If we want to execute certain logic before authentication, we do this by inserting a filter before the authentication filter.

The filters in Spring Security architecture are typical HTTP filters. We can create filters by implementing the `Filter` interface from the jakarta.servlet package. As for any other HTTP filter, you need to override the `doFilter()` method to implement its logic. This method receives as parameters the `ServletRequest`, `ServletResponse`, and `FilterChain`:

- `ServletRequest`—Represents the HTTP request. We use the `ServletRequest` object to retrieve details about the request.

- `ServletResponse`—Represents the HTTP response. We use the `ServletResponse` object to alter the response before sending it back to the client or further along the filter chain.
- `FilterChain`—Represents the chain of filters. We use the `FilterChain` object to forward the request to the next filter in the chain.

Note

Starting with Spring Boot 3, Jakarta EE replaces the old Java EE specification. Due to this change, you'll observe that some packages changed their prefix from "javax" to "jakarta". For example, types such as `Filter`, `ServletRequest`, and `ServletResponse` were previously in the javax.servlet package, but you now find them in the jakarta.servlet package.

The "charitalics"filter chain represents a collection of filters with a defined order in which they act. Spring Security provides some filter implementations and their order for us. Among the provided filters

- `BasicAuthenticationFilter` takes care of HTTP Basic authentication, if present.
- `CsrfFilter` takes care of cross-site request forgery (CSRF) protection, which we'll discuss in chapter 9.
- `CorsFilter` takes care of cross-origin resource sharing (CORS) authorization rules, which we'll also discuss in chapter 10.

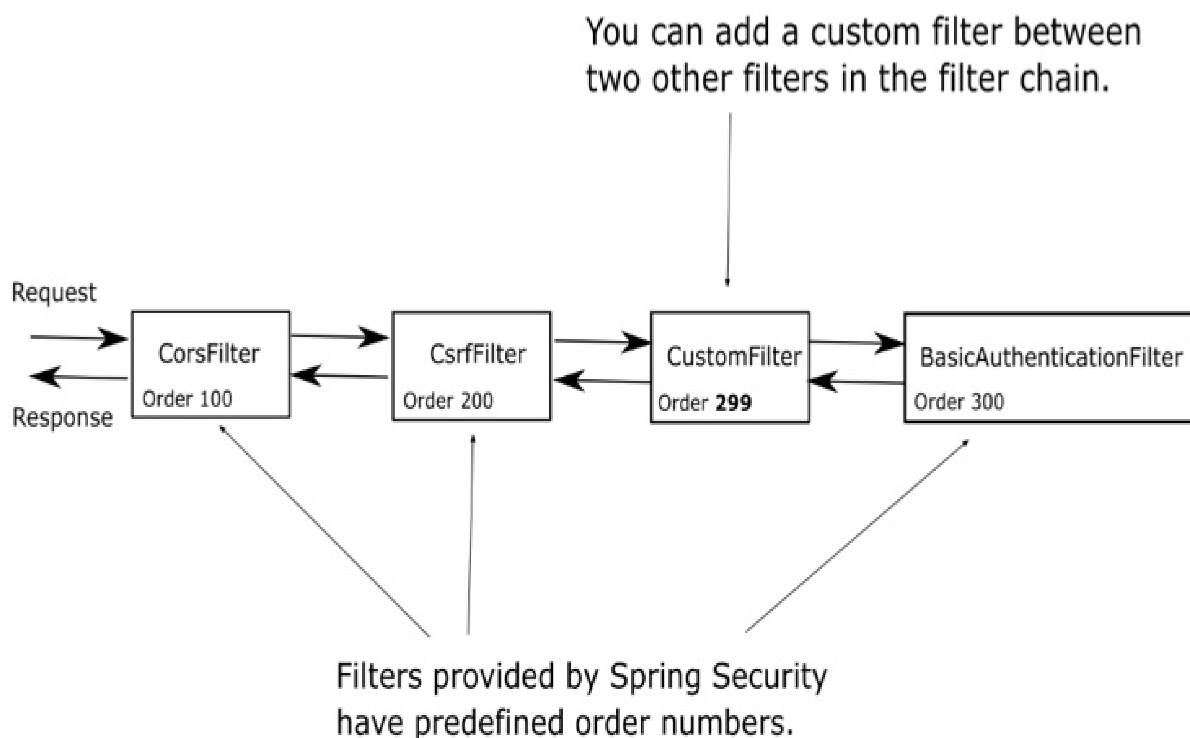
You don't need to know all of the filters as you probably won't touch these directly from your code, but you do need to understand how the filter chain works and to be aware of a few implementations. In this book, I only explain those filters that are essential to the various topics we discuss.

It is important to understand that an application doesn't necessarily have instances of all these filters in the chain. The chain is longer or shorter depending on how you configure the application. For example, in chapters 2 and 3, you learned that you need to call the `httpBasic()` method of the `HttpSecurity` class if you want to use the HTTP Basic authentication method. What happens is that if you call the `httpBasic()` method, an instance of `BasicAuthenticationFilter` is added to the chain. Similarly, depending on the configurations you write, the definition of the filter chain is

affected.

You add a new filter to the chain relative to another one (figure 5.4). Or, you can add a filter either before, after, or at the position of a known one. Each position is, in fact, an index (a number), and you might find it also referred to as “the order.”

Figure 5.4 Each filter has an order number. This determines the order in which filters are applied to a request. You can add custom filters along with the filters provided by Spring Security.



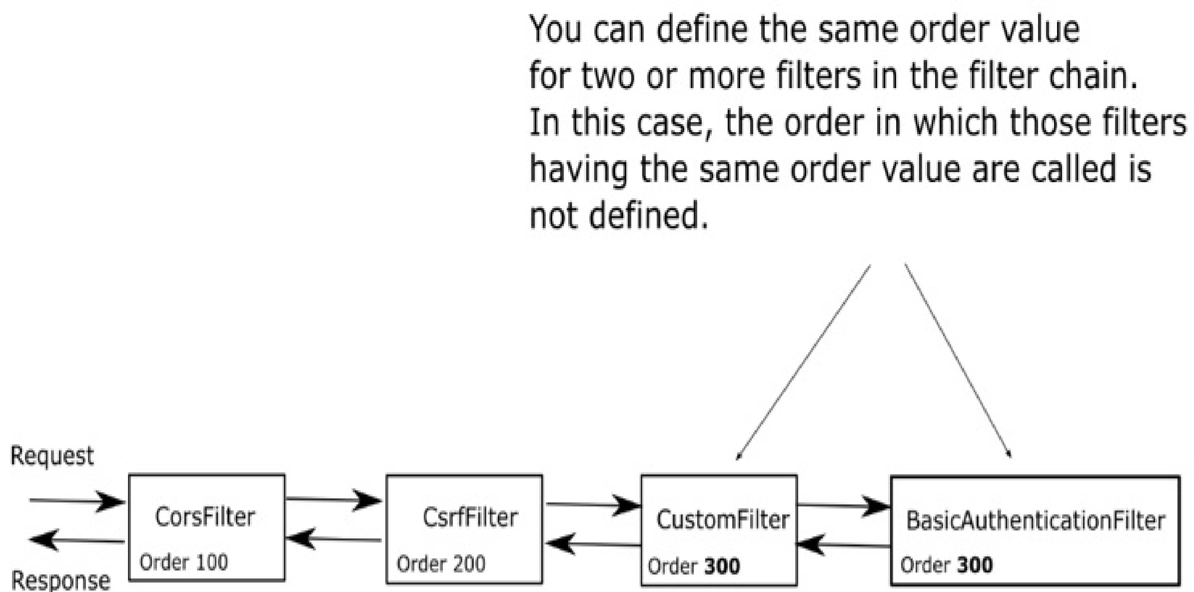
If you want to learn more details about which filters Spring Security provides and what is the order in which these are configured, you can take a look in the `FilterOrderRegistration` from org.springframework.security.config.annotation.web.builders package.

You can add two or more filters in the same position (figure 5.5). In section 5.4, we’ll encounter a common case in which this might occur, one which usually creates confusion among developers.

NOTE

If multiple filters have the same position, the order in which they are called is not defined.

Figure 5.5 You might have multiple filters with the same order value in the chain. In this case, Spring Security doesn't guarantee the order in which they are called.



5.2 Adding a filter before an existing one in the chain

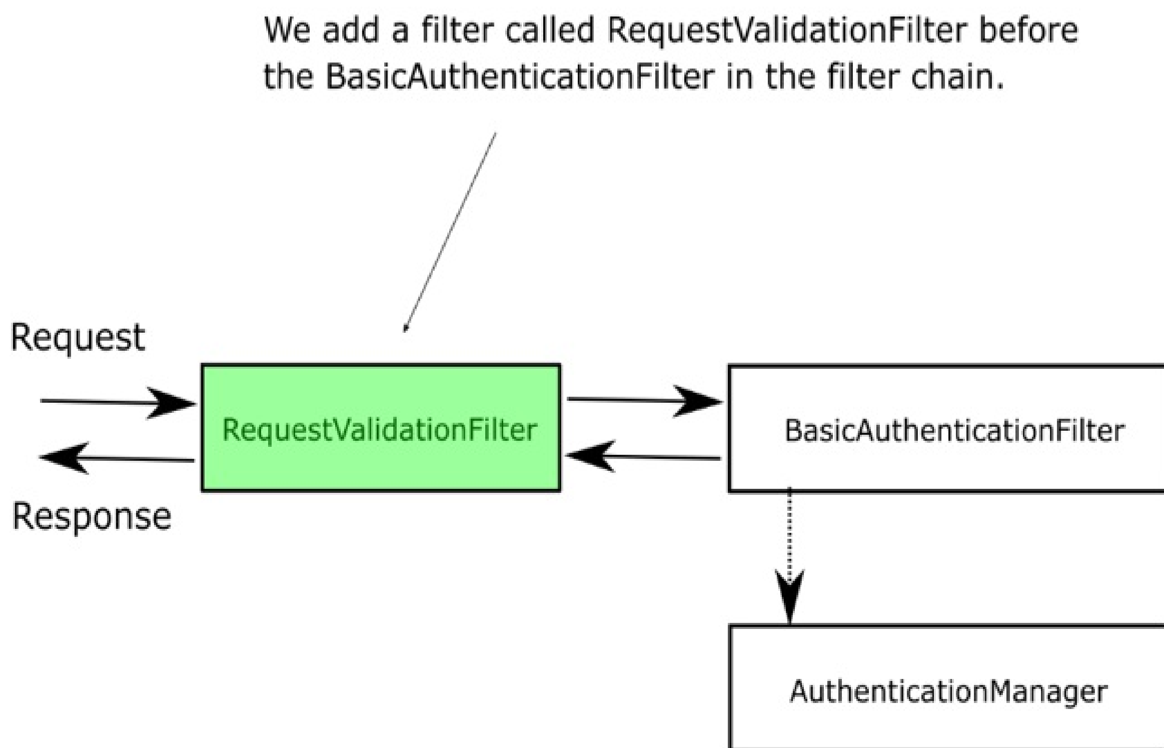
In this section, we discuss applying custom HTTP filters before an existing one in the filter chain. You might find scenarios in which this is useful. To approach this in a practical way, we'll work on a project for our example. With this example, you'll easily learn to implement a custom filter and apply it before an existing one in the filter chain. You can then adapt this example to any similar requirement you might find in a production application.

For our first custom filter implementation, let's consider a trivial scenario. We want to make sure that any request has a header called `Request-Id` (see project ssia-ch5-ex1). We assume that our application uses this header for

tracking requests and that this header is mandatory. At the same time, we want to validate these assumptions before the application performs authentication. The authentication process might involve querying the database or other resource-consuming actions that we don't want the application to execute if the format of the request isn't valid. How do we do this? To solve the current requirement only takes two steps, and in the end, the filter chain looks like the one in figure 5.6:

1. *Implement the filter*. Create a `RequestValidationFilter` class that checks that the needed header exists in the request.
2. *Add the filter to the filter chain*. Do this in the configuration class, overriding the `configure()` method.

Figure 5.6 For our example, we add a `RequestValidationFilter`, which acts before the authentication filter. The `RequestValidationFilter` ensures that authentication doesn't happen if the validation of the request fails. In our case, the request must have a mandatory header named `Request-Id`.



To accomplish step 1, implementing the filter, we define a custom filter. The

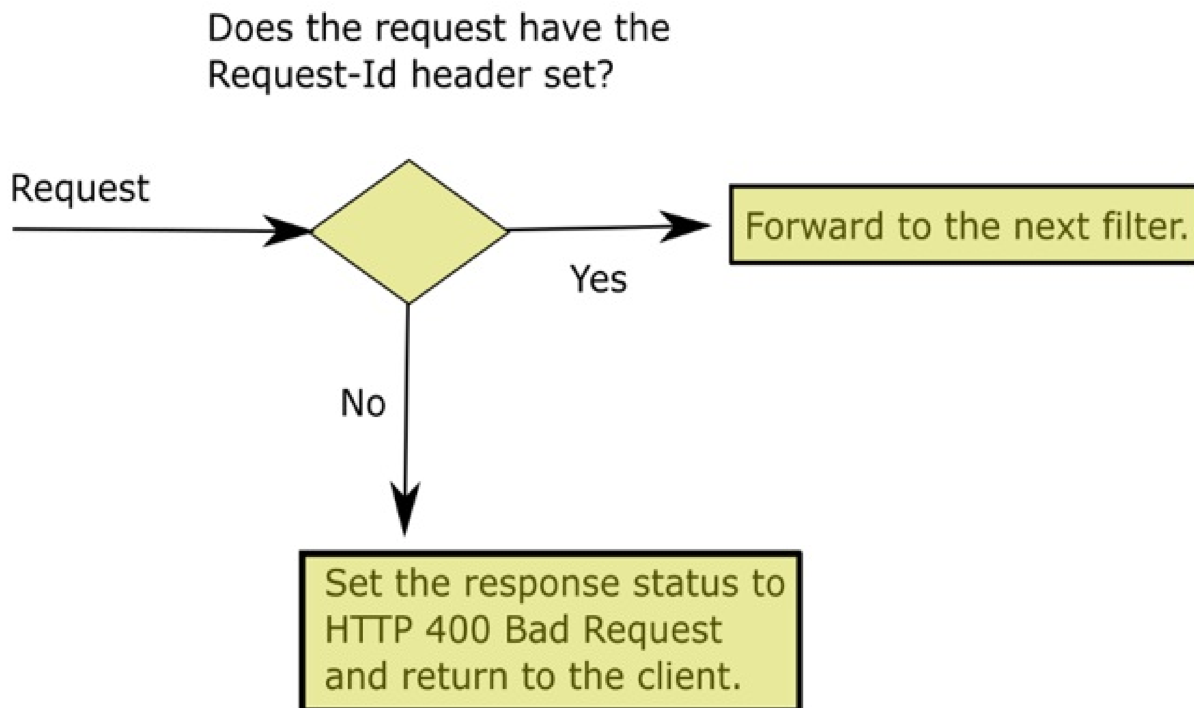
next listing shows the implementation.

Listing 5.1 Implementing a custom filter

```
public class RequestValidationFilter
  implements Filter {      #A

  @Override
  public void doFilter(
     ServletRequest servletRequest,
     ServletResponse servletResponse,
     FilterChain filterChain)
     throws IOException, ServletException {
     // ...
  }
}
```

Inside the `doFilter()` method, we write the logic of the filter. In our example, we check if the `Request-Id` header exists. If it does, we forward the request to the next filter in the chain by calling the `doFilter()` method. If the header doesn't exist, we set an HTTP status 400 Bad Request on the response without forwarding it to the next filter in the chain (figure 5.7). Listing 5.2 presents the logic.

Figure 5.7 The custom filter we add before authentication checks whether the `Request-Id` header exists. If the header exists on the request, the application forwards the request to be authenticated. If the header doesn't exist, the application sets the HTTP status 400 Bad Request and returns to the client.



Listing 5.2 Implementing the logic in the `doFilter()` method

```
@Override
public void doFilter(
  ServletRequest request,
  ServletResponse response,
  FilterChain filterChain)
    throws IOException,
           ServletException {

  var httpRequest = (HttpServletRequest) request;
  var httpResponse = (HttpServletResponse) response;

  String requestId = httpRequest.getHeader("Request-Id");

  if (requestId == null || requestId.isBlank()) {
      httpResponse.setStatus(HttpServletResponse.SC_BAD_REQUEST);
      return;     #A
  }

  filterChain.doFilter(request, response);    #B

}
```

To implement step 2, applying the filter within the configuration class, we use the `addFilterBefore()` method of the `HttpSecurity` object because we want the application to execute this custom filter before authentication. This method receives two parameters:

- "charitalics"An instance of the custom filter we want to add to the chain—In our example, this is an instance of the `RequestValidationFilter` class presented in listing 5.1.
- "charitalics"The type of filter before which we add the new instance—For this example, because the requirement is to execute the filter logic before authentication, we need to add our custom filter instance before the authentication filter. The class `BasicAuthenticationFilter` defines the default type of the authentication filter.

Until now, we have referred to the filter dealing with authentication generally as the "charitalics"authentication filter. You'll find out in the next chapters that Spring Security also configures other filters. In chapter 9, we'll discuss cross-site request forgery (CSRF) protection, and in chapter 10, we'll discuss cross-origin resource sharing (CORS). Both capabilities also rely on filters.

Listing 5.3 shows how to add the custom filter before the authentication filter in the configuration class. To make the example simpler, I use the `permitAll()` method to allow all unauthenticated requests.

Listing 5.3 Configuring the custom filter before authentication

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.addFilterBefore(     #A
       new RequestValidationFilter(), BasicAuthenticationFilter.c
       .authorizeRequests(c -> c.anyRequest().permitAll());


    return http.build();
  }
}
```

We also need a controller class and an endpoint to test the functionality. The next listing defines the controller class.

Listing 5.4 The controller class

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }
}
```

You can now run and test the application. Calling the endpoint without the header generates a response with HTTP status 400 Bad Request. If you add the header to the request, the response status becomes HTTP 200 OK, and you'll also see the response body, `Hello!` To call the endpoint without the `Request-Id` header, we use this cURL command:

```
curl -v http://localhost:8080/hello
```

This call generates the following (truncated) response:

```
...
< HTTP/1.1 400
...
```

To call the endpoint and provide the `Request-Id` header, we use this cURL command:

```
curl -H "Request-Id:12345" http://localhost:8080/hello
```

This call generates the following (truncated) response:

```
Hello!
```

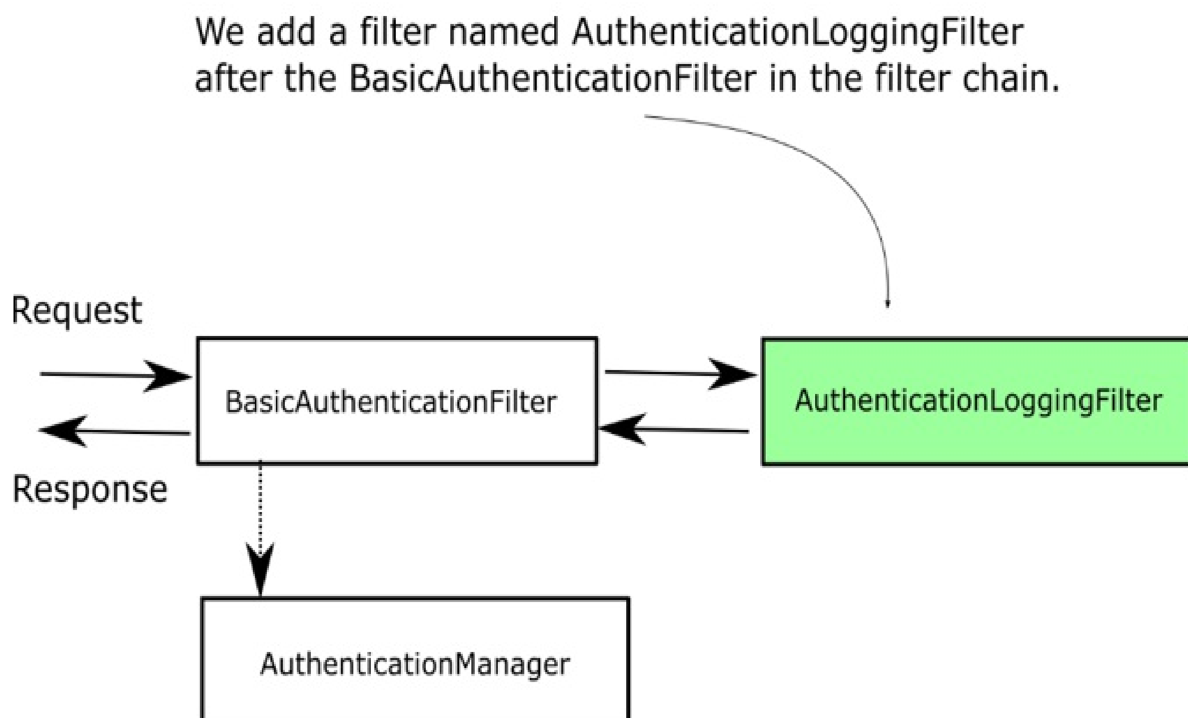
5.3 Adding a filter after an existing one in the chain

In this section, we discuss adding a filter after an existing one in the filter chain. You use this approach when you want to execute some logic after

something already existing in the filter chain. Let’s assume that you have to execute some logic after the authentication process. Examples for this could be notifying a different system after certain authentication events or simply for logging and tracing purposes (figure 5.8). As in section 5.1, we implement an example to show you how to do this. You can adapt it to your needs for a real-world scenario.

For our example, we log all successful authentication events by adding a filter after the authentication filter (figure 5.8). We consider that what bypasses the authentication filter represents a successfully authenticated event and we want to log it. Continuing the example from section 5.1, we also log the request ID received through the HTTP header.

Figure 5.8 We add the `AuthenticationLoggingFilter` after the `BasicAuthenticationFilter` to log the requests that the application authenticates.



The following listing presents the definition of a filter that logs requests that pass the authentication filter.

Listing 5.5 Defining a filter to log requests

```
public class AuthenticationLoggingFilter implements Filter {

  private final Logger logger =
          Logger.getLogger(
          AuthenticationLoggingFilter.class.getName());

  @Override
  public void doFilter(
    ServletRequest request,
    ServletResponse response,
    FilterChain filterChain)
      throws IOException, ServletException {

      var httpRequest = (HttpServletRequest) request;

      var requestId =
        httpRequest.getHeader("Request-Id");     #A

      logger.info("Successfully authenticated      #B
                   request with id " +  requestId);   #B

      filterChain.doFilter(request, response);     #C
  }
}
```

To add the custom filter in the chain after the authentication filter, you call the `addFilterAfter()` method of `HttpSecurity`. The next listing shows the implementation.

Listing 5.6 Adding a custom filter after an existing one in the filter chain

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.addFilterBefore(
            new RequestValidationFilter(),
            BasicAuthenticationFilter.class)
        .addFilterAfter(      #A
            new AuthenticationLoggingFilter(),
            BasicAuthenticationFilter.class)
        .authorizeRequests(c -> c.anyRequest().permitAll());
```

```
    return http.build();
  }
}
```

Running the application and calling the endpoint, we observe that for every successful call to the endpoint, the application prints a log line in the console. For the call

```
curl -H "Request-Id:12345" http://localhost:8080/hello
```

the response body is

```
Hello!
```

In the console, you can see a line similar to this:

```
INFO 5876 --- [nio-8080-exec-2] c.l.s.f.AuthenticationLoggingFilt
```

5.4 Adding a filter at the location of another in the chain

In this section, we discuss adding a filter at the location of another one in the filter chain. You use this approach especially when providing a different implementation for a responsibility that is already assumed by one of the filters known by Spring Security. A typical scenario is authentication.

Let’s assume that instead of the HTTP Basic authentication flow, you want to implement something different. Instead of using a username and a password as input credentials based on which the application authenticates the user, you need to apply another approach. Some examples of scenarios that you could encounter are

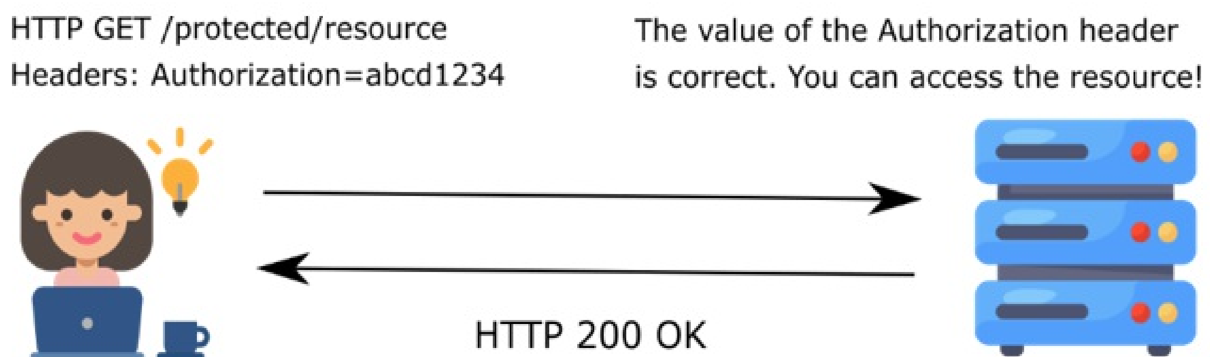
- Identification based on a static header value for authentication
- Using a symmetric key to sign the request for authentication
- Using a one-time password (OTP) in the authentication process

In our first scenario, identification based on a static key for authentication, the client sends a string to the app in the header of HTTP request, which is

always the same. The application stores these values somewhere, most probably in a database or a secrets vault. Based on this static value, the application identifies the client.

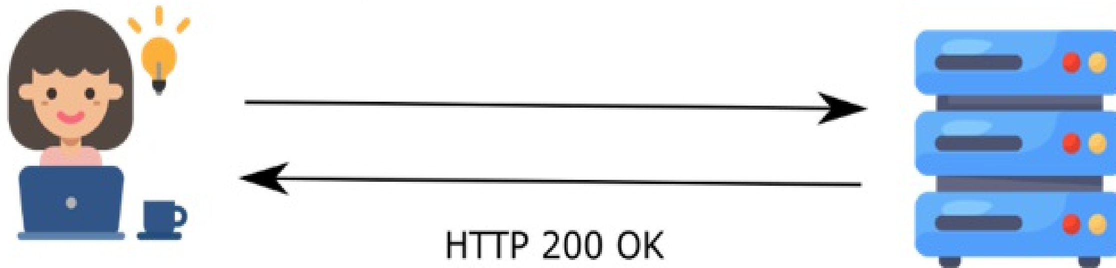
This approach (figure 5.9) offers weak security related to authentication, but architects and developers often choose it in calls between backend applications for its simplicity. The implementations also execute fast because these don't need to do complex calculations, as in the case of applying a cryptographic signature. This way, static keys used for authentication represent a compromise where developers rely more on the infrastructure level in terms of security and also don't leave the endpoints wholly unprotected.

Figure 5.9 The request contains a header with the value of the static key. If this value matches the one known by the application, it accepts the request.



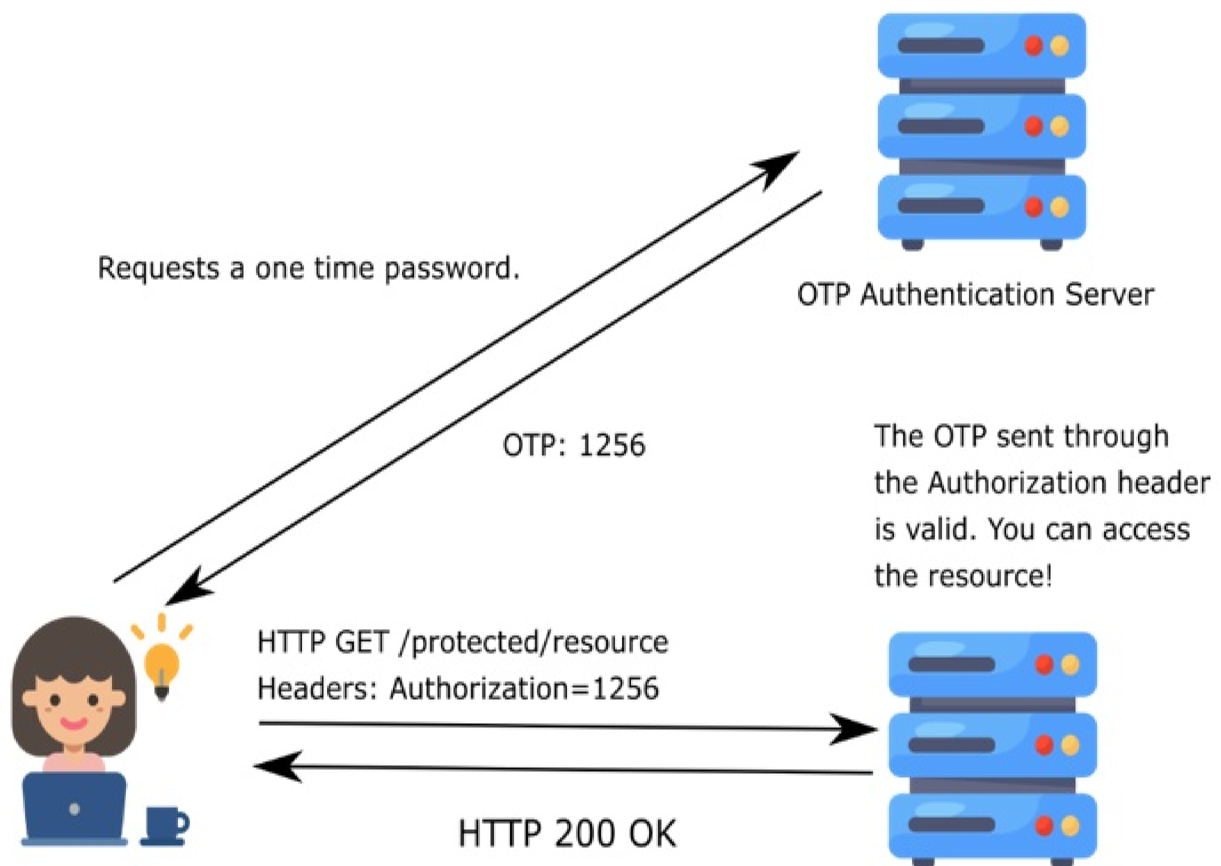
In our second scenario, using symmetric keys to sign and validate requests, both client and server know the value of a key (client and server share the key). The client uses this key to sign a part of the request (for example, to sign the value of specific headers), and the server checks if the signature is valid using the same key (figure 5.10). The server can store individual keys for each client in a database or a secrets vault. Similarly, you can use a pair of asymmetric keys.

Figure 5.10 The `Authorization` header contains a value signed with a key known by both client and server (or a private key for which the server has the public key). The application checks the signature and, if correct, allows the request.



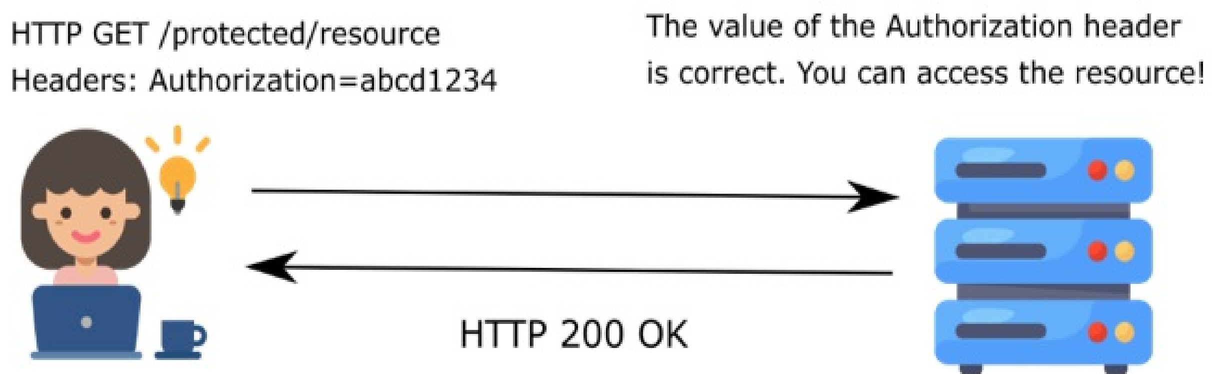
And finally, for our third scenario, using an OTP in the authentication process, the user receives the OTP via a message or by using an authentication provider app like Google Authenticator (figure 5.11).

Figure 5.11 To access the resource, the client has to use a one-time password (OTP). The client obtains the OTP from a third-party authentication server. Generally, applications use this approach during login when multifactor authentication is required.



Let's implement an example to demonstrate how to apply a custom filter. To keep the case relevant but straightforward, we focus on configuration and consider a simple logic for authentication. In our scenario, we have the value of a static key, which is the same for all requests. To be authenticated, the user must add the correct value of the static key in the `Authorization` header as presented in figure 5.12. You can find the code for this example in the project ssia-ch5-ex2.

Figure 5.12 The client adds a static key in the `Authorization` header of the HTTP request. The server checks if it knows the key before authorizing the requests.



We start with implementing the filter class, named `StaticKeyAuthenticationFilter`. This class reads the value of the static key from the properties file and verifies if the value of the `Authorization` header is equal to it. If the values are the same, the filter forwards the request to the next component in the filter chain. If not, the filter sets the value 401 Unauthorized to the HTTP status of the response without forwarding the request in the filter chain. Listing 5.7 defines the `StaticKeyAuthenticationFilter` class.

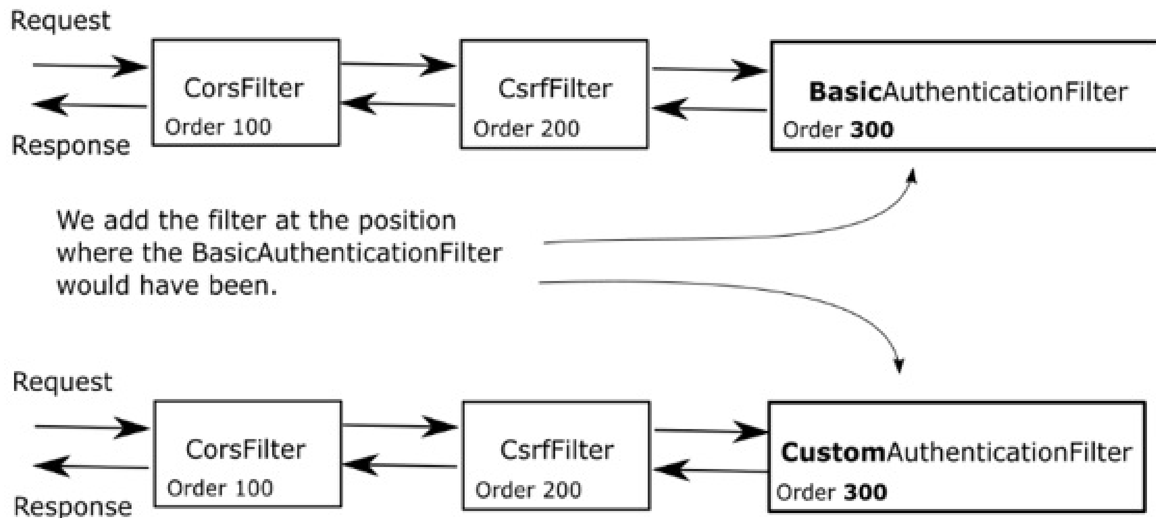
Listing 5.7 The definition of the `StaticKeyAuthenticationFilter` class

```
@Component      #A
public class StaticKeyAuthenticationFilter
  implements Filter {     #B

  @Value("${authorization.key}")      #C
  private String authorizationKey;
```

```
  @Override
  public void doFilter(ServletRequest request,
                       ServletResponse response,
                       FilterChain filterChain)
    throws IOException, ServletException {

    var httpRequest = (HttpServletRequest) request;
    var httpResponse = (HttpServletResponse) response;

    String authentication =     #D
           httpRequest.getHeader("Authorization");

    if (authorizationKey.equals(authentication)) {
        filterChain.doFilter(request, response);
    } else {
        httpResponse.setStatus(
            HttpServletResponse.SC_UNAUTHORIZED);
    }
  }
}
```

Once we define the filter, we add it to the filter chain at the position of the class `BasicAuthenticationFilter` by using the `addFilterAt()` method (figure 5.13).

Figure 5.13 We add our custom authentication filter at the location where the class `BasicAuthenticationFilter` would have been if we were using HTTP Basic as an authentication method. This means our custom filter has the same ordering value.



But remember what we discussed in section 5.1. When adding a filter at a specific position, Spring Security does not assume it is the only one at that position. You might add more filters at the same location in the chain. In this case, Spring Security doesn't guarantee in which order these will act. I tell you this again because I've seen many people confused by how this works. Some developers think that when you apply a filter at a position of a known one, it will be replaced. This is not the case! We must make sure not to add filters that we don't need to the chain.

NOTE

I do advise you not to add multiple filters at the same position in the chain. When you add more filters in the same location, the order in which they are used is not defined. It makes sense to have a definite order in which filters are called. Having a known order makes your application easier to understand and maintain.

In listing 5.8, you can find the definition of the configuration class that adds the filter. Observe that we don't call the `httpBasic()` method from the `HttpSecurity` class here because we don't want the `BasicAuthenticationFilter` instance to be added to the filter chain.

Listing 5.8 Adding the filter in the configuration class

```
@Configuration
public class ProjectConfig {

  private final StaticKeyAuthenticationFilter filter;    #A

  // omitted constructor

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {
    http.addFilterAt(filter,     #B
           BasicAuthenticationFilter.class)
        .authorizeRequests(c -> c.anyRequest().permitAll());


    return http.build();
  }
```

```
}
```

To test the application, we also need an endpoint. For that, we define a controller, as given in listing 5.4. You should add a value for the static key on the server in the application.properties file, as shown in this code snippet:

```
authorization.key=SD9cICjl1e
```

NOTE

Storing passwords, keys, or any other data that is not meant to be seen by everybody in the properties file is never a good idea for a production application. In our examples, we use this approach for simplicity and to allow you to focus on the Spring Security configurations we make. But in real-world scenarios, make sure to use a secrets vault to store such kinds of details.

We can now test the application. We expect that the app allows requests having the correct value for the `Authorization` header and rejects others, returning an HTTP 401 Unauthorized status on the response. The next code snippets present the `curl` calls used to test the application. If you use the same value you set on the server side for the `Authorization` header, the call is successful, and you'll see the response body, `Hello!` The call

```
curl -H "Authorization:SD9cICjl1e" http://localhost:8080/hello
```

returns this response body:

```
Hello!
```

With the following call, if the `Authorization` header is missing or is incorrect, the response status is HTTP 401 Unauthorized:

```
curl -v http://localhost:8080/hello
```

The response status is

```
...
< HTTP/1.1 401
...
```

In this case, because we don't configure a `UserDetailsService`, Spring Boot automatically configures one, as you learned in chapter 2. But in our scenario, you don't need a `UserDetailsService` at all because the concept of the user doesn't exist. We only validate that the user requesting to call an endpoint on the server knows a given value. Application scenarios are not usually this simple and often require a `UserDetailsService`. But, if you anticipate or have such a case where this component is not needed, you can disable autoconfiguration. To disable the configuration of the default `UserDetailsService`, you can use the `exclude` attribute of the `@SpringBootApplication` annotation on the main class like this:

```
@SpringBootApplication(exclude =
  {UserDetailsServiceAutoConfiguration.class })
```

5.5 Filter implementations provided by Spring Security

In this section, we discuss classes provided by Spring Security, which implement the `Filter` interface. In the examples in this chapter, we define the filter by implementing this interface directly.

Spring Security offers a few abstract classes that implement the `Filter` interface and for which you can extend your filter definitions. These classes also add functionality your implementations could benefit from when you extend them. For example, you could extend the `GenericFilterBean` class, which allows you to use initialization parameters that you would define in a web.xml descriptor file where applicable. A more useful class that extends the `GenericFilterBean` is `OncePerRequestFilter`. When adding a filter to the chain, the framework doesn't guarantee it will be called only once per request. `OncePerRequestFilter`, as the name suggests, implements logic to make sure that the filter's `doFilter()` method is executed only one time per request.

If you need such functionality in your application, use the classes that Spring provides. But if you don't need them, I'd always recommend you to go as simple as possible with your implementations. Too often, I've seen developers extending the `GenericFilterBean` class instead of implementing

the `Filter` interface in functionalities that don't require the custom logic added by the `GenericFilterBean` class. When asked why, it seems they don't know. They probably copied the implementation as they found it in examples on the web.

To make it crystal clear how to use such a class, let's write an example. The logging functionality we implemented in section 5.3 makes a great candidate for using `OncePerRequestFilter`. We want to avoid logging the same requests multiple times. Spring Security doesn't guarantee the filter won't be called more than once, so we have to take care of this ourselves. The easiest way is to implement the filter using the `OncePerRequestFilter` class. I wrote this in a separate project called ssia-ch5-ex3.

In listing 5.9, you find the change I made for the `AuthenticationLoggingFilter` class. Instead of implementing the `Filter` interface directly, as was the case in the example in section 5.3, now it extends the `OncePerRequestFilter` class. The method we override here is `doFilterInternal()`. You find this code in project ssia-ch5-ex3.

Listing 5.9 Extending the `OncePerRequestFilter` class

```
public class AuthenticationLoggingFilter
  extends OncePerRequestFilter {      #A

  private final Logger logger =
          Logger.getLogger(
            AuthenticationLoggingFilter.class.getName());

  @Override
  protected void doFilterInternal(     #B
    HttpServletRequest request,          #C
    HttpServletResponse response,        #C
    FilterChain filterChain) throws
      ServletException, IOException {

      String requestId = request.getHeader("Request-Id");

      logger.info("Successfully authenticated request with id " +
                   requestId);

