

# Spring Security IN ACTION

Laurențiu Spilcă



MANNING



**MEAP Edition  
Manning Early Access Program  
Spring Security in Action  
Version 7**

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to  
[manning.com](https://manning.com)

©Manning Publications Co. To comment go to [liveBook](#)

Licensed to Ashwin Sakharkar <[ashwinsakharkar.44@gmail.com](mailto:ashwinsakharkar.44@gmail.com)>

# welcome

---

Thank you for purchasing the MEAP of Spring Security in Action. You've already made a significant step forward by understanding the importance of security in software applications. I hope you will find plenty of things that will help you secure your Spring applications. Thanks to your help, the book will become a great learning resource for a lot of people when it's finished.

Spring framework is the leading choice for developing enterprise applications. For applications developed with the Spring framework, Spring Security is the first choice for establishing application-level security. An increasing number of developers understand how important security is and what the repercussions could be when it hasn't been given the proper attention. Most people participating in the development of a software system today are aware of OWASP and at least OWASP's top ten vulnerabilities.

This book is about using Spring Security in Spring applications. To get the most out of this book, you should already know how to use the Spring framework and Spring Boot for application development.

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

- How Spring Security works and its application in different scenarios, from using it with the standard servlet-based applications to reactive applications
- Usability of Spring Security according to your application's architecture, from small applications to using OAuth2 and OpenID Connect in bigger systems
- Various ways to apply, and best practices for applying, configurations in Spring Security
- Writing unit tests for the functionality related to Spring Security
- What to take into consideration when deploying your applications in an orchestrated environment

I have used Spring and Spring Security for some years already, with a variety of systems from the technical perspective to the business domain. However, as you might already be aware, this subject's complexity makes touching all the cases a challenging task. This is why your feedback is invaluable and will significantly help in the improvement of this book.

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

-- Laurentiu Spilca

# *brief contents*

---

## **PART 1: FIRST STEPS**

- 1 Security today*
- 2 Hello Spring Security*

## **PART 2: IMPLEMENTATION**

- 3 Managing users*
- 4 Dealing with passwords*
- 5 Implementing authentication*
- 6 Hands-on: A small secured web application*
- 7 Configuring authorization: restricting access*
- 8 Configuring authorization: applying restrictions*
- 9 Implementing filters*
- 10 Applying CSRF protection and CORS*
- 11 Hands-on: A separation of responsibilities*
- 12 How does OAuth 2 work?*
- 13 OAuth 2 – Implementing the authorization server*
- 14 OAuth 2 – Implementing the resource server*
- 15 OAuth 2 – Using JWT and cryptographic signatures*
- 16 Global Method Security – Pre/Post Authorization*
- 17 Global Method Security – Pre/Post Filtering*
- 18 Hands-on: An OAuth 2 application*

*19 Spring Security for reactive apps*

*20 Spring Security testing*

## APPENDIXES

*A Creating the Spring Boot project*

# Part 1.

## *First steps*

Security is one of the essential non-functional qualities of a software system. One of the crucial aspects you'll learn in this book is that you should consider security from the very beginning stages of the development of an app. In part 1, we start by discussing the place of security in the development process of an application in chapter 1. Then, in chapter 2, I'll introduce you to the basic components of the Spring Security's backbone architecture by implementing a few straightforward projects.

The purpose of this part is to get you started with Spring Security, especially if you just started learning this framework. However, even if you already consider you to know some aspects of application-level security and the underlying architecture of Spring Security, I still recommend you read this part as a refresher.

# 1

## *Security today*

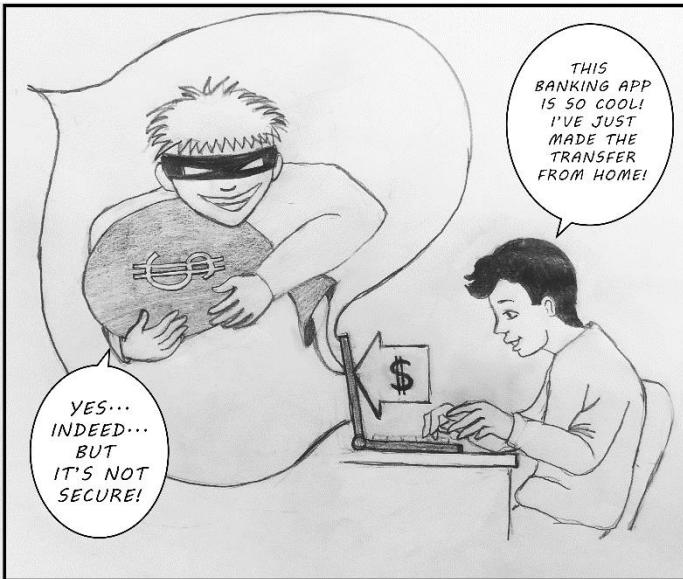
### This chapter covers

- 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
- Common vulnerabilities that you'll encounter at the application level

Today more and more developers are becoming aware of security. It's not, unfortunately, a common practice to take responsibility for security from the beginning of the development of a software application. This attitude should change, and everyone involved in developing a software system should learn to consider security from the start.

Generally, as developers, we start by learning that the purpose of an application is to solve business cases. 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 ages of learning development, 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 functionalities, there are lots of aspects hidden in the final result.

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



**Figure 1.1** A user mainly thinks about the functional requirements. Sometimes you might see them aware also of performance – which is non-functional, but it's quite unusual unfortunately that they care about security. Non-functional requirements tend to be more transparent than functional ones.

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

But before starting, I'd like to make you aware of the following: depending on how much experience you have, you might or not find this chapter cumbersome. Don't worry too much if you don't understand absolutely all the aspects at the moment. For now, with this chapter, I want to show you a big picture of the security-related concepts. Throughout the book, we'll 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 more details. Here and there, you'll find references to other materials (books, articles, documentation) that are useful for you to read further on specific subjects.

## 1.1 Spring Security - the what and the why

In this section, we'll 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 them.

If we'd go to the official website, <https://spring.io/projects/spring-security>, they describe Spring Security as a powerful and highly customizable framework for authentication and access control. I would simply say it is a framework that enormously simplifies 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 open-source software released under the Apache 2.0 license. You can access the source code of the project on GitHub at <https://github.com/spring-projects/spring-security/>, and I highly recommend that you contribute to the project as well.

**NOTE** You can use Spring Security for both standard web servlets as well as reactive applications. To use it, you need at least Java 8, although the examples in this book use Java 11, which is the latest long term supported version.

I can guess that if you opened this book, you work on