      filterChain.doFilter(request, response);
  }
```

```
}
```

A few short observations about the `OncePerRequestFilter` class that you might find useful:

- "charitalics"It supports only HTTP requests, but that's actually what we always use. The advantage is that it casts the types, and we directly receive the requests as `HttpServletRequest` and `HttpServletResponse`. Remember, with the `Filter` interface, we had to cast the request and the response.
- "charitalics"You can implement logic to decide if the filter is applied or not. Even if you added the filter to the chain, you might decide it doesn't apply for certain requests. You set this by overriding the `shouldNotFilter(HttpServletRequest)` method. By default, the filter applies to all requests.
- "charitalics"By default, a `OncePerRequestFilter` "charitalics"doesn't apply to asynchronous requests or error dispatch requests. You can change this behavior by overriding the methods `shouldNotFilterAsyncDispatch()` and `shouldNotFilterErrorDispatch()`.

If you find any of these characteristics of the `OncePerRequestFilter` useful in your implementation, I recommend you use this class to define your filters.

5.6 Summary

- The first layer of the web application architecture, which intercepts HTTP requests, is a filter chain. As for other components in Spring Security architecture, you can customize them to match your requirements.
- You can customize the filter chain by adding new filters before an existing one, after an existing one, or at the position of an existing filter.
- You can have multiple filters at the same position of an existing filter. In this case, the order in which the filters are executed is not defined.
- Changing the filter chain helps you customize authentication and authorization to match precisely the requirements of your application.

6 Implementing authentication

This chapter covers

- Implementing authentication logic using a custom `AuthenticationProvider`
- Using the HTTP Basic and form-based login authentication methods
- Understanding and managing the `SecurityContext` component

In chapters 3 and 4, we covered a few of the components acting in the authentication flow. We discussed `UserDetails` and how to define the prototype to describe a user in Spring Security. We then used `UserDetails` in examples where you learned how the `UserDetailsService` and `UserDetailsManager` contracts work and how you can implement these. We discussed and used the leading implementations of these interfaces in examples as well. Finally, you learned how a `PasswordEncoder` manages passwords and how to use one, as well as the Spring Security crypto module (SSCM) with its encryptors and key generators.

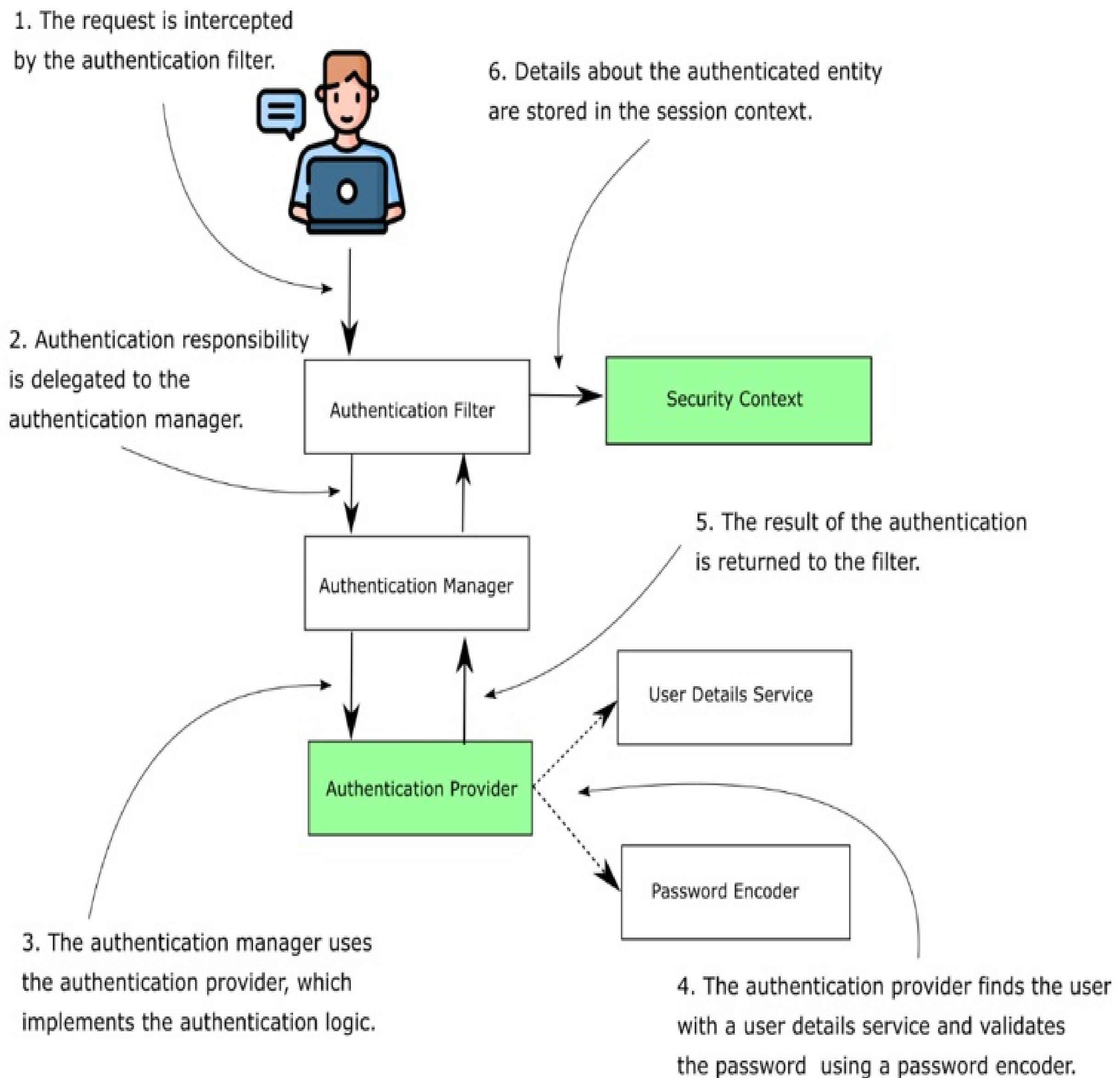
The `AuthenticationProvider` layer, however, is the one responsible for the logic of authentication. The `AuthenticationProvider` is where you find the conditions and instructions that decide whether to authenticate a request or not. The component that delegates this responsibility to the `AuthenticationProvider` is the `AuthenticationManager`, which receives the request from the HTTP filter layer, which we discussed in chapter 5. In this chapter, let's look at the authentication process, which has only two possible results:

- "charitalics"The entity making the request is not authenticated. The user is not recognized, and the application rejects the request without delegating to the authorization process. Usually, in this case, the response status sent back to the client is HTTP 401 Unauthorized.
- "charitalics"The entity making the request is authenticated. The details about the requester are stored such that the application can use these for authorization. As you'll find out in this chapter, `SecurityContext` is

responsible for the details about the current authenticated request.

To remind you of the actors and the links between them, figure 6.1 provides the diagram that you also saw in chapter 2.

Figure 6.1 The authentication flow in Spring Security. This process defines how the application identifies someone making a request. In the figure, the components discussed in this chapter are shaded. Here, the `AuthenticationProvider` implements the authentication logic in this process, while the `SecurityContext` stores the details about the authenticated request.



This chapter covers the remaining parts of the authentication flow (the shaded

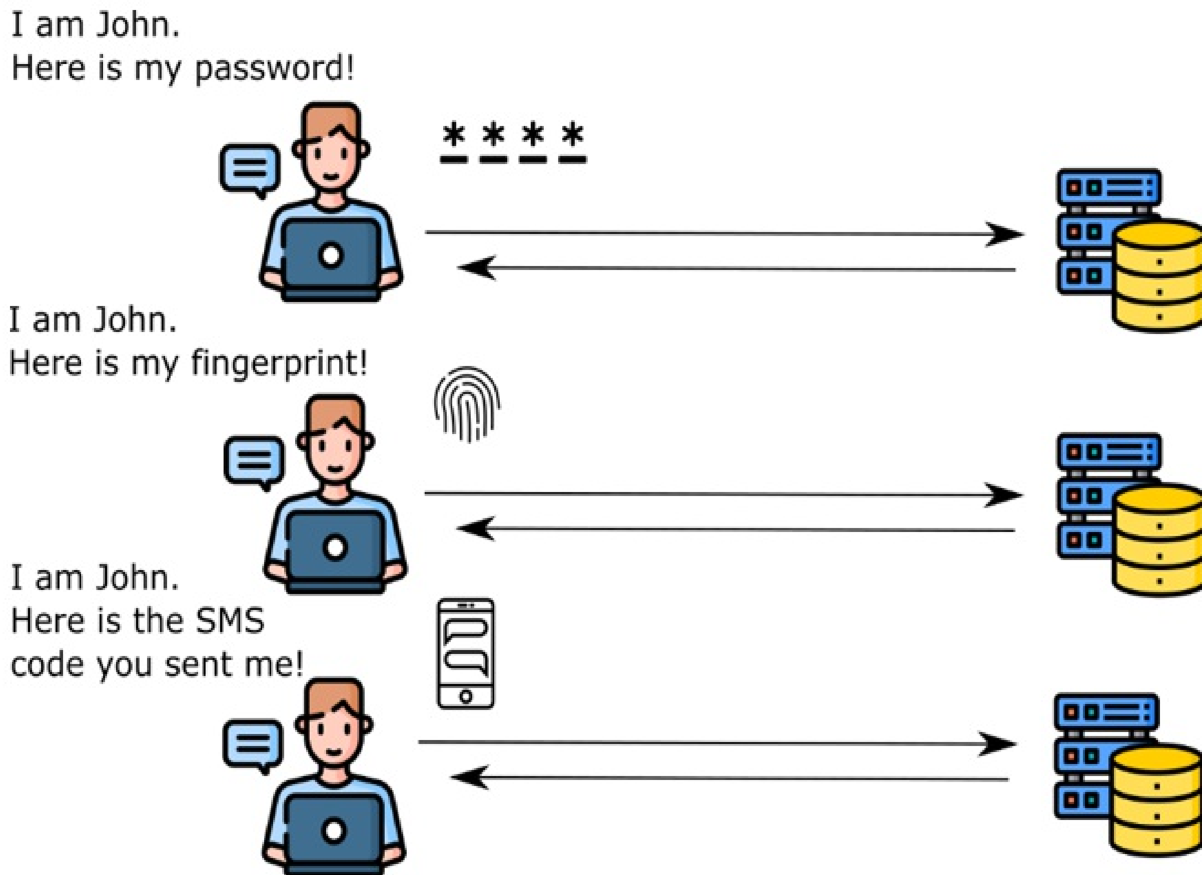
boxes in figure 6.1). Then, in chapters 7 and 8, you'll learn how authorization works, which is the process that follows authentication in the HTTP request. First, we need to discuss how to implement the `AuthenticationProvider` interface. You need to know how Spring Security understands a request in the authentication process.

To give you a clear description of how to represent an authentication request, we'll start with the `Authentication` interface. Once we discuss this, we can go further and observe what happens with the details of a request after successful authentication. After successful authentication, we can then discuss the `SecurityContext` interface and the way Spring Security manages it. Near the end of the chapter, you'll learn how to customize the HTTP Basic authentication method. We'll also discuss another option for authentication that we can use in our applications—the form-based login.

6.1 Understanding the AuthenticationProvider

In enterprise applications, you might find yourself in a situation in which the default implementation of authentication based on username and password does not apply. Additionally, when it comes to authentication, your application may require the implementation of several scenarios (figure 6.2). For example, you might want the user to be able to prove who they are by using a code received in an SMS message or displayed by a specific application. Or, you might need to implement authentication scenarios where the user has to provide a certain kind of key stored in a file. You might even need to use a representation of the user's fingerprint to implement the authentication logic. A framework's purpose is to be flexible enough to allow you to implement any of these required scenarios.

Figure 6.2 For an application, you might need to implement authentication in different ways. While in most cases a username and a password are enough, in some cases, the user-authentication scenario might be more complicated.



A framework usually provides a set of the most commonly used implementations, but it cannot, of course, cover all the possible options. In terms of Spring Security, you can use the `AuthenticationProvider` contract to define any custom authentication logic. In this section, you learn to represent the authentication event by implementing the `Authentication` interface and then creating your custom authentication logic with an `AuthenticationProvider`. To achieve our goal

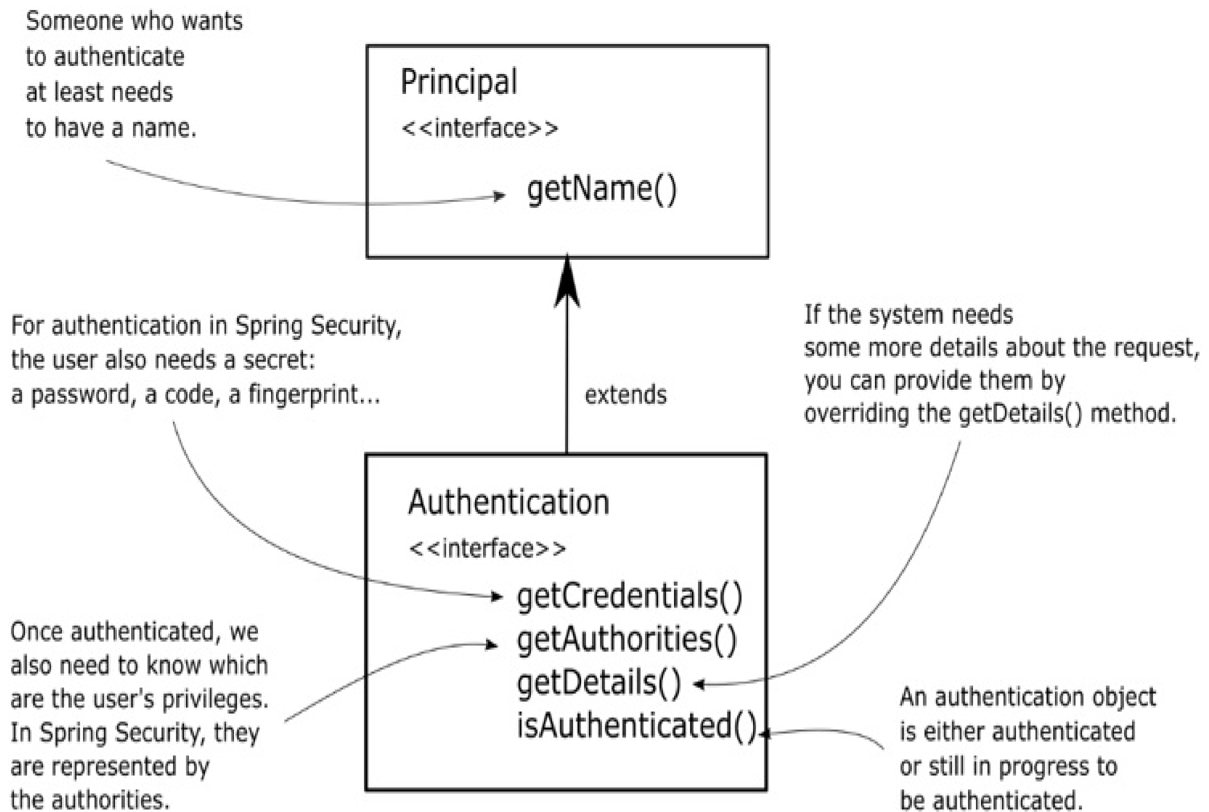
- In section 6.1.1, we analyze how Spring Security represents the authentication event.
- In section 6.1.2, we discuss the `AuthenticationProvider` contract, which is responsible for the authentication logic.
- In section 6.1.3, you write custom authentication logic by implementing the `AuthenticationProvider` contract in an example.

6.1.1 Representing the request during authentication

In this section, we discuss how Spring Security understands a request during the authentication process. It is important to touch on this before diving into implementing custom authentication logic. As you'll learn in section 6.1.2, to implement a custom `AuthenticationProvider`, you first need to understand how to describe the authentication event itself. In this section, we take a look at the contract representing authentication and discuss the methods you need to know.

`Authentication` is one of the essential interfaces involved in the process with the same name. The `Authentication` interface represents the authentication request event and holds the details of the entity that requests access to the application. You can use the information related to the authentication request event during and after the authentication process. The user requesting access to the application is called a "charitalics"principal. If you've ever used Java Security in any app, you learned that in Java Security, an interface named `Principal` represents the same concept. The `Authentication` interface of Spring Security extends this contract (figure 6.3).

Figure 6.3 The `Authentication` contract inherits from the `Principal` contract. `Authentication` adds requirements such as the need for a password or the possibility to specify more details about the authentication request. Some of these details, like the list of authorities, are Spring Security-specific.



The `Authentication` contract in Spring Security not only represents a principal, it also adds information on whether the authentication process finishes, as well as a collection of authorities. The fact that this contract was designed to extend the `Principal` contract from Java Security is a plus in terms of compatibility with implementations of other frameworks and applications. This flexibility allows for more facile migrations to Spring Security from applications that implement authentication in another fashion.

Let’s find out more about the design of the `Authentication` interface, in the following listing.

Listing 6.1 The `Authentication` interface as declared in Spring Security

```
public interface Authentication extends Principal, Serializable {

  Collection<? extends GrantedAuthority> getAuthorities();
  Object getCredentials();
  Object getDetails();
```

```
  Object getPrincipal();
  boolean isAuthenticated();
  void setAuthenticated(boolean isAuthenticated)
     throws IllegalArgumentException;
}
```

For the moment, the only methods of this contract that you need to learn are these:

- `isAuthenticated()`—Returns true if the authentication process ends or false if the authentication process is still in progress.
- `getCredentials()`—Returns a password or any secret used in the process of authentication.
- `getAuthorities()`—Returns a collection of granted authorities for the authenticated request.

We'll discuss the other methods for the `Authentication` contract in later chapters, where appropriate to the implementations we look at then.

6.1.2 Implementing custom authentication logic

In this section, we discuss implementing custom authentication logic. We analyze the Spring Security contract related to this responsibility to understand its definition. With these details, you implement custom authentication logic with a code example in section 6.1.3.

The `AuthenticationProvider` in Spring Security takes care of the authentication logic. The default implementation of the `AuthenticationProvider` interface delegates the responsibility of finding the system's user to a `UserDetailsService`. It uses the `PasswordEncoder` as well for password management in the process of authentication. The following listing gives the definition of the `AuthenticationProvider`, which you need to implement to define a custom authentication provider for your application.

Listing 6.2 The `AuthenticationProvider` interface

```
public interface AuthenticationProvider {

  Authentication authenticate(Authentication authentication)
```

```
    throws AuthenticationException;

  boolean supports(Class<?> authentication);
}
```

The `AuthenticationProvider` responsibility is strongly coupled with the `Authentication` contract. The `authenticate()` method receives an `Authentication` object as a parameter and returns an `Authentication` object. We implement the `authenticate()` method to define the authentication logic. We can quickly summarize the way you should implement the `authenticate()` method with three bullets:

- The method should throw an `AuthenticationException` if the authentication fails.
- If the method receives an authentication object that is not supported by your implementation of `AuthenticationProvider`, then the method should return null. This way, we have the possibility of using multiple `Authentication` types separated at the HTTP-filter level.
- The method should return an `Authentication` instance representing a fully authenticated object. For this instance, the `isAuthenticated()` method returns true, and it contains all the necessary details about the authenticated entity. Usually, the application also removes sensitive data like a password from this instance. After a successful authentication, the password is no longer required and keeping these details can potentially expose them to unwanted eyes.

The second method in the `AuthenticationProvider` interface is `supports(Class<?> authentication)`. You can implement this method to return true if the current `AuthenticationProvider` supports the type provided as an `Authentication` object. Observe that even if this method returns true for an object, there is still a chance that the `authenticate()` method rejects the request by returning null. Spring Security is designed like this to be more flexible and to allow you to implement an `AuthenticationProvider` that can reject an authentication request based on the request's details, not only by its type.

An analogy of how the authentication manager and authentication provider work together to validate or invalidate an authentication request is having a

more complex lock for your door. You can open this lock either by using a card or an old fashioned physical key (figure 6.4). The lock itself is the authentication manager that decides whether to open the door. To make that decision, it delegates to the two authentication providers: one that knows how to validate the card or the other that knows how to verify the physical key. If you present a card to open the door, the authentication provider that works only with physical keys complains that it doesn't know this kind of authentication. But the other provider supports this kind of authentication and verifies whether the card is valid for the door. This is actually the purpose of the -`supports()` methods.

Besides testing the authentication type, Spring Security adds one more layer for flexibility. The door's lock can recognize multiple kinds of cards. In this case, when you present a card, one of the authentication providers could say, "I understand this as being a card. But it isn't the type of card I can validate!" This happens when `supports()` returns true but `authenticate()` returns null.

Figure 6.4 The `AuthenticationManager` delegates to one of the available authentication providers. The `AuthenticationProvider` might not support the provided type of authentication. On the other hand, if it does support the object type, it might not know how to authenticate that specific object. The authentication is evaluated, and an `AuthenticationProvider` that can say if the request is correct or not responds to the `AuthenticationManager`.

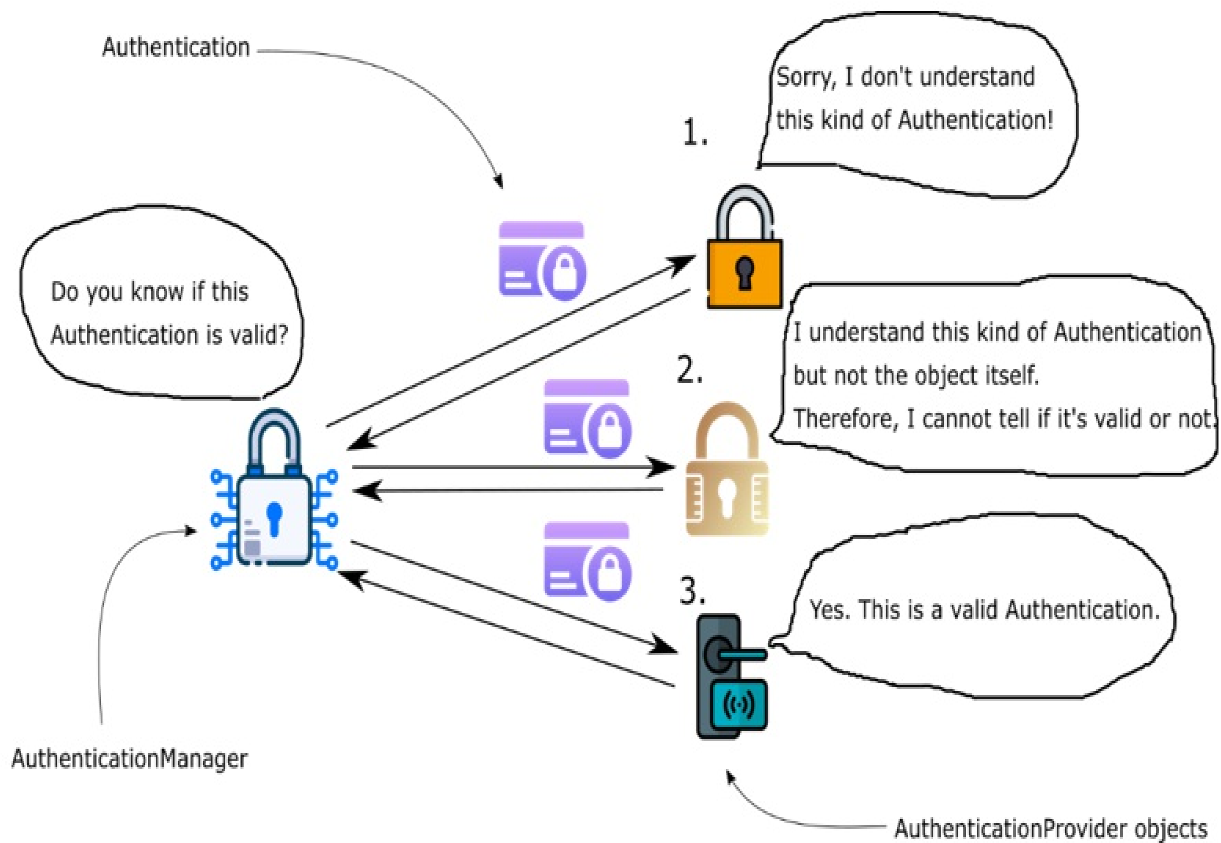
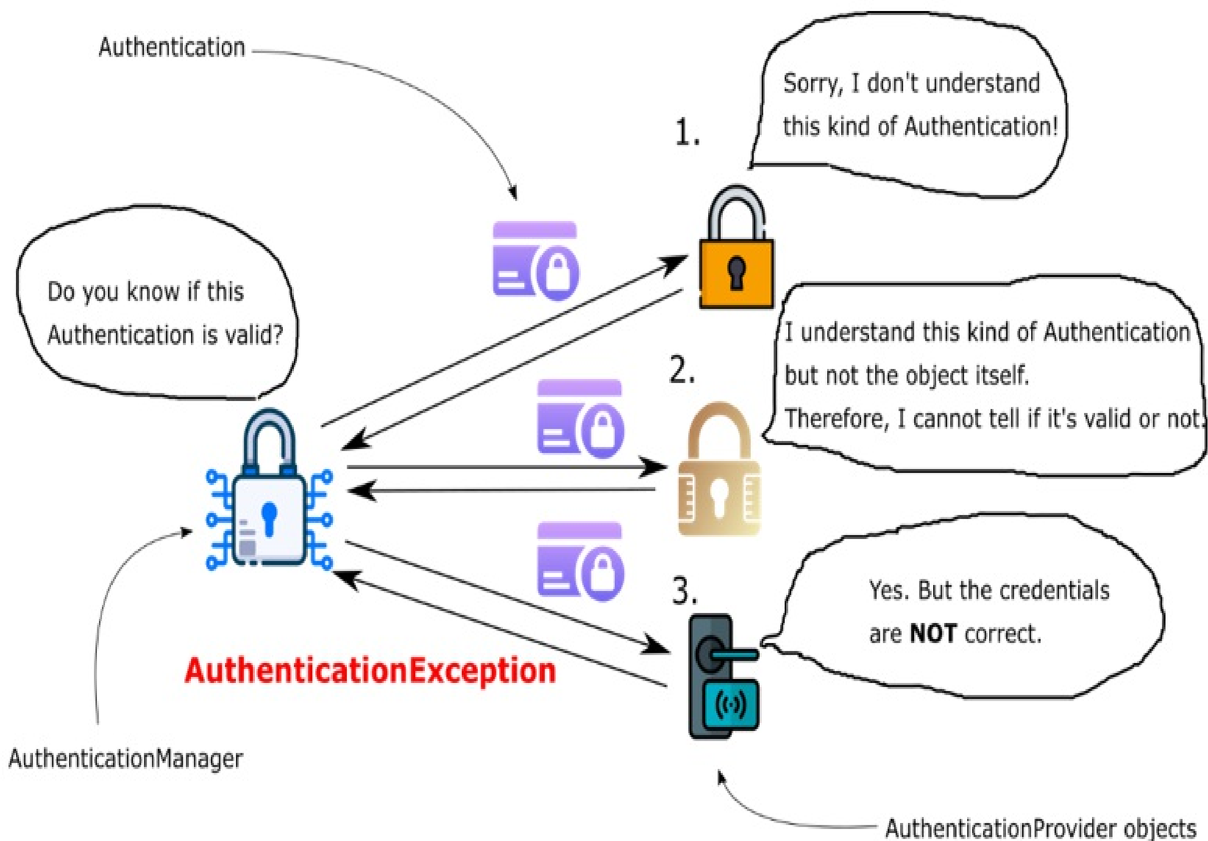


Figure 6.5 shows the alternative scenario where one of the `AuthenticationProvider` objects recognizes the `Authentication` but decides it's not valid. The result in this case will be an `AuthenticationException` which ends up as a 401 Unauthorized HTTP status in the HTTP response in a web app.

Figure 6.5 If none of the `AuthenticationProvider` objects recognize the `Authentication` or any of them rejects it, the result is an `AuthenticationException`.



6.1.3 Applying custom authentication logic

In this section, we implement custom authentication logic. You can find this example in the project ssia-ch6-ex1. With this example, you apply what you've learned about the `Authentication` and `AuthenticationProvider` interfaces in sections 6.1.1 and 6.1.2. In listings 6.3 and 6.4, we build, step by step, an example of how to implement a custom `AuthenticationProvider`. These steps, also presented in figure 6.5, follow:

1. Declare a class that implements the `AuthenticationProvider` contract.
2. Decide which kinds of `Authentication` objects the new `AuthenticationProvider` supports:
3. Implement the `supports(Class<?> c)` method to specify which type of authentication is supported by the `AuthenticationProvider` that we define.
4. Implement the `authenticate(Authentication a)` method to implement the authentication logic.

5. Register an instance of the new `AuthenticationProvider` implementation with Spring Security.

Listing 6.3 Overriding the `supports()` method of the `AuthenticationProvider`

```
@Component
public class CustomAuthenticationProvider
  implements AuthenticationProvider {

  // Omitted code

  @Override
  public boolean supports(Class<?> authenticationType) {
    return authenticationType
            .equals(UsernamePasswordAuthenticationToken.class);
  }
}
```

In listing 6.3, we define a new class that implements the `AuthenticationProvider` interface. We mark the class with `@Component` to have an instance of its type in the context managed by Spring. Then, we have to decide what kind of `Authentication` interface implementation this `AuthenticationProvider` supports. That depends on what type we expect to be provided as a parameter to the `authenticate()` method. If we don't customize anything at the authentication filter level (as discussed in chapter 5), then the class `UsernamePasswordAuthenticationToken` defines the type. This class is an implementation of the `Authentication` interface and represents a standard authentication request with username and password.

With this definition, we made the `AuthenticationProvider` support a specific kind of key. Once we have specified the scope of our `AuthenticationProvider`, we implement the authentication logic by overriding the `authenticate()` method as shown in following listing.

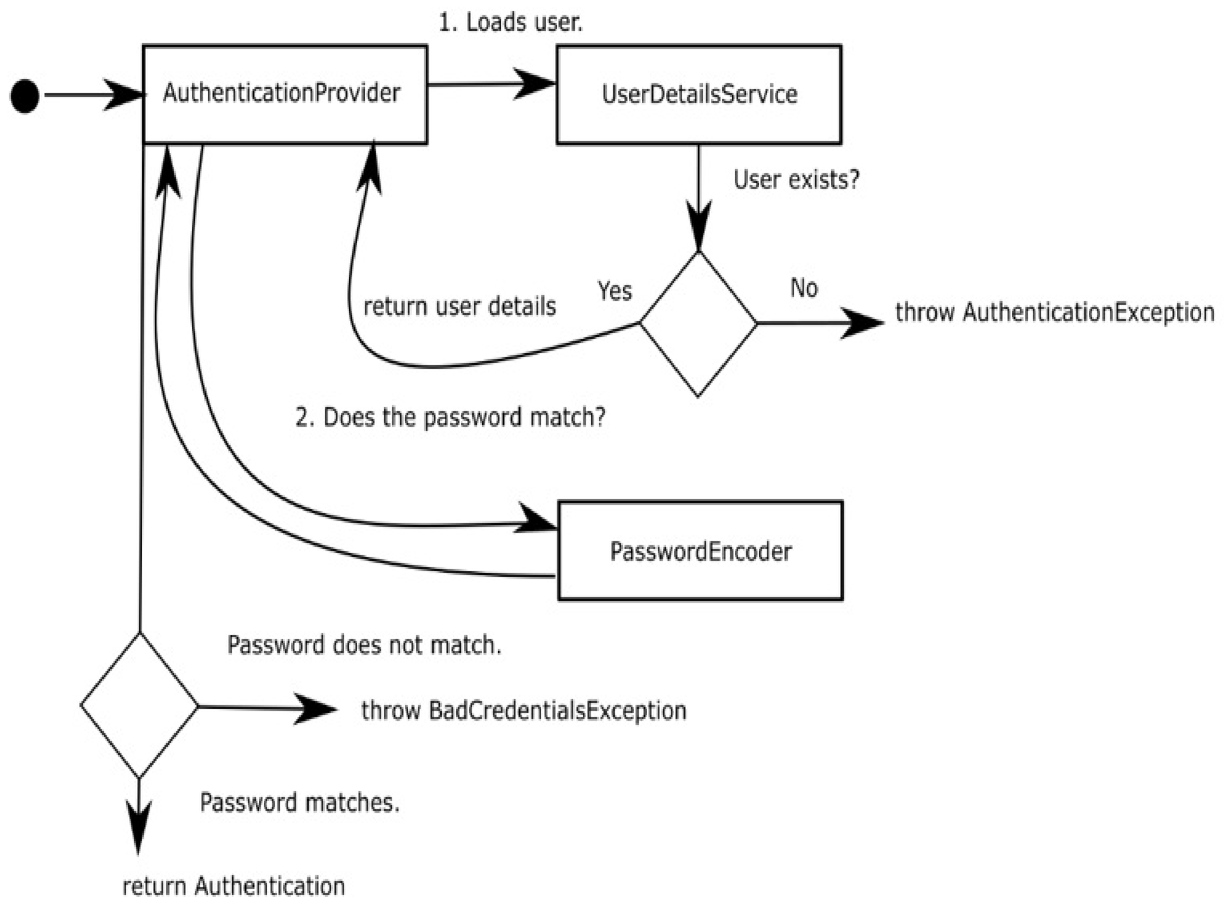
Listing 6.4 Implementing the authentication logic

```
@Component
public class CustomAuthenticationProvider
  implements AuthenticationProvider {

  private final UserDetailsService userDetailsService;
```

```
  private final PasswordEncoder passwordEncoder;

  // Omitted constructor

  @Override
  public Authentication authenticate(Authentication authenticatio
    String username = authentication.getName();
    String password = authentication.getCredentials().toString();

    UserDetails u = userDetailsService.loadUserByUsername(usernam

    if (passwordEncoder.matches(password, u.getPassword())) {
      return new UsernamePasswordAuthenticationToken(
            username,
            password,
            u.getAuthorities());     #A
    } else {
      throw new BadCredentialsException
                  ("Something went wrong!");    #B
    }
  }

  // Omitted code
}
```

The logic in listing 6.4 is simple, and figure 6.6 shows this logic visually. We make use of the `UserDetailsService` implementation to get the `UserDetails`. If the user doesn't exist, the `loadUserByUsername()` method should throw an `AuthenticationException`. In this case, the authentication process stops, and the HTTP filter sets the response status to HTTP 401 Unauthorized. If the username exists, we can check further the user's password with the `matches()` method of the `PasswordEncoder` from the context. If the password does not match, then again, an `AuthenticationException` should be thrown. If the password is correct, the `AuthenticationProvider` returns an instance of `Authentication` marked as "authenticated," which contains the details about the request.

Figure 6.6 The custom authentication flow implemented by the `AuthenticationProvider`. To validate the authentication request, the `AuthenticationProvider` loads the user details with a provided implementation of `UserDetailsService`, and if the password matches, validates the password with a `PasswordEncoder`. If either the user does not exist or the password is incorrect, the `AuthenticationProvider` throws an `AuthenticationException`.



To plug in the new implementation of the `AuthenticationProvider`, we define a `SecurityFilterChain` bean. This is demonstrated in the following listing.

Listing 6.5 Registering the `AuthenticationProvider` in the configuration class

```
@Configuration
public class ProjectConfig {

  private final AuthenticationProvider authenticationProvider;

  // Omitted constructor

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());
```

```
    http.authenticationProvider(authenticationProvider);

    http.authorizeHttpRequests(c -> c.anyRequest().authenticated(

    return http.build();
  }

  // Omitted code
}
```

NOTE

In listing 6.5, I use the dependency injection with a field declared with the `AuthenticationProvider` interface. Spring recognizes the `AuthenticationProvider` as an interface (which is an abstraction). But Spring knows that it needs to find in its context an instance of an implementation for that specific interface. In our case, the implementation is the instance of `CustomAuthenticationProvider`, which is the only one of this type that we declared and added to the Spring context using the `@Component` annotation. For a refresher on dependency injection I recommend you read Spring Start Here (Manning, 2021), another book I wrote.

That's it! You successfully customized the implementation of the `AuthenticationProvider`. You can now customize the authentication logic for your application where you need it.

How to fail in application design

Incorrectly applying a framework leads to a less maintainable application. Worse is sometimes those who fail in using the framework believe that it's the framework's fault. Let me tell you a story.

One winter, the head of development in a company I worked with as a consultant called me to help them with the implementation of a new feature. They needed to apply a custom authentication method in a component of their system developed with Spring in its early days. Unfortunately, when implementing the application's class design the developers didn't rely properly on Spring Security's backbone architecture.

They only relied on the filter chain, reimplementing entire features from Spring Security as custom code.

Developers observed that with time, customizations became more and more difficult. But nobody took action to redesign the component properly to use the contracts as intended in Spring Security. Much of the difficulty came from not knowing Spring's capabilities. One of the lead developers said, "It's only the fault of this Spring Security! This framework is hard to apply, and it's difficult to use with any customization." I was a bit shocked at his observation. I know that Spring Security is sometimes difficult to understand, and the framework is known for not having a soft learning curve. But I've never experienced a situation in which I couldn't find a way to design an easy-to-customize class with Spring Security!

We investigated together, and I realized the application developers only used maybe 10% of what Spring Security offers. Then, I presented a two-day workshop on Spring Security, focusing on what (and how) we could do for the specific system component they needed to change.

Everything ended with the decision to completely rewrite a lot of custom code to rely correctly on Spring Security and, thus, make the application easier to extend to meet their concerns for security implementations. We also discovered some other issues unrelated to Spring Security, but that's another story.

A few lessons for you to take from this story:

- A framework, and especially one widely used in applications, is written with the participation of many smart individuals. Even so, it's hard to believe that it can be badly implemented. Always analyze your application before concluding that any problems are the framework's fault.

- When deciding to use a framework, make sure you understand, at least, its basics well.

- Be mindful of the resources you use to learn about the framework. Sometimes, articles you find on the web show you how to do quick workarounds and not necessarily how to correctly implement a class design.

· Use multiple sources in your research. To clarify your misunderstandings, write a proof of concept when unsure how to use something.

· If you decide to use a framework, use it as much as possible for its intended purpose. For example, say you use Spring Security, and you observe that for security implementations, you tend to write more custom code instead of relying on what the framework offers. You should raise a question on why this happens.

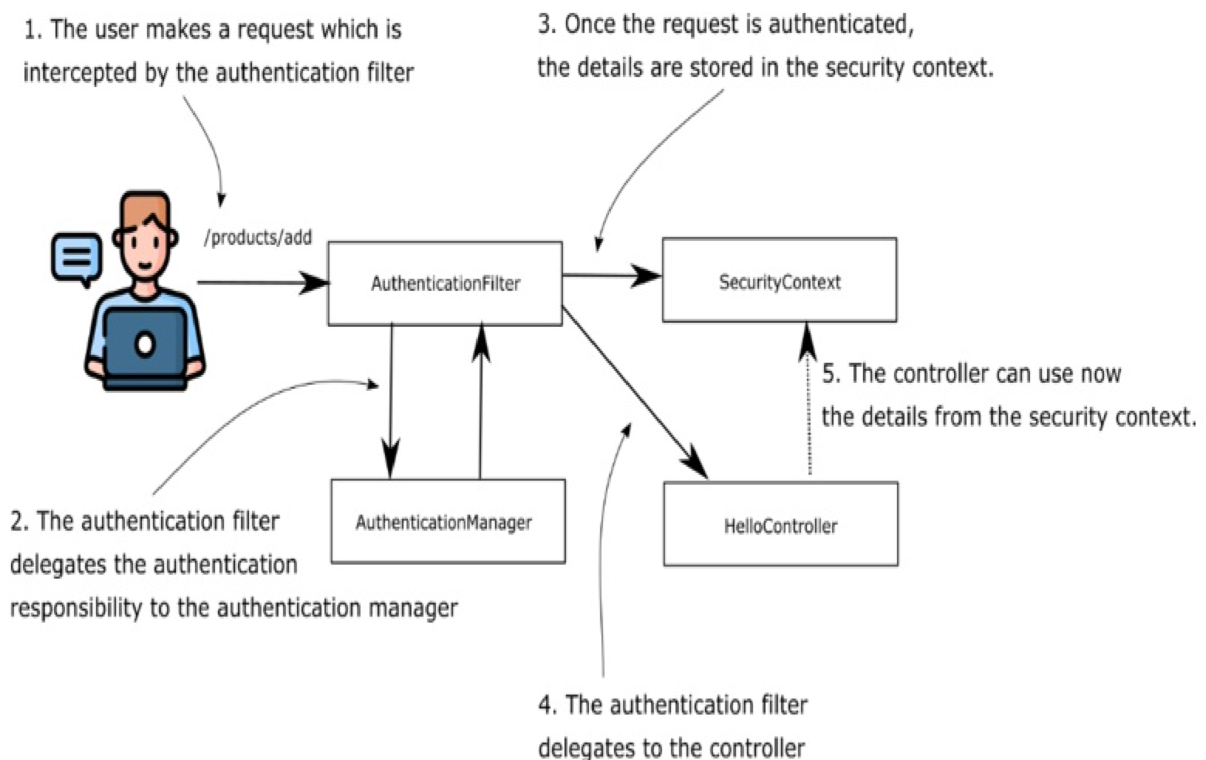
When we rely on functionalities implemented by a framework, we enjoy several benefits. We know they are tested and there are fewer changes that include vulnerabilities. Also, a good framework relies on abstractions, which help you create maintainable applications. Remember that when you write your own implementations, you're more susceptible to including vulnerabilities.

6.2 Using the SecurityContext

This section discusses the security context. We analyze how it works, how to access data from it, and how the application manages it in different thread-related scenarios. Once you finish this section, you'll know how to configure the security context for various situations. This way, you can use the details about the authenticated user stored by the security context in configuring authorization in chapters 7 and 8.

It is likely that you will need details about the authenticated entity after the authentication process ends. You might, for example, need to refer to the username or the authorities of the currently authenticated user. Is this information still accessible after the authentication process finishes? Once the `AuthenticationManager` completes the authentication process successfully, it stores the `Authentication` instance for the rest of the request. The instance storing the `Authentication` object is called the "charitalics"security context.

Figure 6.7 After successful authentication, the authentication filter stores the details of the authenticated entity in the security context. From there, the controller implementing the action mapped to the request can access these details when needed.



The security context of Spring Security is described by the `SecurityContext` interface. The following listing defines this interface.

Listing 6.6 The `SecurityContext` interface

```
public interface SecurityContext extends Serializable {

  Authentication getAuthentication();
  void setAuthentication(Authentication authentication);
}
```

As you can observe from the contract definition, the primary responsibility of the `SecurityContext` is to store the `Authentication` object. But how is the `SecurityContext` itself managed? Spring Security offers three strategies to manage the `SecurityContext` with an object in the role of a manager. It's named the `SecurityContextHolder`:

- `MODE_THREADLOCAL`—Allows each thread to store its own details in the security context. In a thread-per-request web application, this is a common approach as each request has an individual thread.

- `MODE_INHERITABLETHREADLOCAL`—Similar to `MODE_THREADLOCAL` but also instructs Spring Security to copy the security context to the next thread in case of an asynchronous method. This way, we can say that the new thread running the `@Async` method inherits the security context. The `@Async` annotation is used with methods to instruct Spring to call the annotated method on a separate thread.
- `MODE_GLOBAL`—Makes all the threads of the application see the same security context instance.

Besides these three strategies for managing the security context provided by Spring Security, in this section, we also discuss what happens when you define your own threads that are not known by Spring. As you will learn, for these cases, you need to explicitly copy the details from the security context to the new thread. Spring Security cannot automatically manage objects that are not in Spring's context, but it offers some great utility classes for this.

6.2.1 Using a holding strategy for the security context

The first strategy for managing the security context is the `MODE_THREADLOCAL` strategy. This strategy is also the default for managing the security context used by Spring Security. With this strategy, Spring Security uses `ThreadLocal` to manage the context. `ThreadLocal` is an implementation provided by the JDK. This implementation works as a collection of data but makes sure that each thread of the application can see only the data stored in its dedicated part of the collection. This way, each request has access to its security context. No thread will have access to another's `ThreadLocal`. And that means that in a web application, each request can see only its own security context. We could say that this is also what you generally want to have for a backend web application.

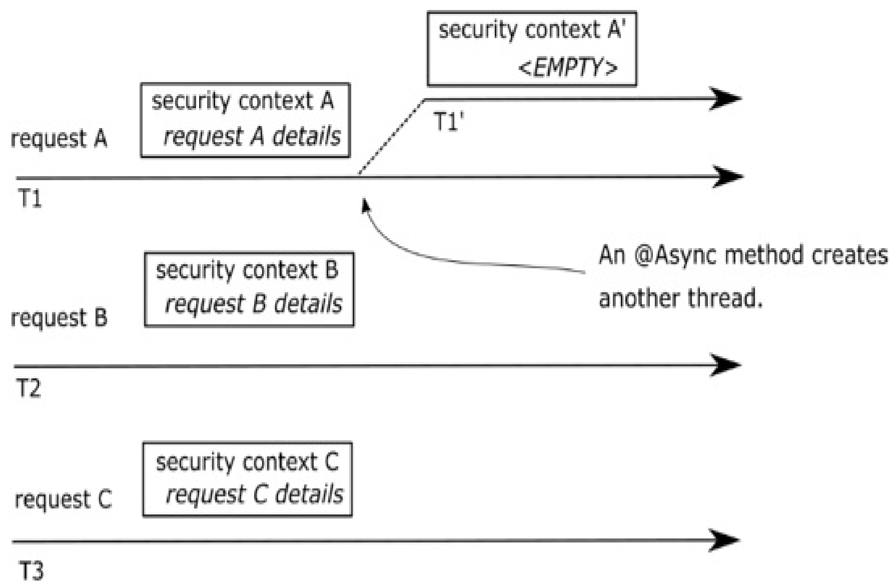
Figure 6.8 offers an overview of this functionality. Each request (A, B, and C) has its own allocated thread (T1, T2, and T3). This way, each request only sees the details stored in their own security context. But this also means that if a new thread is created (for example, when an asynchronous method is called), the new thread will have its own security context as well. The details from the parent thread (the original thread of the request) are not copied to the security context of the new thread.

NOTE

Here we discuss a traditional servlet application where each request is tied to a thread. This architecture only applies to the traditional servlet application where each request has its own thread assigned. It does not apply to reactive applications. We'll discuss the security for reactive approaches in detail in chapter 17.

Being the default strategy for managing the security context, this process does not need to be explicitly configured. Just ask for the security context from the holder using the static `getContext()` method wherever you need it after the end of the authentication process. In listing 6.7, you find an example of obtaining the security context in one of the endpoints of the application. From the security context, you can further get the `Authentication` object, which stores the details about the authenticated entity. You can find the examples we discuss in this section as part of the project ssia-ch6-ex2.

Figure 6.8 Each request has its own thread, represented by an arrow. Each thread has access only to its own security context details. When a new thread is created (for example, by an `@Async` method), the details from the parent thread aren't copied.



Listing 6.7 Obtaining the `SecurityContext` from the `SecurityContextHolder`

```
@GetMapping("/hello")
public String hello() {
  SecurityContext context = SecurityContextHolder.getContext();
  Authentication a = context.getAuthentication();

  return "Hello, " + a.getName() + "!";
}
```

Obtaining the authentication from the context is even more comfortable at the endpoint level, as Spring knows to inject it directly into the method parameters. You don’t need to refer every time to the `SecurityContextHolder` class explicitly. This approach, as presented in the following listing, is better.

Listing 6.8 Spring injects `Authentication` value in the parameter of the method

```
@GetMapping("/hello")
public String hello(Authentication a) {      #A
  return "Hello, " + a.getName() + "!";
}
```

When calling the endpoint with a correct user, the response body contains the username. For example,

```
curl -u user:99ff79e3-8ca0-401c-a396-0a8625ab3bad http://localhos
Hello, user!
```

6.2.2 Using a holding strategy for asynchronous calls

It is easy to stick with the default strategy for managing the security context. And in a lot of cases, it is the only thing you need. `MODE_THREADLOCAL` offers you the ability to isolate the security context for each thread, and it makes the security context more natural to understand and manage. But there are also cases in which this does not apply.

The situation gets more complicated if we have to deal with multiple threads per request. Look at what happens if you make the endpoint asynchronous. The thread that executes the method is no longer the same thread that serves the request. Think about an endpoint like the one presented in the next listing.

Listing 6.9 An `@Async` method served by a different thread

```
@GetMapping("/bye")
@Async      #A
public void goodbye() {
  SecurityContext context = SecurityContextHolder.getContext();
  String username = context.getAuthentication().getName();

  // do something with the username
}
```

To enable the functionality of the `@Async` annotation, I have also created a configuration class and annotated it with `@EnableAsync`, as shown here:

```
@Configuration
@EnableAsync
public class ProjectConfig {
```

```
}
```

NOTE

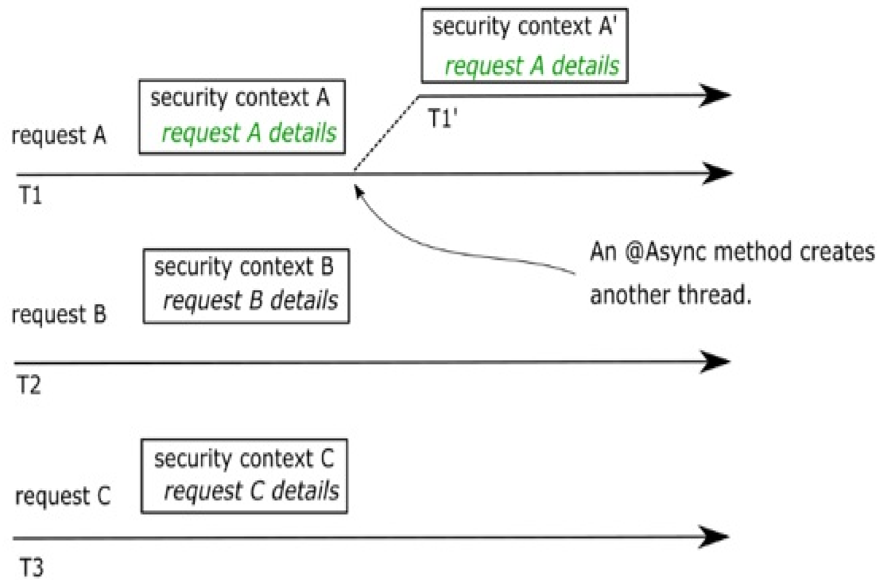
Sometimes in articles or forums, you find that the configuration annotations are placed over the main class. For example, you might find that certain examples use the `@EnableAsync` annotation directly over the main class. This approach is technically correct because we annotate the main class of a Spring Boot application with the `@SpringBootApplication` annotation, which includes the `@Configuration` characteristic. But in a real-world application, we prefer to keep the responsibilities apart, and we never use the main class as a configuration class. To make things as clear as possible for the examples in this book, I prefer to keep these annotations over the `@Configuration` class, similar to how you'll find them in practical scenarios.

If you try the code as it is now, it throws a `NullPointerException` on the line that gets the name from the authentication, which is

```
String username = context.getAuthentication().getName()
```

This is because the method executes now on another thread that does not inherit the security context. For this reason, the `Authorization` object is null and, in the context of the presented code, causes a `NullPointerException`. In this case, you could solve the problem by using the `MODE_INHERITABLETHREADLOCAL` strategy. This can be set either by calling the `SecurityContextHolder.setStrategyName()` method or by using the system property `spring.security.strategy`. By setting this strategy, the framework knows to copy the details of the original thread of the request to the newly created thread of the asynchronous method (figure 6.9).

Figure 6.9 When using the `MODE_INHERITABLETHREADLOCAL`, the framework copies the security context details from the original thread of the request to the security context of the new thread.



The next listing presents a way to set the security context management strategy by calling the `setStrategyName()` method.

Listing 6.10 Using `InitializingBean` to set `SecurityContextHolder` mode

```
@Configuration
@EnableAsync
public class ProjectConfig {

  @Bean
  public InitializingBean initializingBean() {
    return () -> SecurityContextHolder.setStrategyName(
      SecurityContextHolder.MODE_INHERITABLETHREADLOCAL);
  }
}
```

Calling the endpoint, you will observe now that the security context is propagated correctly to the next thread by Spring. Additionally,

`Authentication` is not null anymore.

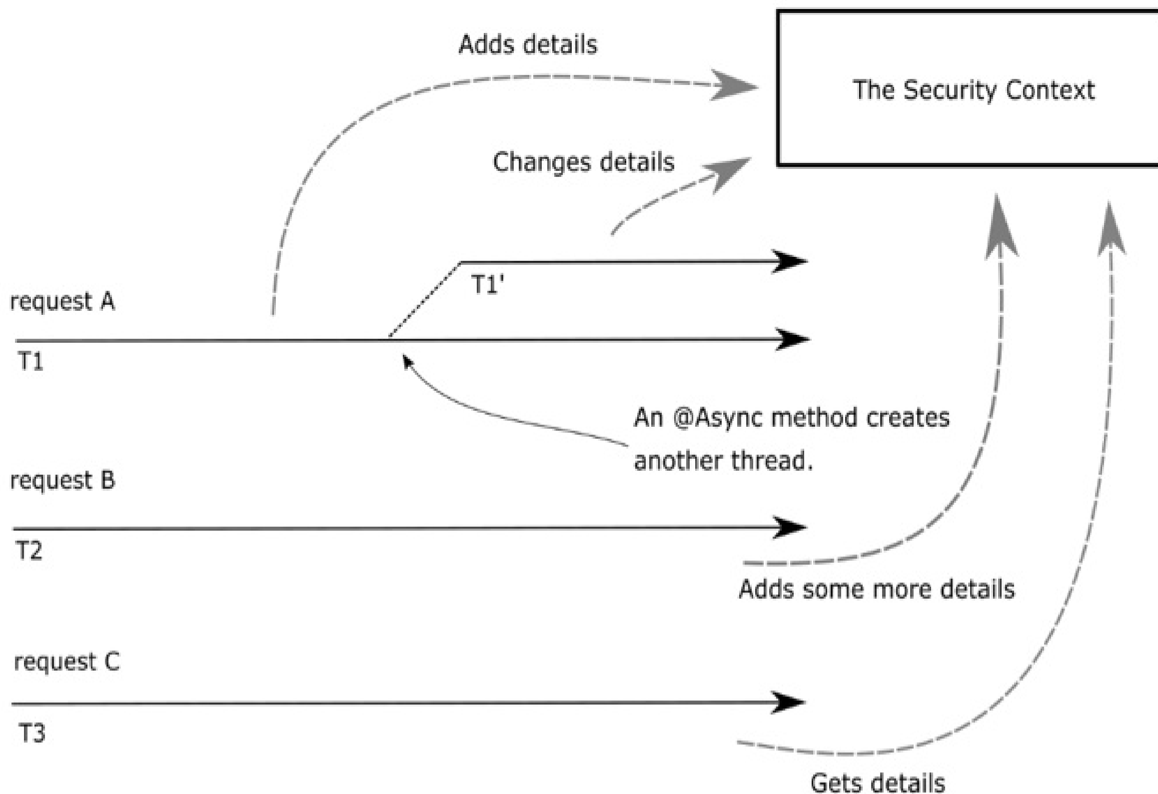
NOTE

This works, however, only when the framework itself creates the thread (for example, in case of an `@Async` method). If your code creates the thread, you will run into the same problem even with the `MODE_INHERITABLETHREADLOCAL` strategy. This happens because, in this case, the framework does not know about the thread that your code creates. We’ll discuss how to solve the issues of these cases in sections 6.2.4 and 6.2.5.

6.2.3 Using a holding strategy for standalone applications

If what you need is a security context shared by all the threads of the application, you change the strategy to `MODE_GLOBAL` (figure 6.10). You would not use this strategy for a web server as it doesn’t fit the general picture of the application. A backend web application independently manages the requests it receives, so it really makes more sense to have the security context separated per request instead of one context for all of them. But this can be a good use for a standalone application.

Figure 6.10 With `MODE_GLOBAL` used as the security context management strategy, all the threads access the same security context. This implies that these all have access to the same data and can change that information. Because of this, race conditions can occur, and you have to take care of synchronization.



As the following code snippet shows, you can change the strategy in the same way we did with `MODE_INHERITABLETHREADLOCAL`. You can use the method `SecurityContextHolder.setStrategyName()` or the system property `spring.security.strategy`:

```
@Bean
public InitializingBean initializingBean() {
  return () -> SecurityContextHolder.setStrategyName(
    SecurityContextHolder.MODE_GLOBAL);
}
```

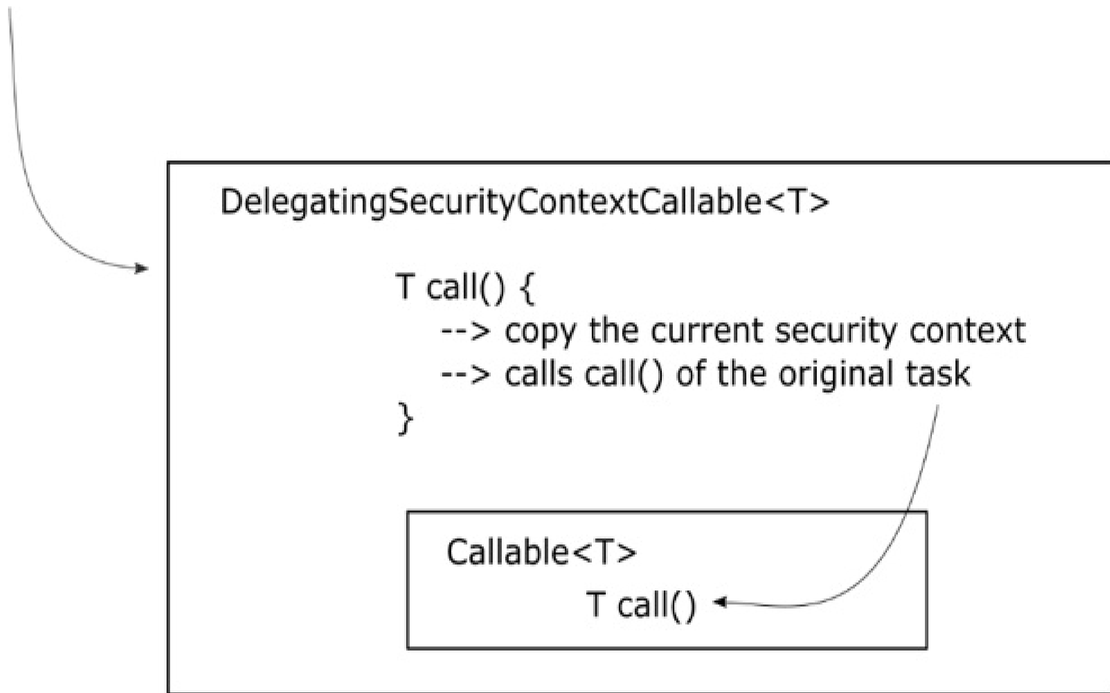
Also, be aware that the `SecurityContext` is not thread safe. So, with this strategy where all the threads of the application can access the `SecurityContext` object, you need to take care of concurrent access.

6.2.4 Forwarding the security context with DelegatingSecurityContextRunnable

You have learned that you can manage the security context with three modes provided by Spring Security: `MODE_THREADLOCAL`, `MODE_INHERITEDTHREADLOCAL`, and `MODE_GLOBAL`. By default, the framework only makes sure to provide a security context for the thread of the request, and this security context is only accessible to that thread. But the framework doesn't take care of newly created threads (for example, in case of an asynchronous method). And you learned that for this situation, you have to explicitly set a different mode for the management of the security context. But we still have a singularity: what happens when your code starts new threads without the framework knowing about them? Sometimes we name these "charitalics"self-managed threads because it is we who manage them, not the framework. In this section, we apply some utility tools provided by Spring Security that help you propagate the security context to newly created threads.

No specific strategy of the `SecurityContextHolder` offers you a solution to self-managed threads. In this case, you need to take care of the security context propagation. One solution for this is to use the `DelegatingSecurityContextRunnable` to decorate the tasks you want to execute on a separate thread. The `DelegatingSecurityContextRunnable` extends `Runnable`. You can use it following the execution of the task when there is no value expected. If you have a return value, then you can use the `Callable<T>` alternative, which is `DelegatingSecurityContextCallable<T>`. Both classes represent tasks executed asynchronously, as any other `Runnable` or `Callable`. Moreover, these make sure to copy the current security context for the thread that executes the task. As figure 6.11 shows, these objects decorate the original tasks and copy the security context to the new threads.

Figure 6.11 `DelegatingSecurityContextCallable` is designed as a decorator of the `Callable` object. When building such an object, you provide the callable task that the application executes asynchronously. `DelegatingSecurityContextCallable` copies the details from the security context to the new thread and then executes the task.



Listing 6.11 presents the use of `DelegatingSecurityContextCallable`. Let's start by defining a simple endpoint method that declares a `Callable` object. The `Callable` task returns the username from the current security context.

Listing 6.11 Defining a `Callable` object and executing it as a task on a separate thread

```
@GetMapping("/ciao")
public String ciao() throws Exception {
  Callable<String> task = () -> {
     SecurityContext context = SecurityContextHolder.getContext()
     return context.getAuthentication().getName();
  };

  // Omitted code
}
```

We continue the example by submitting the task to an `ExecutorService`. The response of the execution is retrieved and returned as a response body by the endpoint.

Listing 6.12 Defining an `ExecutorService` and submitting the task

```
@GetMapping("/ciao")
public String ciao() throws Exception {
  Callable<String> task = () -> {
      SecurityContext context = SecurityContextHolder.getContext(
      return context.getAuthentication().getName();
  };

  ExecutorService e = Executors.newCachedThreadPool();
  try {
     return "Ciao, " + e.submit(task).get() + "!";
  } finally {
     e.shutdown();
  }
}
```

If you run the application as is, you get nothing more than a `NullPointerException`. Inside the newly created thread to run the callable task, the authentication does not exist anymore, and the security context is empty. To solve this problem, we decorate the task with `DelegatingSecurityContextCallable`, which provides the current context to the new thread, as provided by this listing.

Listing 6.13 Running the task decorated by `DelegatingSecurityContextCallable`

```
@GetMapping("/ciao")
public String ciao() throws Exception {
  Callable<String> task = () -> {
    SecurityContext context = SecurityContextHolder.getContext();
    return context.getAuthentication().getName();
  };

  ExecutorService e = Executors.newCachedThreadPool();
  try {
    var contextTask = new DelegatingSecurityContextCallable<>(tas
    return "Ciao, " + e.submit(contextTask).get() + "!";
  } finally {
    e.shutdown();
  }
}
```

Calling the endpoint now, you can observe that Spring propagated the security context to the thread in which the tasks execute:

```
curl -u user:2eb3f2e8-debd-420c-9680-48159b2ff905
[CA]http://localhost:8080/ciao
```

The response body for this call is

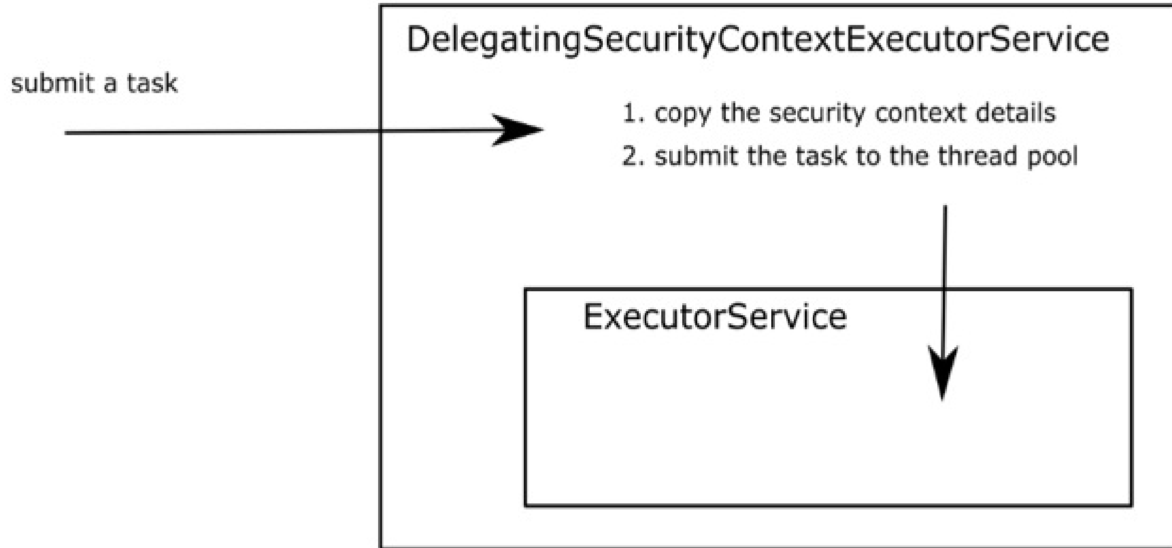
```
Ciao, user!
```

6.2.5 Forwarding the security context with DelegatingSecurityContextExecutorService

When dealing with threads that our code starts without letting the framework know about them, we have to manage propagation of the details from the security context to the next thread. In section 6.2.4, you applied a technique to copy the details from the security context by making use of the task itself. Spring Security provides some great utility classes like `DelegatingSecurityContextRunnable` and `DelegatingSecurityContextCallable`. These classes decorate the tasks you execute asynchronously and also take the responsibility to copy the details from security context such that your implementation can access those from the newly created thread. But we have a second option to deal with the security context propagation to a new thread, and this is to manage propagation from the thread pool instead of from the task itself. In this section, you learn how to apply this technique by using more great utility classes provided by Spring Security.

An alternative to decorating tasks is to use a particular type of `Executor`. In the next example, you can observe that the task remains a simple `Callable<T>`, but the thread still manages the security context. The propagation of the security context happens because an implementation called `DelegatingSecurityContextExecutorService` decorates the `ExecutorService`. The `DelegatingSecurityContextExecutorService` also takes care of the security context propagation, as presented in figure 6.12.

Figure 6.12 `DelegatingSecurityContextExecutorService` decorates an `ExecutorService` and propagates the security context details to the next thread before submitting the task.



The code in listing 6.14 shows how to use a `DelegatingSecurityContextExecutorService` to decorate an `ExecutorService` such that when you submit the task, it takes care to propagate the details of the security context.

Listing 6.14 Propagating the `SecurityContext`

```
@GetMapping("/hola")
public String hola() throws Exception {
  Callable<String> task = () -> {
    SecurityContext context = SecurityContextHolder.getContext();
    return context.getAuthentication().getName();
  };

  ExecutorService e = Executors.newCachedThreadPool();
  e = new DelegatingSecurityContextExecutorService(e);
  try {
    return "Hola, " + e.submit(task).get() + "!";
  } finally {
    e.shutdown();
  }
}
```

Call the endpoint to test that the `DelegatingSecurityContextExecutorService` correctly delegated the

security context:

```
curl -u user:5a5124cc-060d-40b1-8aad-753d3da28dca http://localhos
```

The response body for this call is

```
Hola, user!
```

Note

Of the classes that are related to concurrency support for the security context, I recommend you be aware of the ones presented in table 6.1.

Spring offers various implementations of the utility classes that you can use in your application to manage the security context when creating your own threads. In section 6.2.4, you implemented `DelegatingSecurityContextCallable`. In this section, we use `DelegatingSecurityContextExecutorService`. If you need to implement security context propagation for a scheduled task, then you will be happy to hear that Spring Security also offers you a decorator named `DelegatingSecurityContextScheduledExecutorService`. This mechanism is similar to the `DelegatingSecurityContextExecutorService` that we presented in this section, with the difference that it decorates a `ScheduledExecutorService`, allowing you to work with scheduled tasks.

Additionally, for more flexibility, Spring Security offers you a more abstract version of a decorator called `DelegatingSecurityContextExecutor`. This class directly decorates an `Executor`, which is the most abstract contract of this hierarchy of thread pools. You can choose it for the design of your application when you want to be able to replace the implementation of the thread pool with any of the choices the language provides you.

Table 6.1 Objects responsible for delegating the security context to a separate thread

Class	Description

`DelegatingSecurityContextExecutor`	Implements the `Ex` interface and is de decorate an `Execu` with the capability forwarding the sec context to the thre created by its poo
`DelegatingSecurityContextExecutorService`	Implements the `ExecutorService` and is designed to an `ExecutorServ` with the capability forwarding the sec context to the thre created by its poo
`DelegatingSecurityContextScheduledExecutorService`	Implements the `ScheduledExecut` interface and is de decorate a `ScheduledExecut` object with the ca forwarding the sec context to the thre created by its poo
`DelegatingSecurityContextRunnable`	Implements the `Ru` interface and repr task that is execut different thread w returning a respon a normal `Runnabl` able to propagate context to use on

	thread.
DelegatingSecurityContextCallable	Implements the `Ca` interface and repr task that is execut different thread ar eventually return Above a normal `C` it is also able to p security context to the new thread.

6.3 Understanding HTTP Basic and form-based login authentications

Up to now, we've only used HTTP Basic as the authentication method, but throughout this book, you'll learn that there are other possibilities as well. The HTTP Basic authentication method is simple, which makes it an excellent choice for examples and demonstration purposes or proof of concept. But for the same reason, it might not fit all of the real-world scenarios that you'll need to implement.

In this section, you learn more configurations related to HTTP Basic. As well, we discover a new authentication method called `formLogin`. For the rest of this book, we'll discuss other methods for authentication, which match well with different kinds of architectures. We'll compare these such that you understand the best practices as well as the anti-patterns for authentication.

6.3.1 Using and configuring HTTP Basic

You are aware that HTTP Basic is the default authentication method, and we have observed the way it works in various examples in chapter 3. In this section, we add more details regarding the configuration of this authentication method.

For theoretical scenarios, the defaults that HTTP Basic authentication comes with are great. But in a more complex application, you might find the need to customize some of these settings. For example, you might want to implement a specific logic for the case in which the authentication process fails. You might even need to set some values on the response sent back to the client in this case. So let's consider these cases with practical examples to understand how you can implement this. I want to point out again how you can set this method explicitly, as shown in the following listing. You can find this example in the project ssia-ch6-ex3.

Listing 6.15 Setting the HTTP Basic authentication method

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

     http.httpBasic(Customizer.withDefaults());

     return http.build();
  }
}
```

You can call the `httpBasic()` method of the `HttpSecurity` instance with a parameter of type `Customizer`. This parameter allows you to set up some configurations related to the authentication method, for example, the realm name, as shown in listing 6.16. You can think about the realm as a protection space that uses a specific authentication method. For a complete description, refer to RFC 2617 at https://tools.ietf.org/html/rfc2617.

Listing 6.16 Configuring the realm name for the response of failed authentications

```
@Bean
public SecurityFilterChain configure(HttpSecurity http)
  throws Exception {

  http.httpBasic(c -> {
    c.realmName("OTHER");
    c.authenticationEntryPoint(new CustomEntryPoint());
  });
```

```
  http.authorizeHttpRequests(c -> c.anyRequest().authenticated())

  return http.build();
}
```

Listing 6.16 presents an example of changing the realm name. The lambda expression used is, in fact, an object of type `Customizer<HttpBasicConfigurer<HttpSecurity>>`. The parameter of type `HttpBasicConfigurer<HttpSecurity>` allows us to call the `realmName()` method to rename the realm. You can use cURL with the `-v` flag to get a verbose HTTP response in which the realm name is indeed changed. However, note that you'll find the `WWW-Authenticate` header in the response only when the HTTP response status is 401 Unauthorized and not when the HTTP response status is 200 OK. Here's the call to cURL:

```
curl -v http://localhost:8080/hello
```

The response of the call is

```
/
...
< WWW-Authenticate: Basic realm="OTHER"
...
```

Also, by using a `Customizer`, we can customize the response for a failed authentication. You need to do this if the client of your system expects something specific in the response in the case of a failed authentication. You might need to add or remove one or more headers. Or you can have some logic that filters the body to make sure that the application doesn't expose any sensitive data to the client.

NOTE

Always exercise caution about the data that you expose outside of the system. One of the most common mistakes (which is also part of the OWASP top ten vulnerabilities - https://owasp.org/www-project-top-ten/) is exposing sensitive data. Working with the details that the application sends to the client for a failed authentication is always a point of risk for revealing confidential information.

To customize the response for a failed authentication, we can implement an `AuthenticationEntryPoint`. Its `commence()` method receives the `HttpServletRequest`, the `HttpServletResponse`, and the `AuthenticationException` that cause the authentication to fail. Listing 6.17 demonstrates a way to implement the `AuthenticationEntryPoint`, which adds a header to the response and sets the HTTP status to 401 Unauthorized.

NOTE

It's a little bit ambiguous that the name of the `AuthenticationEntryPoint` interface doesn't reflect its usage on authentication failure. In the Spring Security architecture, this is used directly by a component called `ExceptionTranslationManager`, which handles any `AccessDeniedException` and `AuthenticationException` thrown within the filter chain. You can view the `ExceptionTranslationManager` as a bridge between Java exceptions and HTTP responses.

Listing 6.17 Implementing an `AuthenticationEntryPoint`

```
public class CustomEntryPoint
  implements AuthenticationEntryPoint {

  @Override
  public void commence(
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse,
    AuthenticationException e)
      throws IOException, ServletException {

      httpServletResponse
        .addHeader("message", "Luke, I am your father!");
      httpServletResponse
        .sendError(HttpStatus.UNAUTHORIZED.value());

    }
}
```

You can then register the `CustomEntryPoint` with the HTTP Basic method in the configuration class. The following listing presents the configuration class for the custom entry point.

Listing 6.18 Setting the custom `AuthenticationEntryPoint`

```
@Bean
public SecurityFilterChain configure(HttpSecurity http)
  throws Exception {

  http.httpBasic(c -> {
    c.realmName("OTHER");
    c.authenticationEntryPoint(new CustomEntryPoint());
  });


  http.authorizeHttpRequests().anyRequest().authenticated();

  return http.build();
}
```

If you now make a call to an endpoint such that the authentication fails, you should find in the response the newly added header:

```
curl -v http://localhost:8080/hello
```

The response of the call is

```
...
< HTTP/1.1 401
< Set-Cookie: JSESSIONID=459BAFA7E0E6246A463AD19B07569C7B; Path=/
< message: Luke, I am your father!
...
```

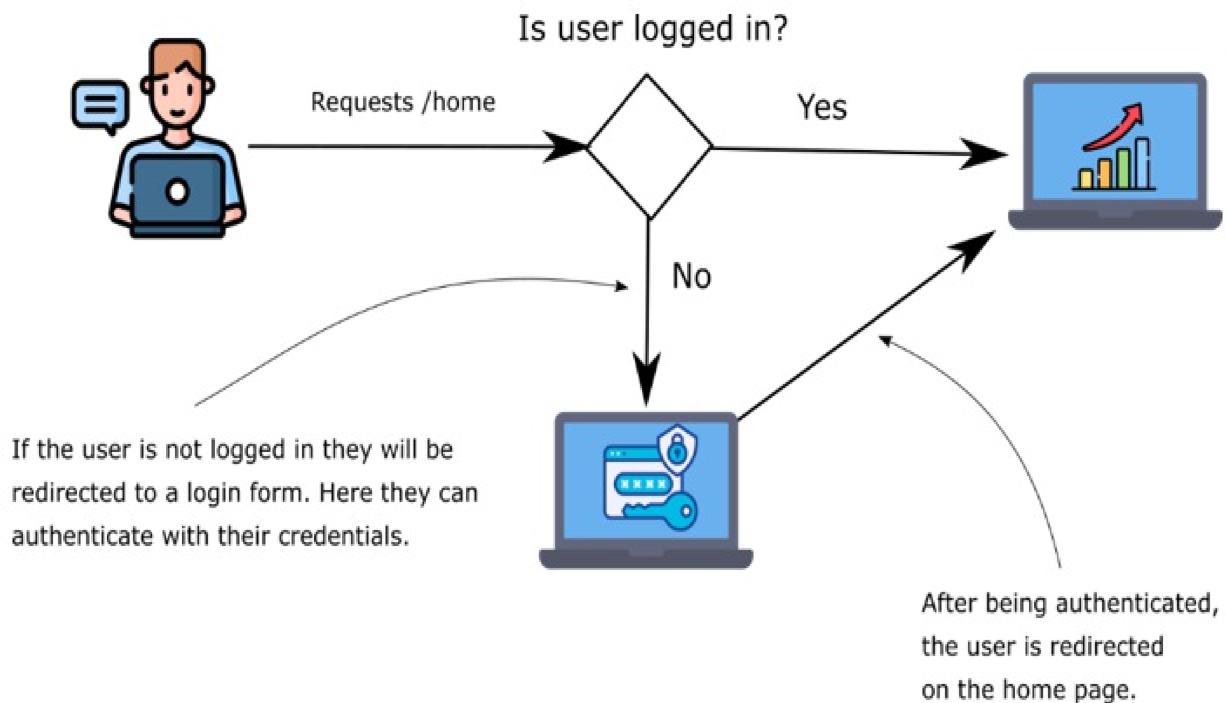
6.3.2 Implementing authentication with form-based login

When developing a web application, you would probably like to present a user-friendly login form where the users can input their credentials. As well, you might like your authenticated users to be able to surf through the web pages after they logged in and to be able to log out. For a small web application, you can take advantage of the form-based login method. In this section, you learn to apply and configure this authentication method for your application. To achieve this, we write a small web application that uses form-based login. Figure 6.13 describes the flow we'll implement. The examples in this section are part of the project ssia-ch6-ex4.

NOTE

I link this method to a small web application because, this way, we use a server-side session for managing the security context. For larger applications that require horizontal scalability, using a server-side session for managing the security context is undesirable. We'll discuss these aspects in more detail in chapters 12 through 15 when dealing with OAuth 2.

Figure 6.13 Using form-based login. An unauthenticated user is redirected to a form where they can use their credentials to authenticate. Once the application authenticates them, they are redirected to the homepage of the application.



To change the authentication method to form-based login, using the `HttpSecurity` object of the `SecurityFilterChain` bean, instead of `httpBasic()`, call the `formLogin()` method of the `HttpSecurity` parameter. The following listing presents this change.

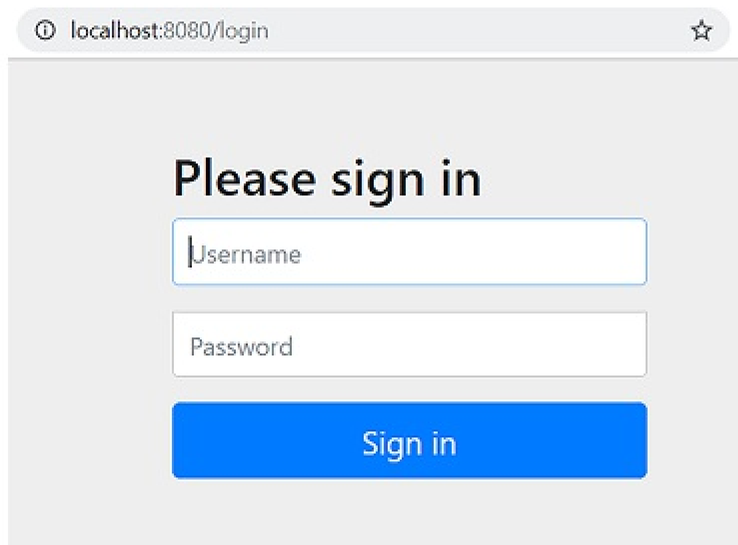
Listing 6.19 Changing the authentication method to a form-based login

```
@Configuration
public class ProjectConfig {
```

```
  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.formLogin(Customizer.withDefaults());

    http.authorizeHttpRequests(c -> c.anyRequest().authenticated(

    return http.build();
  }
}
```

Even with this minimal configuration, Spring Security has already configured a login form, as well as a log-out page for your project. Starting the application and accessing it with the browser should redirect you to a login page (figure 6.14).

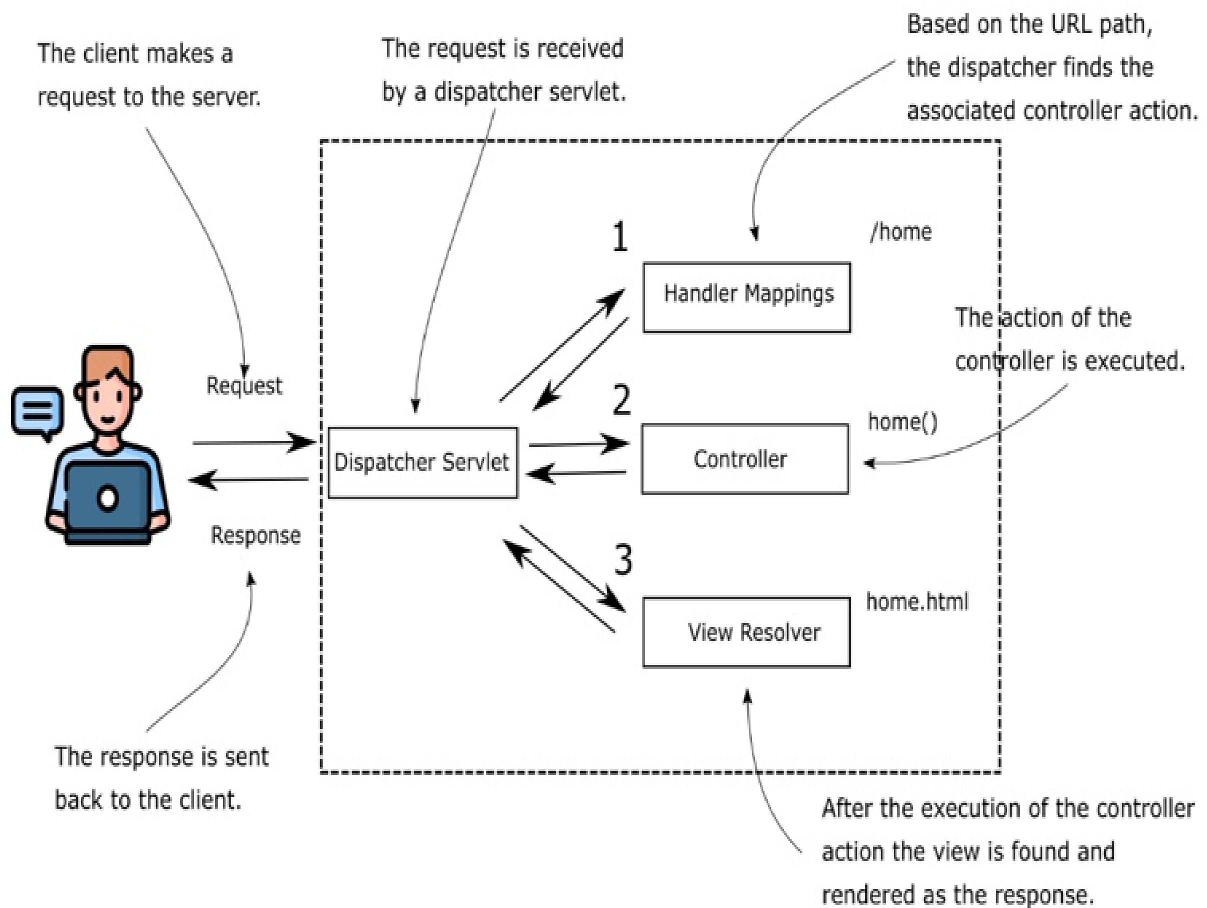
Figure 6.14 The default login page auto-configured by Spring Security when using the `formLogin()` method.



You can log in using the default provided credentials as long as you do not register your `UserDetailsService`. These are, as we learned in chapter 2, username “user” and a UUID password that is printed in the console when the application starts. After a successful login, because there is no other page defined, you are redirected to a default error page. The application relies on the same architecture for authentication that we encountered in previous

examples. So, like figure 6.14 shows, you need to implement a controller for the homepage of the application. The difference is that instead of having a simple JSON-formatted response, we want the endpoint to return HTML that can be interpreted by the browser as our web page. Because of this, we choose to stick to the Spring MVC flow and have the view rendered from a file after the execution of the action defined in the controller. Figure 6.15 presents the Spring MVC flow for rendering the homepage of the application.

Figure 6.15 A simple representation of the Spring MVC flow. The dispatcher finds the controller action associated with the given path, /home, in this case. After executing the controller action, the view is rendered, and the response is sent back to the client.



To add a simple page to the application, you first have to create an HTML file in the resources/static folder of the project. I call this file home.html. Inside it, type some text that you will be able to find afterward in the browser. You can just add a heading (for example, `<h1>Welcome</h1>`). After creating

the HTML page, a controller needs to define the mapping from the path to the view. The following listing presents the definition of the action method for the home.html page in the controller class.

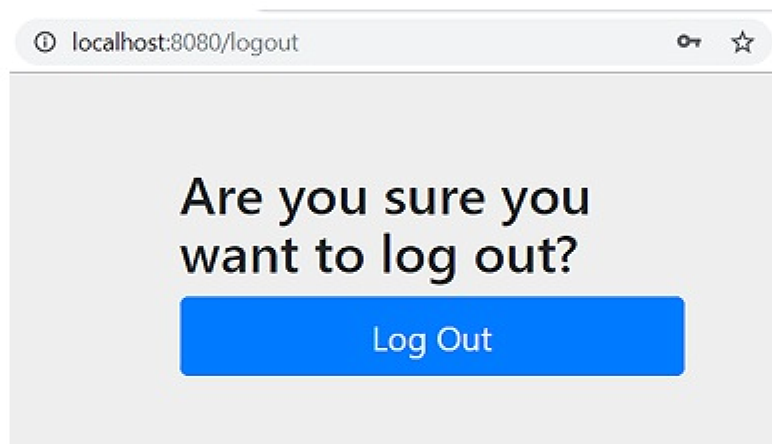
Listing 6.20 Defining the action method of the controller for the home.html page

```
@Controller
public class HelloController {

  @GetMapping("/home")
  public String home() {
    return "home.html";
  }
}
```

Mind that it is not a `@RestController` but a simple `@Controller`. Because of this, Spring does not send the value returned by the method in the HTTP response. Instead, it finds and renders the view with the name home.html.

Trying to access the /home path now, you are first asked if you want to log in. After a successful login, you are redirected to the homepage, where the welcome message appears. You can now access the /logout path, and this should redirect you to a log-out page (figure 6.16).

Figure 6.16 The log-out page configured by Spring Security for the form-based login authentication method.



After attempting to access a path without being logged in, the user is

automatically redirected to the login page. After a successful login, the application redirects the user back to the path they tried to originally access. If that path does not exist, the application displays a default error page. The `formLogin()` method returns an object of type `FormLoginConfigurer<HttpSecurity>`, which allows us to work on customizations. For example, you can do this by calling the `defaultSuccessUrl()`method, as shown in the following listing.

Listing 6.21 Setting a default success URL for the login form

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
  throws Exception {

  http.formLogin(c -> c.defaultSuccessUrl("/home", true));

  http.authorizeHttpRequests(c -> c.anyRequest().authenticated())

  return http.build();
}
```

If you need to go even more in depth with this, using the `AuthenticationSuccessHandler` and `AuthenticationFailureHandler` objects offers a more detailed customization approach. These interfaces let you implement an object through which you can apply the logic executed for authentication. If you want to customize the logic for successful authentication, you can define an `AuthenticationSuccessHandler`. The `onAuthenticationSuccess()` method receives the servlet request, servlet response, and the `Authentication` object as parameters. In listing 6.22, you'll find an example of implementing the `onAuthenticationSuccess()`method to make different redirects depending on the granted authorities of the logged-in user.

Listing 6.22 Implementing an `AuthenticationSuccessHandler`

```
@Component
public class CustomAuthenticationSuccessHandler
  implements AuthenticationSuccessHandler {

  @Override
  public void onAuthenticationSuccess(
```

```
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse,
    Authentication authentication)
      throws IOException {

      var authorities = authentication.getAuthorities();

      var auth =
              authorities.stream()
                .filter(a -> a.getAuthority().equals("read"))
                .findFirst();      #A

      if (auth.isPresent()) {      #B
        httpServletResponse
          .sendRedirect("/home");
      } else {
        httpServletResponse
          .sendRedirect("/error");
      }
   }
}
```

There are situations in practical scenarios when a client expects a certain format of the response in case of failed authentication. They may expect a different HTTP status code than 401 Unauthorized or additional information in the body of the response. The most typical case I have found in applications is to send a "charitalics"request identifier. This request identifier has a unique value used to trace back the request among multiple systems, and the application can send it in the body of the response in case of failed authentication. Another situation is when you want to sanitize the response to make sure that the application doesn't expose sensitive data outside of the system. You might want to define custom logic for failed authentication simply by logging the event for further investigation.

If you would like to customize the logic that the application executes when authentication fails, you can do this similarly with an `AuthenticationFailureHandler` implementation. For example, if you want to add a specific header for any failed authentication, you could do something like that shown in listing 6.23. You could, of course, implement any logic here as well. For the `AuthenticationFailureHandler`, `onAuthenticationFailure()` receives the request, response, and the `Authentication` object.

Listing 6.23 Implementing an `AuthenticationFailureHandler`

```
@Component
public class CustomAuthenticationFailureHandler
  implements AuthenticationFailureHandler {

  @Override
  public void onAuthenticationFailure(
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse,
    AuthenticationException e)  {

    try {

      httpServletResponse.setHeader("failed",
         LocalDateTime.now().toString());
      httpServletResponse.sendRedirect("/error");

    } catch (IOException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

To use the two objects, you need to register them in the `configure()` method on the `FormLoginConfigurer` object returned by the `formLogin()` method. The following listing shows how to do this.

Listing 6.24 Registering the handler objects in the configuration class

```
@Configuration
public class ProjectConfig {

  private final CustomAuthenticationSuccessHandler
[CA]authenticationSuccessHandler;
  private final CustomAuthenticationFailureHandler
[CA]authenticationFailureHandler;

// Omitted constructor

@Bean
public UserDetailsService uds() {
  var uds = new InMemoryUserDetailsManager();

  uds.createUser(
```

```
      User.withDefaultPasswordEncoder()
           .username("john")
           .password("12345")
           .authorities("read")
           .build()
  );

  uds.createUser(
      User.withDefaultPasswordEncoder()
            .username("bill")
            .password("12345")
            .authorities("write")
            .build()
  );

  return uds;
}


@Bean
public SecurityFilterChain configure(HttpSecurity http)
  throws Exception {

   http.formLogin(c ->
     c.successHandler(authenticationSuccessHandler)
      .failureHandler(authenticationFailureHandler)
   );

   http.authorizeHttpRequests(c -> c.anyRequest().authenticated()

   return http.build();
  }
}
```

For now, if you try to access the /home path using HTTP Basic with the proper username and password, you are returned a response with the status HTTP 302 Found. This response status code is how the application tells you that it is trying to do a redirect. Even if you have provided the right username and password, it won't consider these and will instead try to send you to the login form as requested by the `formLogin` method. You can, however, change the configuration to support both the HTTP Basic and the form-based login methods, as in the following listing.

Listing 6.25 Using form-based login and HTTP Basic together

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
  throws Exception {

  http.formLogin(c ->
     c.successHandler(authenticationSuccessHandler)
      .failureHandler(authenticationFailureHandler)
  );

  http.httpBasic(Customizer.withDefaults());

  http.authorizeHttpRequests(c -> c.anyRequest().authenticated())

  return http.build();
}
```

Accessing the /home path now works with both the form-based login and HTTP Basic authentication methods:

```
curl -u user:cdd430f6-8ebc-49a6-9769-b0f3ce571d19
[CA]http://localhost:8080/home
```

The response of the call is

```
<h1>Welcome</h1>
```

6.4 Summary

- The `AuthenticationProvider` is the component that allows you to implement custom authentication logic.
- When you implement custom authentication logic, it’s a good practice to keep the responsibilities decoupled. For user management, the `AuthenticationProvider` delegates to a `UserDetailsService`, and for the responsibility of password validation, the `AuthenticationProvider` delegates to a `PasswordEncoder`.
- The `SecurityContext` keeps details about the authenticated entity after successful authentication.
- You can use three strategies to manage the security context: `MODE_THREADLOCAL`, `MODE_INHERITABLETHREADLOCAL`, and `MODE_GLOBAL`. Access from different threads to the security context details works differently depending on the mode you choose.

- Remember that when using the shared-thread local mode, it’s only applied for threads that are managed by Spring. The framework won’t copy the security context for the threads that are not governed by it.
- Spring Security offers you great utility classes to manage the threads created by your code, about which the framework is now aware. To manage the `SecurityContext` for the threads that you create, you can use
  - `DelegatingSecurityContextRunnable`
  - `DelegatingSecurityContextCallable`
  - `DelegatingSecurityContextExecutor`
- Spring Security autoconfigures a form for login and an option to log out with the form-based login authentication method, `formLogin()`. It is straightforward to use when developing small web applications.
- The `formLogin` authentication method is highly customizable. Moreover, you can use this type of authentication together with the HTTP Basic method.

7 Configuring endpoint-level authorization: Restricting access

This chapter covers

- Defining authorities and roles
- Applying authorization rules on endpoints

Some years ago, I was skiing in the beautiful Carpathian mountains when I witnessed this funny scene. About ten, maybe fifteen people were queuing up to get into the cabin to go to the top of the ski slope. A well-known pop artist showed up, accompanied by two bodyguards. He confidently strode up, expecting to skip the queue because he was famous. Reaching the head of the line, he got a surprise. “Ticket, please!” said the person managing the boarding, who then had to explain, “Well, you first need a ticket, and second, there is no priority line for this boarding, sorry. The queue ends there.” He pointed to the end of the queue. As in most cases in life, it doesn’t matter who you are. We can say the same about software applications. It doesn’t matter who you are when trying to access a specific functionality or data!

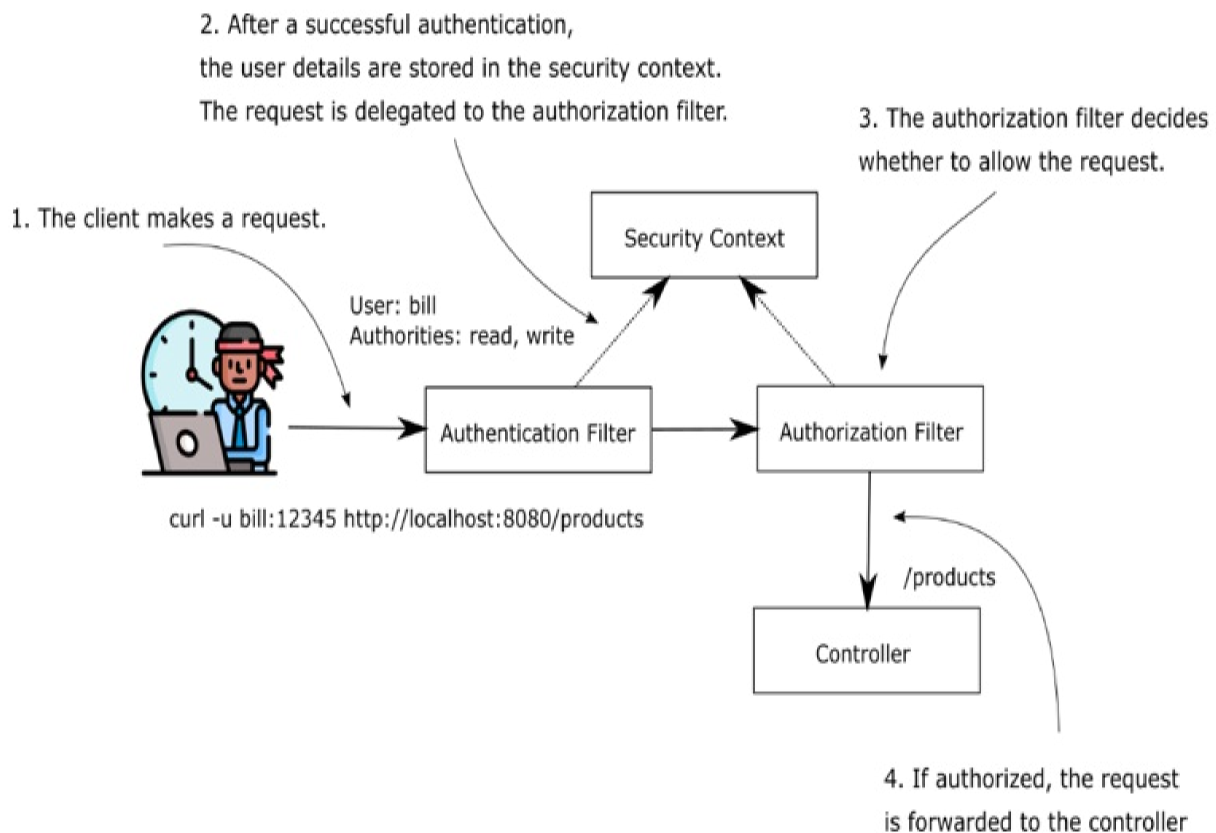
Up to now, we’ve only discussed authentication, which is, as you learned, the process in which the application identifies the caller of a resource. In the examples we worked on in the previous chapters, we didn’t implement any rule to decide whether to approve a request. We only cared if the system knew the user or not. In most applications, it doesn’t happen that all the users identified by the system can access every resource in the system. In this chapter, we’ll discuss authorization. "charitalics"Authorization is the process during which the system decides if an identified client has permission to access the requested resource (figure 7.1).

Figure 7.1 Authorization is the process during which the application decides whether an authenticated entity is allowed to access a resource. Authorization always happens after authentication.

In Spring Security, once the application ends the authentication flow, it delegates the request to an authorization filter. The filter allows or rejects the request based on the configured authorization rules (figure 7.2).

Figure 7.2 When the client makes the request, the authentication filter authenticates the user. After successful authentication, the authentication filter stores the user details in the security context and forwards the request to the authorization filter. The authorization filter decides

whether the call is permitted. To decide whether to authorize the request, the authorization filter uses the details from the security context.



To cover all the essential details on authorization, in this chapter we'll follow these steps:

1. Gain an understanding of what an authority is and apply access rules on all endpoints based on a user's authorities.
2. Learn how to group authorities in roles and how to apply authorization rules based on a user's roles.

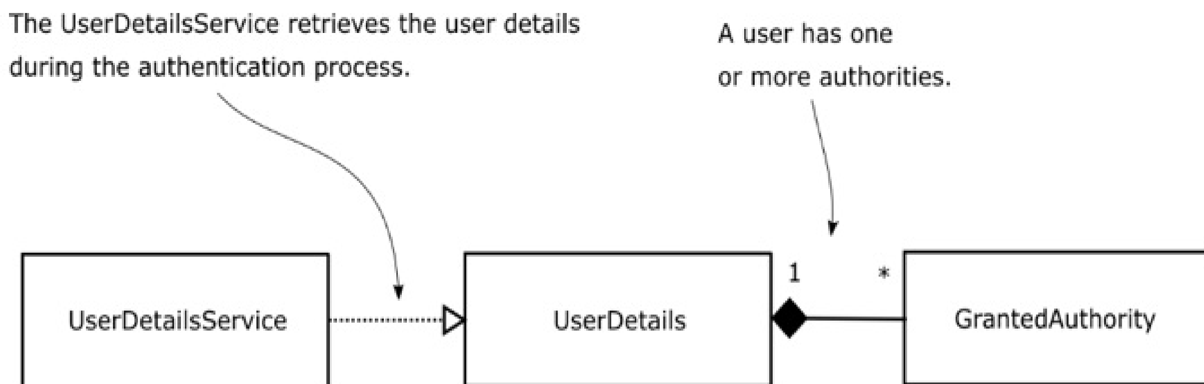
In chapter 8, we'll continue with selecting endpoints to which we'll apply the authorization rules. For now, let's look at authorities and roles and how these can restrict access to our applications.

7.1 Restricting access based on authorities and roles

In this section, you learn about the concepts of authorization and roles. You use these to secure all the endpoints of your application. You need to understand these concepts before you can apply them in real-world scenarios, where different users have different permissions. Based on what privileges users have, they can only execute a specific action. The application provides privileges as authorities and roles.

In chapter 3, you implemented the `GrantedAuthority` interface. I introduced this contract when discussing another essential component: the `UserDetails` interface. We didn't work with `GrantedAuthority` then because, as you'll learn in this chapter, this interface is mainly related to the authorization process. We can now return to `GrantedAuthority` to examine its purpose. Figure 7.3 presents the relationship between the `UserDetails` contract and the `GrantedAuthority` interface. Once we finish discussing this contract, you'll learn how to use these rules individually or for specific requests.

Figure 7.3 A user has one or more authorities (actions that a user can do). During the authentication process, the `UserDetailsService` obtains all the details about the user, including the authorities. The application uses the authorities as represented by the `GrantedAuthority` interface for authorization after it successfully authenticates the user.



Listing 7.1 shows the definition of the `GrantedAuthority` contract. An "charitalics"authority is an action that a user can perform with a system resource. An authority has a name that the `getAuthority()` behavior of the object returns as a `String`. We use the name of the authority when defining the custom authorization rule. Often an authorization rule can look like this: "Jane is allowed to "charitalics"delete the product records," or "John is allowed to "charitalics"read the document records." In these cases,

"charitalics"delete and "charitalics"read are the granted authorities. The application allows the users Jane and John to perform these actions, which often have names like read, write, or delete.

Listing 7.1 The `GrantedAuthority` contract

```
public interface GrantedAuthority extends Serializable {
  String getAuthority();
}
```

The `UserDetails`, which is the contract describing the user in Spring Security, has a collection of `GrantedAuthority` instances as presented in figure 7.3. You can allow a user one or more privileges. The `getAuthorities()` method returns the collection of `GrantedAuthority` instances. In listing 7.2, you can review this method in the `UserDetails` contract. We implement this method so that it returns all the authorities granted for the user. After authentication ends, the authorities are part of the details about the user that logged in, which the application can use to grant permissions.

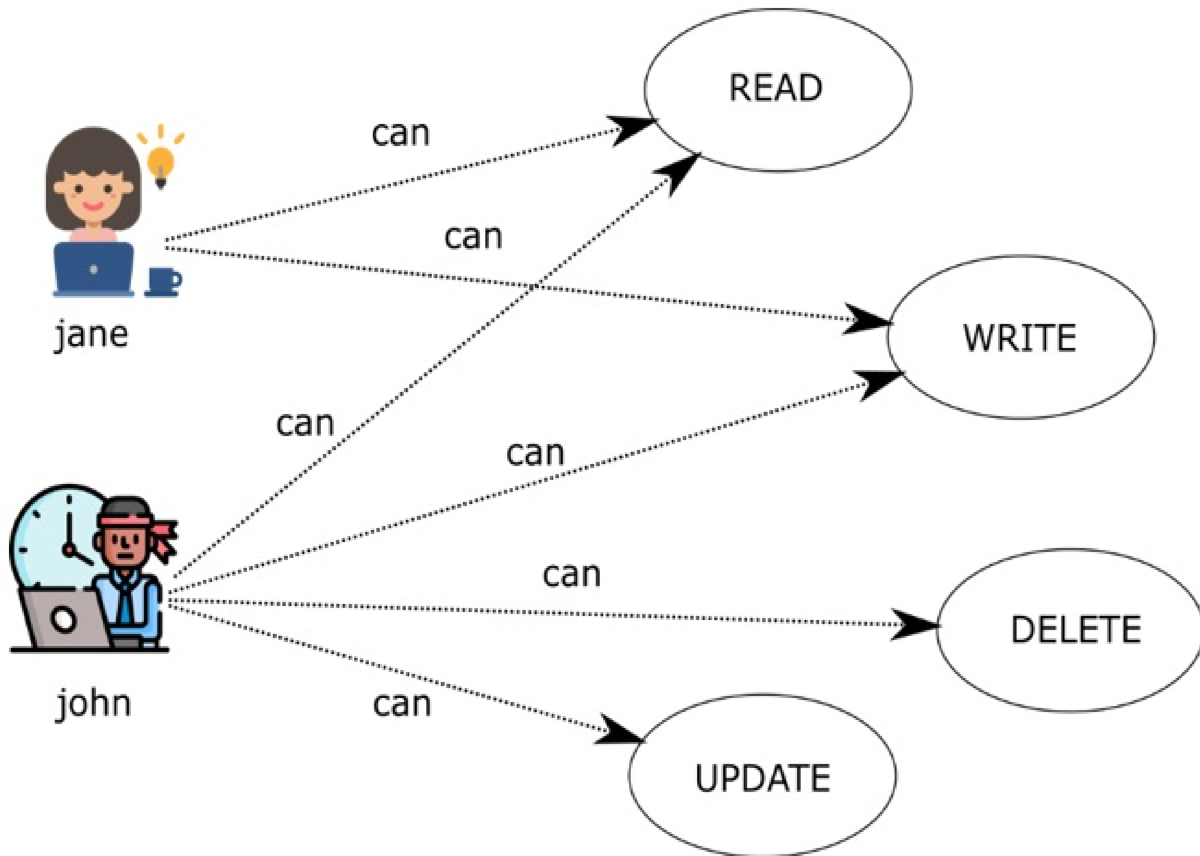
Listing 7.2 The `getAuthorities()` method from the `UserDetails` contract

```
public interface UserDetails extends Serializable {
  Collection<? extends GrantedAuthority> getAuthorities();

  // Omitted code
}
```

7.1.1 Restricting access for all endpoints based on user authorities

In this section, we discuss limiting access to endpoints for specific users. Up to now in our examples, any authenticated user could call any endpoint of the application. From now on, you'll learn to customize this access. In the apps you find in production, you can call some of the endpoints of the application even if you are unauthenticated, while for others, you need special privileges (figure 7.4). We'll write several examples so that you learn various ways in which you can apply these restrictions with Spring Security.

Figure 7.4 Authorities are actions that users can perform in the application. Based on these actions, you implement the authorization rules. Only users having specific authorities can make a particular request to an endpoint. For example, Jane can only read and write to the endpoint, while John can read, write, delete, and update the endpoint.



Now that you remember the `UserDetails` and `GrantedAuthority` contracts and the relationship between them, it is time to write a small app that applies an authorization rule. With this example, you learn a few alternatives to configure access to endpoints based on the user's authorities. We start a new project that I name ssia-ch7-ex1. I show you three ways in which you can configure access as mentioned using these methods:

- `hasAuthority()`—Receives as parameters only one authority for which the application configures the restrictions. Only users having that authority can call the endpoint.
- `hasAnyAuthority()`—Can receive more than one authority for which the application configures the restrictions. I remember this method as "has any of the given authorities." The user must have at least one of the

specified authorities to make a request.
I recommend using this method or the `hasAuthority()` method for their simplicity, depending on the number of privileges you assign the users. These are simple to read in configurations and make your code easier to understand.

- `access()`—Offers you unlimited possibilities for configuring access because the application builds the authorization rules based on a custom object named `AuthorizationManager` you implement. You can provide any implementation for the `AuthorizationManager` contract depending on your case. Spring Security provides a few implementations as well. The most common implementation is the `WebExpressionAuthorizationManager` which helps you apply authorization rules based on Spring Expression Language (SpEL). But using the `access()` method can make the authorization rules more difficult to read and understand. For this reason, I recommend it as the lesser solution and only if you cannot apply the `hasAnyAuthority()` or `hasAuthority()` methods.

The only dependencies needed in your pom.xml file are `spring-boot-starter-web` and `spring-boot-starter-security`. These dependencies are enough to approach all three solutions previously enumerated. You can find this example in the project ssia-ch7-ex1

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

We also add an endpoint in the application to test our authorization configuration:

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
```

```
  }
}
```

In a configuration class, we declare an `InMemoryUserDetailsManager` as our `UserDetailsService` and add two users, John and Jane, to be managed by this instance. Each user has a different authority. You can see how to do this in the following listing.

Listing 7.3 Declaring the `UserDetailsService` and assigning users

```
@Configuration
public class ProjectConfig {

  @Bean     #A
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();    #B

    var user1 = User.withUsername("john")      #C
                    .password("12345")
                    .authorities("READ")
                    .build();

    var user2 = User.withUsername("jane")      #D
                    .password("12345")
                    .authorities("WRITE")
                    .build();

    manager.createUser(user1);      #E
    manager.createUser(user2);

    return manager;
  }

  @Bean    #F
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

The next thing we do is add the authorization configuration. In chapter 2 when we worked on the first example, you saw how we could make all the endpoints accessible for everyone. To do that, you created a `SecurityFilterChain` bean in the app’s context, similar to what you see in the next listing.

Listing 7.4 Making all the endpoints accessible for everyone without authentication

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
        c -> c.anyRequest().permitAll()
    );      #A

    return http.build();
  }
}
```

The `authorizeHttpRequests()` method lets us continue with specifying authorization rules on endpoints. The `anyRequest()` method indicates that the rule applies to all the requests, regardless of the URL or HTTP method used. The `permitAll()` method allows access to all requests, authenticated or not.

Let's say we want to make sure that only users having `WRITE` authority can access all endpoints. For our example, this means only Jane. We can achieve our goal and restrict access this time based on a user's authorities. Take a look at the code in the following listing.

Listing 7.5 Restricting access to only users having `WRITE` authority

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());
```

```
    http.authorizeHttpRequests(
        c -> c.anyRequest()
          .hasAuthority("WRITE")
    );   #A

    return http.build();
  }
}
```

You can see that I replaced the `permitAll()` method with the `hasAuthority()` method. You provide the name of the authority allowed to the user as a parameter of the `hasAuthority()` method. The application needs, first, to authenticate the request and then, based on the user's authorities, the app decides whether to allow the call.

We can now start to test the application by calling the endpoint with each of the two users. When we call the endpoint with user Jane, the HTTP response status is 200 OK, and we see the response body "Hello!" When we call it with user John, the HTTP response status is 403 Forbidden, and we get an empty response body back. For example, calling this endpoint with user Jane,

```
curl -u jane:12345 http://localhost:8080/hello
```

we get this response:

```
Hello!
```

Calling the endpoint with user John,

```
curl -u john:12345 http://localhost:8080/hello
```

we get this response:

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

In a similar way, you can use the `hasAnyAuthority()` method. This method

has the parameter `varargs`; this way, it can receive multiple authority names. The application permits the request if the user has at least one of the authorities provided as a parameter to the method. You could replace `hasAuthority()` in the previous listing with `hasAnyAuthority("WRITE")`, in which case, the application works precisely in the same way. If, however, you replace `hasAuthority()` with `hasAnyAuthority("WRITE", "READ")`, then requests from users having either authority are accepted. For our case, the application allows the requests from both John and Jane. In the following listing, you can see how you can apply the `hasAnyAuthority()` method.

Listing 7.6 Applying the `hasAnyAuthority()` method

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
       c -> c.anyRequest()
            .hasAnyAuthority("WRITE", "READ");
    );      #A

    return http.build();
  }
}
```

You can successfully call the endpoint now with any of our two users. Here’s the call for John:

```
curl -u john:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

And the call for Jane:

```
curl -u jane:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

To specify access based on user authorities, the third way you find in practice is the `access()` method. The `access()` method is more general, however. It receives as a parameter an `AuthorizationManager` implementation. You can provide any implementation for this object that can apply any kind of logic that defines the authorization rules. This method is powerful, and it doesn't refer only to authorities. However, this method also makes the code more difficult to read and understand. For this reason, I recommend it as the last option, and only if you can't apply one of the `hasAuthority()` or `hasAnyAuthority()` methods presented earlier in this section.

To make this method easier to understand, I first present it as an alternative to specifying authorities with the `hasAuthority()` and `hasAnyAuthority()` methods. As you learn in this example, you'll use an `AuthorizationManager` implementation where you have to provide a SpEL expression as a parameter. The authorization rule we defined becomes more challenging to read, and this is why I don't recommend this approach for simple rules. However, the `access()` method has the advantage of allowing you to customize rules through the `AuthorizationManager` implementation you provide as a parameter. And this is really powerful! As with SpEL expressions, you can basically define any condition.

Note

In most situations, you can implement the required restrictions with the `hasAuthority()` and `hasAnyAuthority()` methods, and I recommend that you use these. Use the `access()` method only if the other two options do not fit and you want to implement more generic authorization rules.

I start with a simple example to match the same requirement as in the previous cases. If you only need to test if the user has specific authorities, the expression you need to use with the `access()` method can be one of the following:

* `hasAuthority('WRITE')`—Stipulates that the user needs the `WRITE` authority to call the endpoint.
* `hasAnyAuthority('READ', 'WRITE')`—Specifies that the user needs one of either the `READ` or `WRITE` authorities. With this expression, you can enumerate all the authorities for which you want to allow access.

Observe that these expressions have the same name as the methods presented earlier in this section. The following listing demonstrates how you can use the `access()`method.

Listing 7.7 Using the `access()` method to configure access to the endpoints

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.anyRequest()
              .access("hasAuthority('WRITE')")
    );     #A

    return http.build();
  }
}
```

The example presented in listing 7.7 proves how the `access()` method complicates the syntax if you use it for straightforward requirements. In such a case, you should instead use the `hasAuthority()` or the `hasAnyAuthority()` method directly. But the `access()` method is not all evil. As I stated earlier, it offers you flexibility. You'll find situations in real-world scenarios in which you could use it to write more complex expressions, based on which the application grants access. You wouldn't be able to implement these scenarios without the `access()` method.

In listing 7.8, you find the `access()` method applied with an expression that's

not easy to write otherwise. Precisely, the configuration presented in listing 7.8 defines two users, John and Jane, who have different authorities. The user John has only read authority, while Jane has read, write, and delete authorities. The endpoint should be accessible to those users who have read authority but not to those that have delete authority.

NOTE

In Spring apps, you find various styles and conventions for naming authorities. Some developers use all caps, other use all small letters. In my opinion, all of these choices are OK as long as you keep these consistent in your app. In this book, I use different styles in the examples so that you can observe more approaches that you might encounter in real-world scenarios.

It is a hypothetical example, of course, but it’s simple enough to be easy to understand and complex enough to prove why the `access()` method is more powerful. To implement this with the `access()` method, you can use an `AuthorizationManager` implementation that takes a SpEL expression. The SpEL expression must reflect the requirement. For example:

```
"hasAuthority('read') and !hasAuthority('delete')"
```

The next listing illustrates how to apply the `access()` method with a more complex expression. You can find this example in the project named ssia-ch7-ex2.

Listing 7.8 Applying the `access()` method with a more complex expression

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();

    var user1 = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

    var user2 = User.withUsername("jane")
```

```
            .password("12345")
            .authorities("read", "write", "delete")
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    String expression =
           """hasAuthority('read') and        #A
            !hasAuthority('delete')  #A
           """;       #A

    http.authorizeHttpRequests(
      c -> c.anyRequest()
               .access(new WebExpressionAuthorizationManager(expr

    );


    return http.build();
  }
}
```

Let’s test our application now by calling the /hello endpoint for user John:

```
curl -u john:12345 http://localhost:8080/hello
```

The body of the response is

```
Hello!
```

And calling the endpoint with user Jane:

```
curl -u jane:12345 http://localhost:8080/hello
```

The body of the response is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

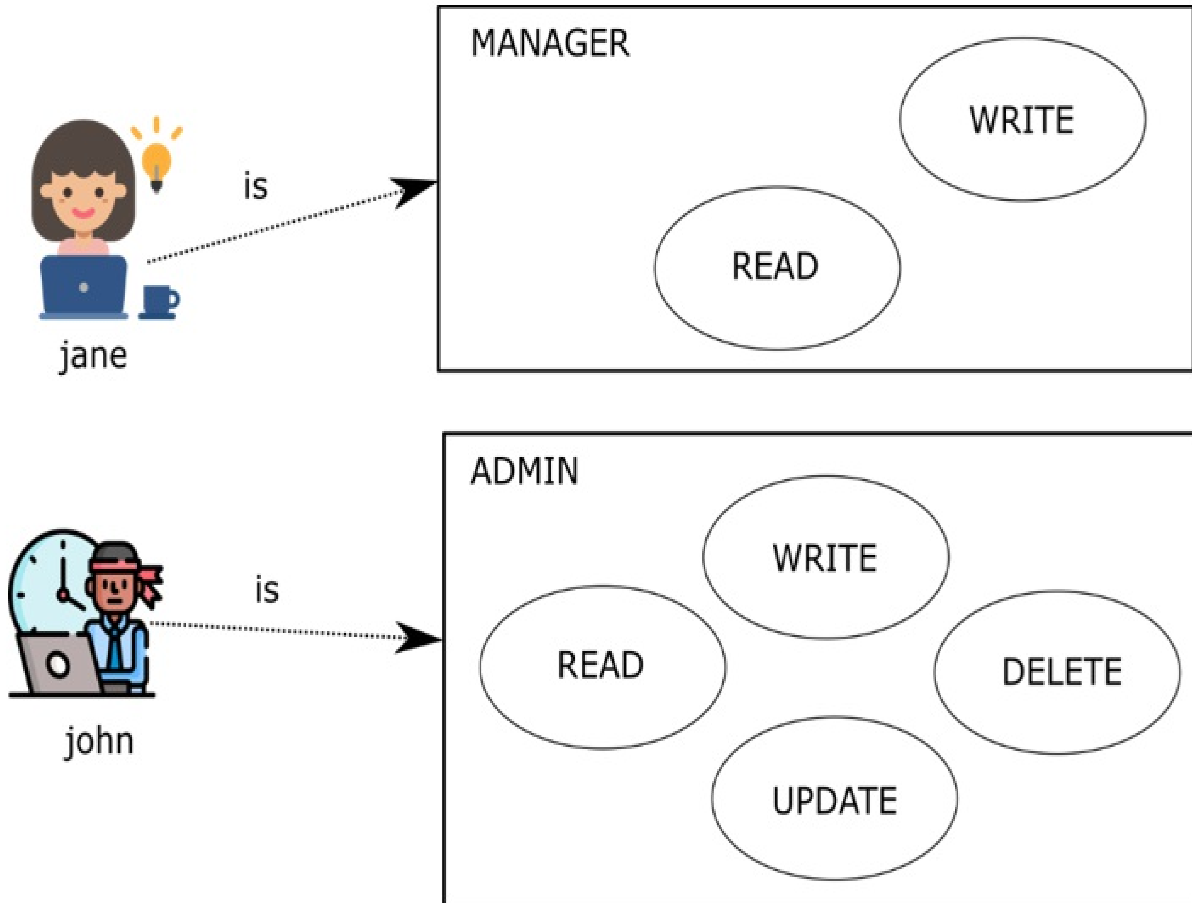
The user John has only read authority and can call the endpoint successfully. But Jane also has delete authority and is not authorized to call the endpoint. The HTTP status for the call by Jane is 403 Forbidden.

With these examples, you can see how to set constraints regarding the authorities that a user needs to have to access some specified endpoints. Of course, we haven’t yet discussed selecting which requests to be secured based on the path or the HTTP method. We have, instead, applied the rules for all requests regardless of the endpoint exposed by the application. Once we finish doing the same configuration for user roles, we discuss how to select the endpoints to which you apply the authorization configurations.

7.1.2 Restricting access for all endpoints based on user roles

In this section, we discuss restricting access to endpoints based on roles. Roles are another way to refer to what a user can do (figure 7.5). You find these as well in real-world applications, so this is why it is important to understand roles and the difference between roles and authorities. In this section, we apply several examples using roles so that you’ll know all the practical scenarios in which the application uses roles and how to write configurations for these cases.

Figure 7.5 Roles are coarse grained. Each user with a specific role can only do the actions granted by that role. When applying this philosophy in authorization, a request is allowed based on the purpose of the user in the system. Only users who have a specific role can call a certain endpoint.



Spring Security understands authorities as fine-grained privileges on which we apply restrictions. Roles are like badges for users. These give a user privileges for a group of actions. Some applications always provide the same groups of authorities to specific users. Imagine, in your application, a user can either only have read authority or have all: read, write, and delete authorities. In this case, it might be more comfortable to think that those users who can only read have a role named READER, while the others have the role ADMIN. Having the ADMIN role means that the application grants you read, write, update, and delete privileges. You could potentially have more roles. For example, if at some point the requests specify that you also need a user who is only allowed to read and write, you can create a third role named MANAGER for your application.

NOTE

When using an approach with roles in the application, you won’t have to

define authorities anymore. The authorities exist, in this case as a concept, and can appear in the implementation requirements. But in the application, you only have to define a role to cover one or more such actions a user is privileged to do.

The names that you give to roles are like the names for authorities—it's your own choice. We could say that roles are coarse grained when compared with authorities. Behind the scenes, anyway, roles are represented using the same contract in Spring Security, `GrantedAuthority`. When defining a role, its name should start with the `ROLE_` prefix. At the implementation level, this prefix specifies the difference between a role and an authority. You find the example we work on in this section in the project ssia-ch7-ex3. In the next listing, take a look at the change I made to the previous example.

Listing 7.9 Setting roles for users

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();

    var user1 = User.withUsername("john")
                    .password("12345")
                    .authorities("ROLE_ADMIN")     #A
                    .build();

    var user2 = User.withUsername("jane")
                    .password("12345")
                    .authorities("ROLE_MANAGER")
                    .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  // Omitted code

}
```

To set constraints for user roles, you can use one of the following methods:

- `hasRole()`—Receives as a parameter the role name for which the application authorizes the request.
- `hasAnyRole()`—Receives as parameters the role names for which the application approves the request.
- `access()`—Uses an `AuthorizationManager` to specify the role or roles for which the application authorizes requests. In terms of roles, you could use `hasRole()` or `hasAnyRole()` as SpEL expressions together with the `WebExpressionAuthorizationManager` implementation.

As you observe, the names are similar to the methods presented in section 7.1.1. We use these in the same way, but to apply configurations for roles instead of authorities. My recommendations are also similar: use the `hasRole()` or `hasAnyRole()` methods as your first option, and fall back to using `access()` only when the previous two don't apply. In the next listing, you can see what the `configure()` method looks like now.

Listing 7.10 Configuring the app to accept only requests from admins

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.anyRequest().hasRole("ADMIN")
    );       #A


    return http.build();
  }
}
```

NOTE

A critical thing to observe is that we use the `ROLE_` prefix only to declare the role. But when we use the role, we do it only by its name.

When testing the application, you should observe that user John can access the endpoint, while Jane receives an HTTP 403 Forbidden. To call the endpoint with user John, use

```
curl -u john:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

And to call the endpoint with user Jane, use

```
curl -u jane:12345 http://localhost:8080/hello
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

When building users with the `User` builder class as we did in the example for this section, you specify the role by using the `roles()` method. This method creates the `GrantedAuthority` object and automatically adds the `ROLE_` prefix to the names you provide.

NOTE

Make sure the parameter you provide for the `roles()` method does not include the `ROLE_` prefix. If that prefix is inadvertently included in the `role()` parameter, the method throws an exception. In short, when using the `authorities()` method, include the `ROLE_` prefix. When using the `roles()` method, do not include the `ROLE_` prefix.

In the following listing, you can see the correct way to use the `roles()`

method instead of the `authorities()` method when you design access based on roles. You can also compare listing 7.11 with listing 7.9 to observe the difference between using authorities and roles.

Listing 7.11 Setting up roles with the `roles()` method

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();

    var user1 = User.withUsername("john")
                    .password("12345")
                    .roles("ADMIN")     #A
                    .build();

    var user2 = User.withUsername("jane")
                    .password("12345")
                    .roles("MANAGER")
                    .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  // Omitted code
}
```

More on the `access()` method

In sections 7.1.1 and 7.1.2, you learned to use the `access()` method to apply authorization rules referring to authorities and roles. In general, in an application the authorization restrictions are related to authorities and roles. But it's important to remember that the `access()` method is generic and it only depends on what implementation of the `AuthorizationManager` contract you provide as a parameter. Moreover, in our example, we only used the `WebExpressionAuthorizationManager` implementation which applies the authorization restrictions based on a SpEL expression. With the examples I present, I focus on teaching you how to apply it for authorities and roles, but

in practice, `WebExpressionAuthorizationManager` receives any SpEL expression. It doesn't need to be related to authorities and roles.

A straightforward example would be to configure access to the endpoint to be allowed only after 12:00 pm. To solve something like this, you can use the following SpEL expression:

```
T(java.time.LocalTime).now().isAfter(T(java.time.LocalTime).of(12
```

For more about SpEL expressions, see the Spring Framework documentation:

https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions

We could say that with the `access()` method, you can basically implement any kind of rule. The possibilities are endless. Just don't forget that in applications, we always strive to keep syntax as simple as possible. Complicate your configurations only when you don't have any other choice. You'll find this example applied in the project ssia-ch7-ex4.

7.1.3 Restricting access to all endpoints

In this section, we discuss restricting access to all requests. You learned in section 5.2 that by using the `permitAll()` method, you can permit access for all requests. You learned as well that you can apply access rules based on authorities and roles. But what you can also do is deny all requests. The `denyAll()` method is just the opposite of the `permitAll()`method. In the next listing, you can see how to use the `denyAll()` method.

Listing 7.12 Using the `denyAll()` method to restrict access to endpoints

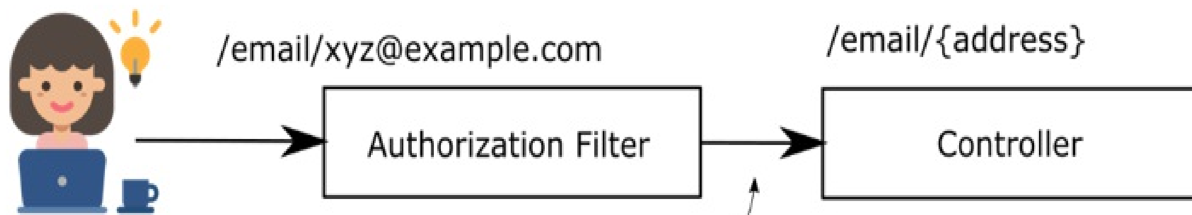
```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {
```

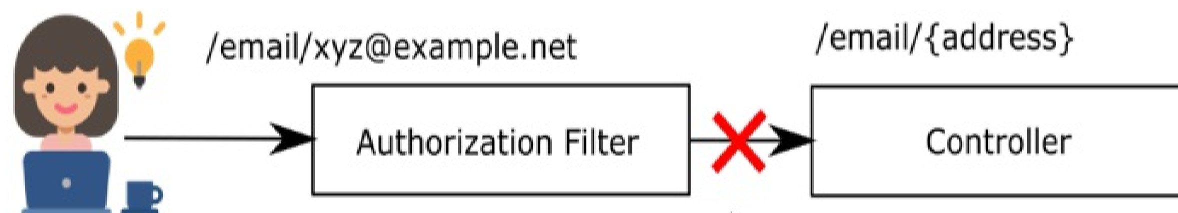
```
    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
       c -> c.anyRequest().denyAll()
    );      #A

    return http.basic();
  }
}
```

So, where could you use such a restriction? You won't find it used as much as the other methods, but there are cases in which requirements make it necessary. Let me show you a couple of cases to clarify this point.

Let's assume that you have an endpoint receiving as a path variable an email address. What you want is to allow requests that have the value of the variable addresses ending in .com. You don't want the application to accept any other format for the email address. (You'll learn in the next chapter how to apply restrictions for a group of requests based on the path and HTTP method and even for path variables.) For this requirement, you use a regular expression to group requests that match your rule and then use the `denyAll()` method to instruct your application to deny all these requests (figure 7.6).

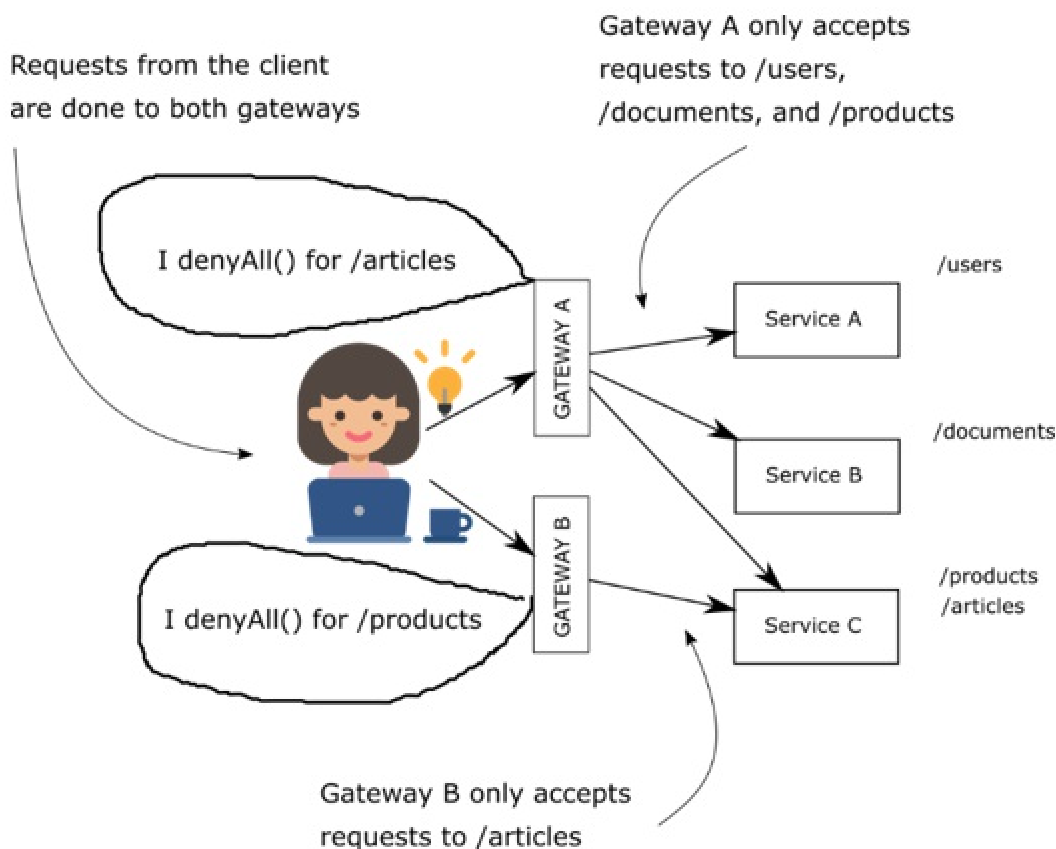
Figure 7.6 When the user calls the endpoint with a value of the parameter ending in .com, the application accepts the request. When the user calls the endpoint and provides an email address ending in .net, the application rejects the call. To achieve such behavior, you can use the `denyAll()` method for all endpoints for which the value of the parameter doesn't end with .com.





You can also imagine an application designed as in figure 7.7. A few services implement the use cases of the application, which are accessible by calling endpoints available at different paths. But to call an endpoint, the client requests another service that we can call a gateway. In this architecture, there are two separate services of this type. In figure 7.7, I called these Gateway A and Gateway B. The client requests Gateway A if they want to access the /products path. But for the /articles path, the client has to request Gateway B. Each of the gateway services is designed to deny all requests to other paths that these do not serve. This simplified scenario can help you easily understand the `denyAll()` method. In a production application, you could find similar cases in more complex architectures.

Figure 7.7 Access is done through Gateway A and Gateway B. Each of the gateways only delivers requests for specific paths and denies all others.



Applications in production face various architectural requirements, which could look strange sometimes. A framework must allow the needed flexibility for any situation you might encounter. For this reason, the `denyAll()` method is as important as all the other options you learned in his chapter.

7.2 Summary

- Authorization is the process during which the application decides if an authenticated request is permitted or not. Authorization always happens after authentication.
- You configure how the application authorizes requests based on the authorities and roles of an authenticated user.
- In your application, you can also specify that certain requests are possible for unauthenticated users.
- You can configure your app to reject any request, using the `denyAll()` method, or permit any requests, using the `permitAll()`method.

8 Configuring endpoint-level authorization: Applying restrictions

This chapter covers

- Selecting requests to apply restrictions using matcher methods
- Learning best-case scenarios for each matcher method

In chapter 7, you learned how to configure access based on authorities and roles. But we only applied the configurations for all of the endpoints. In this chapter, you'll learn how to apply authorization constraints to a specific group of requests. In production applications, it's less probable that you'll apply the same rules for all requests. You have endpoints that only some specific users can call, while other endpoints might be accessible to everyone. Each application, depending on the business requirements, has its own custom authorization configuration. Let's discuss the options you have to refer to different requests when you write access configurations.

Even though we didn't call attention to it, the first matcher method you used was the `anyRequest()` method. As you used it in the previous chapters, you know now that it refers to all requests, regardless of the path or HTTP method. It is the way you say "any request" or, sometimes, "any other request."

First, let's talk about selecting requests by path; then we can also add the HTTP method to the scenario. To choose the requests to which we apply authorization configuration, we use the `requestMatchers()` method.

8.1 Using the requestMatchers() method to select endpoints

In this section, you learn how to use the `requestMatchers()` method in general so that in sections 8.2 through 8.4, we can continue describing

various approaches you have to select HTTP requests for which you need to apply authorization restrictions. By the end of this chapter, you'll be able to apply the `requestMatchers()` method in any authorization configurations you might need to write for your application's requirements. Let's start with a straightforward example.

We create an application that exposes two endpoints: /hello and /ciao. We want to make sure that only users having the ADMIN role can call the /hello endpoint. Similarly, we want to make sure that only users having the MANAGER role can call the /ciao endpoint. You can find this example in the project ssia-ch8-ex1. The following listing provides the definition of the controller class.

Listing 8.1 The definition of the controller class

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }

  @GetMapping("/ciao")
  public String ciao() {
    return "Ciao!";
  }
}
```

In the configuration class, we declare an `InMemoryUserDetailsManager` as our `UserDetailsService` instance and add two users with different roles. The user John has the ADMIN role, while Jane has the MANAGER role. To specify that only users having the ADMIN role can call the endpoint /hello when authorizing requests, we use the `requestMatchers()` method. The next listing presents the definition of the configuration class.

Listing 8.2 The definition of the configuration class

```
@Configuration
public class ProjectConfig {
```

```
  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();

    var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")
            .build();

    var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER")
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.requestMatchers("/hello").hasRole("ADMIN")      #A
            .requestMatchers("/ciao").hasRole("MANAGER")    #B

    );


    return http.build();
  }

}
```

You can run and test this application. When you call the endpoint /hello with user John, you get a successful response. But if you call the same endpoint with user Jane, the response status returns an HTTP 403 Forbidden.

Similarly, for the endpoint /ciao, you can only use Jane to get a successful result. For user John, the response status returns an HTTP 403 Forbidden. You can see the example calls using cURL in the next code snippets. To call the endpoint /hello for user John, use

```
curl -u john:12345 http://localhost:8080/hello
```

The response body is

```
Hello!
```

To call the endpoint /hello for user Jane, use

```
curl -u jane:12345 http://localhost:8080/hello
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

To call the endpoint /ciao for user Jane, use

```
curl -u jane:12345 http://localhost:8080/ciao
```

The response body is

```
Ciao!
```

To call the endpoint /ciao for user John, use

```
curl -u john:12345 http://localhost:8080/ciao
```

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/ciao"
```

```
}
```

If you now add any other endpoint to your application, it is accessible by default to anyone, even unauthenticated users. Let’s assume you add a new endpoint /hola as presented in the next listing.

Listing 8.3 Adding a new endpoint for path /hola to the application

```
@RestController
public class HelloController {

  // Omitted code

  @GetMapping("/hola")
  public String hola() {
    return "Hola!";
  }
}
```

Now when you access this new endpoint, you see that it is accessible with or without having a valid user. The next code snippets display this behavior. To call the endpoint /hola without authenticating, use

```
curl http://localhost:8080/hola
```

The response body is

```
Hola!
```

To call the endpoint /hola for user John, use

```
curl -u john:12345 http://localhost:8080/hola
```

The response body is

```
Hola!
```

You can make this behavior more visible if you like by using the `permitAll()` method. You do this by using the `anyRequest()` matcher method at the end of the configuration chain for the request authorization, as presented in listing 8.4.

NOTE

It is good practice to make all your rules explicit. Listing 8.4 clearly and unambiguously indicates the intention to permit requests to endpoints for everyone, except for the endpoints /hello and /ciao.

Listing 8.4 Marking additional requests explicitly as accessible without authentication

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
       c -> c.mvcMatchers("/hello").hasRole("ADMIN")
           .mvcMatchers("/ciao").hasRole("MANAGER")
           .anyRequest().permitAll()      #A

    );


    return http.build();
  }
}
```

NOTE

When you use matchers to refer to requests, the order of the rules should be from particular to general. This is why the `anyRequest()` method cannot be called before a more specific `requestMatchers()` method.

Unauthenticated vs. failed authentication

If you have designed an endpoint to be accessible to anyone, you can call it without providing a username and a password for authentication. In this case, Spring Security won’t do the authentication. If you, however, provide a

username and a password, Spring Security evaluates them in the authentication process. If they are wrong (not known by the system), authentication fails, and the response status will be 401 Unauthorized. To be more precise, if you call the /hola endpoint for the configuration presented in listing 8.4, the app returns the body “Hola!” as expected, and the response status is 200 OK. For example,

```
curl http://localhost:8080/hola
```

The response body is

```
Hola!
```

But if you call the endpoint with invalid credentials, the status of the response is 401 Unauthorized. In the next call, I use an invalid password:

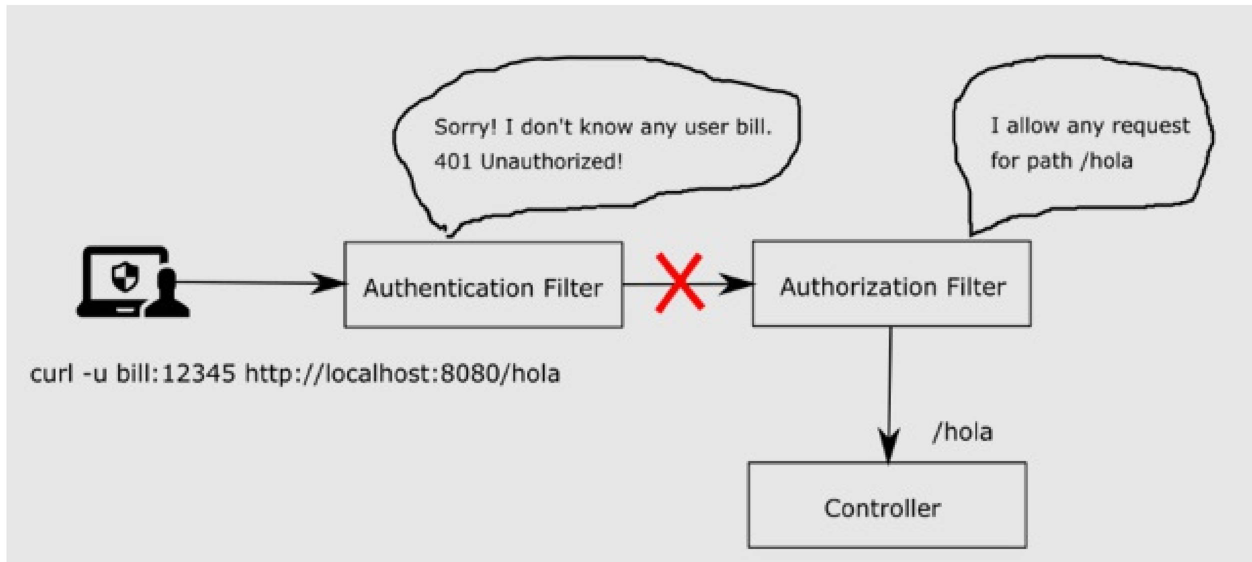
```
curl -u bill:abcde http://localhost:8080/hola
```

The response body is

```
{
    "status":401,
    "error":"Unauthorized",
    "message":"Unauthorized",
    "path":"/hola"
}
```

This behavior of the framework might look strange, but it makes sense as the framework evaluates any username and password if you provide them in the request. As you learned in chapter 7, the application always does authentication before authorization, as this figure shows.

The authorization filter allows any request to the /hola path. But because the application first executes the authentication logic, the request is never forwarded to the authorization filter. Instead, the authentication filter replies with an HTTP 401 Unauthorized.



In conclusion, any situation in which authentication fails will generate a response with the status 401 Unauthorized, and the application won’t forward the call to the endpoint. The `permitAll()` method refers to authorization configuration only, and if authentication fails, the call will not be allowed further.

You could decide, of course, to make all the other endpoints accessible only for authenticated users. To do this, you would change the `permitAll()` method with `authenticated()` as presented in the following listing. Similarly, you could even deny all other requests by using the `denyAll()` method.

Listing 8.5 Making other requests accessible for all authenticated users

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
       c -> c.requestMatchers("/hello").hasRole("ADMIN")
```

```
            .requestMatchers("/ciao").hasRole("MANAGER")
            .anyRequest().authenticated()      #A
    );

    return http.build();
  }
}
```

Here, at the end of this section, you’ve become familiar with how you should use matcher methods to refer to requests for which you want to configure authorization restrictions. Now we must go more in depth with the syntaxes you can use.

In most practical scenarios, multiple endpoints can have the same authorization rules, so you don’t have to set them up endpoint by endpoint. As well, you sometimes need to specify the HTTP method, not only the path, as we’ve done until now.

Sometimes, you only need to configure rules for an endpoint when its path is called with HTTP GET. In this case, you’d need to define different rules for HTTP POST and HTTP DELETE. In the next section, we take each type of matcher method and discuss these aspects in detail.

8.2 Selecting requests to apply authorization restrictions

In this section, we deep dive into configuring request matchers. Using the `requestMatchers()` method is a common approach to refer to requests for applying authorization configuration. So I expect you to have many opportunities to use this method to refer to requests in the applications you develop.

This matcher uses the standard ANT syntax (table 8.1) for referring to paths. This syntax is the same one you use when writing endpoint mappings with annotations like `@RequestMapping`, `@GetMapping`, `@PostMapping`, and so forth. The two methods you can use to declare MVC matchers are as follows:

- `requestMatchers(HttpMethod method, String... patterns)`—Lets you specify both the HTTP method to which the restrictions apply and the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same path.
- `requestMatchers(String... patterns)`—Simpler and easier to use if you only need to apply authorization restrictions based on paths. The restrictions can automatically apply to any HTTP method used with the path.

In this section, we approach multiple ways of using `requestMatchers()` methods. To demonstrate this, we start by writing an application that exposes multiple endpoints.

For the first time, we write endpoints that can be called with other HTTP methods besides GET. You might have observed that until now, I've avoided using other HTTP methods. The reason for this is that Spring Security applies, by default, protection against cross-site request forgery (CSRF). In chapter 9, we'll discuss how Spring Security mitigates this vulnerability by using CSRF tokens. But to make things simpler for the current example and to be able to call all endpoints, including those exposed with POST, PUT, or DELETE, we need to disable CSRF protection in our `configure()` method:

```
http.csrf(
  c -> c.disable()
);
```

NOTE

We disable CSRF protection now only to allow you to focus for the moment on the discussed topic: matcher methods. But don't rush to consider this a good approach. In chapter 9, we'll discuss in detail the CSRF protection provided by Spring Security.

We start by defining four endpoints to use in our tests:

- /a using the HTTP method GET
- /a using the HTTP method POST
- /a/b using the HTTP method GET

- /a/b/c using the HTTP method GET

With these endpoints, we can consider different scenarios for authorization configuration. In listing 8.6, you can see the definitions of these endpoints. You can find this example in the project ssia-ch8-ex2.

Listing 8.6 Definition of the four endpoints for which we configure authorization

```
@RestController
public class TestController {

  @PostMapping("/a")
  public String postEndpointA() {
    return "Works!";
  }

  @GetMapping("/a")
  public String getEndpointA() {
    return "Works!";
  }

  @GetMapping("/a/b")
  public String getEnpointB() {
    return "Works!";
  }

  @GetMapping("/a/b/c")
  public String getEnpointC() {
    return "Works!";
  }
}
```

We also need a couple of users with different roles. To keep things simple, we continue using an `InMemoryUserDetailsManager`. In the next listing, you can see the definition of the `UserDetailsService` in the configuration class.

Listing 8.7 The definition of the `UserDetailsService`

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();   #A
```

```
    var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")       #B
            .build();

    var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER")     #C
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();      #D
  }
}
```

Let’s start with the first scenario. For requests done with an HTTP GET method for the /a path, the application needs to authenticate the user. For the same path, requests using an HTTP POST method don’t require authentication. The application denies all other requests. The following listing shows the configurations that you need to write to achieve this setup.

Listing 8.8 Authorization configuration for the first scenario, /a

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.requestMatchers(HttpMethod.GET, "/a")
              .authenticated()     #A
```

```
            .requestMatchers(HttpMethod.POST, "/a")
               .permitAll()     #B
            .anyRequest()
               .denyAll()    #C

    );


    http.csrf(
      c -> c.disable()
    );      #D

    return http.build();
  }
}
```

In the next code snippets, we analyze the results on the calls to the endpoints for the configuration presented in listing 8.8. For the call to path /a using the HTTP method POST without authenticating, use this cURL command:

```
curl -XPOST http://localhost:8080/a
```

The response body is

```
Works!
```

When calling path /a using HTTP GET without authenticating, use

```
curl -XGET http://localhost:8080/a
```

The response is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/a"
}
```

If you want to change the response to a successful one, you need to authenticate with a valid user. For the following call

```
curl -u john:12345 -XGET http://localhost:8080/a
```

the response body is

```
Works!
```

But user John isn’t allowed to call path /a/b, so authenticating with his credentials for this call generates a 403 Forbidden:

```
curl -u john:12345 -XGET http://localhost:8080/a/b
```

The response is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/a/b"
}
```

With this example, you now know how to differentiate requests based on the HTTP method. But, what if multiple paths have the same authorization rules? Of course, we can enumerate all the paths for which we apply authorization rules, but if we have too many paths, this makes reading code uncomfortable. As well, we might know from the beginning that a group of paths with the same prefix always has the same authorization rules. We want to make sure that if a developer adds a new path to the same group, it doesn’t also change the authorization configuration. To manage these cases, we use "charitalics"path expressions. Let’s prove these in an example.

For the current project, we want to ensure that the same rules apply for all requests for paths starting with /a/b. These paths in our case are /a/b and /a/b/c. To achieve this, we use the ** operator. You can find this example in the project ssia-ch8-ex3.

Listing 8.9 Changes in the configuration class for multiple paths

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
```

```
  public void configure(HttpSecurity http)
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.requestMatchers( "/a/b/**").authenticated()    #A
            .anyRequest().permitAll();

    );

    http.csrf(
      c -> c.disable()
    );

    return http.build();
  }
}
```

With the configuration given in listing 8.9, you can call path /a without being authenticated, but for all paths prefixed with /a/b, the application needs to authenticate the user. The next code snippets present the results of calling the /a, /a/b, and /a/b/c endpoints. First, to call the /a path without authenticating, use

```
curl http://localhost:8080/a
```

The response body is

```
Works!
```

To call the /a/b path without authenticating, use

```
curl http://localhost:8080/a/b
```

The response is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/a/b"
}
```

To call the /a/b/c path without authenticating, use

```
curl http://localhost:8080/a/b/c
```

The response is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/a/b/c"
}
```

As presented in the previous examples, the `**` operator refers to any number of pathnames. You can use it as we have done in the last example so that you can match requests with paths having a known prefix. You can also use it in the middle of a path to refer to any number of pathnames or to refer to paths ending in a specific pattern like `/a/**/c`. Therefore, `/a/**/c` would not only match /a/b/c but also /a/b/d/c and a/b/c/d/e/c and so on. If you only want to match one pathname, then you can use a single `*`. For example, `a/*/c` would match a/b/c and a/d/c but not a/b/d/c.

Because you generally use path variables, you can find it useful to apply authorization rules for such requests. You can even apply rules referring to the path variable value. Do you remember the discussion from section 8.1 about the `denyAll()` method and restricting all requests?

Let’s turn now to a more suitable example of what you have learned in this section. We have an endpoint with a path variable, and we want to deny all requests that use a value for the path variable that has anything else other than digits. You can find this example in the project ssia-ch8-ex4. The following listing presents the controller.

Listing 8.10 The definition of an endpoint with a path variable in a controller class

```
@RestController
public class ProductController {

  @GetMapping("/product/{code}")
  public String productCode(@PathVariable String code) {
    return code;
```

```
  }
}
```

The next listing shows you how to configure authorization such that only calls that have a value containing only digits are always permitted, while all other calls are denied.

Listing 8.11 Configuring the authorization to permit only specific digits

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(
      c -> c.requestMatchers
          [CA]("/product/{code:^[0-9]*$}")      #A
              .permitAll()
         .anyRequest()
              .denyAll()

    );


    return http.build();
  }
}
```

NOTE

When using parameter expressions with a regex, make sure to not have a space between the name of the parameter, the colon (:), and the regex, as displayed in the listing.

Running this example, you can see the result as presented in the following code snippets. The application only accepts the call when the path variable value has only digits. To call the endpoint using the value `1234a`, use

```
curl http://localhost:8080/product/1234a
```

The response is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/product/1234a"
}
```

To call the endpoint using the value `12345`, use

```
curl http://localhost:8080/product/12345
```

The response is

```
12345
```

We discussed a lot and included plenty of examples of how to refer to requests using `requestMatchers()` method. Table 8.1 is a refresher for the path expressions you used in this section. You can simply refer to it later when you want to remember any of them.

Table 8.1 Common expressions used for path matching with MVC matchers

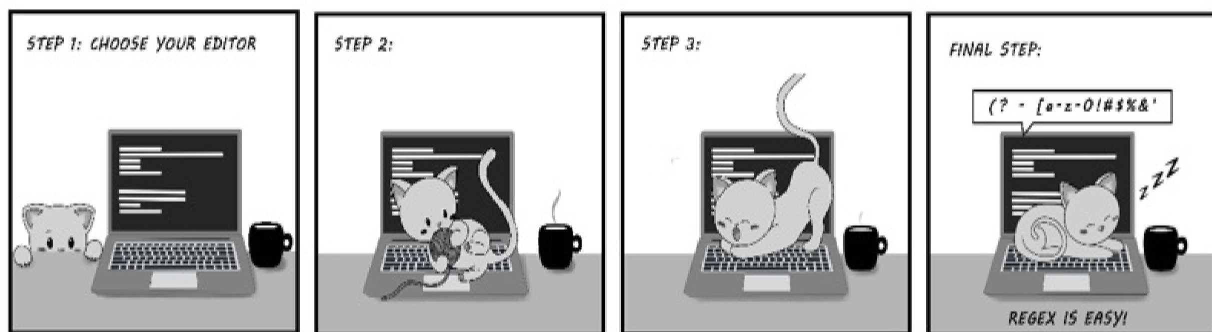
Expression	Description
`/a`	Only path /a.
`/a/*`	The * operator replaces one pathname. In this case, it matches /a/b or /a/c, but not /a/b/c.
`/a/**`	The ** operator replaces multiple pathnames. In this case, /a as well as /a/b and /a/b/c are a match for this expression.

`/a/{param}`	This expression applies to the path /a with a given path parameter.
`/a/{param:regex}`	This expression applies to the path /a with a given path parameter only when the value of the parameter matches the given regular expression.

8.2.1 Using regular expressions with request matchers

In this section, we discuss regular expression (regex). You should already be aware of what regular expressions are, but you don't need to be an expert in the subject. Any of the books recommended at https://www.regular-expressions.info/books.html are excellent resources from which you can learn about the subject in more depth. For writing regex, I also often use online generators like (figure 8.1).

Figure 8.1 Letting your cat play over the keyboard is not the best solution for generating regular expressions (regex). To learn how to generate regexes you can use an online generator like https://regexr.com/.



You learned in sections 8.2 and 8.3 that in most cases, you can use path expression syntaxes to refer to requests to which you apply authorization configurations. In some cases, however, you might have requirements that are more particular, and you cannot solve those with path expressions. An example of such a requirement could be this: "Deny all requests when paths contain specific symbols or characters." For these scenarios, you need to use

a more powerful expression like a regex.

You can use regexes to represent any format of a string, so they offer limitless possibilities for this matter. But they have the disadvantage of being difficult to read, even when applied to simple scenarios. For this reason, you might prefer to use path expressions and fall back to regexes only when you have no other option. To implement a regex request matcher you can use the `requestMatchers()` method with a `RegexRequestMatcher` implementation as parameter.

To prove how regex matchers work, let's put them into action with an example: building an application that provides video content to its users. The application that presents the video gets its content by calling the endpoint /video/{country}/{language}. For the sake of the example, the application receives the country and language in two path variables from where the user makes the request. We consider that any authenticated user can see the video content if the request comes from the US, Canada, or the UK, or if they use English.

You can find this example implemented in the project ssia-ch8-ex5. The endpoint we need to secure has two path variables, as shown in the following listing. This makes the requirement complicated to implement with request matchers.

Listing 8.12 The definition of the endpoint for the controller class

```
@RestController
public class VideoController {

  @GetMapping("/video/{country}/{language}")
  public String video(@PathVariable String country,
                      @PathVariable String language) {
    return "Video allowed for " + country + " " + language;
  }
}
```

For a condition on a single path variable, we can write a regex directly in the path expression. We referred to such an example in section 8.2, but I didn't go in depth about it at that time because we weren't discussing regexes.

Let’s assume you have the endpoint /email/{email}. You want to apply a rule using a matcher only to the requests that send as a value of the `email` parameter an address ending in .com. In that case, you write a request matcher as presented by the next code snippet. You can find the complete example of this in the project ssia-ch8-ex6.

```
http.authorizeHttpRequests(
   c -> c.requestMatchers("/email/{email:.*(?:.+@.+\\.com)}" ).pe
         .anyRequest().denyAll();

);
```

If you test such a restriction, you find that the application only accepts emails ending in .com. For example, to call the endpoint to jane@example.com, you can use this command:

```
curl http://localhost:8080/email/jane@example.com
```

The response body is

```
Allowed for email jane@example.com
```

And to call the endpoint to jane@example.net, you use this command:

```
curl http://localhost:8080/email/jane@example.net
```

The response body is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":/email/jane@example.net
}
```

It is fairly easy and makes it even clearer why we encounter regex matchers less frequently. But, as I said earlier, requirements are complex sometimes. You’ll find it handier to use regex matchers when you find something like the following:

- Specific configurations for all paths containing phone numbers or email addresses

- Specific configurations for all paths having a certain format, including what is sent through all the path variables

Back to our regex matchers example (ssia-ch8-ex6): when you need to write a more complex rule, eventually referring to more path patterns and multiple path variable values, it's easier to write a regex matcher. In listing 8.13, you find the definition for the configuration class that uses a regex matcher to solve the requirement given for the /video/{country}/{language} path. We also add two users with different authorities to test the implementation.

Listing 8.13 The configuration class using a regex matcher

```
@Configuration
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var uds = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("john")
                 .password("12345")
                 .authorities("read")
                 .build();

    var u2 = User.withUsername("jane")
                .password("12345")
                .authorities("read", "premium")
                .build();

    uds.createUser(u1);
    uds.createUser(u2);

    return uds;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {
```

```
    http.httpBasic(Customizer.withDefaults());

    http.authorizeHttpRequests(

      c -> c.regexMatchers(".*/(us|uk|ca)+/(en|fr).*")    #A
                .authenticated()
            .anyRequest()
                .hasAuthority("premium");     #B

    );

  }
}
```

Running and testing the endpoints confirm that the application applied the authorization configurations correctly. The user John can call the endpoint with the country code `US` and language `en`, but he can't call the endpoint for the country code `FR` and language `fr` due to the restrictions we configured. Calling the /video endpoint and authenticating user John for the US region and the English language looks like this:

```
curl -u john:12345 http://localhost:8080/video/us/en
```

The response body is

```
Video allowed for us en
```

Calling the /video endpoint and authenticating user John for the FR region and the French language looks like this:

```
curl -u john:12345 http://localhost:8080/video/fr/fr
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/video/fr/fr"
}
```

Having premium authority, user Jane makes both calls with success. For the first call,

```
curl -u jane:12345 http://localhost:8080/video/us/en
```

the response body is

```
Video allowed for us en
```

And for the second call,

```
curl -u jane:12345 http://localhost:8080/video/fr/fr
```

the response body is

```
Video allowed for fr fr
```

Regexes are powerful tools. You can use them to refer to paths for any given requirement. But because regexes are hard to read and can become quite long, they should remain your last choice. Use these only if path expressions don’t offer you a solution to your problem.

In this section, I used the most simple example I could imagine so that the needed regex is short. But with more complex scenarios, the regex can become much longer. Of course, you’ll find experts who say any regex is easy to read. For example, a regex used to match an email address might look like the one in the next code snippet. Can you easily read and understand it?

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)
```

8.3 Summary

- In real-world scenarios, you often apply different authorization rules for different requests.
- You specify the requests for which authorization rules are configured based on path and HTTP method. To do this, you use the `requestMatchers()` method.
- When requirements are too complex to be solved with path expressions, you can implement them with the more powerful regexes.

9 Configuring Cross-Site Request Forgery (CSRF) protection

This chapter covers

- Understanding CSRF attacks
- Implementing cross-site request forgery protection
- Customizing CSRF protection

You have learned about the filter chain and its purpose in the Spring Security architecture. We worked on several examples in chapter 5, where we customized the filter chain. But Spring Security also adds its own filters to the chain. In this chapter, we'll discuss the filter that configures CSRF (Cross-Site Request Forgery) protection. You'll learn to customize these filters to make a perfect fit for your scenarios.

You have probably observed that in most of the examples up to now, we only implemented our endpoints with HTTP GET. Moreover, when we needed to configure HTTP POST, we also had to add a supplementary instruction to the configuration to disable CSRF protection. The reason why you can't directly call an endpoint with HTTP POST is because of CSRF protection, which is enabled by default in Spring Security.

We'll now discuss CSRF protection and when to use it in your applications. CSRF is a widespread type of attack, and applications vulnerable to CSRF can force users to execute unwanted actions on a web application after authentication. You don't want the applications you develop to be CSRF vulnerable and allow attackers to trick your users into making unwanted actions.

Because it's essential to understand how to mitigate these vulnerabilities, we start by reviewing what CSRF is and how it works. We then discuss the CSRF token mechanism that Spring Security uses to mitigate CSRF vulnerabilities. We continue with obtaining a token and use it to call an

endpoint with the HTTP POST method. We prove this with a small application using REST endpoints. Once you learn how Spring Security implements its CSRF token mechanism, we discuss how to use it in real-world application scenarios. Finally, you learn possible customizations of the CSRF token mechanism in Spring Security.

9.1 How CSRF protection works in Spring Security

In this section, we discuss how Spring Security implements CSRF protection. I consider it essential first to understand the underlying mechanism of CSRF protection. I encounter many situations in which misunderstanding the way CSRF protection works leads developers to misuse it, either disabling it in scenarios where it should be enabled or the other way around. Like any other feature in a framework, you have to use it correctly to bring value to your applications.

As an example, consider this scenario (figure 9.1): you are at work, where you use a web tool to store and manage your files. With this tool, in a web interface you can add new files, add new versions for your records, and even delete them. You receive an email asking you to open a page for a specific reason (for example, a promotion at your favorite store). You open the page, but the page is blank or it redirects you to a known website (the online shop of your favorite store). You go back to your work but observe that all your files are gone!

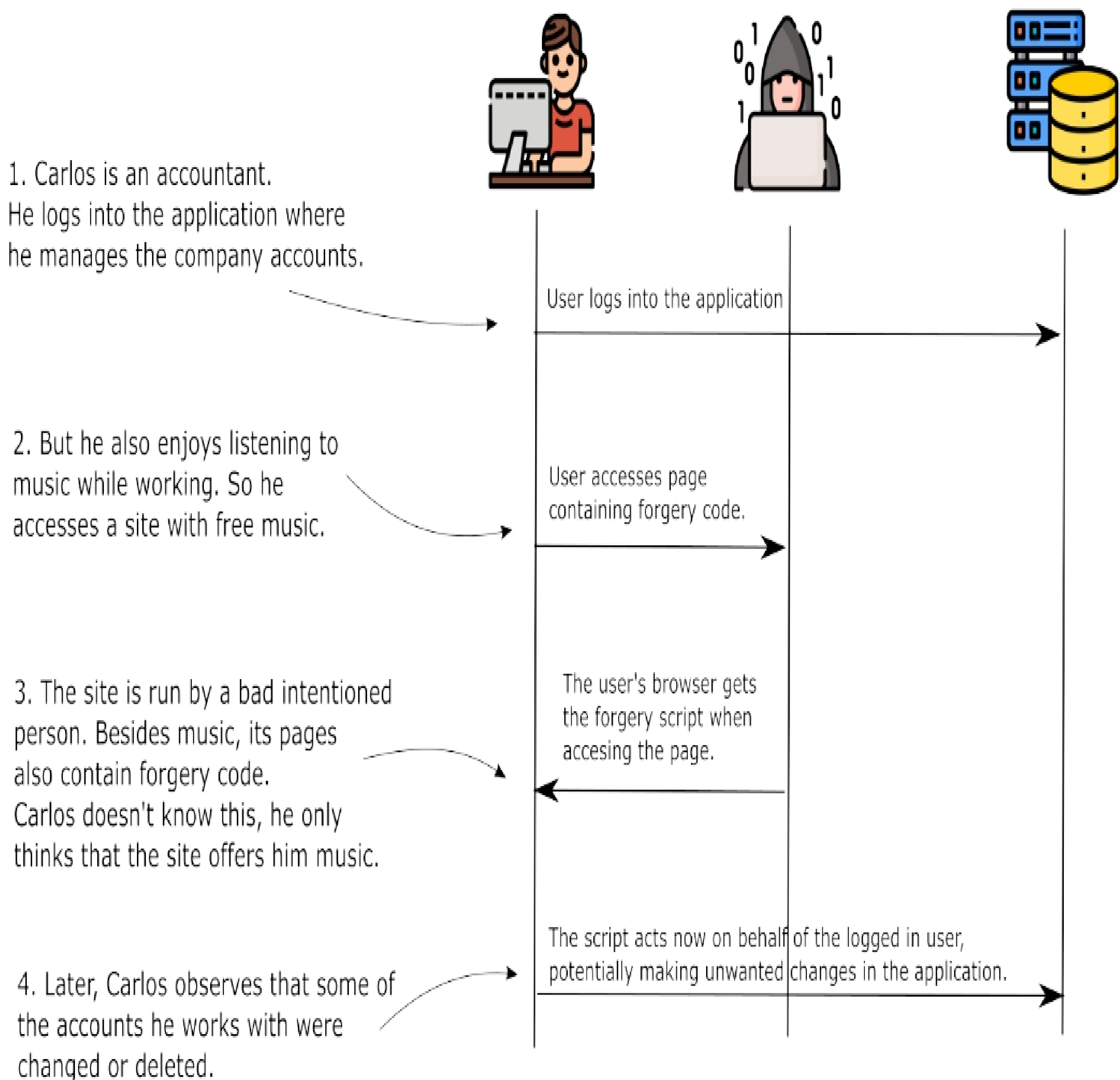
What happened? You were logged into your work application so you could manage your files. When you add, change, or delete a file, the web page you interact with calls some endpoints from the server to execute these operations. When you opened the foreign page by clicking the unknown link in the email, that page called your app’s backend and executed actions on your behalf (it deleted your files).

It could do that because you logged in previously, so the server trusted that the actions came from you. You might think that someone couldn’t trick you so easily into clicking a link from a foreign email or message, but trust me, this happens to a lot of people. Most web app users aren’t aware of security risks. So it’s wiser if you who know all the tricks, protect your users, and

build secure apps rather than rely on your apps' users to protect themselves.

CSRF attacks assume that a user is logged into a web application. They're tricked by the attacker into opening a page that contains scripts that execute actions in the same application the user was working on. Because the user has already logged in (as we've assumed from the beginning), the forgery code can now impersonate the user and do actions on their behalf.

Figure 9.1 After the user logs into their account, they access a page containing forgery code. This code impersonates the user and can execute unwanted actions on behalf of the user.



How do we protect our users from such scenarios? What CSRF protection wants to ensure is that only the frontend of web applications can perform mutating operations (by convention, HTTP methods other than GET, HEAD, TRACE, or OPTIONS). Then, a foreign page, like the one in our example, can't act on behalf of the user.

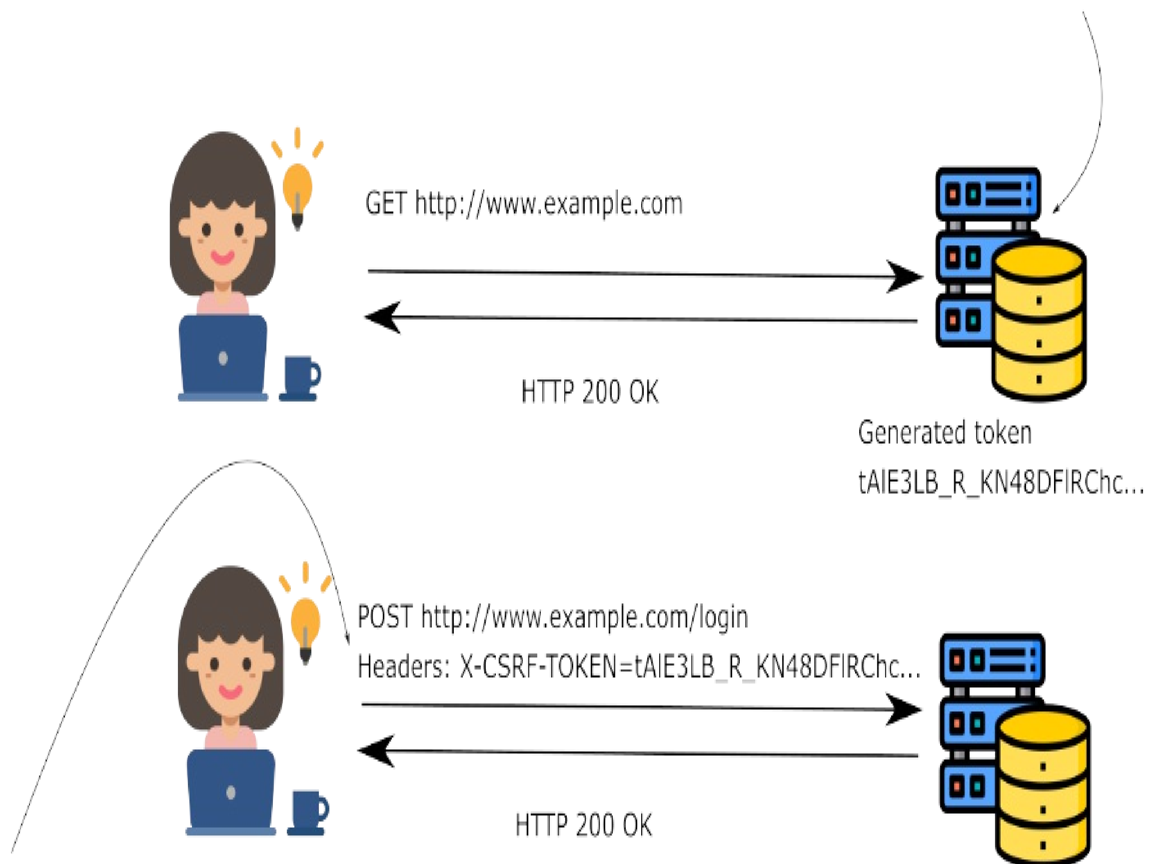
How can we achieve this? What you know for sure is that before being able to do any action that could change data, a user must send a request using HTTP GET to see the web page at least once. When this happens, the application generates a unique token. The application now accepts only requests for mutating operations (POST, PUT, DELETE, and so forth) that contain this unique value in the header.

The application considers that knowing the value of the token is proof that it is the app itself making the mutating request and not another system. Any page containing mutating calls, like POST, PUT, DELETE, and so on, should receive through the response the CSRF token, and the page must use this token when making mutating calls.

The starting point of CSRF protection is a filter in the filter chain called `CsrfFilter`. The `CsrfFilter` intercepts requests and allows all those that use these HTTP methods: GET, HEAD, TRACE, and OPTIONS. For all other requests, the filter expects to receive a header containing a token. If this header does not exist or contains an incorrect token value, the application rejects the request and sets the status of the response to HTTP 403 Forbidden.

What is this token, and where does it come from? These tokens are nothing more than string values. You have to add the token in the header of the request when you use any method other than GET, HEAD, TRACE, or OPTIONS. If you don't add the header containing the token, the application doesn't accept the request, as presented in figure 9.2.

Figure 9.2 To make a POST request, the client needs to add a header containing the CSRF token. The application generates a CSRF token when the page is loaded (via a GET request), and the token is added to all requests that can be made from the loaded page. This way, only the loaded page can make mutating requests.



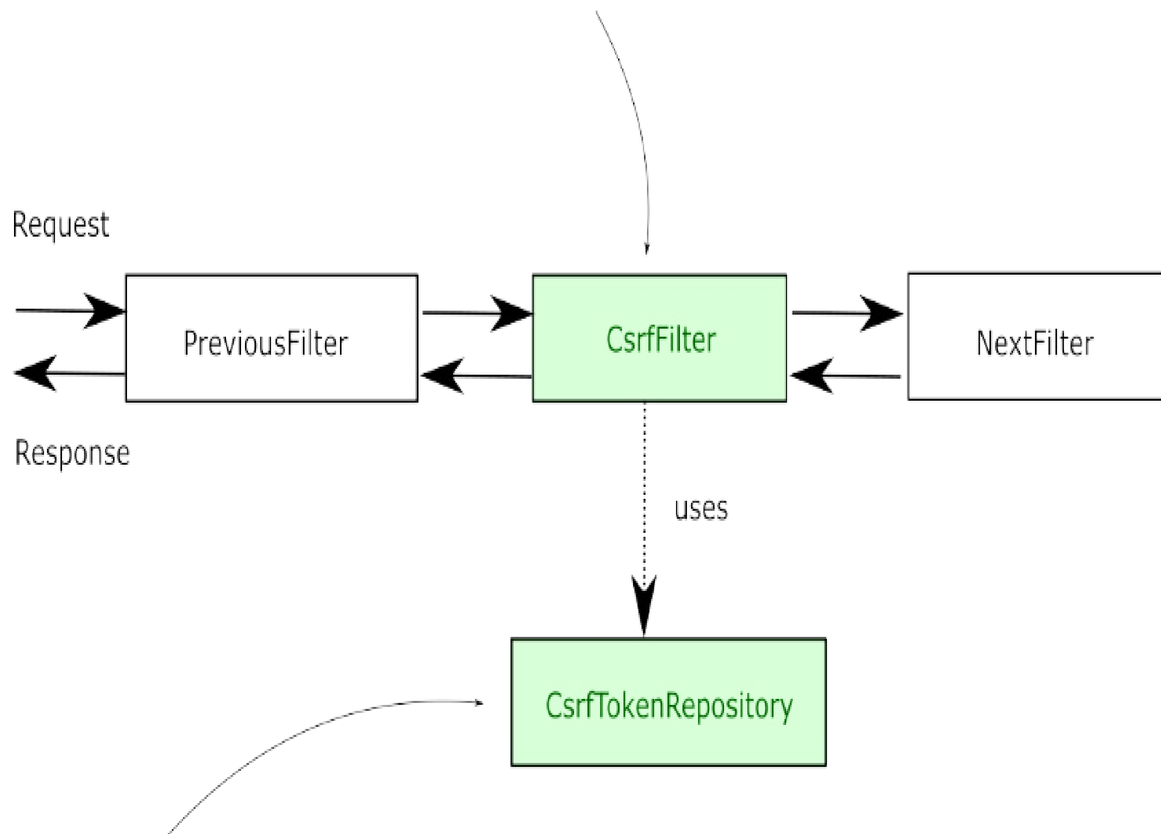
The `CsrfFilter` (figure 9.3) uses a component named `CsrfTokenRepository` to manage the CSRF token values that generate new tokens, store tokens, and eventually invalidate these. By default, the `CsrfTokenRepository` stores the token on the HTTP session and generates the tokens as random string values. In most cases, this is enough, but as you'll learn in section 9.3, you can use your own implementation of `CsrfTokenRepository` if the default one doesn't apply to the requirements you need to implement.

In this section, I explained how CSRF protection works in Spring Security with plenty of text and figures. But I want to enforce your understanding with a small code example as well. You'll find this code as part of the project named ssia-ch9-ex1. Let's create an application that exposes two endpoints.

We can call one of these with HTTP GET and the other with HTTP POST.

As you know by now, you are not able to call endpoints with POST directly without disabling CSRF protection. In this example, you learn how to call the POST endpoint without disabling CSRF protection. You need to obtain the CSRF token so that you can use it in the header of the call, which you do with HTTP POST.

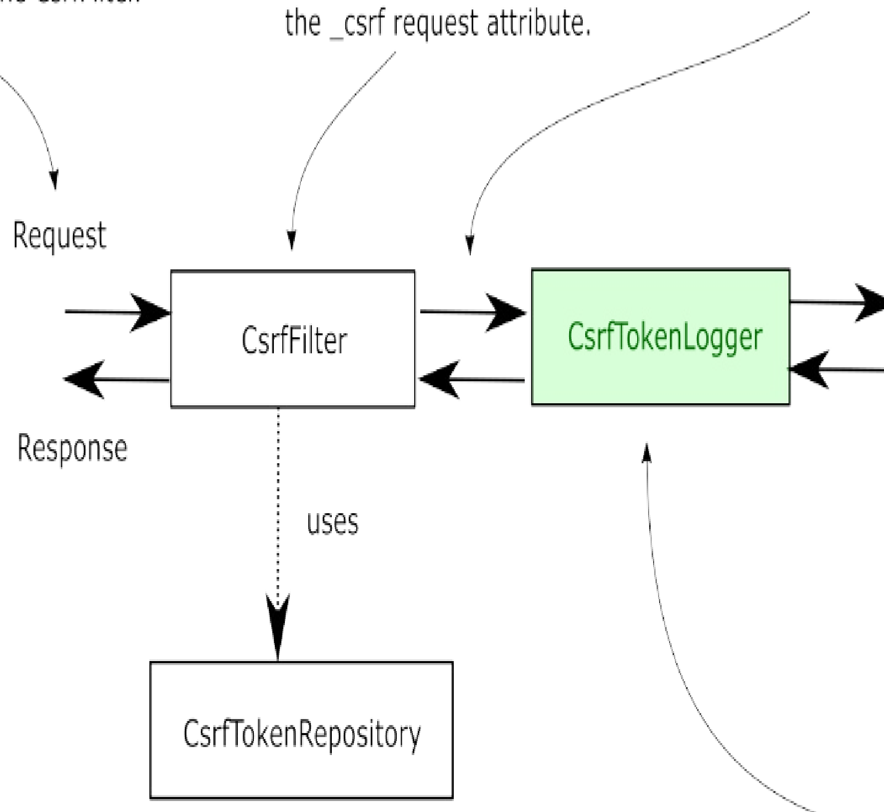
Figure 9.3 The `CsrfFilter` is one of the filters in the filter chain. It receives the request and eventually forwards it to the next filter in the chain. To manage CSRF tokens, `CsrfFilter` uses a `CsrfTokenRepository`.



As you learn with this example, the `CsrfFilter` adds the generated CSRF

token to the attribute of the HTTP request named `_csrf` (figure 9.4). If we know this, we know that after the `CsrfFilter`, we can find this attribute and take the value of the token from it. For this small application, we choose to add a custom filter after the `CsrfFilter`, as you learned in chapter 5. You use this custom filter to print in the console of the application the CSRF token that the app generates when we call the endpoint using HTTP GET. We can then copy the value of the token from the console and use it to make the mutating call with HTTP POST. In the following listing, you can find the definition of the controller class with the two endpoints that we use for a test.

Figure 9.4 Adding the `CsrfTokenLogger` (shaded) after the `CsrfFilter`. This way, the `CsrfTokenLogger` can obtain the value of the token from the `_csrf` attribute of the request where the `CsrfFilter` stores it. The `CsrfTokenLogger` prints the CSRF token in the application console, where we can access it and use it to call an endpoint with the HTTP POST method.



Listing 9.1 The controller class with two endpoints

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String getHello() {
    return "Get Hello!";
  }

  @PostMapping("/hello")
  public String postHello() {
    return "Post Hello!";
  }
```

```
}
```

Listing 9.2 defines the custom filter we use to print the value of the CSRF token in the console. I named the custom filter `CsrfTokenLogger`. When called, the filter obtains the value of the CSRF token from the `_csrf` request attribute and prints it in the console. The name of the request attribute, `_csrf`, is where the `CsrfFilter` sets the value of the generated CSRF token as an instance of the class `CsrfToken`. This instance of `CsrfToken` contains the string value of the CSRF token. You can obtain it by calling the `getToken()` method.

Listing 9.2 The definition of the custom filter class

```
public class CsrfTokenLogger implements Filter {

  private Logger logger =
          Logger.getLogger(CsrfTokenLogger.class.getName());

  @Override
  public void doFilter(
    ServletRequest request,
    ServletResponse response,
    FilterChain filterChain)
      throws IOException, ServletException {

      CsrfToken o =
       (CsrfToken) request.getAttribute("_csrf");    #A


      logger.info("CSRF token " + token.getToken());

      filterChain.doFilter(request, response);
  }
}
```

In the configuration class, we add the custom filter. The next listing presents the configuration class. Observe that I don’t disable CSRF protection in the listing.

Listing 9.3 Adding the custom filter in the configuration class

```
@Configuration
public class ProjectConfig {
```

```
  @Bean
  public SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.addFilterAfter(
            new CsrfTokenLogger(), CsrfFilter.class)
        .authorizeRequests(
            c -> c.anyRequest().permitAll()
        );


    return http.build();
  }
}
```

We can now test the endpoints. We begin by calling the endpoint with HTTP GET. Because the default implementation of the `CsrfTokenRepository` interface uses the HTTP session to store the token value on the server side, we also need to remember the session ID. For this reason, I add the `-v` flag to the call so that I can see more details from the response, including the session ID. Calling the endpoint

```
curl -v http://localhost:8080/hello
```

returns this (truncated) response:

```
...
< Set-Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206;
...
Get Hello!
```

Following the request in the application console, you can find a log line that contains the CSRF token:

```
INFO 21412 --- [nio-8080-exec-1] c.l.ssia.filters.CsrfTokenLogger
```

NOTE

You might ask yourself, how do clients get the CSRF token? They can neither guess it nor read it in the server logs. I designed this example such that it’s easier for you to understand how CSRF protection implementation

works. As you’ll find in section 9.2, the backend application has the responsibility to add the value of the CSRF token in the HTTP response to be used by the client.

If you call the endpoint using the HTTP POST method without providing the CSRF token, the response status is 403 Forbidden, as this command line shows:

```
curl -XPOST http://localhost:8080/hello
```

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

But if you provide the correct value for the CSRF token, the call is successful. You also need to specify the session ID (`JSESSIONID`) because the default implementation of `CsrfTokenRepository` stores the value of the CSRF token on the session:

```
curl -X POST   http://localhost:8080/hello
-H 'Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206'
-H 'X-CSRF-TOKEN: tAlE3LB_R_KN48DFlRChc…'
```

The response body is

```
Post Hello!
```

9.2 Using CSRF protection in practical scenarios

In this section, we discuss applying CSRF protection in practical situations. Now that you know how CSRF protection works in Spring Security, you need to know where you should use it in the real world. Which kinds of applications need to use CSRF protection?

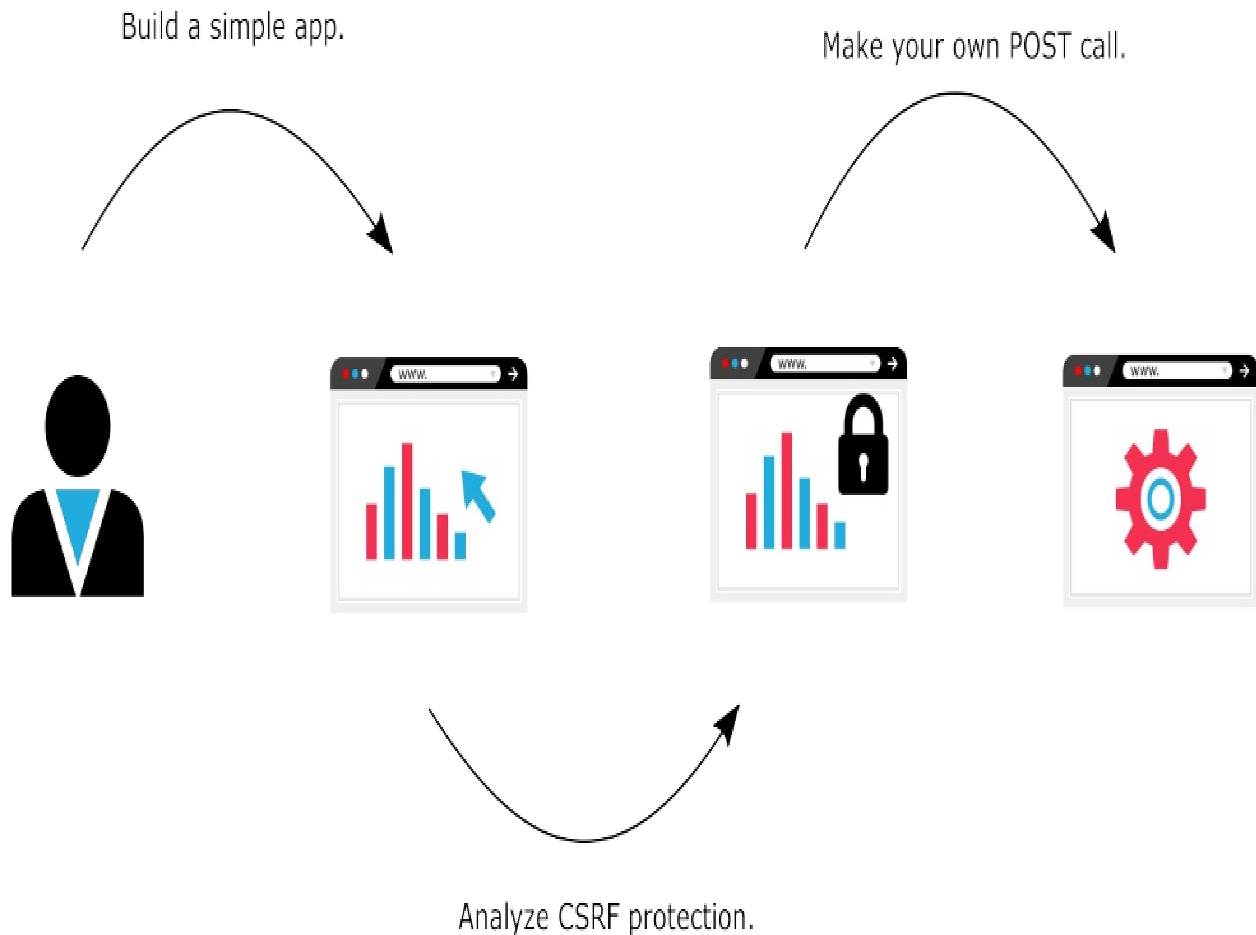
You use CSRF protection for web apps running in a browser, where you

should expect that mutating operations can be done by the browser that loads the displayed content of the app. The most basic example I can provide here is a simple web application developed on the standard Spring MVC flow. We already made such an application when discussing form login in chapter 6, and that web app actually used CSRF protection. Did you notice that the login operation in that application used HTTP POST? Then why didn't we need to do anything explicitly about CSRF in that case? The reason why we didn't observe this was because we didn't develop any mutating operation within it ourselves.

For the default form login, Spring Security correctly applies CSRF protection for us. The framework takes care of adding the CSRF token to the login request. Let's now develop a similar application to look closer at how CSRF protection works. As figure 9.5 shows, in this section we

- Build an example of a web application with the login form
- Look at how the default implementation of the login uses CSRF tokens
- Implement an HTTP POST call from the main page

Figure 9.5 The plan. In this section, we start by building and analyzing a simple app to understand how Spring Security applies CSRF protection, and then we write our own POST call.



In this example application, you’ll notice that the HTTP POST call won’t work until we correctly use the CSRF tokens, and you’ll learn how to apply the CSRF tokens in a form on such a web page. To implement this application, we start by creating a new Spring Boot project. You can find this example in the project ssia-ch9-ex2. The next code snippet presents the needed dependencies:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Then we need, of course, to configure the form login and at least one user. The following listing presents the configuration class, which defines the `UserDetailsService`, adds a user, and configures the `formLogin` method.

Listing 9.4 The definition of the configuration class

```
public class ProjectConfig {

  @Bean     #A
  public UserDetailsService uds() {
    var uds = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("mary")
                 .password("12345")
                 .authorities("READ")
                 .build();

    uds.createUser(u1);

    return uds;
  }

  @Bean     #B
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }

  @Bean      #C
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.formLogin(
      c -> c.defaultSuccessUrl("/main", true)
    );


    http.authorizeHttpRequests(
      c -> c.anyRequest().authenticated()
    );


    return http.build();
  }
}
```

We add a controller class for the main page in a package named controllers

and in a main.html file in the resources/templates folder of the Maven project. The main.html file can remain empty for the moment because on the first execution of the application, we only focus on how the login page uses the CSRF tokens. The following listing presents the `MainController` class, which serves the main page.

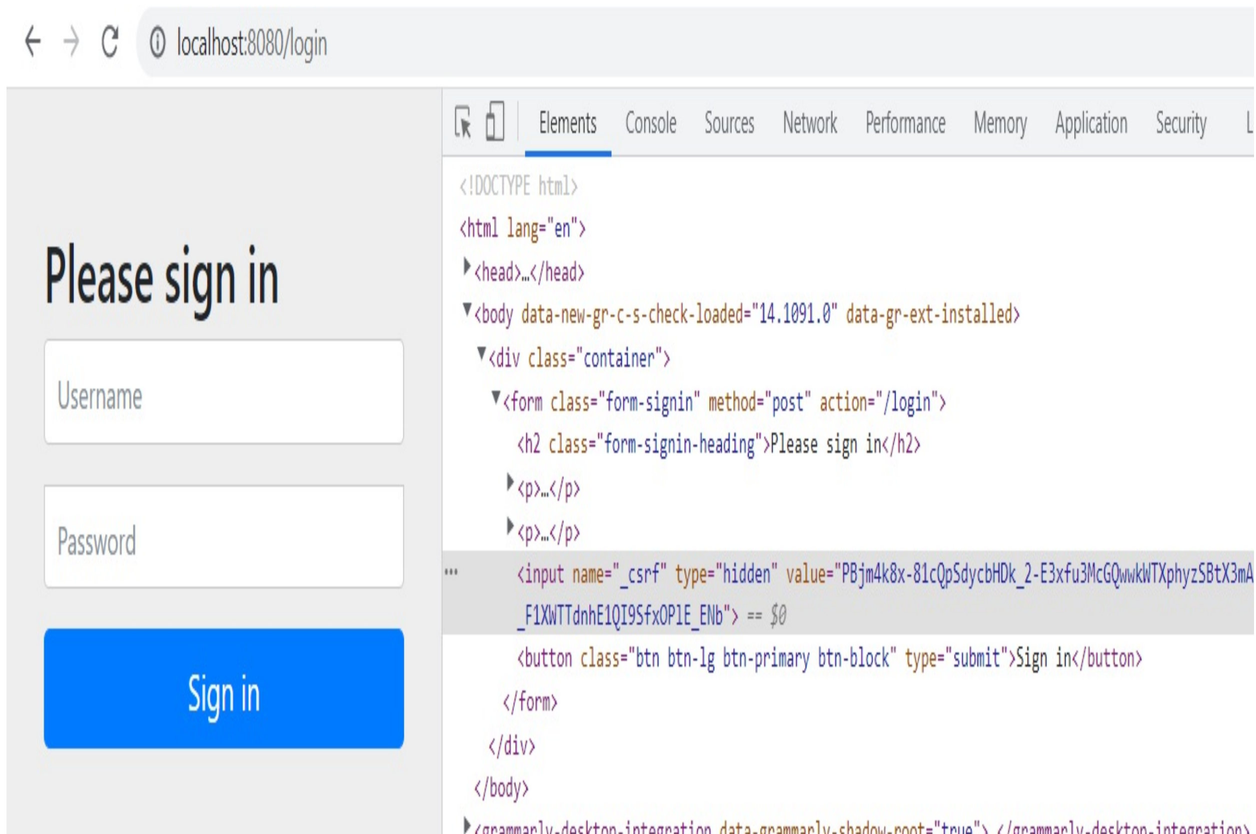
Listing 9.5 The definition of the `MainController` class

```
@Controller
public class MainController {

  @GetMapping("/main")
  public String main() {
    return "main.html";
  }
}
```

After running the application, you can access the default login page. If you inspect the form using the inspect element function of your browser, you can observe that the default implementation of the login form sends the CSRF token. This is why your login works with CSRF protection enabled even if it uses an HTTP POST request! Figure 9.6 shows how the login form sends the CSRF token through hidden input.

But what about developing our own endpoints that use POST, PUT, or DELETE as HTTP methods? For these, we have to take care of sending the value of the CSRF token if CSRF protection is enabled. To test this, let’s add an endpoint using HTTP POST to our application. We call this endpoint from the main page, and we create a second controller for this, called `ProductController`. Within this controller, we define an endpoint, /product/add, that uses HTTP POST. Further, we use a form on the main page to call this endpoint. The next listing defines the `ProductController` class.

Figure 9.6 The default form login uses a hidden input to send the CSRF token in the request. This is why the login request that uses an HTTP POST method works with CSRF protection enabled.



Listing 9.6 The definition of the `ProductController` class

```
@Controller
@RequestMapping("/product")
public class ProductController {

  private Logger logger =
    Logger.getLogger(ProductController.class.getName());

  @PostMapping("/add")
  public String add(@RequestParam String name) {
    logger.info("Adding product " + name);
    return "main.html";
  }
}
```

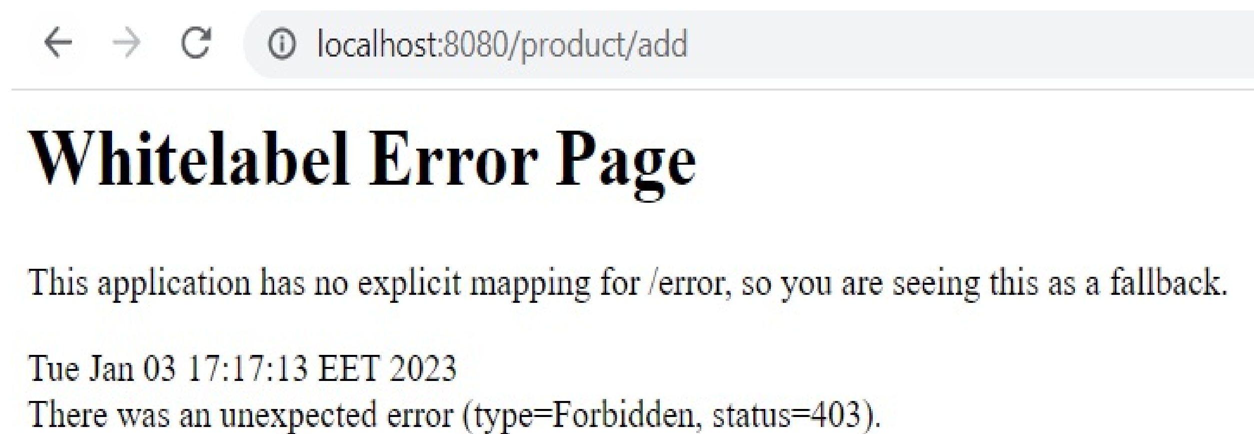
The endpoint receives a request parameter and prints it in the application console. The following listing shows the definition of the form defined in the main.html file.

Listing 9.7 The definition of the form in the main.html page

```
<form action="/product/add" method="post">
   <span>Name:</span>
   <span><input type="text" name="name" /></span>
   <span><button type="submit">Add</button></span>
</form>
```

Now you can rerun the application and test the form. What you’ll observe is that when submitting the request, a default error page is displayed, which confirms an HTTP 403 Forbidden status on the response from the server (figure 9.7). The reason for the HTTP 403 Forbidden status is the absence of the CSRF token.

Figure 9.7 Without sending the CSRF token, the server won’t accept the request done with the HTTP POST method. The application redirects the user to a default error page, which confirms that the status on the response is HTTP 403 Forbidden.



To solve this problem and make the server allow the request, we need to add the CSRF token in the request done through the form. An easy way to do this is to use a hidden input component, as you saw in the default form login. You can implement this as presented in the following listing.

Listing 9.8 Adding the CSRF token to the request done through the form

```
<form action="/product/add" method="post">
   <span>Name:</span>
   <span><input type="text" name="name" /></span>
   <span><button type="submit">Add</button></span>

   <input type="hidden"      #A
```

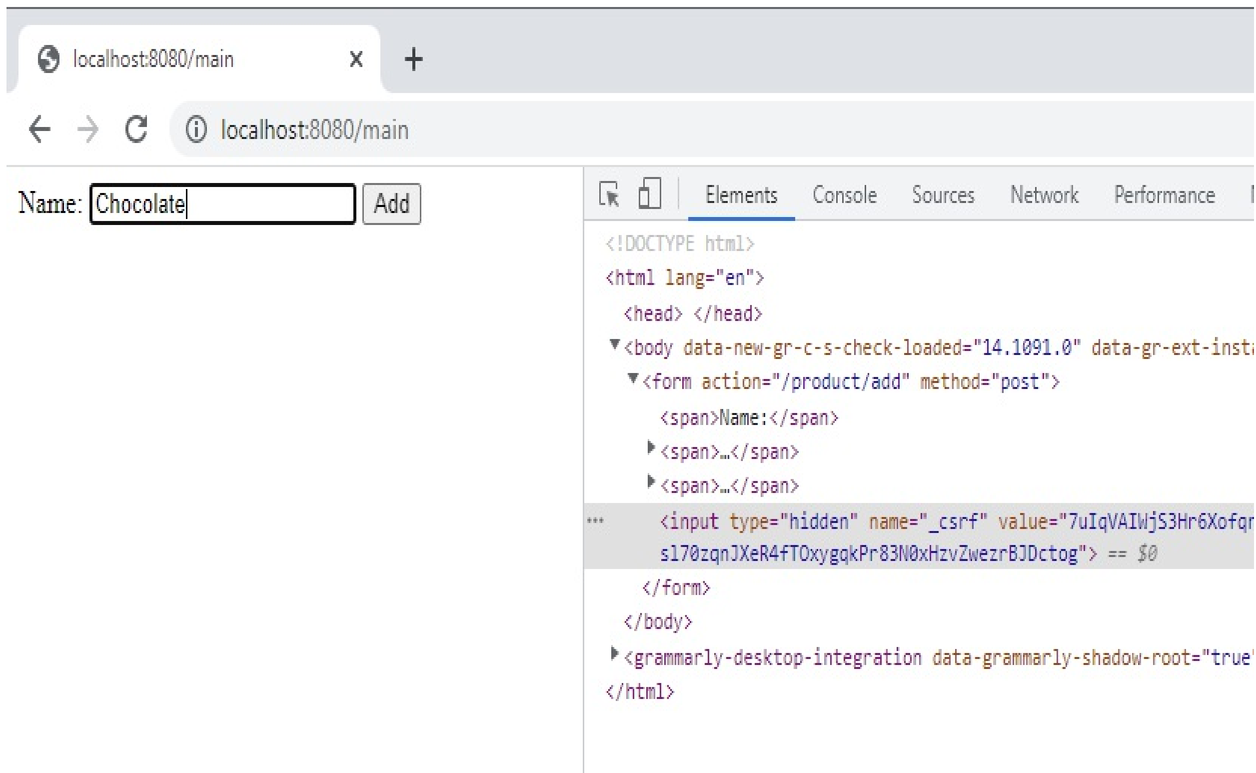
```
          th:name="${_csrf.parameterName}"  #B
          th:value="${_csrf.token}" />      #B
</form>
```

NOTE

In the example, we use Thymeleaf because it provides a straightforward way to obtain the request attribute value in the view. In our case, we need to print the CSRF token. Remember that the `CsrfFilter` adds the value of the token in the `_csrf` attribute of the request. It’s not mandatory to do this with Thymeleaf. You can use any alternative of your choice to print the token value to the response.

After rerunning the application, you can test the form again. This time the server accepts the request, and the application prints the log line in the console, proving that the execution succeeds. Also, if you inspect the form, you can find the hidden input with the value of the CSRF token (figure 9.8).

Figure 9.8 The form defined on the main page now sends the value for the CSRF token in the request. This way, the server allows the request and executes the controller action. In the source code for the page, you can now find the hidden input used by the form to send the CSRF token in the request.



After submitting the form, you should find in the application console a line similar to this one:

```
INFO 20892 --- [nio-8080-exec-7] c.l.s.controllers.ProductControl
```

Of course, for any action or asynchronous JavaScript request your page uses to call a mutating action, you need to send a valid CSRF token. This is the most common way used by an application to make sure the request doesn’t come from a third party. A third-party request could try to impersonate the user to execute actions on their behalf.

CSRF tokens work well in an architecture where the same server is responsible for both the frontend and the backend, mainly for its simplicity. But CSRF tokens don’t work well when the client is independent of the backend solution it consumes. This scenario happens when you have a mobile application as a client or a web frontend developed independently. A web client developed with a framework like Angular, ReactJS, or Vue.js is ubiquitous in web application architectures, and this is why you need to know how to implement the security approach for these cases as well. We’ll discuss these kinds of designs in part 4 of this book.

In chapters 13 through 16, you'll learn to implement the OAuth 2 specification, which has excellent advantages in decoupling the component. This makes the authentication from the resources for which the application authorizes the client.

NOTE

It might look like a trivial mistake, but in my experience, I see it too many times in applications—never use HTTP GET with mutating operations! Do not implement behavior that changes data and allows it to be called with an HTTP GET endpoint. Remember that calls to HTTP GET endpoints don't require a CSRF token.

9.3 Customizing CSRF protection

In this section, you learn how to customize the CSRF protection solution that Spring Security offers. Because applications have various requirements, any implementation provided by a framework needs to be flexible enough to be easily adapted to different scenarios. The CSRF protection mechanism in Spring Security is no exception. In this section, the examples let you apply the most frequently encountered needs in the customization of the CSRF protection mechanism. These are

- Configuring paths for which CSRF applies
- Managing CSRF tokens

We use CSRF protection only when the page that consumes resources produced by the server is itself generated by the same server. It can be a web application where the consumed endpoints are exposed by a different origin, as we discussed in section 9.2, or a mobile application. In the case of mobile applications, you can use the OAuth 2 flow, which we'll discuss in chapters 13 through 16.

By default, CSRF protection applies to any path for endpoints called with HTTP methods other than GET, HEAD, TRACE, or OPTIONS. You already know from chapter 5 how to completely disable CSRF protection. But what if you want to disable it only for some of your application paths? You can do

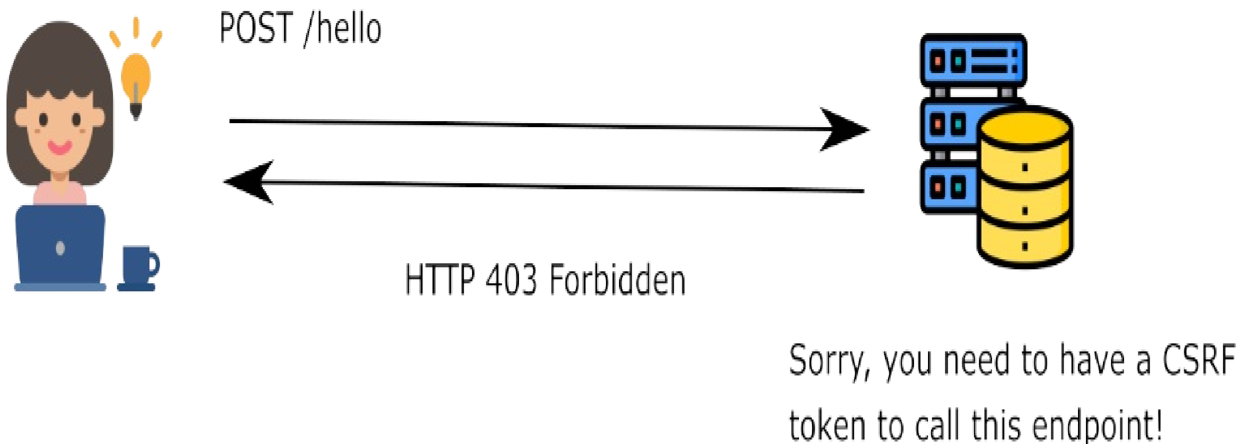
this configuration quickly with a `Customizer` object, similar to the way we customized HTTP Basic for form-login methods in chapter 6. Let's try this with an example.

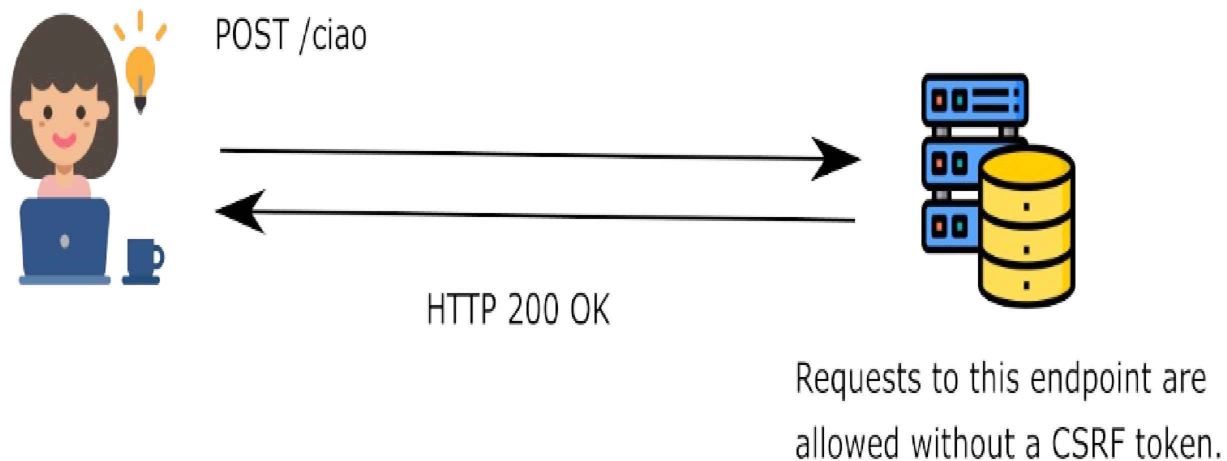
In our example, we create a new project and add only the web and security dependencies as presented in the next code snippet. You can find this example in the project ssia-ch9-ex3. Here are the dependencies:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

In this application, we add two endpoints called with HTTP POST, but we want to exclude one of these from using CSRF protection (figure 9.9). Listing 9.9 defines the controller class for this, which I name `HelloController`.

Figure 9.9 The application requires a CSRF token for the /hello endpoint called with HTTP POST but allows HTTP POST requests to the /ciao endpoint without a CSRF token.





Listing 9.9 The definition of the `HelloController` class

```
@RestController
public class HelloController {

  @PostMapping("/hello")     #A
  public String postHello() {
    return "Post Hello!";
  }

  @PostMapping("/ciao")      #B
  public String postCiao() {
    return "Post Ciao";
  }
}
```

To make customizations on CSRF protection, you can use the `csrf()` method of the `HttpSecurity` object in the `configuration()` method with a `Customizer` object. The next listing presents this approach.

Listing 9.10 A `Customizer` object for the configuration of CSRF protection

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.csrf(c -> {     #A
        c.ignoringRequestMatchers("/ciao");
    });

    http.authorizeRequests(
         c -> c.anyRequest().permitAll()
    );

    return http.build();
}
```

Calling the `ignoringRequestMatchers(String paths)` method, you can specify the path expressions representing the paths that you want to exclude from the CSRF protection mechanism. A more general approach is to use a `RequestMatcher`. Using this allows you to apply the exclusion rules with regular path expressions as well as with regexes (regular expressions). When using the `ignoringRequestMatchers()` method of the `CsrfCustomizer` object, you can provide any `RequestMatcher` as a parameter. The next code snippet shows how to use the `ignoringRequestMatchers()` method with an `MvcRequestMatcher` instead of using `ignoringRequestMatchers()` with a path given as a `String` value:

```
HandlerMappingIntrospector i = new HandlerMappingIntrospector();
MvcRequestMatcher r = new MvcRequestMatcher(i, "/ciao");
c.ignoringRequestMatchers(r);
```

Or, you can similarly use a regex matcher as in the next code snippet:

```
String pattern = ".*[0-9].*";
```

```
String httpMethod = HttpMethod.POST.name();
RegexRequestMatcher r = new RegexRequestMatcher(pattern, httpMeth
c.ignoringRequestMatchers(r);
```

Another need often found in the requirements of the application is customizing the management of CSRF tokens. As you learned, by default, the application stores CSRF tokens in the HTTP session on the server side. This simple approach is suitable for small applications, but it’s not great for applications that serve a large number of requests and that require horizontal scaling. The HTTP session is stateful and reduces the scalability of the application.

Let’s suppose you want to change the way the application manages tokens and store them somewhere in a database rather than in the HTTP session. Spring Security offers three contracts that you need to implement to do this:

- `CsrfToken`—Describes the CSRF token itself
- `CsrfTokenRepository`—Describes the object that creates, stores, and loads CSRF tokens
- `CsrfTokenRequestHandler` – Describes and object that manages the way in which the generated CSRF token is set on the HTTP request.

The `CsrfToken` object has three main characteristics that you need to specify when implementing the contract (listing 9.11 defines the `CsrfToken` contract):

- The name of the header in the request that contains the value of the CSRF token (default named `X-CSRF-TOKEN`)
- The name of the attribute of the request that stores the value of the token (default named `_csrf`)
- The value of the token

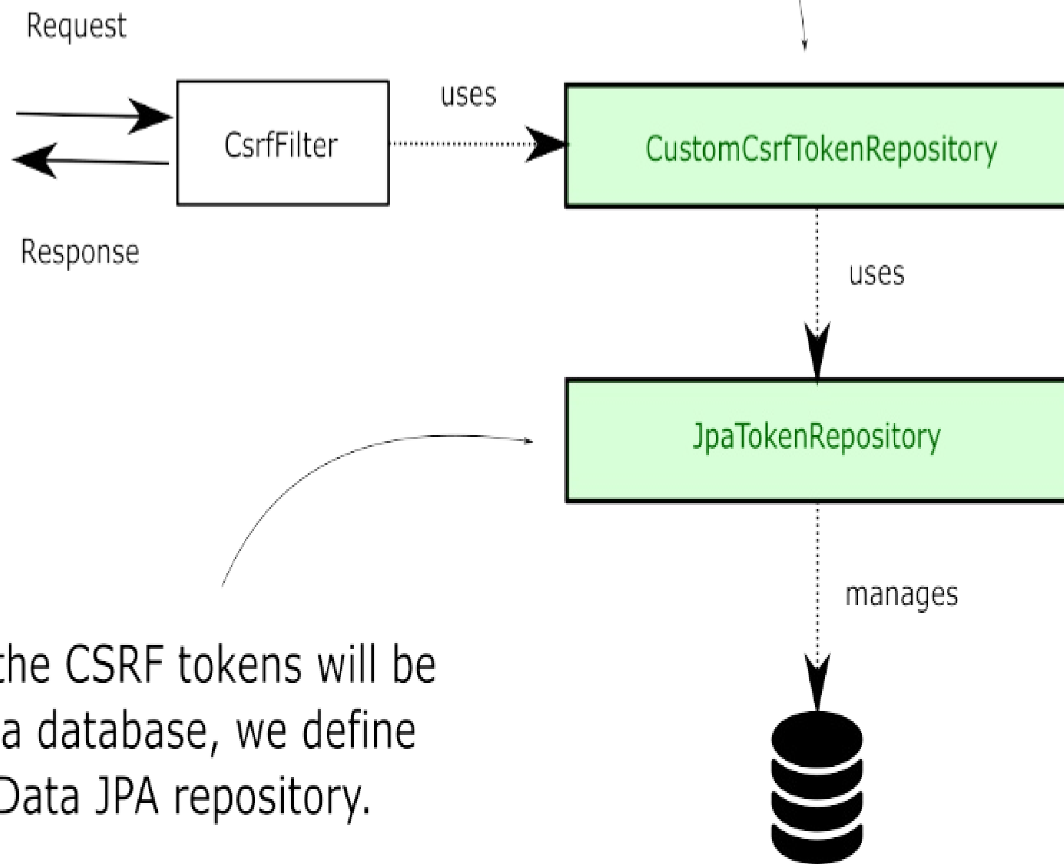
Listing 9.11 The definition of the `CsrfToken` interface

```
public interface CsrfToken extends Serializable {

  String getHeaderName();
  String getParameterName();
  String getToken();
}
```

Generally, you only need the instance of the `CsrfToken` type to store the three details in the attributes of the instance. For this functionality, Spring Security offers an implementation called `DefaultCsrfToken` that we also use in our example. `DefaultCsrfToken` implements the `CsrfToken` contract and creates immutable instances containing the required values: the name of the request attribute and header, and the token itself.

The interface `CsrfTokenRepository` is the contract that represents the component that manages CSRF tokens. To change the way the application manages the tokens, you need to implement the `CsrfTokenRepository` interface, which allows you to plug your custom implementation into the framework. Let’s change the current application we use in this section to add a new implementation for `CsrfTokenRepository`, which stores the tokens in a database. Figure 9.10 presents the components we implement for this example and the link between them.

Figure 9.10 The `CsrfToken` uses a custom implementation of `CsrfTokenRepository`. This custom implementation uses a `JpaRepository` to manage CSRF tokens in a database.



In our example, we use a table in a database to store CSRF tokens. We assume the client has an ID to identify themselves uniquely. The application needs this identifier to obtain the CSRF token and validate it. Generally, this unique ID would be obtained during login and should be different each time the user logs in. This strategy of managing tokens is similar to storing them in memory. In this case, you use a session ID. So the new identifier for this example merely replaces the session ID.

An alternative to this approach would be to use CSRF tokens with a defined lifetime. With such an approach, tokens expire after a time you define. You can store tokens in the database without linking them to a specific user ID. You only need to check if a token provided via an HTTP request exists and is

not expired to decide whether you allow that request.

Exercise

Once you finish with this example where we use an identifier to which we assign the CSRF token, implement the second approach where you use CSRF tokens that expire.

To make our example shorter, we only focus on the implementation of the `CsrfTokenRepository`, and we need to consider that the client already has a generated identifier. To work with the database, we need to add a couple more dependencies to the pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
</dependency>
```

In the application.properties file, we need to add the properties for the database connection:

```
spring.datasource.url=jdbc:mysql://localhost/spring?useLegacyDate
spring.datasource.username=root
spring.datasource.password=
spring.sql.init.mode=always
```

To allow the application to create the needed table in the database at the start, you can add the schema.xml file in the resources folder of the project. This file should contain the query for creating the table, as presented by this code snippet:

```
CREATE TABLE IF NOT EXISTS `spring`.`token` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `identifier` VARCHAR(45) NULL,
    `token` TEXT NULL,
PRIMARY KEY (`id`));
```

We use Spring Data with a JPA implementation to connect to the database, so

we need to define the entity class and the `JpaRepository` class. In a package named entities, we define the JPA entity as presented in the following listing.

Listing 9.12 The definition of the JPA entity class

```
@Entity
public class Token {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  private String identifier;   #A
  private String token;    #B

  // Omitted code

}
```

The `JpaTokenRepository`, which is our `JpaRepository` contract, can be defined as shown in the following listing. The only method you need is `findTokenByIdentifier()`, which gets the CSRF token from the database for a specific client.

Listing 9.13 The definition of the `JpaTokenRepository` interface

```
public interface JpaTokenRepository
  extends JpaRepository<Token, Integer> {

  Optional<Token> findTokenByIdentifier(String identifier);
}
```

With access to the implemented database, we can now start writing the `CsrfTokenRepository` implementation, which I call `CustomCsrfTokenRepository`. The next listing defines this class, which overrides the three methods of `CsrfTokenRepository`.

Listing 9.14 The implementation of the `CsrfTokenRepository` contract

```
@Component
public class CustomCsrfTokenRepository implements CsrfTokenReposi
```

```
  private final JpaTokenRepository jpaTokenRepository;

  // Omitted constructor

  @Override
  public CsrfToken generateToken(
    HttpServletRequest httpServletRequest) {
    // ...
  }

  @Override
  public void saveToken(
    CsrfToken csrfToken,
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) {
    // ...
  }

  @Override
  public CsrfToken loadToken(
    HttpServletRequest httpServletRequest) {
    // ...
  }
}
```

`CustomCsrfTokenRepository` injects an instance of `JpaTokenRepository` from the Spring context to gain access to the database. `CustomCsrfTokenRepository` uses this instance to retrieve or to save the CSRF tokens in the database. The CSRF protection mechanism calls the `generateToken()` method when the application needs to generate a new token. In listing 9.15, you find the implementation of this method for our exercise. We use the `UUID` class to generate a new random UUID value, and we keep the same names for the request header and attribute, `X-CSRF-TOKEN` and `_csrf`, as in the default implementation offered by Spring Security.

Listing 9.15 The implementation of the `generateToken()` method

```
@Override
public CsrfToken generateToken(HttpServletRequest httpServletRequ
  String uuid = UUID.randomUUID().toString();
  return new DefaultCsrfToken("X-CSRF-TOKEN", "_csrf", uuid);
}
```

The `saveToken()` method saves a generated token for a specific client. In the

case of the default CSRF protection implementation, the application uses the HTTP session to identify the CSRF token. In our case, we assume that the client has a unique identifier. The client sends the value of its unique ID in the request with the header named `X-IDENTIFIER`. In the method logic, we check whether the value exists in the database. If it exists, we update the database with the new value of the token. If not, we create a new record for this ID with the new value of the CSRF token. The following listing presents the implementation of the `saveToken()` method.

Listing 9.16 The implementation of the `saveToken()` method

```
@Override
public void saveToken(
   CsrfToken csrfToken,
   HttpServletRequest httpServletRequest,
   HttpServletResponse httpServletResponse) {
    String identifier =
        httpServletRequest.getHeader("X-IDENTIFIER");

    Optional<Token> existingToken =      #A
       jpaTokenRepository.findTokenByIdentifier(identifier);

    if (existingToken.isPresent()) {      #B
       Token token = existingToken.get();
       token.setToken(csrfToken.getToken());
    } else {      #C
       Token token = new Token();
       token.setToken(csrfToken.getToken());
       token.setIdentifier(identifier);
       jpaTokenRepository.save(token);
    }
}
```

The `loadToken()` method implementation loads the token details, if these exist, or returns null, otherwise. The following listing shows this implementation.

Listing 9.17 The implementation of the `loadToken()` method

```
@Override
public CsrfToken loadToken(
  HttpServletRequest httpServletRequest) {
```

```
  String identifier = httpServletRequest.getHeader("X-IDENTIFIER"

  Optional<Token> existingToken =
    jpaTokenRepository
      .findTokenByIdentifier(identifier);

  if (existingToken.isPresent()) {
    Token token = existingToken.get();
    return new DefaultCsrfToken(
                  "X-CSRF-TOKEN",
                  "_csrf",
                  token.getToken());
  }

  return null;
}
```

We use a custom implementation of the `CsrfTokenRepository` to declare a bean in the configuration class. We then plug the bean into the CSRF protection mechanism with the `csrfTokenRepository()` method of `CsrfConfigurer`. The next listing defines this configuration class.

Listing 9.18 The configuration class for the custom `CsrfTokenRepository`

```
@Configuration
public class ProjectConfig {

  private final CustomCsrfTokenRepository customTokenRepository;

  // Omitted constructor

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.csrf(c -> {
      c.csrfTokenRepository(customTokenRepository);    #A
    });

    http.authorizeHttpRequests(
      c -> c.anyRequest().permitAll()
    );

    return http.build();
  }
```

```
}
```

The last piece we need to plug in for everything to work fine is a `CsrfTokenRequestHandler`. Fortunately, we can use an implementation that Spring Security provides – the `CsrfTokenRequestAttributeHandler`. This implementation simply uses the `generateToken()` method of the `CsrfTokenRepository` to generate a new token when an endpoint is called using the HTTP GET method. It then add the generated `CrsfToken` on the request as an attribute.

You can customize the simple behavior of the `CsrfTokenRequestAttributeHandler` object by extending its class. For example, the default implementation Spring Security uses (named `XorCsrfTokenRequestAttributeHandler`) has a more complex behavior. This implementation generates a random value using a `SecuredRandom` object and then mixes its byte array with the token generated by the `CsrfTokenRepository` using an XOR logic operation.

But to avoid adding to much complexity to our example and allow you focus on the configuration part, we will set up a simple `CsrfTokenRequestAttributeHandler` to handle the management of the CSRF token on the HTTP request object. Listing 9.19 shows you how to configure the `CsrfTokenRequestAttributeHandler` in the configuration class.

Listing 9.19 The configuration class for the custom `CsrfTokenRepository`

```
@Configuration
public class ProjectConfig {

  private final CustomCsrfTokenRepository customTokenRepository;

  // Omitted constructor

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.csrf(c -> {
      c.csrfTokenRepository(customTokenRepository);
      c.csrfTokenRequestHandler(     #A
```

```
        new CsrfTokenRequestAttributeHandler()    #A
      );     #A
    });

    http.authorizeHttpRequests(
      c -> c.anyRequest().permitAll()
    );

    return http.build();
  }

}
```

In the definition of the controller class presented in listing 9.9, we also add an endpoint that uses the HTTP GET method. We need this method to obtain the CSRF token when testing our implementation:

```
@GetMapping("/hello")
public String getHello() {
  return "Get Hello!";
}
```

You can now start the application and test the new implementation for managing the token. We call the endpoint using HTTP GET to obtain a value for the CSRF token. When making the call, we have to use the client's ID within the `X-IDENTIFIER` header, as assumed from the requirement. A new value of the CSRF token is generated and stored in the database. Here's the call:

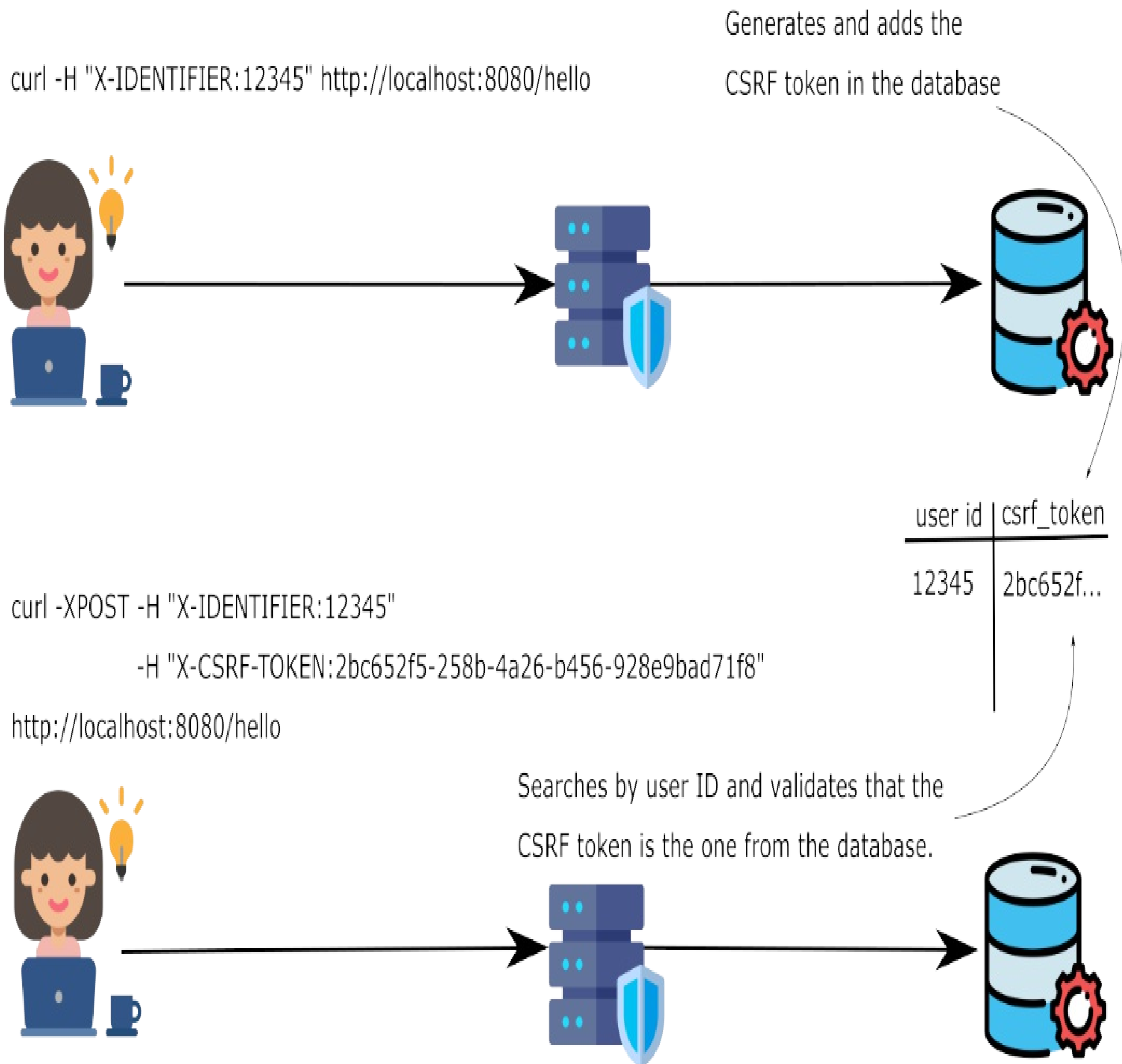
```
curl -H "X-IDENTIFIER:12345" http://localhost:8080/hello
Get Hello!
```

If you search the token table in the database, you find that the application added a new record for the client with identifier 12345. In my case, the generated value for the CSRF token, which I can see in the database, is `2bc652f5-258b-4a26-b456-928e9bad71f8`. We use this value to call the /hello endpoint with the HTTP POST method, as the next code snippet presents. Of course, we also have to provide the client ID that's used by the application to retrieve the token from the database to compare with the one we provide in the request. Figure 9.11 describes the flow.

```
curl -XPOST -H "X-IDENTIFIER:12345" -H "X-CSRF-TOKEN:2bc652f5-258
```

```
Post Hello!
```

Figure 9.11 First, the GET request generates the CSRF token and stores its value in the database. Any following POST request must send this value. Then, the `CsrfFilter` checks if the value in the request corresponds with the one in the database. Based on this, the request is accepted or rejected.



If we try to call the /hello endpoint with POST without providing the needed headers, we get a response back with the HTTP status 403 Forbidden. To confirm this, call the endpoint with

```
curl -XPOST http://localhost:8080/hello
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

9.4 Summary

- A cross-site request forgery (CSRF) is a type of attack where the user is tricked into accessing a page containing a forgery script. This script can impersonate a user logged into an application and execute actions on their behalf.
- CSRF protection is by default enabled in Spring Security.
- The entry point of CSRF protection logic in the Spring Security architecture is an HTTP filter.
- You can customize the capability that offers CSRF protection. Spring Security offers three simple contracts that you can implement and plug in to define custom CSRF protection capabilities:
  - `CsrfToken`—Describes the CSRF token itself
  - `CsrfTokenRepository`—Describes the object that creates, stores, and loads CSRF tokens
  - `CsrfTokenRequestHandler` – Describes and object that manages the way in which the generated CSRF token is set on the HTTP request.

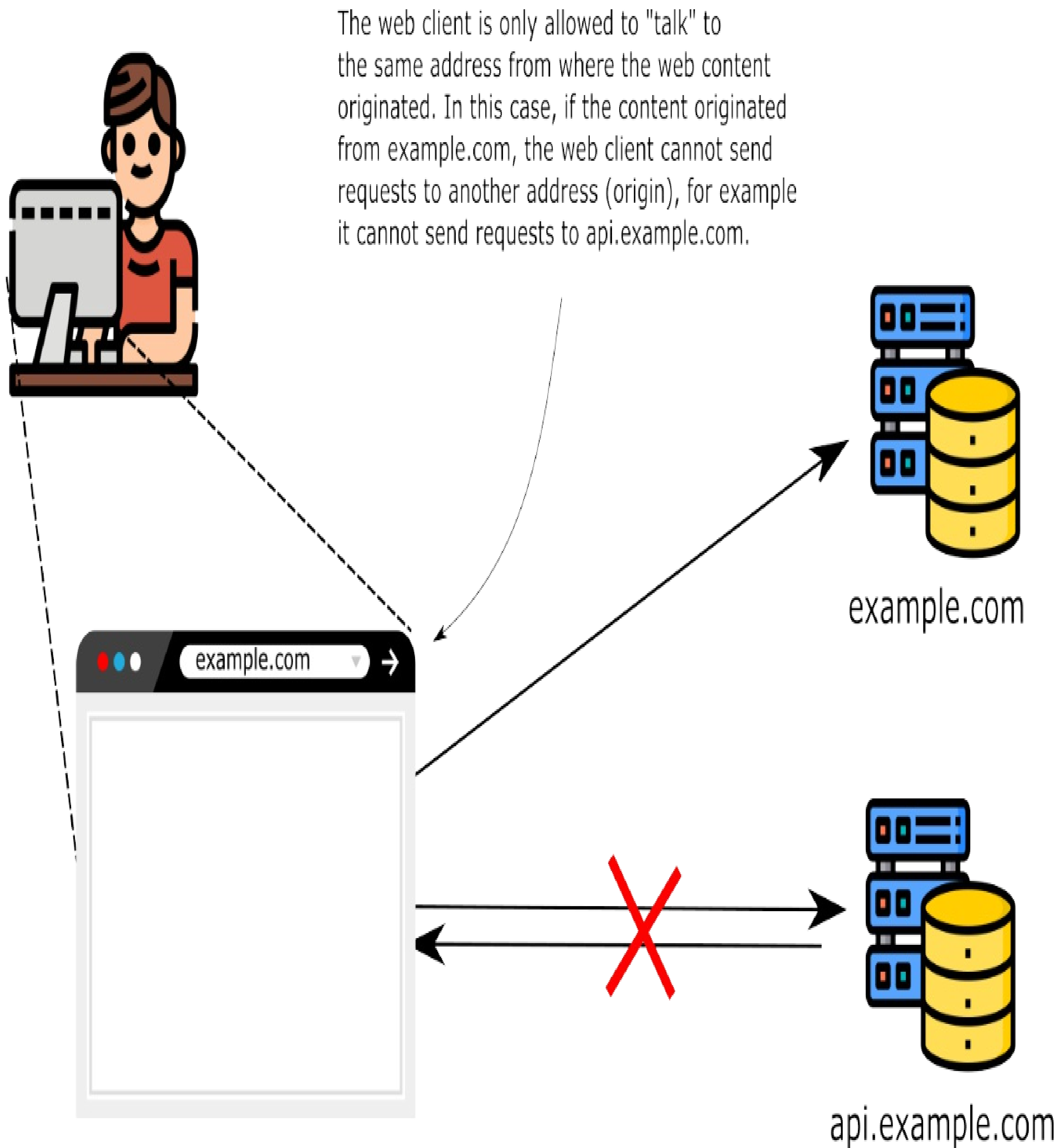
10 Configuring Cross-Origin Resource Sharing (CORS)

This chapter covers

- What is cross-origin resource sharing (CORS)
- Applying cross-origin resource sharing configurations

In this chapter, we discuss cross-origin resource sharing (CORS) and how to apply it with Spring Security. First, what is CORS and why should you care? The necessity for CORS came from web applications. By default, browsers don't allow requests made for any domain other than the one from which the site is loaded. For example, if you access the site from example.com, the browser won't let the site make requests to api.example.com. Figure 10.1 shows this concept.

Figure 10.1 Cross-origin resource sharing (CORS). When accessed from example.com, the website cannot make requests to api.example.com because they would be cross-domain requests.



We can briefly say that an app uses the CORS mechanism to relax this strict policy and allow requests made between different origins in some conditions. You need to know this because it's likely you will have to apply it to your applications, especially nowadays where the frontend and backend are separate applications. It is common that a frontend application is developed using a framework like Angular, ReactJS, or Vue and hosted at a domain like example.com, but it calls endpoints on the backend hosted at another domain

like api.example.com.

For this chapter, we develop some examples from which you can learn how to apply CORS policies for your web applications. We also describe some details that you need to know such that you avoid leaving security breaches in your applications.

10.1 How does CORS work?

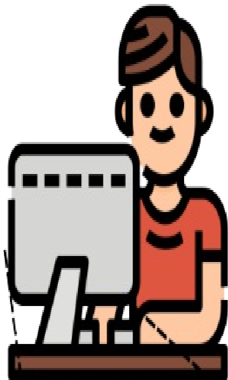
In this section, we discuss how CORS applies to web applications. If you are the owner of example.com, for example, and for some reason the developers from example.org decide to call your REST endpoints (api.example.com) from their website, they won't be able to. The same situation can happen if a domain loads your application using an `iframe`, for example (see figure 10.2).

NOTE

An `iframe` is an HTML element that you use to embed content generated by a web page into another web page (for example, to integrate the content from example.org inside a page from example.com).

Any situation in which an application makes calls between two different domains is prohibited. But, of course, you can find cases in which you need to make such calls. In these situations, CORS allows you to specify from which domain your application allows requests and what details can be shared. The CORS mechanism works based on HTTP headers (figure 10.3). The most important are

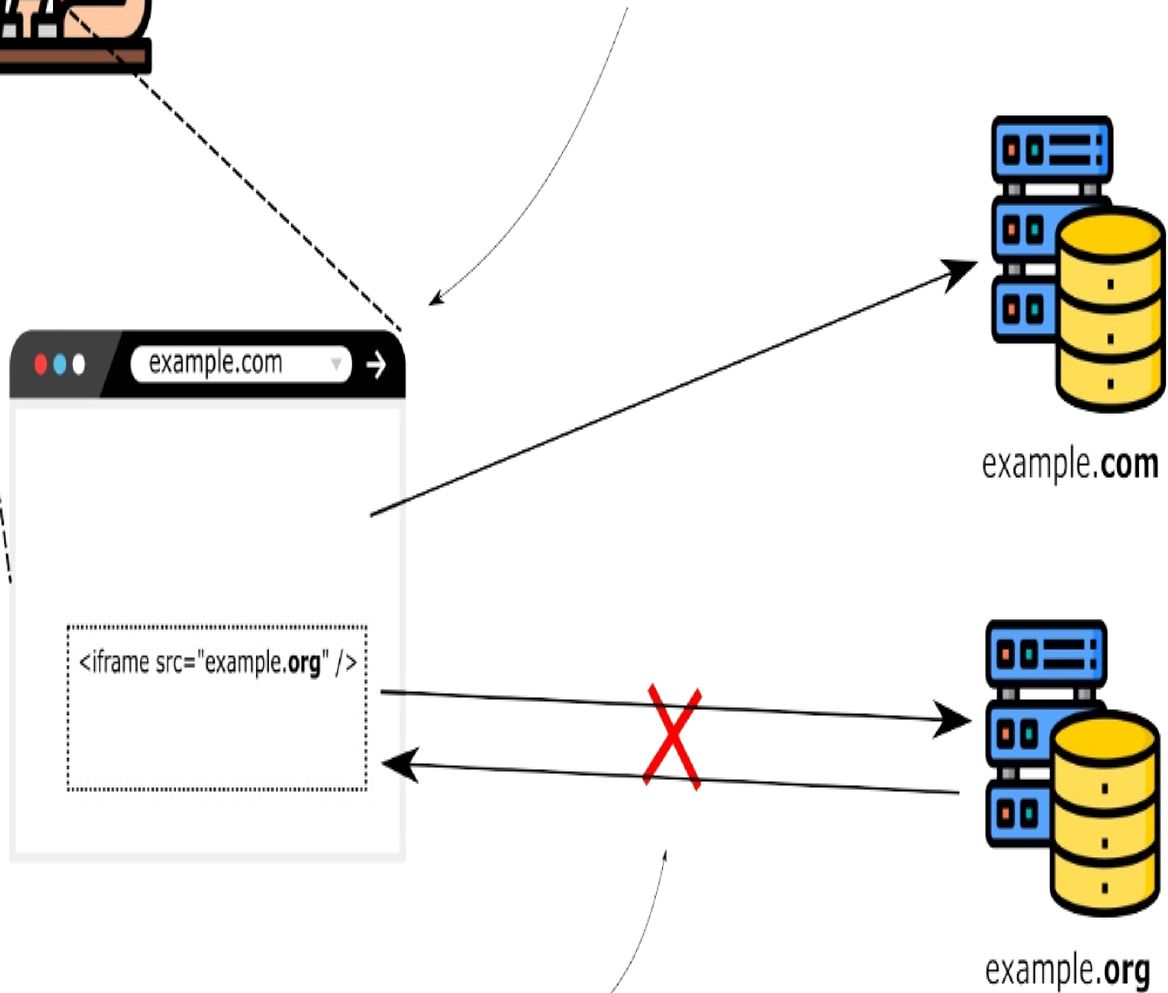
Figure 10.2 Even if the example.org page is loaded in an `iframe` from the example.com domain, the calls from the content loaded in example.org won't load. Even if the application makes a request, the browser won't accept the response.



You have opened example.**com**,

but the page uses an <iframe />

to nest content from example.**org**.



The browser doesn't allow the content

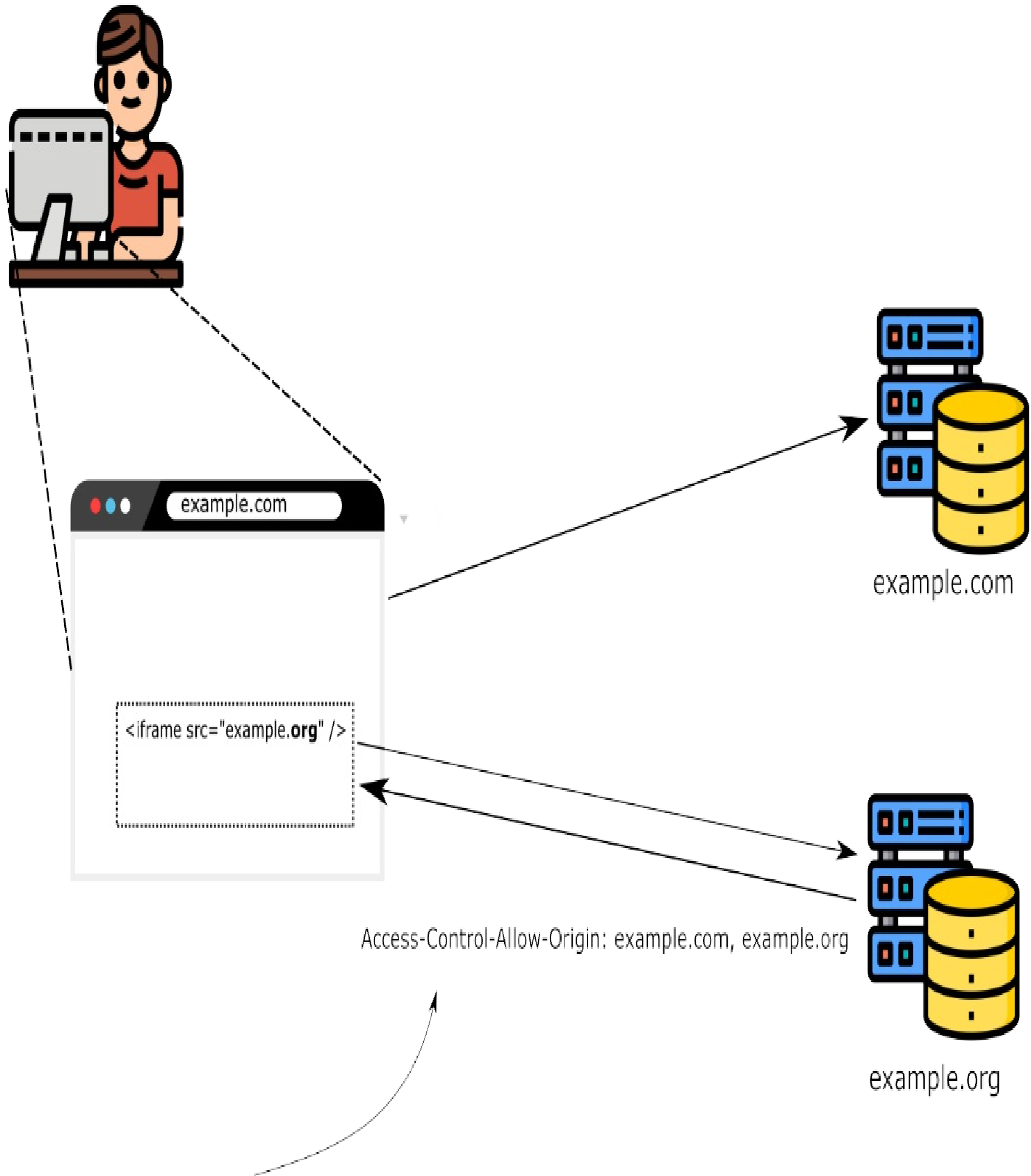
from example.org because it is cross-domain.

- `Access-Control-Allow-Origin`—Specifies the foreign domains (origins) that can access resources on your domain.
- `Access-Control-Allow-Methods`—Lets us refer only to some HTTP methods in situations in which we want to allow access to a different domain, but only to specific HTTP methods. You use this if you’re

going to enable example.com to call some endpoint, but only with HTTP GET, for example.

- `Access-Control-Allow-Headers`—Adds limitations to which headers you can use in a specific request. For example, you don't want the client to be able to send a specific header for a given request.

Figure 10.3 Enabling cross-origin requests. The example.org server adds the `Access-Control-Allow-Origin` header to specify the origins of the request for which the browser should accept the response. If the domain from where the call was made is enumerated in the origins, the browser accepts the response.



With Spring Security, by default, none of these headers are added to the response. So let’s start at the beginning: what happens when you make a

cross-origin call if you don't configure CORS in your application. When the application makes the request, it expects that the response has an `Access-Control-Allow-Origin` header containing the origins accepted by the server. If this doesn't happen, as in the case of default Spring Security behavior, the browser won't accept the response. Let's demonstrate this with a small web application. We create a new project using the dependencies presented by the next code snippet. You can find this example in the project ssia-ch10-ex1.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

We define a controller class having an action for the main page and a REST endpoint. Because the class is a normal Spring MVC `@Controller` class, we also have to add the `@ResponseBody` annotation explicitly to the endpoint. The following listing defines the controller.

Listing 10.1 The definition of the controller class

```
@Controller
public class MainController {

  private Logger logger =      #A
    Logger.getLogger(MainController.class.getName());

  @GetMapping("/")   #B
  public String main() {
    return "main.html";
  }

  @PostMapping("/test")
  @ResponseBody
  public String test() {    #C
    logger.info("Test method called");
```

```
    return "HELLO";
  }
}
```

Further, we need to define the configuration class where we disable CSRF protection to make the example simpler and allow you to focus only on the CORS mechanism. Also, we allow unauthenticated access to all endpoints. The next listing defines this configuration class.

Listing 10.2 The definition of the configuration class

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

    http.csrf(
      c -> c.disable()
    );

    http.authorizeHttpRequests(
       c -> c.anyRequest().permitAll()
    );


    return http.build();
  }
}
```

Of course, we also need to define the main.html file in the resources/templates folder of the project. The main.html file contains the JavaScript code that calls the /test endpoint. To simulate the cross-origin call, we can access the page in a browser using the domain localhost. From the JavaScript code, we make the call using the IP address 127.0.0.1. Even if localhost and 127.0.0.1 refer to the same host, the browser sees these as different strings and considers these different domains. The next listing defines the main.html page.

Listing 10.3 The main.html page

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <script>
      const http = new XMLHttpRequest();
      const url='http://127.0.0.1:8080/test';    #A
      http.open("POST", url);
      http.send();

      http.onreadystatechange = (e) => {
        document      #B
          .getElementById("output")
          .innerHTML = http.responseText;
      }
    </script>
  </head>
  <body>
    <div id="output"></div>
  </body>
</html>
```

Starting the application and opening the page in a browser with localhost:8080, we can observe that the page doesn’t display anything. We expected to see HELLO on the page because this is what the /test endpoint returns. When we check the browser console, what we see is an error printed by the JavaScript call. The error looks like this:

```
Access to XMLHttpRequest at 'http://127.0.0.1:8080/test' from ori
```

The error message tells us that the response wasn’t accepted because the `Access-Control-Allow-Origin` HTTP header doesn’t exist. This behavior happens because we didn’t configure anything regarding CORS in our Spring Boot application, and by default, it doesn’t set any header related to CORS. So the browser’s behavior of not displaying the response is correct. I would like you, however, to notice that in the application console, the log proves the method was called. The next code snippet shows what you find in the application console:

```
INFO 25020 --- [nio-8080-exec-2] c.l.s.controllers.MainController
```

This aspect is important! I meet many developers who understand CORS as a restriction similar to authorization or CSRF protection. Instead of being a restriction, CORS helps to relax a rigid constraint for cross-domain calls. And

even with restrictions applied, in some situations, the endpoint can be called. This behavior doesn't always happen. Sometimes, the browser first makes a call using the HTTP OPTIONS method to test whether the request should be allowed. We call this test request a *preflight* request. If the preflight request fails, the browser won't attempt to honor the original request.

The preflight request and the decision to make it or not are the responsibility of the browser. You don't have to implement this logic. But it is important to understand it, so you won't be surprised to see cross-origin calls to the backend even if you did not specify any CORS policies for specific domains. This could happen, as well, when you have a client-side app developed with a framework like Angular or ReactJS. Figure 10.4 presents this request flow.

When the browser omits to make the preflight request if the HTTP method is GET, POST, or OPTIONS, it only has some basic headers as described in the official documentation at https://fetch.spec.whatwg.org/#http-cors-protocol

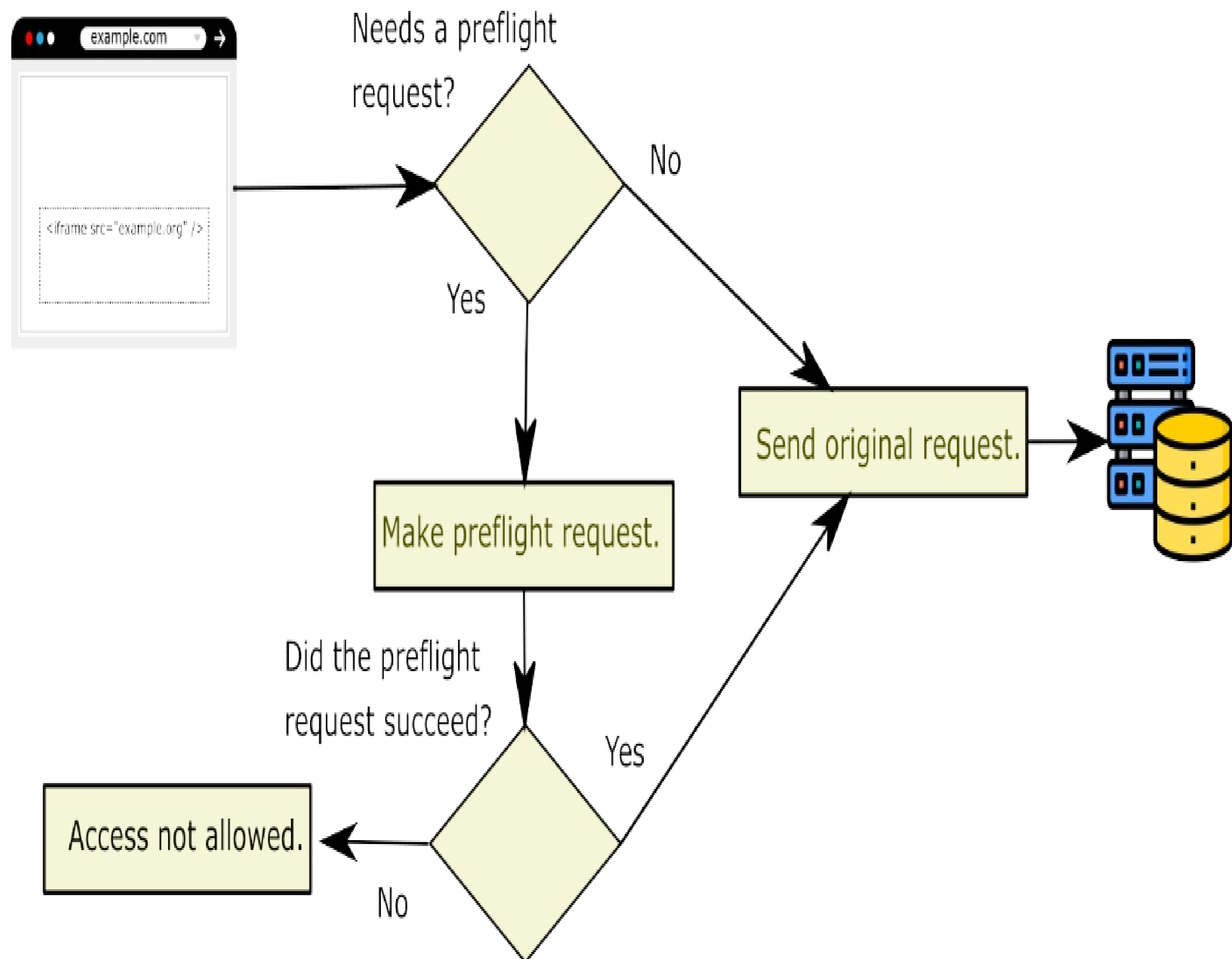
In our example, the browser makes the request, but we don't accept the response if the origin is not specified in the response, as shown in figures 10.1 and 10.2. The CORS mechanism is, in the end, related to the browser and not a way to secure endpoints. The only thing it guarantees is that only origin domains that you allow can make requests from specific pages in the browser.

10.2 Applying CORS policies with the @CrossOrigin annotation

In this section, we discuss how to configure CORS to allow requests from different domains using the `@CrossOrigin` annotation. You can place the `@CrossOrigin` annotation directly above the method that defines the endpoint and configure it using the allowed origins and methods. As you learn in this section, the advantage of using the `@CrossOrigin` annotation is that it makes it easy to configure CORS for each endpoint.

Figure 10.4 For simple requests, the browser sends the original request directly to the server. The browser rejects the response if the server doesn't allow the origin. In some cases, the browser sends a preflight request to test if the server accepts the origin. If the preflight request succeeds,

the browser sends the original request.



We use the application we created in section 10.1 to demonstrate how `@CrossOrigin` works. To make the cross-origin call work in the application, the only thing you need to do is to add the `@CrossOrigin` annotation over the `test()` method in the controller class. The following listing shows how to use the annotation to make the localhost an allowed origin.

Listing 10.4 Making localhost an allowed origin

```
@PostMapping("/test")
@ResponseBody
@CrossOrigin("http://localhost:8080")     #A
public String test() {
  logger.info("Test method called");
  return "HELLO";
}
```

You can rerun and test the application. This should now display on the page the string returned by the /test endpoint: HELLO.

The value parameter of `@CrossOrigin` receives an array to let you define multiple origins; for example, `@CrossOrigin({"example.com", "example.org"})`. You can also set the allowed headers and methods using the `allowedHeaders` attribute and the `methods` attribute of the annotation. For both origins and headers, you can use the asterisk (*) to represent all headers or all origins. But I recommend you exercise caution with this approach. It's always better to filter the origins and headers that you want to allow and never allow any domain to implement code that accesses your applications' resources.

By allowing all origins, you expose the application to cross-site scripting (XSS) requests, which eventually can lead to DDoS attacks. I personally avoid allowing all origins even in test environments. I know that applications sometimes happen to run on wrongly defined infrastructures that use the same data centers for both test and production. It is wiser to treat all layers on which security applies independently, as we discussed in chapter 1, and to avoid assuming that the application doesn't have particular vulnerabilities because the infrastructure doesn't allow it.

The advantage of using `@CrossOrigin` to specify the rules directly where the endpoints are defined is that it creates good transparency of the rules. The disadvantage is that it might become verbose, forcing you to repeat a lot of code. It also imposes the risk that the developer might forget to add the annotation for newly implemented endpoints. In section 10.3, we discuss applying the CORS configuration centralized within the configuration class.

10.3 Applying CORS using a CorsConfigurer

Although using the `@CrossOrigin` annotation is easy, as you learned in section 10.2, you might find it more comfortable in a lot of cases to define CORS configuration in one place. In this section, we change the example we worked on in sections 10.1 and 10.2 to apply CORS configuration in the configuration class using a `Customizer`. In the next listing, you can find the changes we need to make in the configuration class to define the origins we

want to allow.

Listing 10.5 Defining CORS configurations centralized in the configuration class

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity htt
    throws Exception {

     http.cors(c -> {     #A
      CorsConfigurationSource source = request -> {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(
            List.of("example.com", "example.org"));
        config.setAllowedMethods(
            List.of("GET", "POST", "PUT", "DELETE"));
        config.setAllowedHeaders(List.of("*"));
        return config;
      };
      c.configurationSource(source);
    });

    http.csrf(
      c -> c.disable()
    );

    http.authorizeHttpRequests(
      c -> c.anyRequest().permitAll()
    );


    return http.build();
  }
}
```

The `cors()` method that we call from the `HttpSecurity` object receives as a parameter a `Customizer<CorsConfigurer>` object. For this object, we set a `CorsConfigurationSource`, which returns `CorsConfiguration` for an HTTP request. `CorsConfiguration` is the object that states which are the allowed origins, methods, and headers. If you use this approach, you have to specify at least which are the origins and the methods. If you only specify the origins, your application won’t allow the requests. This behavior happens because a

`CorsConfiguration` object doesn't define any methods by default.

In this example, to make the explanation straightforward, I provide the implementation for `CorsConfigurationSource` as a lambda expression using the `SecurityFilterChain` bean directly. I strongly recommend to separate this code in a different class in your applications. In real-world applications, you could have much longer code, so it becomes difficult to read if not separated by the configuration class.

10.4 Summary

- Cross-over resource sharing (CORS) refers to the situation in which a web application hosted on a specific domain tries to access content from another domain.
- By default, the browser doesn't allow cross origin requests to happen. CORS configuration enables you to allow a part of your resources to be called from a different domain in a web application run in the browser.
- You can configure CORS both for an endpoint using the `@CrossOrigin` annotation or centralized in the configuration class using the `cors()` method of the `HttpSecurity` object.

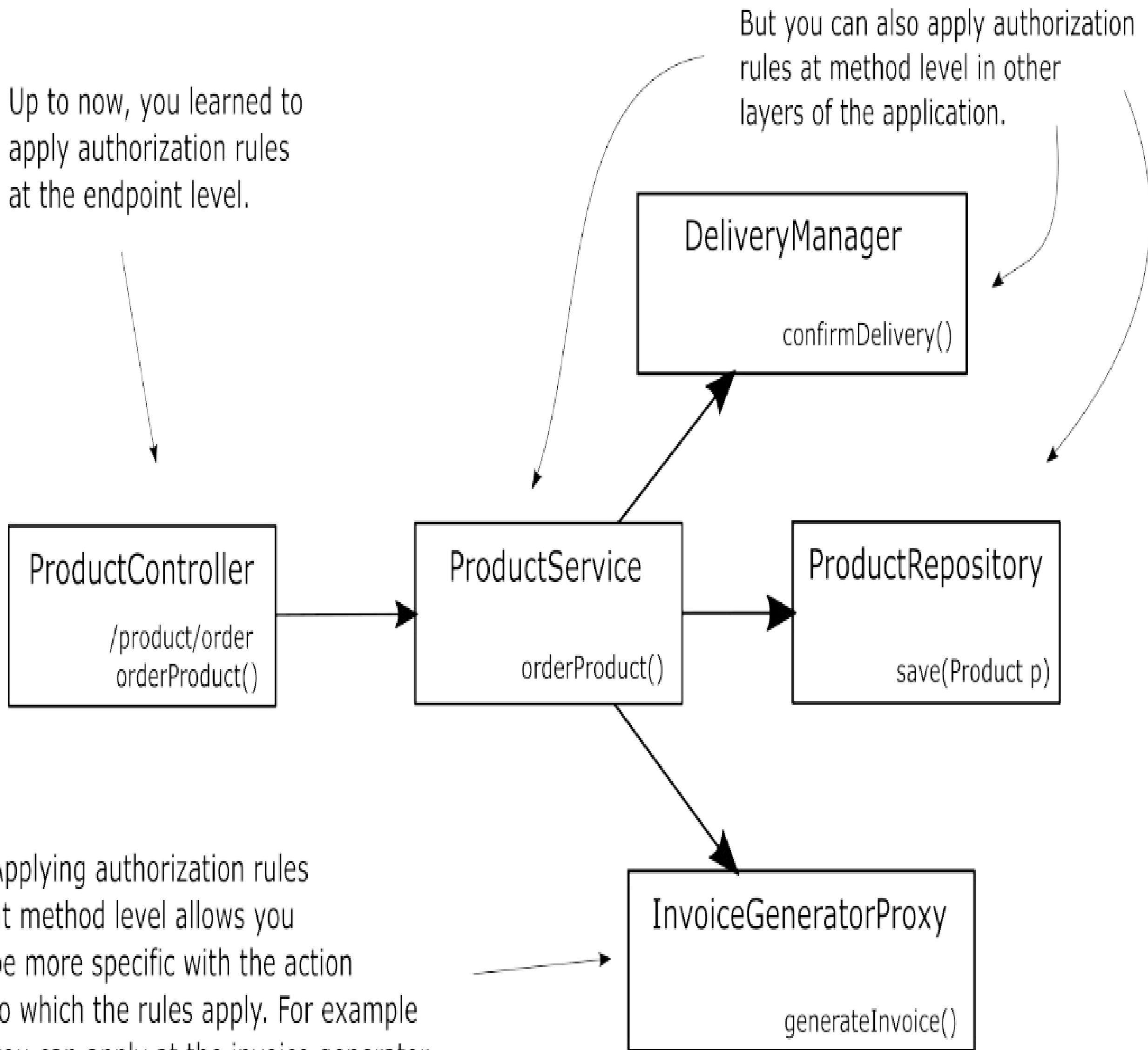
11 Implement authorization at the method level

This chapter covers

- Method security in Spring applications
- Preauthorization of methods based on authorities, roles, and permissions
- Postauthorization of methods based on authorities, roles, and permissions

Up to now, we discussed various ways of configuring authentication. We started with the most straightforward approach, HTTP Basic, in chapter 2, and then I showed you how to set form login in chapter 6. But in terms of authorization, we only discussed configuration at the endpoint level. Suppose your app is not a web application—can’t you use Spring Security for authentication and authorization as well? Spring Security is a good fit for scenarios where your app isn’t used via HTTP endpoints. In this chapter, you’ll learn how to configure authorization at the method level. We’ll use this approach to configure authorization in both web and non-web applications, and we’ll call it *method security* (figure 11.1).

Figure 11.1 Method security enables you to apply authorization rules at any layer of your application. This approach allows you to be more granular and to apply authorization rules at a specifically chosen level.



For non-web applications, method security offers the opportunity to implement authorization rules even if we don’t have endpoints. In web applications, this approach gives us the flexibility to apply authorization rules on different layers of our app, not only at the endpoint level. Let’s dive into the chapter and learn how to apply authorization at the method level with method security.

11.1 Enabling method security

In this section, you learn how to enable authorization at the method level and the different options that Spring Security offers to apply various authorization rules. This approach provides you with greater flexibility in applying authorization. It’s an essential skill that allows you to solve situations in which authorization simply cannot be configured just at the endpoint level.

By default, method security is disabled, so if you want to use this functionality, you first need to enable it. Also, method security offers multiple approaches for applying authorization. We discuss these approaches and then implement them in examples in the following sections of this chapter and in chapter 12. Briefly, you can do two main things with global method security:

- *Call authorization*—Decides whether someone can call a method according to some implemented privilege rules (preauthorization) or if someone can access what the method returns after the method executes (postauthorization).
- *Filtering*—Decides what a method can receive through its parameters (prefiltering) and what the caller can receive back from the method after the method executes (postfiltering). We’ll discuss and implement filtering in chapter 12.

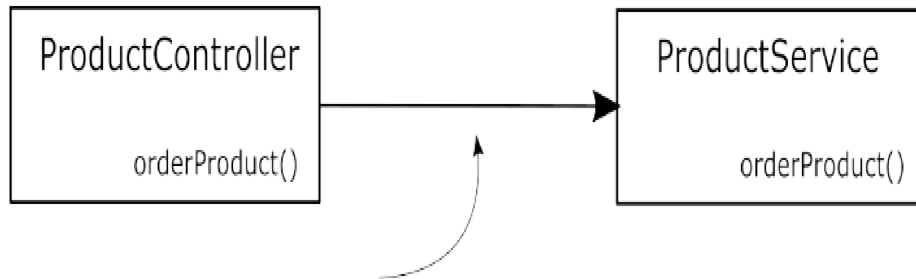
11.1.1 Understanding call authorization

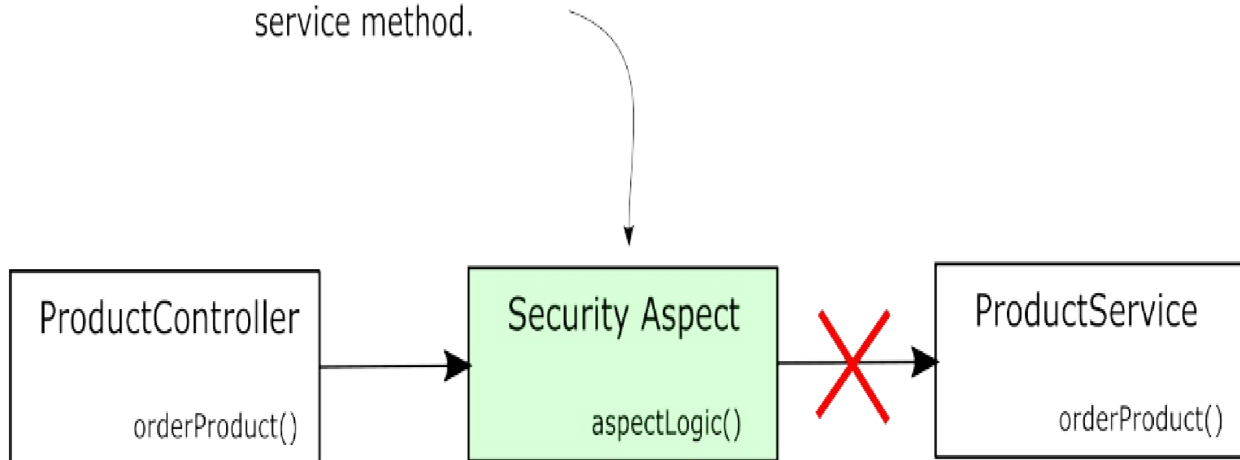
One of the approaches for configuring authorization rules you use with method security is *call authorization*. The call authorization approach refers to applying authorization rules that decide if a method can be called, or that allow the method to be called and then decide if the caller can access the value returned by the method. Often we need to decide if someone can access a piece of logic depending on either the provided parameters or its result. So let’s discuss call authorization and then apply it to some examples.

How does method security work? What’s the mechanism behind applying the authorization rules? When we enable method security in our application, we actually enable a Spring aspect. This aspect intercepts the calls to the method for which we apply authorization rules and, based on these authorization rules, decides whether to forward the call to the intercepted method (figure

11.2).

Figure 11.2 When we enable global method security, an aspect intercepts the call to the protected method. If the given authorization rules aren't respected, the aspect doesn't delegate the call to the protected method.





Plenty of implementations in Spring framework rely on aspect-oriented programming (AOP). Method security is just one of the many components in Spring applications relying on aspects. If you need a refresher on aspects and AOP, I recommend you read chapter 6 of *Spring Start Here (Manning, 2021)* , another book I wrote. Briefly, we classify the call authorization as

- *Preauthorization*—The framework checks the authorization rules before the method call.
- *Postauthorization*—The framework checks the authorization rules after the method executes.

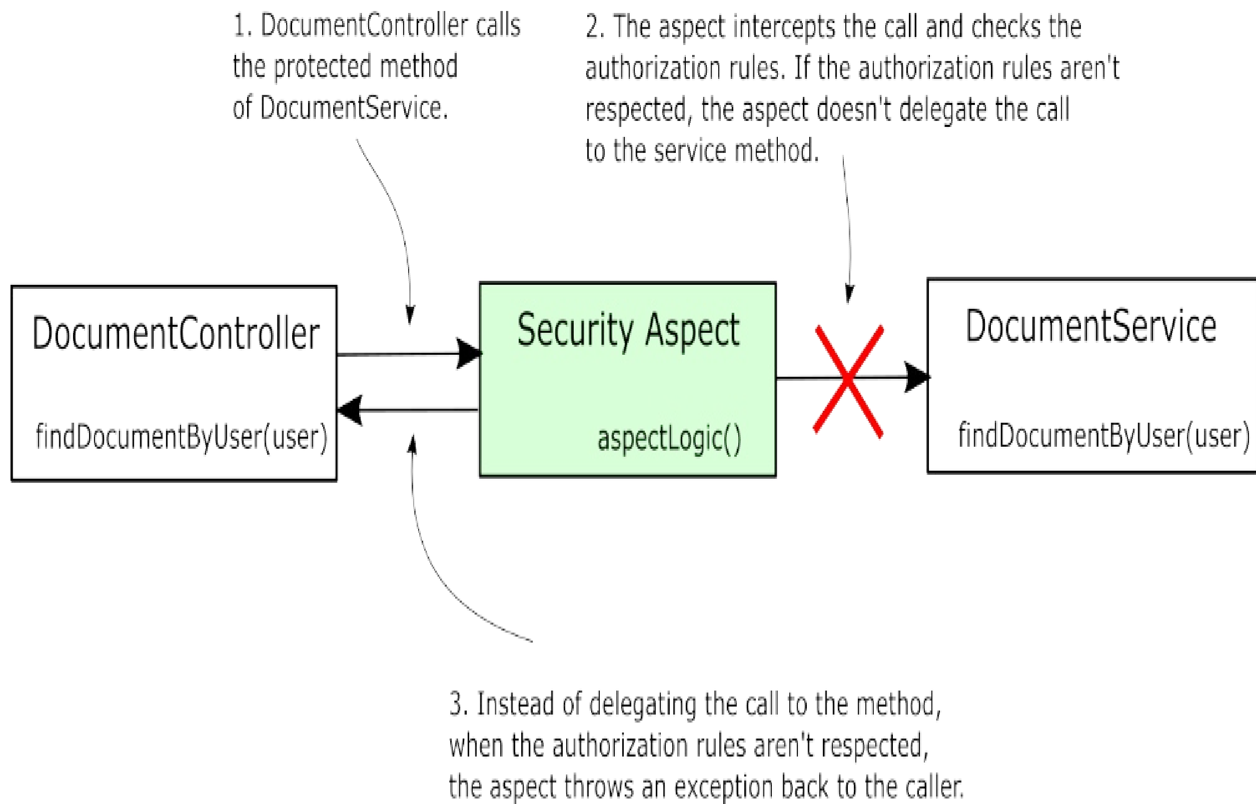
Let's take both approaches, detail them, and implement them with some examples.

Using preauthorization to secure access to methods

Say we have a method `findDocumentsByUser(String username)` that returns to the caller documents for a specific user. The caller provides through the method's parameters the user's name for which the method retrieves the documents. Assume you need to make sure that the authenticated user can only obtain their own documents. Can we apply a rule to this method such that only the method calls that receive the username of the authenticated user as a parameter are allowed? Yes! This is something we do with preauthorization.

When we apply authorization rules that completely forbid anyone to call a method in specific situations, we call this *preauthorization* (figure 11.3). This approach implies that the framework verifies the authorization conditions before executing the method. If the caller doesn't have the permissions according to the authorization rules that we define, the framework doesn't delegate the call to the method. Instead, the framework throws an exception named `AccessDeniedException`. This is by far the most often used approach to global method security.

Figure 11.3 With preauthorization, the authorization rules are verified before delegating the method call further. The framework won't delegate the call if the authorization rules aren't respected, and instead, throws an exception to the method caller.

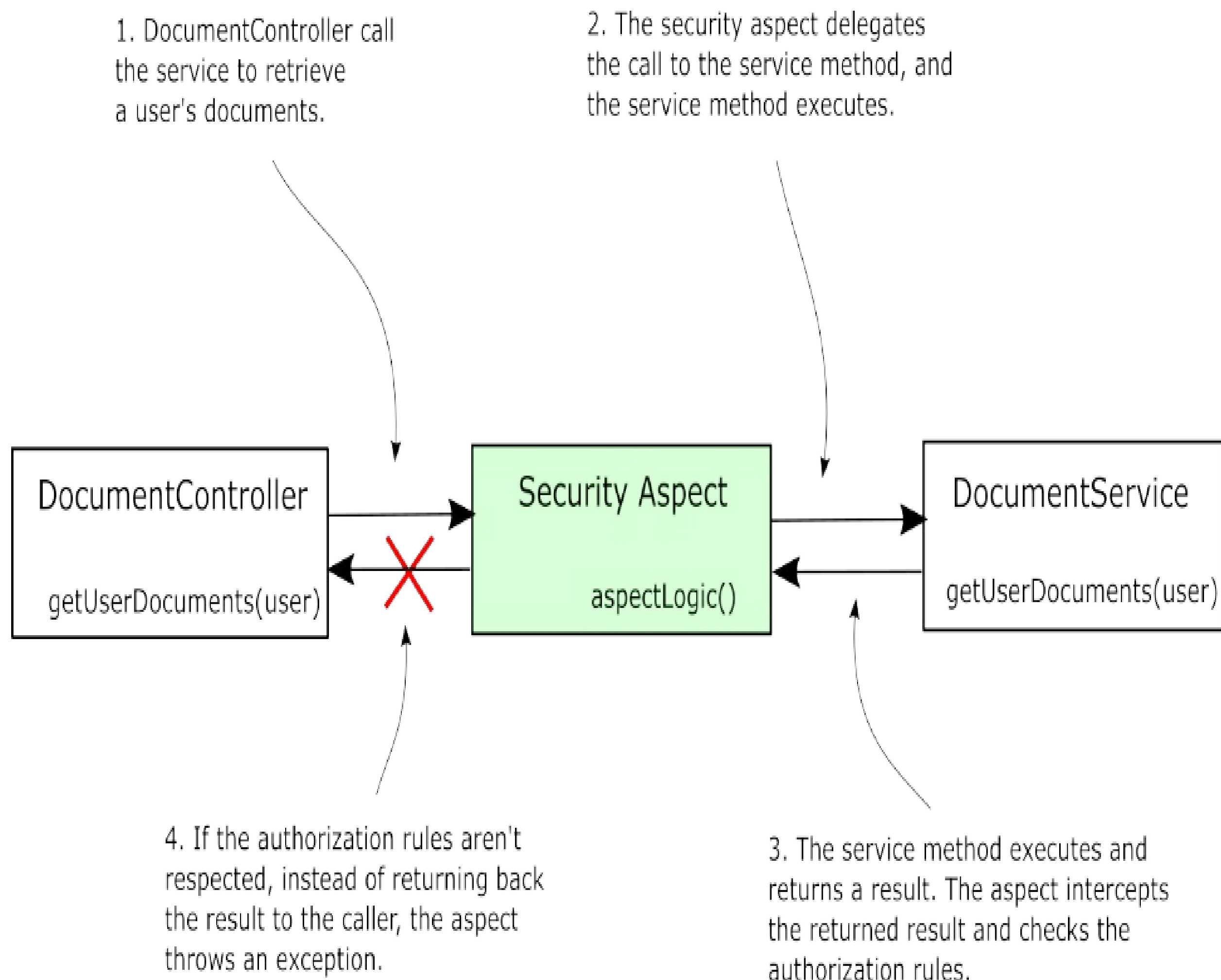


Usually, we don’t want a functionality to be executed at all if some conditions aren’t met. You can apply conditions based on the authenticated user, and you can also refer to the values the method received through its parameters.

Using postauthorization to secure a method call

When we apply authorization rules that allow someone to call a method but not necessarily to obtain the result returned by the method, we’re using *postauthorization* (figure 11.4). With postauthorization, Spring Security checks the authorization rules after the method executes. You can use this kind of authorization to restrict access to the method return in certain conditions. Because postauthorization happens after method execution, you can apply the authorization rules on the result returned by the method.

Figure 11.4 With postauthorization, the aspect delegates the call to the protected method. After the protected method finishes execution, the aspect checks the authorization rules. If the rules aren’t respected, instead of returning the result to the caller, the aspect throws an exception.



Usually, we use postauthorization to apply authorization rules based on what the method returns after execution. But be careful with postauthorization! If the method mutates something during its execution, the change happens whether or not authorization succeeds in the end.

NOTE

Even with the `@Transactional` annotation, a change isn't rolled back if postauthorization fails. The exception thrown by the postauthorization functionality happens after the transaction manager commits the transaction.

11.1.2 Enabling method security in your project

In this section, we work on a project to apply the preauthorization and

postauthorization features offered by method security. Method security isn't enabled by default in a Spring Security project. To use it, you need to first enable it. However, enabling this functionality is straightforward. You do this by simply using the `@EnableMethodSecurity` annotation on the configuration class.

I created a new project for this example, ssia-ch11-ex1. For this project, I wrote a `ProjectConfig` configuration class, as presented in listing 11.1. On the configuration class, we add the `@EnableMethodSecurity` annotation. Method security offers us three approaches to define the authorization rules that we discuss in this chapter:

- The pre-/postauthorization annotations (enabled by default)
- The JSR 250 annotation, `@RolesAllowed`
- The `@Secured` annotation

Because in almost all cases, pre-/postauthorization annotations are the only approach used, we discuss this approach in this chapter. This approach is pre-enabled once you add the `@EnableMethodSecurity` annotation. We present a short overview of the other two options previously mentioned at the end of this chapter.

Listing 11.1 Enabling method security

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {
}
```

You can use global method security with any authentication approach, from HTTP Basic authentication to OAuth 2 (which you'll learn in the 3rd part of this book). To keep it simple and allow you to focus on new details, we provide method security with HTTP Basic authentication. For this reason, the pom.xml file for the projects in this chapter only needs the web and Spring Security dependencies, as the next code snippet presents:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
```

```
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Note

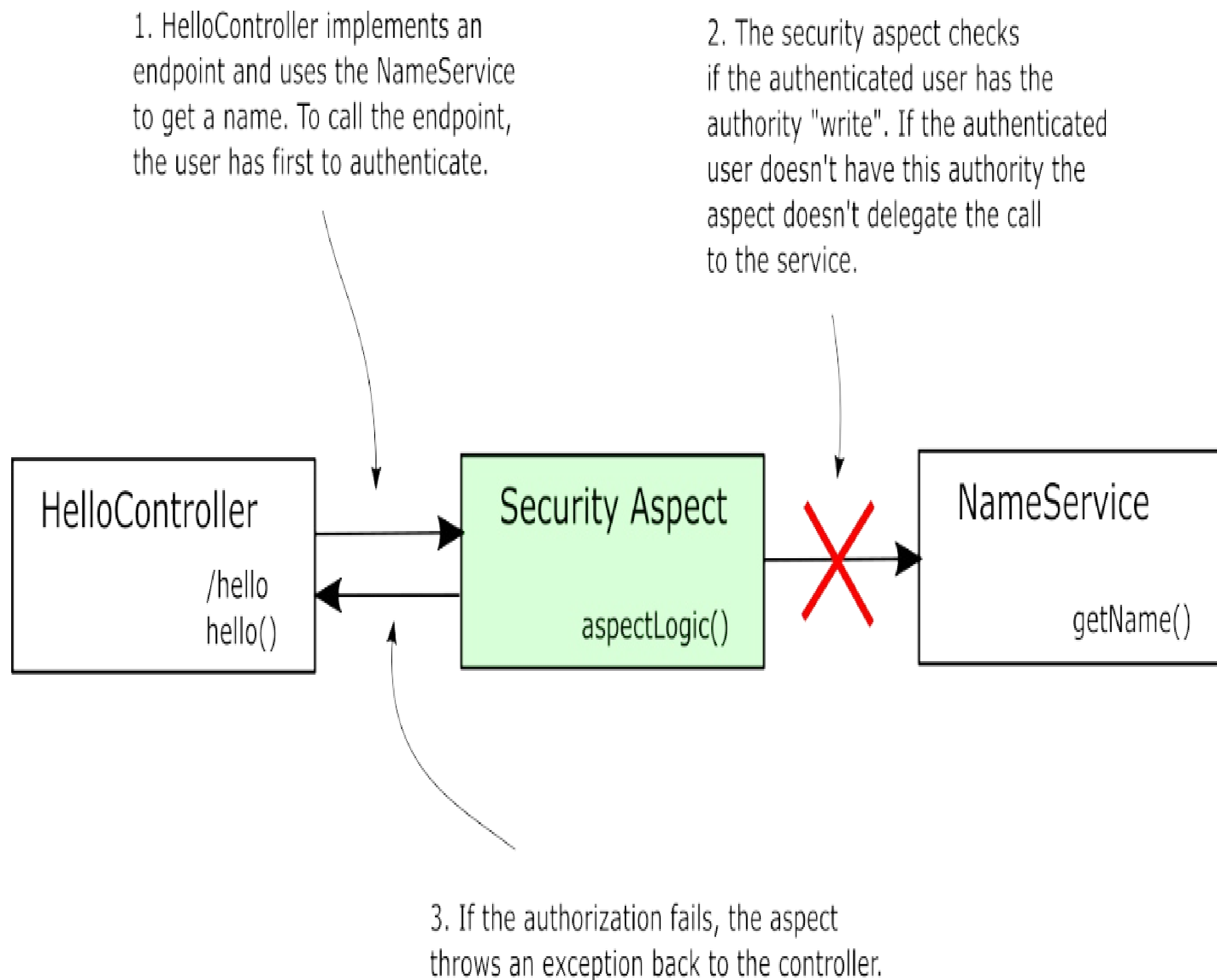
In the previous Spring Security versions, we used the `@EnableGlobalMethodSecurity` annotation, and the pre and post-authorization wasn't enabled by default. If you need to work with method authorization and a previous Spring Security version (older than 6), you will find useful to read chapter 16 from the first edition of Spring Security in Action.

11.2 Applying preauthorization rules

In this section, we implement an example of preauthorization. For our example, we continue with the project `ssia-ch11-ex1` started in section 11.1. As we discussed in section 11.1, preauthorization implies defining authorization rules that Spring Security applies before calling a specific method. If the rules aren't respected, the framework doesn't call the method.

The application we implement in this section has a simple scenario. It exposes an endpoint, `/hello`, which returns the string `"Hello, "` followed by a name. To obtain the name, the controller calls a service method (figure 11.5). This method applies a preauthorization rule to verify the user has write authority.

Figure 11.5 To call the `getName()` method of `NameService`, the authenticated user needs to have write authority. If the user doesn't have this authority, the framework won't allow the call and throws an exception.



I added a `UserDetailsService` and a `PasswordEncoder` to make sure I have some users to authenticate. To validate our solution, we need two users: one user with write authority and another that doesn't have write authority. We prove that the first user can successfully call the endpoint, while for the second user, the app throws an authorization exception when trying to call the method. The following listing shows the complete definition of the configuration class, which defines the `UserDetailsService` and the `PasswordEncoder`.

Listing 11.2 The configuration class for `UserDetailsService` and `PasswordEncoder`

```
@Configuration
@EnableMethodSecurity    #A
public class ProjectConfig {

  @Bean      #B
```

```
  public UserDetailsService userDetailsService() {
    var service = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("natalie")
              .password("12345")
              .authorities("read")
              .build();

    var u2 = User.withUsername("emma")
              .password("12345")
              .authorities("write")
              .build();

    service.createUser(u1);
    service.createUser(u2);

    return service;
  }

  @Bean     #C
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

To define the authorization rule for this method, we use the `@PreAuthorize` annotation. The `@PreAuthorize` annotation receives as a value a Spring Expression Language (SpEL) expression that describes the authorization rule. In this example, we apply a simple rule.

You can define restrictions for users based on their authorities using the `hasAuthority()` method. You learned about the `hasAuthority()` method in chapter 7, where we discussed applying authorization at the endpoint level. The following listing defines the service class, which provides the value for the name.

Listing 11.3 The service class defines the preauthorization rule on the method

```
@Service
public class NameService {

  @PreAuthorize("hasAuthority('write')")      #A
  public String getName() {
    return "Fantastico";
```

```
  }
}
```

We define the controller class in the following listing. It uses `NameService` as a dependency.

Listing 11.4 The controller class implementing the endpoint and using the service

```
@RestController
public class HelloController {

  private final NameService nameService;     #A

  // omitted constructor

  @GetMapping("/hello")
  public String hello() {
    return "Hello, " + nameService.getName();      #B
  }
}
```

You can now start the application and test its behavior. We expect only user Emma to be authorized to call the endpoint because she has write authorization. The next code snippet presents the calls for the endpoint with our two users, Emma and Natalie. To call the /hello endpoint and authenticate with user Emma, use this cURL command:

```
curl -u emma:12345 http://localhost:8080/hello
```

The response body is

```
Hello, Fantastico
```

To call the /hello endpoint and authenticate with user Natalie, use this cURL command:

```
curl -u natalie:12345 http://localhost:8080/hello
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
```

```
  "message":"Forbidden",
  "path":"/hello"
}
```

Similarly, you can use any other expression we discussed in chapter 7 for endpoint authentication. Here's a short recap of them:

- `hasAnyAuthority()`—Specifies multiple authorities. The user must have at least one of these authorities to call the method.
- `hasRole()`—Specifies a role a user must have to call the method.
- `hasAnyRole()`—Specifies multiple roles. The user must have at least one of them to call the method.

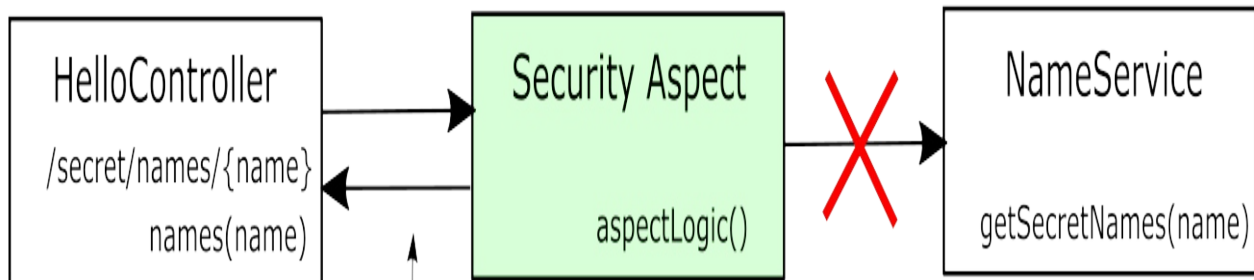
Let's extend our example to prove how you can use the values of the method parameters to define the authorization rules (figure 11.6). You find this example in the project named `ssia-ch11-ex2`.

Figure 11.6 When implementing preauthorization, we can use the values of the method parameters in the authorization rules. In our example, only the authenticated user can retrieve information about their secret names.

1. HelloController implements an endpoint and uses the NameService to get a list of secret names of the user. To call the endpoint, the user has first to authenticate.

2. The security aspect validates that the name provided as a parameter is the same as the name of the user who authenticated.

curl -u **emma**:12345 http://.../secret/names/**natalie**



3. If the name fo the authenticated user is different than the one provided as a parameter, the aspect doesn't delegate the call to the service class, and throws back an exception.

For this project, I defined the same `ProjectConfig` class as in our first example so that we can continue working with our two users, Emma and Natalie. The endpoint now takes a value through a path variable and calls a service class to obtain the “secret names” for a given username. Of course, in this case, the secret names are just an invention of mine referring to a

characteristic of the user, which is something that not everyone can see. I define the controller class as presented in the next listing.

Listing 11.5 The controller class defining an endpoint for testing

```
@RestController
public class HelloController {


  private final NameService nameService;    #A

  // omitted constructor

  @GetMapping("/secret/names/{name}")    #B
  public List<String> names(@PathVariable String name) {
      return nameService.getSecretNames(name);     #C
  }
}
```

Now let's take a look at how to implement the `NameService` class in listing 11.6. The expression we use for authorization now is `#name == authentication.principal`

`.username`. In this expression, we use `#name` to refer to the value of the `getSecretNames()` method parameter called `name`, and we have access directly to the authentication object that we can use to refer to the currently authenticated user. The expression we use indicates that the method can be called only if the authenticated user's username is the same as the value sent through the method's parameter. In other words, a user can only retrieve its own secret names.

Listing 11.6 The `NameService` class defines the protected method

```
@Service
public class NameService {

  private Map<String, List<String>> secretNames =
    Map.of(
     "natalie", List.of("Energico", "Perfecto"),
     "emma", List.of("Fantastico"));

  @PreAuthorize    #A
    ("#name == authentication.principal.username")
```

```
  public List<String> getSecretNames(String name) {
    return secretNames.get(name);
  }
}
```

We start the application and test it to prove it works as desired. The next code snippet shows you the behavior of the application when calling the endpoint, providing the value of the path variable equal to the name of the user:

```
curl -u emma:12345 http://localhost:8080/secret/names/emma
```

The response body is

```
["Fantastico"]
```

When authenticating with the user Emma, we try to get Natalie's secret names. The call doesn't work:

```
curl -u emma:12345 http://localhost:8080/secret/names/natalie
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/secret/names/natalie"
}
```

The user Natalie can, however, obtain her own secret names. The next code snippet proves this:

```
curl -u natalie:12345 http://localhost:8080/secret/names/natalie
```

The response body is

```
["Energico","Perfecto"]
```

NOTE

Remember, you can apply method security to any layer of your application. In the examples presented in this chapter, you find the authorization rules

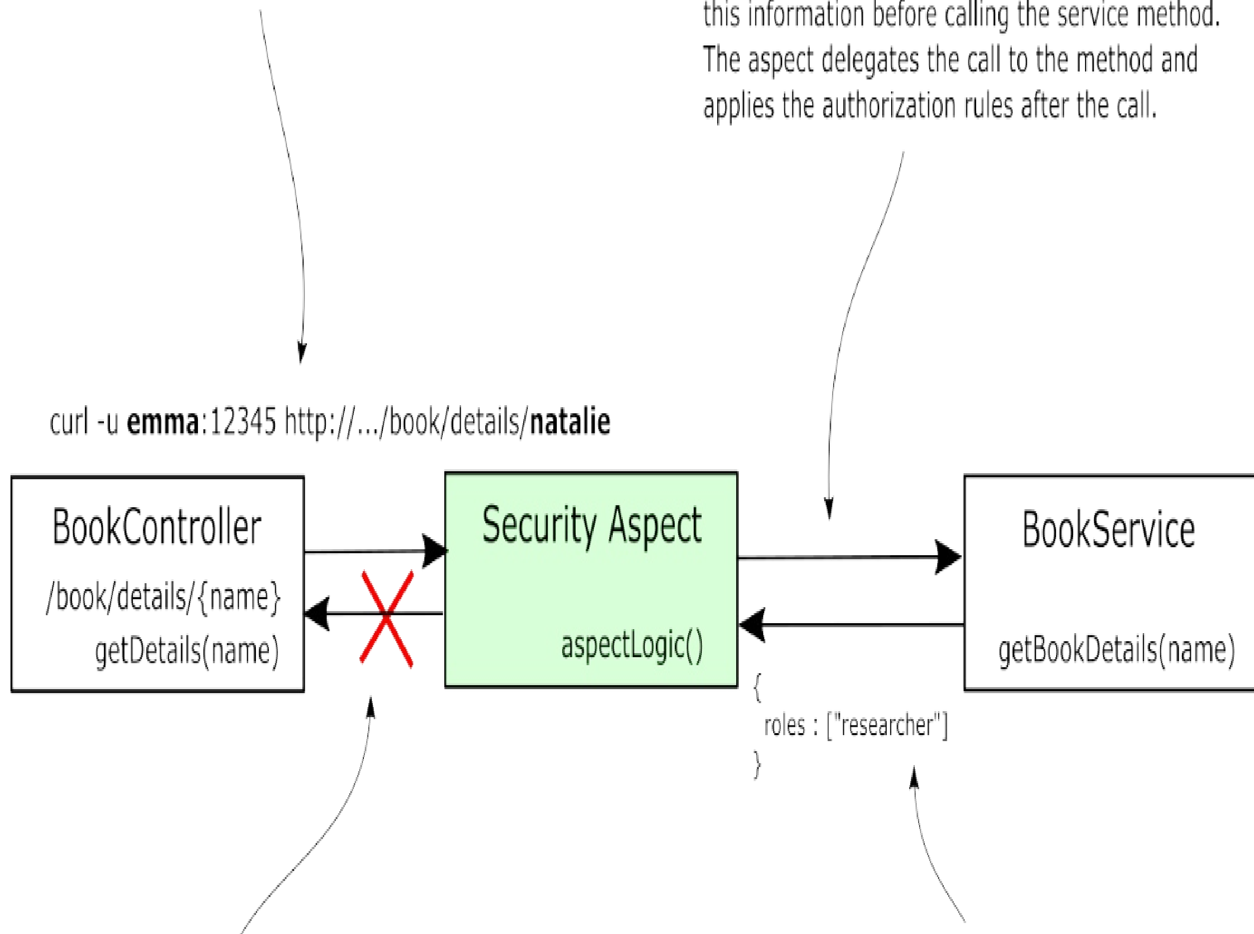
applied for methods of the service classes. But you can apply authorization rules with method security in any part of your application: controllers, repositories, managers, proxies, and so on.

11.3 Applying postauthorization rules

Now say you want to allow a call to a method, but in certain circumstances, you want to make sure the caller doesn't receive the returned value. When we want to apply an authorization rule that is verified after the call of a method, we use postauthorization. It may sound a little bit awkward at the beginning: why would someone be able to execute the code but not get the result? Well, it's not about the method itself, but imagine this method retrieves some data from a data source, say a web service or a database. The conditions you need to add for authorization depend on the received data. So you allow the method to execute, but you validate what it returns and, if it doesn't meet the criteria, you don't let the caller access the return value.

To apply postauthorization rules with Spring Security, we use the `@PostAuthorize` annotation, which is similar to `@PreAuthorize`, discussed in section 11.2. The annotation receives as a value the SpEL defining an authorization rule. We continue with an example in which you learn how to use the `@PostAuthorize` annotation and define postauthorization rules for a method (figure 11.7).

Figure 11.7 With postauthorization, we don't protect the method from being called, but we protect the returned value from being exposed if the defined authorization rules aren't respected.



The scenario for our example, for which I created a project named ssia-ch11-ex3, defines an object `Employee`. Our `Employee` has a name, a list of books, and a list of authorities. We associate each `Employee` to a user of the application. To stay consistent with the other examples in this chapter, we define the same users, Emma and Natalie. We want to make sure that the caller of the method gets the details of the employee only if the employee has read authority. Because we don't know the authorities associated with the employee record until we retrieve the record, we need to apply the authorization rules after the method execution. For this reason, we use the

`@PostAuthorize` annotation.

The configuration class is the same as we used in the previous examples. But, for your convenience, I repeat it in the next listing.

Listing 11.7 Enabling method security and defining users

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var service = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("natalie")
                 .password("12345")
                 .authorities("read")
                 .build();

    var u2 = User.withUsername("emma")
                 .password("12345")
                 .authorities("write")
                 .build();

    service.createUser(u1);
    service.createUser(u2);

    return service;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

We also need to declare a class to represent the `Employee` object with its name, book list, and roles list. The following listing defines the `Employee` class.

Listing 11.8 The definition of the `Employee` class

```
public class Employee {
```

```
  private String name;
  private List<String> books;
  private List<String> roles;

  // Omitted constructor, getters, and setters
}
```

We probably get our employee details from a database. To keep our example shorter, I use a `Map` with a couple of records that we consider as our data source. In listing 11.9, you find the definition of the `BookService` class. The `BookService` class also contains the method for which we apply the authorization rules. Observe that the expression we use with the `@PostAuthorize` annotation refers to the value returned by the method `returnObject`. The postauthorization expression can use the value returned by the method, which is available after the method executes.

Listing 11.9 The `BookService` class defining the authorized method

```
@Service
public class BookService {

  private Map<String, Employee> records =
    Map.of("emma",
           new Employee("Emma Thompson",
               List.of("Karamazov Brothers"),
               List.of("accountant", "reader")),
           "natalie",
           new Employee("Natalie Parker",
               List.of("Beautiful Paris"),
               List.of("researcher"))
        );
  @PostAuthorize     #A
  ➥("returnObject.roles.contains('reader')")
  public Employee getBookDetails(String name) {
      return records.get(name);
  }
}
```

Let’s also write a controller and implement an endpoint to call the method for which we applied the authorization rule. The following listing presents this controller class.

Listing 11.10 The controller class implementing the endpoint

```
@RestController
public class BookController {

  private final BookService bookService;

  // omitted constructor

  @GetMapping("/book/details/{name}")
  public Employee getDetails(@PathVariable String name) {
    return bookService.getBookDetails(name);
  }
}
```

You can now start the application and call the endpoint to observe the app's behavior. In the next code snippets, you find examples of calling the endpoint. Any of the users can access the details of Emma because the returned list of roles contains the string `"reader"`, but no user can obtain the details for Natalie. Calling the endpoint to get the details for Emma and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
  "name":"Emma Thompson",
  "books":["Karamazov Brothers"],
  "roles":["accountant","reader"]
}
```

Calling the endpoint to get the details for Emma and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
  "name":"Emma Thompson",
  "books":["Karamazov Brothers"],
  "roles":["accountant","reader"]
```

```
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/book/details/natalie"
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/book/details/natalie"
}
```

NOTE

You can use both `@PreAuthorize` and `@PostAuthorize` on the same method if your requirements need to have both preauthorization and postauthorization.

11.4 Implementing permissions for methods

Up to now, you learned how to define rules with simple expressions for preauthorization and postauthorization. Now, let’s assume the authorization logic is more complex, and you cannot write it in one line. It’s definitely not

comfortable to write huge SpEL expressions. I never recommend using long SpEL expressions in any situation, regardless if it’s an authorization rule or not. It simply creates hard-to-read code, and this affects the app’s maintainability. When you need to implement complex authorization rules, instead of writing long SpEL expressions, take the logic out in a separate class. Spring Security provides the concept of *permission*, which makes it easy to write the authorization rules in a separate class so that your application is easier to read and understand.

In this section, we apply authorization rules using permissions within a project. I named this project `ssia-ch11-ex4`. In this scenario, you have an application managing documents. Any document has an owner, which is the user who created the document. To get the details of an existing document, a user either has to be an admin or they have to be the owner of the document. We implement a permission evaluator to solve this requirement. The following listing defines the `document`, which is only a plain Java object.

Listing 11.11 The `Document` class

```
public class Document {

  private String owner;

  // Omitted constructor, getters, and setters
}
```

To mock the database and make our example shorter for your comfort, I created a repository class that manages a few document instances in a `Map`. You find this class in the next listing.

Listing 11.12 The `DocumentRepository` class managing a few `Document` instances

```
@Repository
public class DocumentRepository {

  private Map<String, Document> documents =      #A
    Map.of("abc123", new Document("natalie"),
           "qwe123", new Document("natalie"),
           "asd555", new Document("emma"));

  public Document findDocument(String code) {
```

```
    return documents.get(code);      #B
  }
}
```

A service class defines a method that uses the repository to obtain a document by its code. The method in the service class is the one for which we apply the authorization rules. The logic of the class is simple. It defines a method that returns the `Document` by its unique code. We annotate this method with `@PostAuthorize` and use a `hasPermission()` SpEL expression. This method allows us to refer to an external authorization expression that we implement further in this example. Meanwhile, observe that the parameters we provide to the `hasPermission()` method are the `returnObject`, which represents the value returned by the method, and the name of the role for which we allow access, which is `'ROLE_admin'`. You find the definition of this class in the following listing.

Listing 11.13 The `DocumentService` class implementing the protected method

```
@Service
public class DocumentService {

  private final DocumentRepository documentRepository;

  // omitted constructor

  @PostAuthorize    #A
  ("hasPermission(returnObject, 'ROLE_admin')")
  public Document getDocument(String code) {
    return documentRepository.findDocument(code);
  }
}
```

It's our duty to implement the permission logic. And we do this by writing an object that implements the `PermissionEvaluator` contract. The `PermissionEvaluator` contract provides two ways to implement the permission logic:

- *By object and permission*—Used in the current example, it assumes the permission evaluator receives two objects: one that's subject to the authorization rule and one that offers extra details needed for implementing the permission logic.

- *By object ID, object type, and permission*—Assumes the permission evaluator receives an object ID, which it can use to retrieve the needed object. It also receives a type of object, which can be used if the same permission evaluator applies to multiple object types, and it needs an object offering extra details for evaluating the permission.

In the next listing, you find the `PermissionEvaluator` contract with two methods.

Listing 11.14 The `PermissionEvaluator` contract definition

```
public interface PermissionEvaluator {

    boolean hasPermission(
              Authentication a,
              Object subject,
              Object permission);

    boolean hasPermission(
              Authentication a,
              Serializable id,
              String type,
              Object permission);
}
```

For the current example, it's enough to use the first method. We already have the subject, which in our case, is the value returned by the method. We also send the role name `'ROLE_admin'`, which, as defined by the example's scenario, can access any document. Of course, in our example, we could have directly used the name of the role in the permission evaluator class and avoided sending it as a value of the `hasPermission()` object. Here, we only do the former for the sake of the example. In a real-world scenario, which might be more complex, you have multiple methods, and details needed in the authorization process might differ between each of them. For this reason, you have a parameter that you can send the needed details for use in the authorization logic from the method level.

For your awareness and to avoid confusion, I'd also like to mention that you don't have to pass the `Authentication` object. Spring Security automatically provides this parameter value when calling the `hasPermission()` method.

The framework knows the value of the authentication instance because it is already in the `SecurityContext`. In listing 11.15, you find the `DocumentsPermissionEvaluator` class, which in our example implements the `PermissionEvaluator` contract to define the custom authorization rule.

Listing 11.15 Implementing the authorization rule

```
@Component
public class DocumentsPermissionEvaluator
  implements PermissionEvaluator {      #A

  @Override
  public boolean hasPermission(
    Authentication authentication,
    Object target,
    Object permission) {

    Document document = (Document) target;  #B
    String p = (String) permission;      #C

    boolean admin =     #D
      authentication.getAuthorities()
        .stream()
        .anyMatch(a -> a.getAuthority().equals(p));

    return admin ||      #E
      document.getOwner()
        .equals(authentication.getName());

  }

  @Override
  public boolean hasPermission(Authentication authentication,
                               Serializable targetId,
                               String targetType,
                               Object permission) {
    return false;     #F
  }
}
```

To make Spring Security aware of our new `PermissionEvaluator` implementation, we have to define a `MethodSecurityExpressionHandler` bean in the configuration class. The following listing presents how to define a `MethodSecurityExpressionHandler` to make the custom

`PermissionEvaluator` known.

Listing 11.16 Configuring the `PermissionEvaluator` in the configuration class

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {


  private final DocumentsPermissionEvaluator evaluator;

  // omitted constructor

  @Bean    #A
  protected MethodSecurityExpressionHandler createExpressionHandl
    var expressionHandler =      #B
        new DefaultMethodSecurityExpressionHandler();

    expressionHandler.setPermissionEvaluator(
        evaluator);     #C

    return expressionHandler;      #D
  }

  // Omitted definition of the UserDetailsService and PasswordEnc
}
```

NOTE

We use here an implementation for `MethodSecurityExpressionHandler` named `DefaultMethodSecurityExpressionHandler` that Spring Security provides. You could as well implement a custom `MethodSecurityExpressionHandler` to define custom SpEL expressions you use to apply the authorization rules. You rarely need to do this in a real-world scenario, and for this reason, we won't implement such a custom object in our examples. I just wanted to make you aware that this is possible.

I separate the definition of the `UserDetailsService` and `PasswordEncoder` to let you focus only on the new code. In listing 11.17, you find the rest of the configuration class. The only important thing to notice about the users is their roles. User Natalie is an admin and can access any document. User Emma is a manager and can only access her own documents.

Listing 11.17 The full definition of the configuration class

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {

  private final DocumentsPermissionEvaluator evaluator;

  // Omitted constructor

  @Override
  protected MethodSecurityExpressionHandler createExpressionHandl
    var expressionHandler =
        new DefaultMethodSecurityExpressionHandler();

    expressionHandler.setPermissionEvaluator(evaluator);

    return expressionHandler;
  }

  @Bean
  public UserDetailsService userDetailsService() {
    var service = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("natalie")
              .password("12345")
              .roles("admin")
              .build();

     var u2 = User.withUsername("emma")
              .password("12345")
              .roles("manager")
              .build();

     service.createUser(u1);
     service.createUser(u2);

     return service;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

To test the application, we define an endpoint. The following listing presents

this definition.

Listing 11.18 Defining the controller class and implementing an endpoint

```
@RestController
public class DocumentController {

  private final DocumentService documentService;

  // Omitted constructor

  @GetMapping("/documents/{code}")
  public Document getDetails(@PathVariable String code) {
    return documentService.getDocument(code);
  }
}
```

Let’s run the application and call the endpoint to observe its behavior. User Natalie can access the documents regardless of their owner. User Emma can only access the documents she owns. Calling the endpoint for a document that belongs to Natalie and authenticating with the user `"natalie"`, we use this command:

```
curl -u natalie:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{
  "owner":"natalie"
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user `"natalie"`, we use this command:

```
curl -u natalie:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{
  "owner":"emma"
}
```

Calling the endpoint for a document that belongs to Emma and authenticating

with the user `"emma"`, we use this command:

```
curl -u emma:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{
  "owner":"emma"
}
```

Calling the endpoint for a document that belongs to Natalie and authenticating with the user `"emma"`, we use this command:

```
curl -u emma:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/documents/abc123"
}
```

In a similar manner, you can use the second `PermissionEvaluator` method to write your authorization expression. The second method refers to using an identifier and subject type instead of the object itself. For example, say that we want to change the current example to apply the authorization rules before the method is executed, using `@PreAuthorize`. In this case, we don't have the returned object yet. But instead of having the object itself, we have the document's code, which is its unique identifier. Listing 11.19 shows you how to change the permission evaluator class to implement this scenario. I separated the examples in a project named ssia-ch11-ex5, which you can run individually.

Listing 11.19 Changes in the `DocumentsPermissionEvaluator` class

```
@Component
public class DocumentsPermissionEvaluator
  implements PermissionEvaluator {

  private final DocumentRepository documentRepository;
```

```
  // Omitted constructor

  @Override
  public boolean hasPermission(Authentication authentication,
                               Object target,
                               Object permission) {
    return false;      #A
  }

  @Override
  public boolean hasPermission(Authentication authentication,
                                 Serializable targetId,
                                 String targetType,
                                 Object permission) {

    String code = targetId.toString();   #B
    Document document = documentRepository.findDocument(code);

    String p = (String) permission;


    boolean admin =      #C
           authentication.getAuthorities()
              .stream()
              .anyMatch(a -> a.getAuthority().equals(p));

     return admin ||      #D
       document.getOwner().equals(
         authentication.getName());
  }
}
```

Of course, we also need to use the proper call to the permission evaluator with the `@PreAuthorize` annotation. In the following listing, you find the change I made in the `DocumentService` class to apply the authorization rules with the new method.

Listing 11.20 The `DocumentService` class

```
@Service
public class DocumentService {

  private final DocumentRepository documentRepository;

  // Omitted contructor
```

```
  @PreAuthorize    #A
   ("hasPermission(#code, 'document', 'ROLE_admin')")
  public Document getDocument(String code) {
    return documentRepository.findDocument(code);
  }
}
```

You can rerun the application and check the behavior of the endpoint. You should see the same result as in the case where we used the first method of the permission evaluator to implement the authorization rules. The user Natalie is an admin and can access details of any document, while the user Emma can only access the documents she owns. Calling the endpoint for a document that belongs to Natalie and authenticating with the user `"natalie"`, we issue this command:

```
curl -u natalie:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{
  "owner":"natalie"
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user `"natalie"`, we issue this command:

```
curl -u natalie:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{
  "owner":"emma"
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user `"emma"`, we issue this command:

```
curl -u emma:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{
  "owner":"emma"
}
```

Calling the endpoint for a document that belongs to Natalie and authenticating with the user `"emma"`, we issue this command:

```
curl -u emma:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/documents/abc123"
}
```

Using the @Secured and @RolesAllowed annotations

Throughout this chapter, we discussed applying authorization rules with global method security. We started by learning that this functionality is disabled by default and that you can enable it using the `@EnableMethodSecurity` annotation over the configuration class. Moreover, when using pre and post authorization you don't need to specify a certain way to apply the authorization rules using an attribute of the `@EnableMethodSecurity` annotation. We used the annotation like this:

```
@EnableMethodSecurity
```

The `@EnableMethodSecurity` annotation offers two attributes that you can use to enable different annotations. You use the `jsr250Enabled` attribute to enable the `@RolesAllowed` annotation and the `securedEnabled` attribute to enable the `@Secured` annotation. Using these two annotations, `@Secured` and `@RolesAllowed`, is less powerful than using `@PreAuthorize` and `@PostAuthorize`, and the chances that you'll find them in real-world scenarios are small. Even so, I'd like to make you aware of both, but without spending too much time on the details.

You enable the use of these annotations the same way we did for

preauthorization and postauthorization by setting to true the attributes of the `@EnableMethodSecurity`. You enable the attributes that represent the use of one kind of annotation, either `@Secure` or `@RolesAllowed`. You can find an example of how to do this in the next code snippet:

```
@EnableMethodSecurity(
   jsr250Enabled = true,
   securedEnabled = true
)
```

Once you’ve enabled these attributes, you can use the `@RolesAllowed` or `@Secured` annotations to specify which roles or authorities the logged-in user needs to have to call a certain method. The next code snippet shows you how to use the `@RolesAllowed` annotation to specify that only users having the role ADMIN can call the `getName()` method:

```
@Service
public class NameService {

  @RolesAllowed("ROLE_ADMIN")
  public String getName() {
      return "Fantastico";
  }
}
```

Similarily, you can use the `@Secured` annotation instead of the `@RolesAllowed` annotation, as the next code snippet presents:

```
@Service
public class NameService {
  @Secured("ROLE_ADMIN")
  public String getName() {
      return "Fantastico";
  }
}
```

You can now test your example. The next code snippet shows how to do this:

```
curl -u emma:12345 http://localhost:8080/hello
```

The response body is

```
Hello, Fantastico
```

To call the endpoint and authenticating with the user Natalie, use this command:

```
curl -u natalie:12345 http://localhost:8080/hello
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

You find a full example using the `@RolesAllowed` and `@Secured` annotations in the project ssia-ch9-ex6.

11.5 Summary

- Spring Security allows you to apply authorization rules for any layer of the application, not only at the endpoint level. To do this, you enable the method security functionality.
- The method security functionality is disabled by default. To enable it, you use the `@EnableMethodSecurity` annotation over the configuration class of your application.
- You can apply authorization rules that the application checks before the call to a method. If these authorization rules aren’t followed, the framework doesn’t allow the method to execute. When we test the authorization rules before the method call, we’re using preauthorization.
- To implement preauthorization, you use the `@PreAuthorize` annotation with the value of a SpEL expression that defines the authorization rule.
- If we want to only decide after the method call if the caller can use the returned value and if the execution flow can proceed, we use postauthorization.
- To implement postauthorization, we use the `@PostAuthorize` annotation with the value of a SpEL expression that represents the authorization rule.

- When implementing complex authorization logic, you should separate this logic into another class to make your code easier to read. In Spring Security, a common way to do this is by implementing a `PermissionEvaluator`.
- Spring Security offers compatibility with older specifications like the `@RolesAllowed` and `@Secured` annotations. You can use these, but they are less powerful than `@PreAuthorize` and `@PostAuthorize`, and the chances that you'll find these annotations used with Spring in a real-world scenario are very low.

12 Implement filtering at the method level

This chapter covers

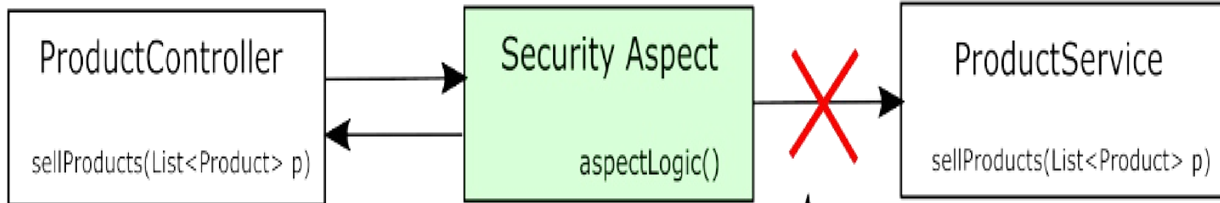
- Using prefiltering to restrict what a method receives as parameter values
- Using postfiltering to restrict what a method returns
- Integrating filtering with Spring Data

In chapter 11, you learned how to apply authorization rules using global method security. We worked on examples using the `@PreAuthorize` and `@PostAuthorize` annotations. By using these annotations, you apply an approach in which the application either allows the method call or it completely rejects the call. Suppose you don't want to forbid the call to a method, but you want to make sure that the parameters sent to it follow some rules. Or, in another scenario, you want to make sure that after someone calls the method, the method's caller only receives an authorized part of the returned value. We name such a functionality filtering, and we classify it in two categories:

- *Prefiltering*—The framework filters the values of the parameters before calling the method.
- *Postfiltering*—The framework filters the returned value after the method call.

Filtering works differently than call authorization (figure 12.1). With filtering, the framework executes the call and doesn't throw an exception if a parameter or returned value doesn't follow an authorization rule you define. Instead, it filters out elements that don't follow the conditions you specify.

Figure 12.1 The client calls the endpoint providing a value that doesn't follow the authorization rule. With preauthorization, the method isn't called at all and the caller receives an exception. With prefiltering, the aspect calls the method but only provides the values that follow the given rules.

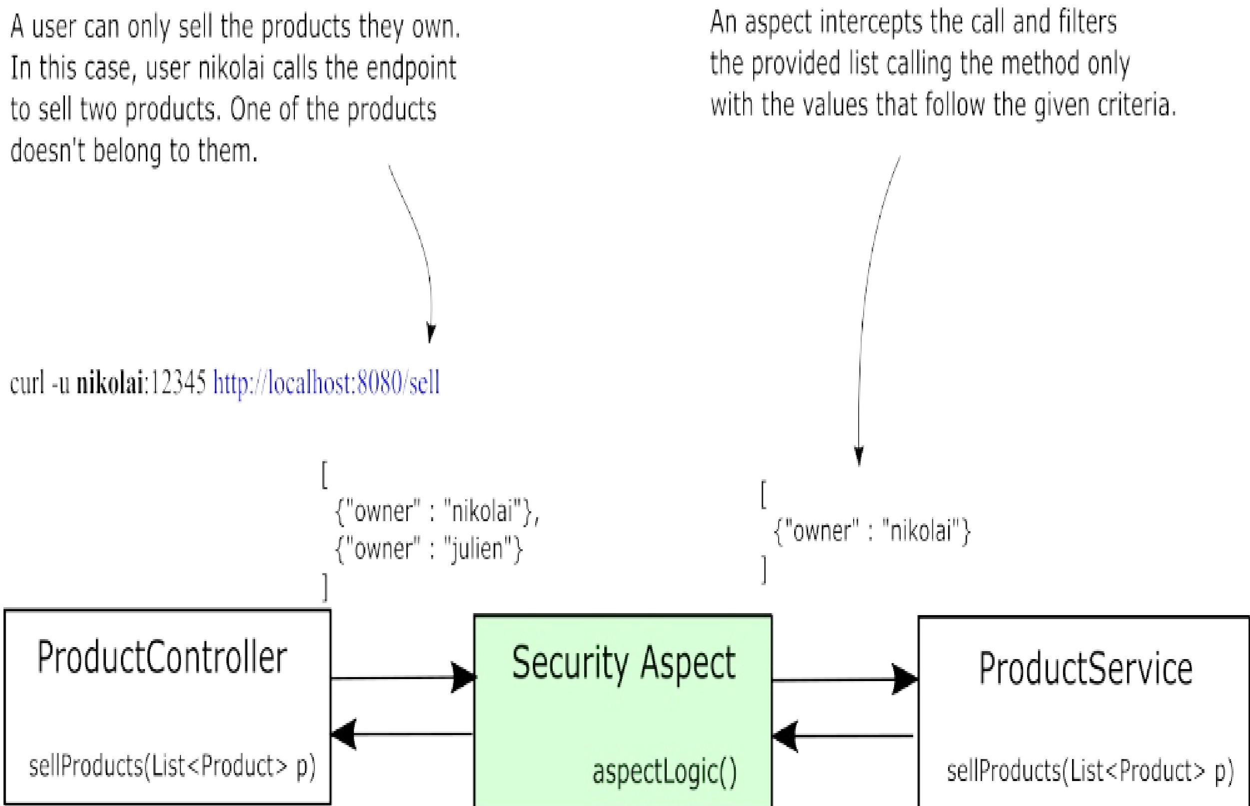


It’s important to mention from the beginning that you can only apply filtering to collections and arrays. You use prefiltering only if the method receives as a parameter an array or a collection of objects. The framework filters this collection or array according to rules you define. Same for postfiltering: you can only apply this approach if the method returns a collection or an array. The framework filters the value the method returns based on rules you specify.

12.1 Applying prefiltering for method authorization

In this section, we discuss the mechanism behind prefiltering, and then we implement prefiltering in an example. You can use filtering to instruct the framework to validate values sent via the method parameters when someone calls a method. The framework filters values that don't match the given criteria and calls the method only with values that do match the criteria. We name this functionality *prefiltering* (figure 12.2).

Figure 12.2 With prefiltering, an aspect intercepts the call to the protected method. The aspect filters the values that the caller provides as the parameter and sends to the method only values that follow the rules you define.

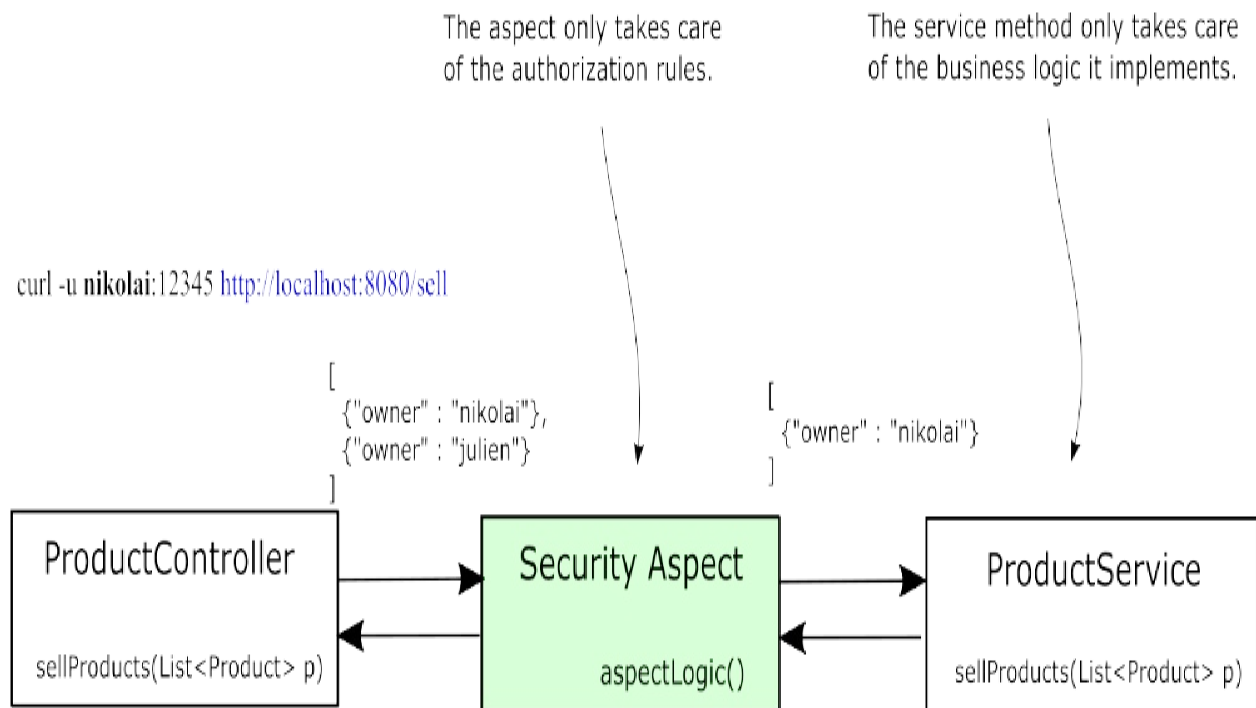


You find requirements in real-world examples where prefiltering applies well because it decouples authorization rules from the business logic the method implements. Say you implement a use case where you process only specific details that are owned by the authenticated user. This use case can be called from multiple places. Still, its responsibility always states that only details of the authenticated user can be processed, regardless of who invokes the use case. Instead of making sure the invoker of the use case correctly applies the authorization rules, you make the case apply its own authorization rules. Of

course, you might do this inside the method. But decoupling authorization logic from business logic enhances the maintainability of your code and makes it easier for others to read and understand it.

As in the case of call authorization, which we discussed in chapter 11, Spring Security also implements filtering by using aspects. Aspects intercept specific method calls and can augment them with other instructions. For prefiltering, an aspect intercepts methods annotated with the `@PreFilter` annotation and filters the values in the collection provided as a parameter according to the criteria you define (figure 12.3).

Figure 12.3 With prefiltering, we decouple the authorization responsibility from the business implementation. The aspect provided by Spring Security only takes care of the authorization rules, and the service method only takes care of the business logic of the use case it implements.



Similar to the `@PreAuthorize` and `@PostAuthorize` annotations we discussed in chapter 11, you set authorization rules as the value of the `@PreFilter` annotation. In these rules, which you provide as SpEL expressions, you use `filterObject` to refer to any element inside the collection or array that you provide as a parameter to the method.

To see prefiltering applied, let’s work on a project. I named this project ssia-

ch12-ex1. Say you have an application for buying and selling products, and its backend implements the endpoint /sell. The application's frontend calls this endpoint when a user sells a product. But the logged-in user can only sell products they own. Let's implement a simple scenario of a service method called to sell the products received as a parameter. With this example, you learn how to apply the `@PreFilter` annotation, as this is what we use to make sure that the method only receives products owned by the currently logged-in user.

Once we create the project, we write a configuration class to make sure we have a couple of users to test our implementation. You find the straightforward definition of the configuration class in listing 12.1. The configuration class that I call `ProjectConfig` only declares a `UserDetailsService` and a `PasswordEncoder`, and I annotate it with `@EnableMethodSecurity`. For the filtering annotation, we still need to use the `@EnableMethodSecurity` annotation and enable the pre-/postauthorization annotations. The provided `UserDetailsService` defines the two users we need in our tests: Nikolai and Julien.

Listing 12.1 Configuring users and enabling method security

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var uds = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("nikolai")
            .password("12345")
            .authorities("read")
            .build();

    var u2 = User.withUsername("julien")
            .password("12345")
            .authorities("write")
            .build();

    uds.createUser(u1);
    uds.createUser(u2);
```

```
    return uds;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

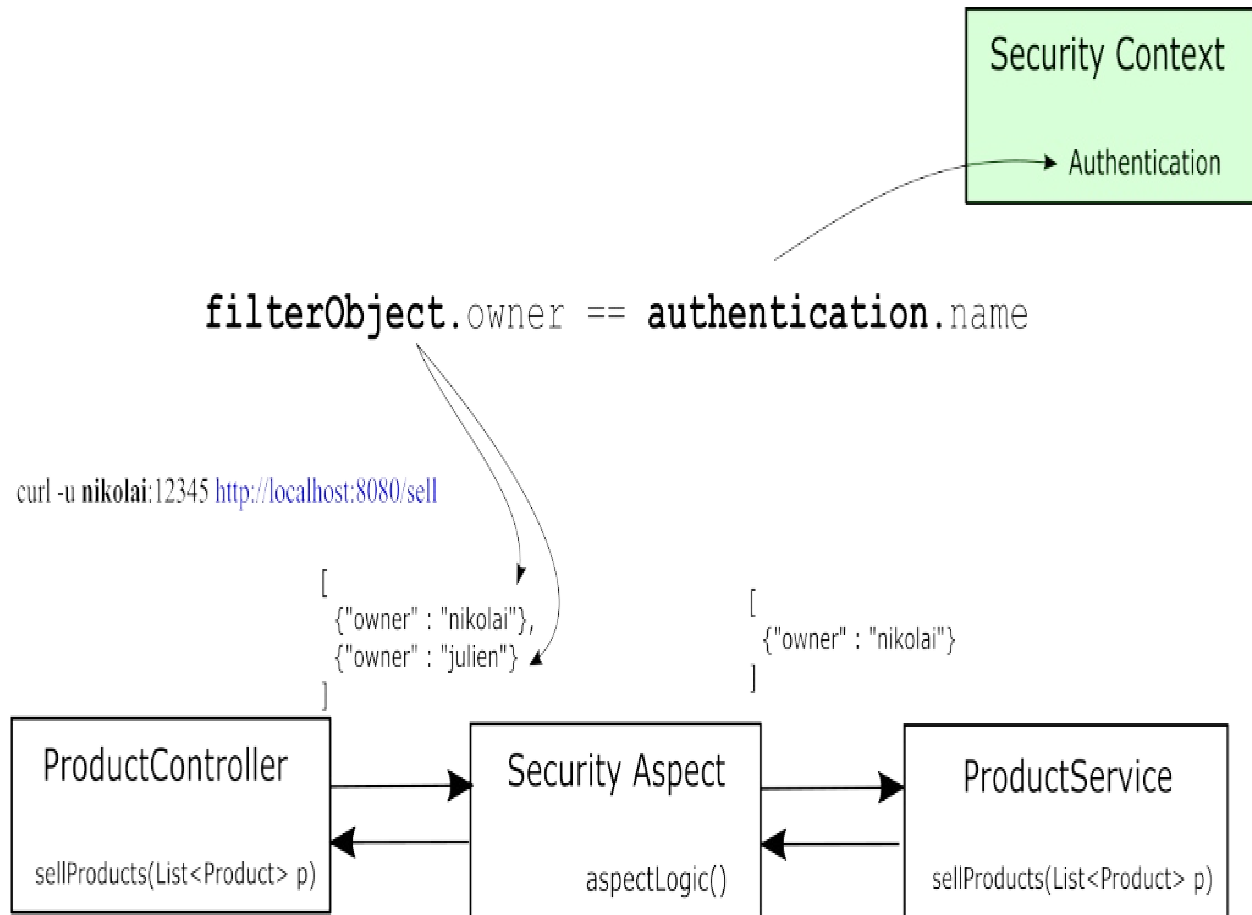
I describe the product using the model class you find in the next listing.

Listing 12.2 The `Product` class definition

```
public class Product {

  private String name;
  private String owner;      #A

  // Omitted constructor, getters, and setters
}
```

The `ProductService` class defines the service method we protect with `@PreFilter`. You can find the `ProductService` class in listing 12.3. In that listing, before the `sellProducts()` method, you can observe the use of the `@PreFilter` annotation. The Spring Expression Language (SpEL) used with the annotation is `filterObject.owner == authentication.name`, which allows only values where the `owner` attribute of the `Product` equals the username of the logged-in user. On the left side of the equals operator in the SpEL expression; we use `filterObject`. With `filterObject`, we refer to objects in the list as parameters. Because we have a list of products, the `filterObject` in our case is of type `Product`. For this reason, we can refer to the product's `owner` attribute. On the right side of the equals operator in the expression; we use the `authentication` object. For the `@PreFilter` and `@PostFilter` annotations, we can directly refer to the authentication object, which is available in the `SecurityContext` after authentication (figure 12.4).

Figure 12.4 When using prefiltering by `filterObject`, we refer to the objects inside the list that the caller provides as a parameter. The authentication object is the one stored after the authentication process in the security context.



The service method returns the list exactly as the method receives it. This way, we can test and validate that the framework filtered the list as we expected by checking the list returned in the HTTP response body.

Listing 12.3 Using the `@PreFilter` annotation in the `ProductService` class

```
@Service
public class ProductService {

  @PreFilter      #A
     ➥("filterObject.owner == authentication.name")
  public List<Product> sellProducts(List<Product> products) {
    // sell products and return the sold products list
    return products;      #B
  }
}
```

To make our tests easier, I define an endpoint to call the protected service method. Listing 12.4 defines this endpoint in a controller class called

`ProductController`. Here, to make the endpoint call shorter, I create a list and directly provide it as a parameter to the service method. In a real-world scenario, this list should be provided by the client in the request body. You can also observe that I use `@GetMapping` for an operation that suggests a mutation, which is non-standard. But know that I do this to avoid dealing with CSRF protection in our example, and this allows you to focus on the subject at hand. You learned about CSRF protection in chapter 9.

Listing 12.4 The controller class implementing the endpoint we use for tests

```
@RestController
public class ProductController {

  private final ProductService productService;

  // omitted constructor

  @GetMapping("/sell")
  public List<Product> sellProduct() {
    List<Product> products = new ArrayList<>();

    products.add(new Product("beer", "nikolai"));
    products.add(new Product("candy", "nikolai"));
    products.add(new Product("chocolate", "julien"));

    return productService.sellProducts(products);
  }
}
```

Let’s start the application and see what happens when we call the /sell endpoint. Observe the three products from the list we provided as a parameter to the service method. I assign two of the products to user Nikolai and the other one to user Julien. When we call the endpoint and authenticate with user Nikolai, we expect to see in the response only the two products associated with her. When we call the endpoint and we authenticate with Julien, we should only find in the response the one product associated with Julien. In the following code snippet, you find the test calls and their results. To call the endpoint /sell and authenticate with user Nikolai, use this command:

```
curl -u nikolai:12345 http://localhost:8080/sell
```

The response body is

```
[
  {"name":"beer","owner":"nikolai"},
  {"name":"candy","owner":"nikolai"}
]
```

To call the endpoint /sell and authenticate with user Julien, use this command:

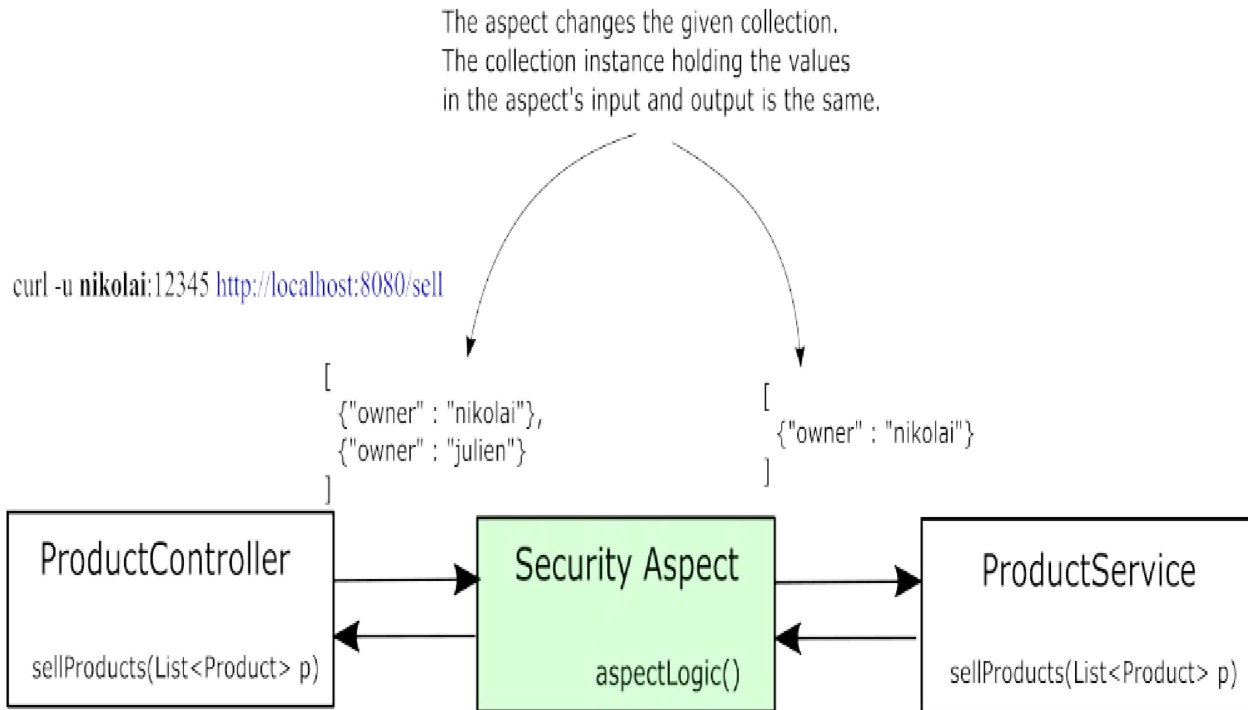
```
curl -u julien:12345 http://localhost:8080/sell
```

The response body is

```
[
  {"name":"chocolate","owner":"julien"}
]
```

What you need to be careful about is the fact that the aspect changes the given collection. In our case, don't expect it to return a new `List` instance. In fact, it's the same instance from which the aspect removed the elements that didn't match the given criteria. This is important to take into consideration. You must always make sure that the collection instance you provide is not immutable. Providing an immutable collection to be processed results in an exception at execution time because the filtering aspect won't be able to change the collection's contents (figure 12.5).

Figure 12.5 The aspect intercepts and changes the collection given as the parameter. You need to provide a mutable instance of a collection so the aspect can change it.



Listing 12.5 presents the same project we worked on earlier in this section, but I changed the `List` definition with an immutable instance as returned by the `List.of()` method to test what happens in this situation.

Listing 12.5 Using an immutable collection

```
@RestController
public class ProductController {

  private final ProductService productService;

  // omitted constructor

  @GetMapping("/sell")
  public List<Product> sellProduct() {
    List<Product> products = List.of(     #A
            new Product("beer", "nikolai"),
            new Product("candy", "nikolai"),
            new Product("chocolate", "julien"));

    return productService.sellProducts(products);
  }
}
```

I separated this example in project ssia-ch12-ex2 folder so that you can test it

yourself as well. Running the application and calling the /sell endpoint results in an HTTP response with status 500 Internal Server Error and an exception in the console log, as presented by the next code snippet:

```
curl -u julien:12345 http://localhost:8080/sell
```

The response body is:

```
{
  "status":500,
  "error":"Internal Server Error",
  "path":"/sell"
}
```

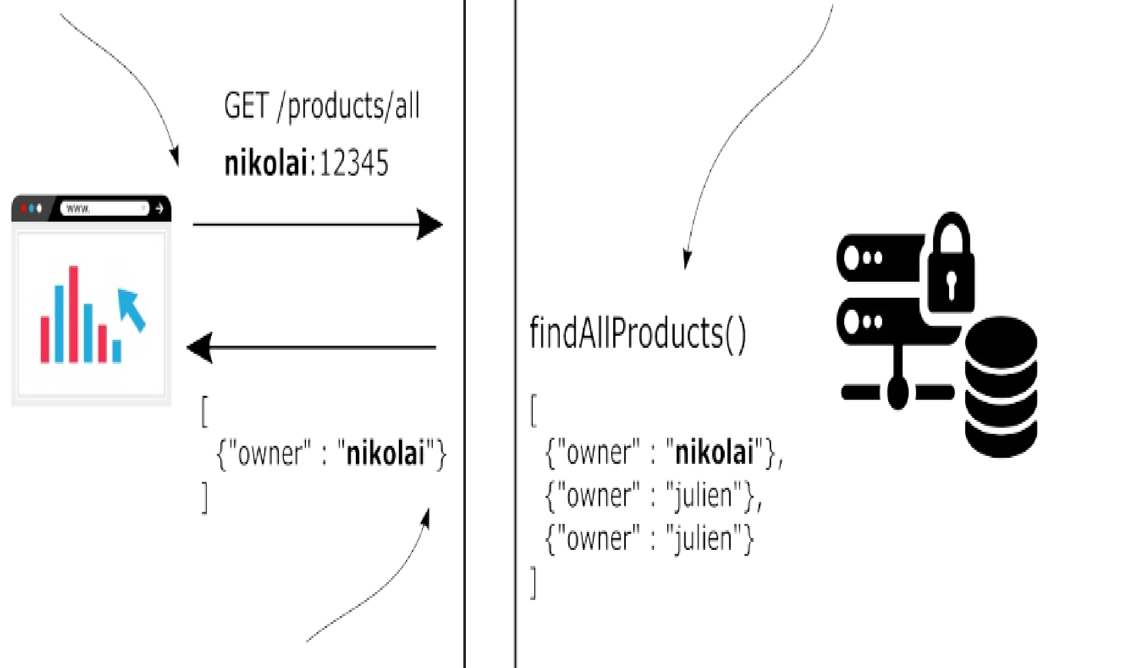
In the application console, you can find an exception similar to the one presented in the following code snippet:

```
java.lang.UnsupportedOperationException: null
    at java.base/java.util.ImmutableCollections.uoe(ImmutableColl
...
```

12.2 Applying postfiltering for method authorization

In this section, we implement postfiltering. Say we have the following scenario. An application that has a frontend implemented in Angular and a Spring-based backend manages some products. Users own products, and they can obtain details only for their products. To get the details of their products, the frontend calls endpoints exposed by the backend (figure 12.6).

Figure 12.6 Postfiltering scenario. A client calls an endpoint to retrieve data it needs to display in the frontend. A postfiltering implementation makes sure that the client only gets data owned by the currently authenticated user.

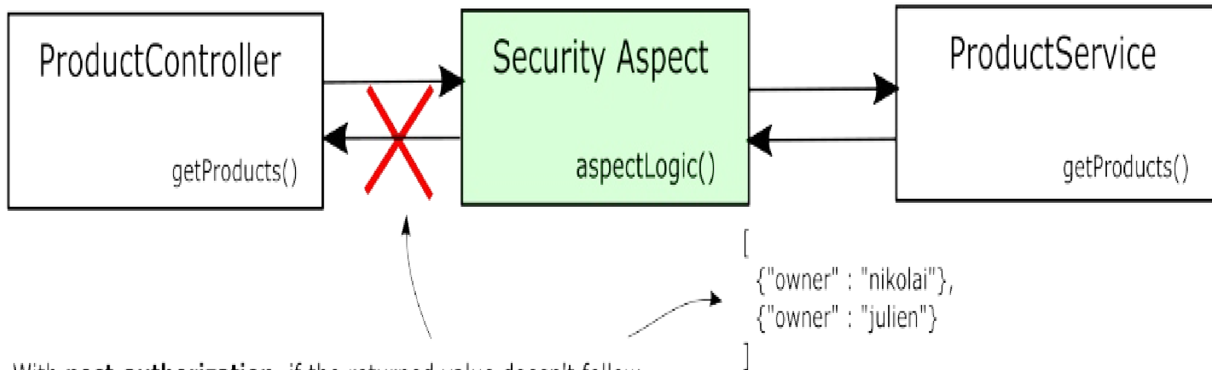


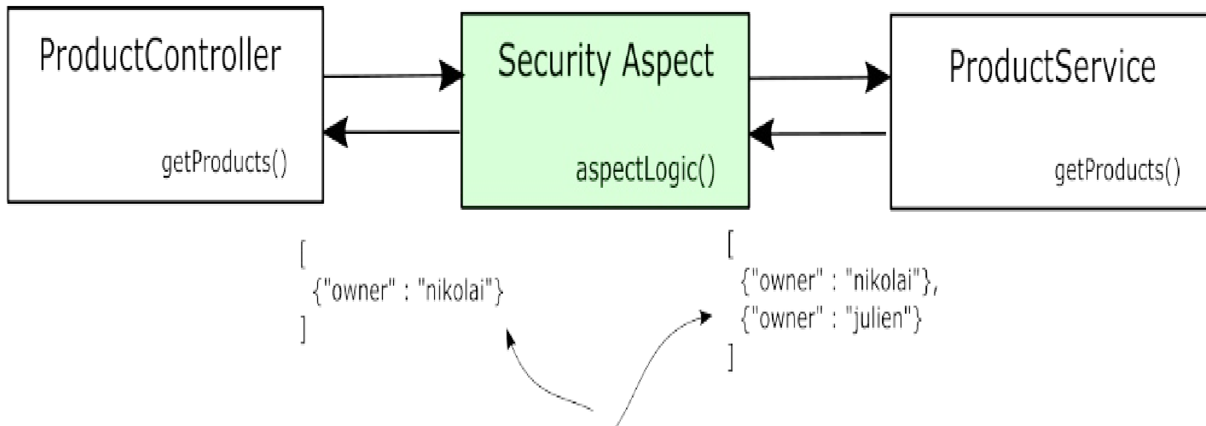
On the backend in a service class the developer wrote a method `List<Product> findProducts()` that retrieves the details of products. The client application displays these details in the frontend. How could the developer make sure that anyone calling this method only receives products they own and not products owned by others? An option to implement this functionality by keeping the authorization rules decoupled from the business rules of the application is called *postfiltering*. In this section, we discuss how postfiltering works and demonstrate its implementation in an application.

Similar to prefiltering, postfiltering also relies on an aspect. This aspect allows a call to a method, but once the method returns, the aspect takes the returned value and makes sure that it follows the rules you define. As in the case of prefiltering, postfiltering changes a collection or an array returned by the method. You provide the criteria that the elements inside the returned collection should follow. The post-filter aspect filters from the returned collection or array those elements that don’t follow your rules.

To apply postfiltering, you need to use the `@PostFilter` annotation. The `@PostFilter` annotation works similar to all the other pre-/post- annotations we used in chapter 11 and in this chapter. You provide the authorization rule as a SpEL expression for the annotation's value, and that rule is the one that the filtering aspect uses as shown in figure 12.7. Also, similar to prefiltering, postfiltering only works with arrays and collections. Make sure you apply the `@PostFilter` annotation only for methods that have as a return type an array or a collection.

Figure 12.7 Postfiltering. An aspect intercepts the collection returned by the protected method and filters the values that don't follow the rules you provide. Unlike postauthorization, postfiltering doesn't throw an exception to the caller when the returned value doesn't follow the authorization rules.





Let’s apply postfiltering in an example. I created a project named ssia-ch12-ex3 for this example. To be consistent, I kept the same users as in our previous examples in this chapter so that the configuration class won’t change. For your convenience, I repeat the configuration presented in the following listing.

Listing 12.6 The configuration class

```
@Configuration
@EnableMethodSecurity
```

```
public class ProjectConfig {

  @Bean
  public UserDetailsService userDetailsService() {
    var uds = new InMemoryUserDetailsManager();

    var u1 = User.withUsername("nikolai")
            .password("12345")
            .authorities("read")
            .build();

    var u2 = User.withUsername("julien")
            .password("12345")
            .authorities("write")
            .build();

    uds.createUser(u1);
    uds.createUser(u2);

    return uds;
  }

  @Bean
  public PasswordEncoder passwordEncoder() {
    return NoOpPasswordEncoder.getInstance();
  }
}
```

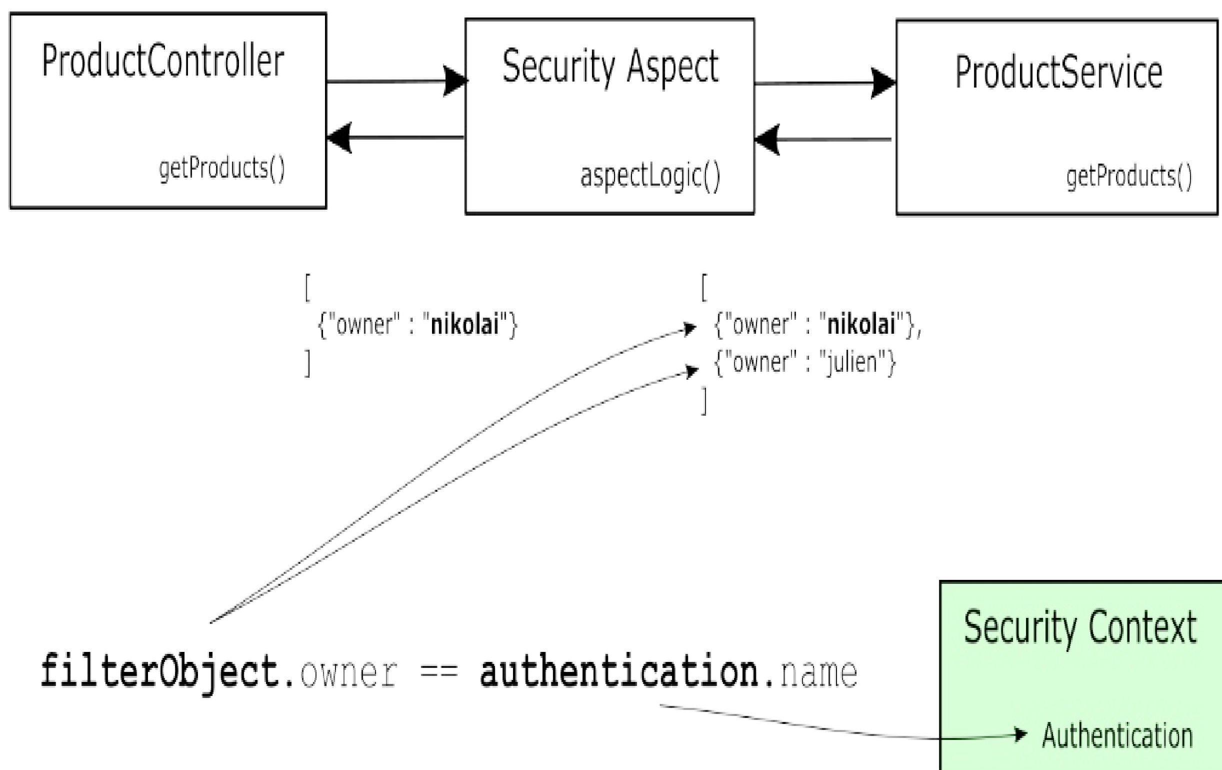
The next code snippet shows that the `Product` class remains unchanged as well:

```
public class Product {

  private String name;
  private String owner;

  // Omitted constructor, getters, and setters
}
```

In the `ProductService` class, we now implement a method that returns a list of products. In a real-world scenario, we assume the application would read the products from a database or any other data source. To keep our example short and allow you to focus on the aspects we discuss, we use a simple collection, as presented in listing 12.7.

I annotate the `findProducts()` method, which returns the list of products, with the `@PostFilter` annotation. The condition I add as the value of the annotation, `filterObject.owner == authentication.name`, only allows products to be returned that have the owner equal to the authenticated user (figure 12.8). On the left side of the equals operator, we use `filterObject` to refer to elements inside the returned collection. On the right side of the operator, we use authentication to refer to the `Authentication` object stored in the `SecurityContext`.

Figure 12.8 In the SpEL expression used for authorization, we use `filterObject` to refer to the objects in the returned collection, and we use `authentication` to refer to the `Authentication` instance from the security context.



Listing 12.7 The `ProductService` class

```
@Service
public class ProductService {

  @PostFilter     #A
```

```
    ➥("filterObject.owner == authentication.name")
  public List<Product> findProducts() {
    List<Product> products = new ArrayList<>();

    products.add(new Product("beer", "nikolai"));
    products.add(new Product("candy", "nikolai"));
    products.add(new Product("chocolate", "julien"));

    return products;
  }
}
```

We define a controller class to make our method accessible through an endpoint. The next listing presents the controller class.

Listing 12.8 The `ProductController` class

```
@RestController
public class ProductController {

  private final ProductService productService;

  // Omitted constructor

  @GetMapping("/find")
  public List<Product> findProducts() {
    return productService.findProducts();
  }
}
```

It’s time to run the application and test its behavior by calling the /find endpoint. We expect to see in the HTTP response body only products owned by the authenticated user. The next code snippets show the result for calling the endpoint with each of our users, Nikolai and Julien. To call the endpoint /find and authenticate with user Julien, use this cURL command:

```
curl -u julien:12345 http://localhost:8080/find
```

The response body is

```
[
  {"name":"chocolate","owner":"julien"}
]
```

To call the endpoint /find and authenticate with user Nikolai, use this cURL command:

```
curl -u nikolai:12345 http://localhost:8080/find
```

The response body is

```
[
  {"name":"beer","owner":"nikolai"},
  {"name":"candy","owner":"nikolai"}
]
```

12.3 Using filtering in Spring Data repositories

In this section, we discuss filtering applied with Spring Data repositories. It's important to understand this approach because we often use databases to persist an application's data. It is pretty common to implement Spring Boot applications that use Spring Data as a high-level layer to connect to a database, be it SQL or NoSQL. We discuss two approaches for applying filtering at the repository level when using Spring Data, and we implement these with examples.

The first approach we take is the one you learned up to now in this chapter: using the `@PreFilter` and `@PostFilter` annotations. The second approach we discuss is direct integration of the authorization rules in queries. As you'll learn in this section, you need to be attentive when choosing the way you apply filtering in Spring Data repositories. As mentioned, we have two options:

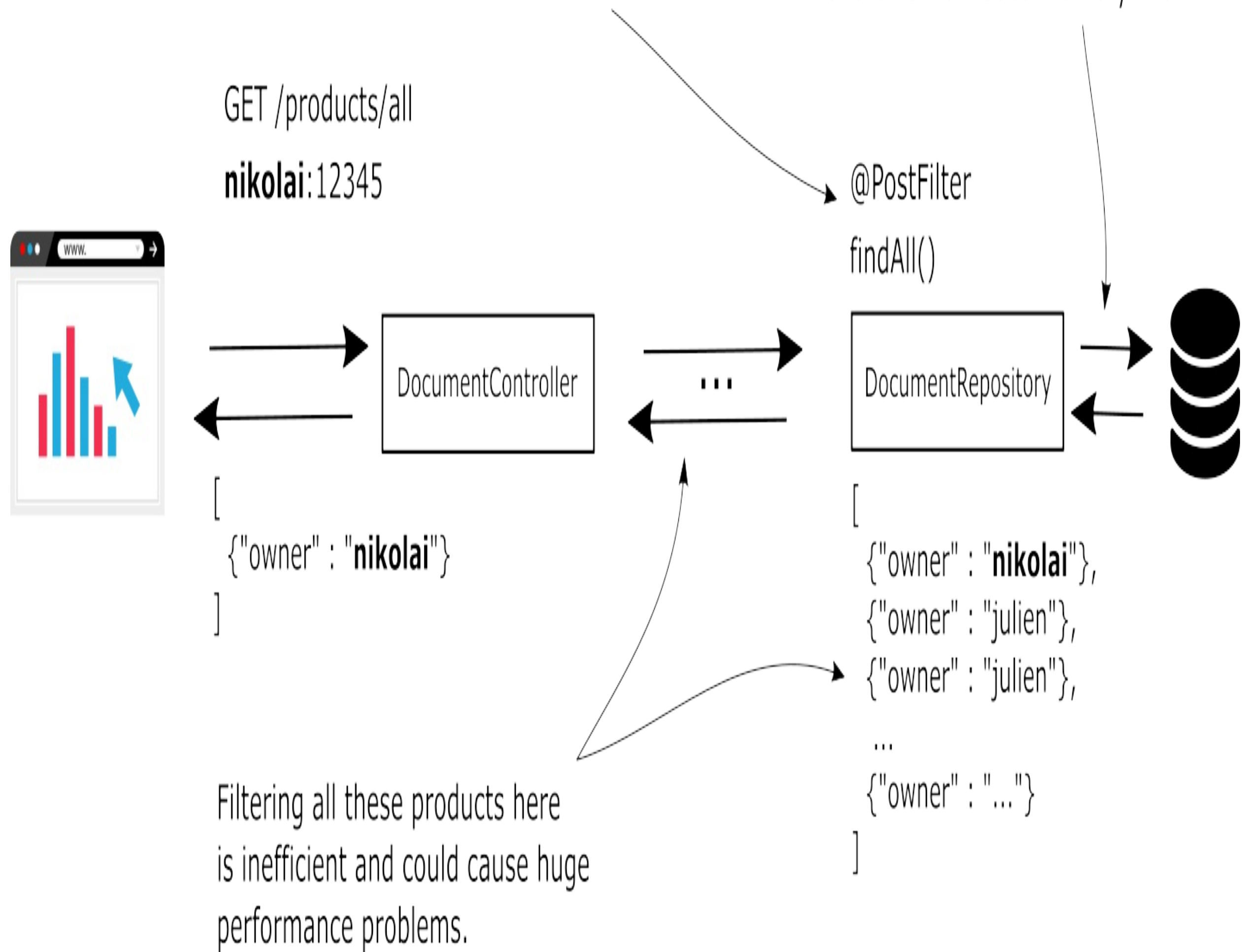
- Using `@PreFilter` and `@PostFilter` annotations
- Directly applying filtering within queries

Using the `@PreFilter` annotation in the case of repositories is the same as applying this annotation at any other layer of your application. But when it comes to postfiltering, the situation changes. Using `@PostFilter` on repository methods technically works fine, but it's rarely a good choice from a performance point of view.

Say you have an application managing the documents of your company. The

developer needs to implement a feature where all the documents are listed on a web page after the user logs in. The developer decides to use the `findAll()` method of the Spring Data repository and annotates it with `@PostFilter` to allow Spring Security to filter the documents such that the method returns only those owned by the currently logged-in user. This approach is clearly wrong because it allows the application to retrieve all the records from the database and then filter the records itself. If we have a large number of documents, calling `findAll()` without pagination could directly lead to an `OutOfMemoryError`. Even if the number of documents isn't big enough to fill the heap, it's still less performant to filter the records in your application rather than retrieving at the start only what you need from the database (figure 12.9).

Figure 12.9 The anatomy of a bad design. When you need to apply filtering at the repository level, it's better to first make sure you only retrieve the data you need. Otherwise, your application can face heavy memory and performance issues.



At the service level, you have no other option than to filter the records in the app. Still, if you know from the repository level that you need to retrieve only records owned by the logged-in user, you should implement a query that extracts from the database only the required documents.

NOTE

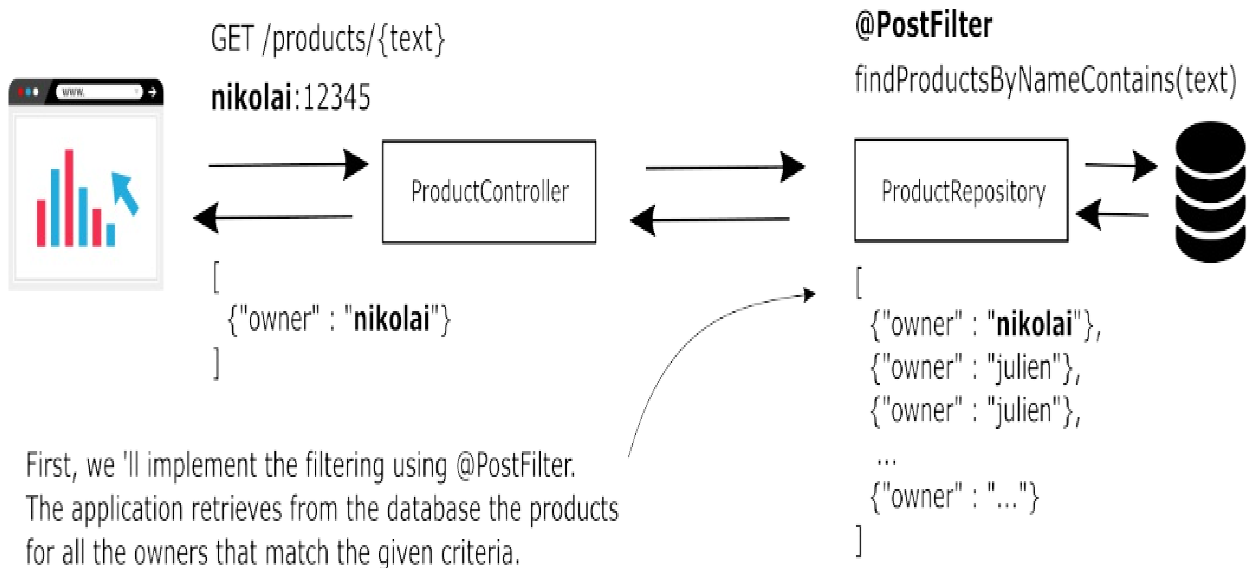
In any situation in which you retrieve data from a data source, be it a database, a web service, an input stream, or anything else, make sure the application retrieves only the data it needs. Avoid as much as possible the

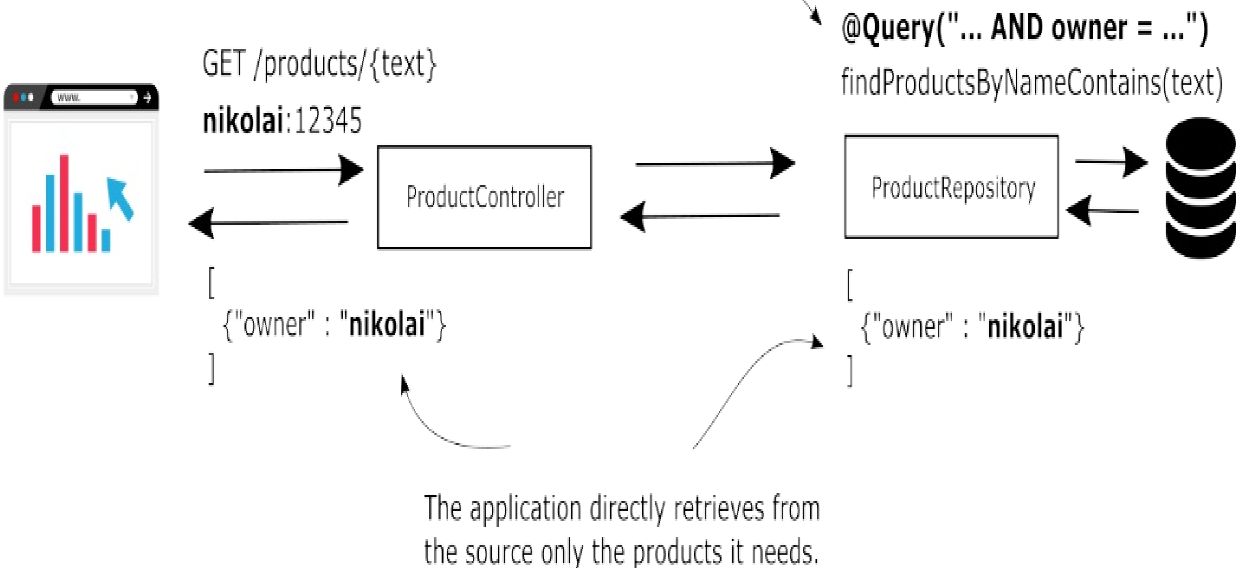
need to filter data inside the application.

Let’s work on an application where we first use the `@PostFilter` annotation on the Spring Data repository method, and then we change to the second approach where we write the condition directly in the query. This way, we have the opportunity to experiment with both approaches and compare them.

I created a new project named ssia-ch12-ex4, where I use the same configuration class as for our previous examples in this chapter. As in the earlier examples, we write an application managing products, but this time we retrieve the product details from a table in our database. For our example, we implement a search functionality for the products (figure 12.10). We write an endpoint that receives a string and returns the list of products that have the given string in their names. But we need to make sure to return only products associated with the authenticated user.

Figure 12.10 In our scenario, we start by implementing the application using `@PostFilter` to filter products based on their owner. Then we change the implementation to add the condition directly on the query. This way, we make sure the application only gets from the source the needed records.





We use Spring Data JPA to connect to a database. For this reason, we also need to add to the pom.xml file the `spring-boot-starter-data-jpa` dependency and a connection driver according to your database management server technology. The next code snippet provides the dependencies I use in the pom.xml file:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <scope>runtime</scope>
</dependency>
```

In the application.properties file, we add the properties Spring Boot needs to create the data source. In the next code snippet, you find the properties I added to my application.properties file:

```
spring.datasource.url=jdbc:mysql://localhost/spring
➥?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

We also need a table in the database for storing the product details that our application retrieves. We define a schema.sql file where we write the script for creating the table, and a data.sql file where we write queries to insert test data in the table. You need to place both files (schema.sql and data.sql) in the resources folder of the Spring Boot project so they will be found and executed at the start of the application. The next code snippet shows you the query used to create the table, which we need to write in the schema.sql file:

```
CREATE TABLE IF NOT EXISTS `spring`.`product` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL,
  `owner` VARCHAR(45) NULL,
  PRIMARY KEY (`id`));
```

In the data.sql file, I write three INSERT statements, which the next code snippet presents. These statements create the test data that we need later to prove the application’s behavior.

```
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VAL
```

```
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VAL
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VAL
```

NOTE

Remember, we used the same names for tables in other examples throughout the book. If you already have tables with the same names from previous examples, you should probably drop those before starting with this project. An alternative is to use a different schema.

To map the product table in our application, we need to write an entity class. The following listing defines the `Product` entity.

Listing 12.9 The `Product` entity class

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String owner;

    // Omitted getters and setters
}
```

For the `Product` entity, we also write a Spring Data repository interface defined in the next listing. Observe that this time we use the `@PostFilter` annotation directly on the method declared by the repository interface.

Listing 12.10 The `ProductRepository` interface

```
public interface ProductRepository
        extends JpaRepository<Product, Integer> {

    @PostFilter      #A
    ➥("filterObject.owner == authentication.name")
    List<Product> findProductByNameContains(String text);
}
```

The next listing shows you how to define a controller class that implements

the endpoint we use for testing the behavior.

Listing 12.11 The `ProductController` class

```
@RestController
public class ProductController {

  private final ProductRepository productRepository;

  // Omitted constructor

  @GetMapping("/products/{text}")
  public List<Product> findProductsContaining(
    @PathVariable String text) {

    return productRepository.findProductByNameContains(text);
  }
}
```

Starting the application, we can test what happens when calling the /products/{text} endpoint. By searching the letter *c* while authenticating with user Nikolai, the HTTP response only contains the product *candy*. Even if *chocolate* contains a *c* as well, because Julien owns it, *chocolate* won't appear in the response. You find the calls and their responses in the next code snippets. To call the endpoint /products and authenticate with user Nikolai, issue this command:

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is

```
[
  {"id":2,"name":"candy","owner":"nikolai"}
]
```

To call the endpoint /products and authenticate with user Julien, issue this command:

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is

```
[
```

```
  {"id":3,"name":"chocolate","owner":"julien"}
]
```

We discussed earlier in this section that using `@PostFilter` in the repository isn't the best choice. We should instead make sure we don't select from the database what we don't need. So how can we change our example to select only the required data instead of filtering data after selection? We can provide SpEL expressions directly in the queries used by the repository classes. To achieve this, we follow two simple steps:

1. We add an object of type `SecurityEvaluationContextExtension` to the Spring context. We can do this using a simple `@Bean` method in the configuration class.
2. We adjust the queries in our repository classes with the proper clauses for selection.

In our project, to add the `SecurityEvaluationContextExtension` bean in the context, we need to change the configuration class as presented in listing 12.12. To keep all the code associated with the examples in the book, I use here another project that named ssia-ch12-ex5.

Listing 12.12 Adding the `SecurityEvaluationContextExtension` to the context

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {

  @Bean    #A
  public SecurityEvaluationContextExtension
    securityEvaluationContextExtension() {

    return new SecurityEvaluationContextExtension();
  }

    // Omitted declaration of the UserDetailsService and Password
}
```

In the `ProductRepository` interface, we add the query prior to the method, and we adjust the WHERE clause with the proper condition using a SpEL expression. The following listing presents the change.

Listing 12.13 Using SpEL in the query in the repository interface

```
public interface ProductRepository
        extends JpaRepository<Product, Integer> {

    @Query("SELECT p FROM Product p
    ➥WHERE p.name LIKE %:text% AND      #A
          ➥p.owner=?#{authentication.name}")
    List<Product> findProductByNameContains(String text);
}
```

We can now start the application and test it by calling the /products/{text} endpoint. We expect that the behavior remains the same as for the case where we used `@PostFilter`. But now, only the records for the right owner are retrieved from the database, which makes the functionality faster and more reliable. The next code snippets present the calls to the endpoint. To call the endpoint /products and authenticate with user Nikolai, we use this command:

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is

```
[
  {"id":2,"name":"candy","owner":"nikolai"}
]
```

To call the endpoint /products and authenticate with user Julien, we use this command:

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is

```
[
  {"id":3,"name":"chocolate","owner":"julien"}
]
```

12.4 Summary

- Filtering is an authorization approach in which the framework validates the input parameters of a method or the value returned by the method

and excludes the elements that don't fulfill some criteria you define. As an authorization approach, filtering focuses on the input and output values of a method and not on the method execution itself.

- You use filtering to make sure that a method doesn't get other values than the ones it's authorized to process and can't return values that the method's caller shouldn't get.
- When using filtering, you don't restrict access to the method, but you restrict what can be sent via the method's parameters or what the method returns. This approach allows you to control the input and output of the method.
- To restrict the values that can be sent via the method's parameters, you use the `@PreFilter` annotation. The `@PreFilter` annotation receives the condition for which values are allowed to be sent as parameters of the method. The framework filters from the collection given as a parameter all values that don't follow the given rule.
- To use the `@PreFilter` annotation, the method's parameter must be a collection or an array. From the annotation's SpEL expression, which defines the rule, we refer to the objects inside the collection using `filterObject`.
- To restrict the values returned by the method, you use the `@PostFilter` annotation. When using the `@PostFilter` annotation, the returned type of the method must be a collection or an array. The framework filters the values in the returned collection according to a rule you define as the value of the `@PostFilter` annotation.
- You can use the `@PreFilter` and `@PostFilter` annotations with Spring Data repositories as well. But using `@PostFilter` on a Spring Data repository method is rarely a good choice. To avoid performance problems, filtering the result should be, in this case, done directly at the database level.
- Spring Security easily integrates with Spring Data, and you use this to avoid issuing `@PostFilter` with methods of Spring Data repositories.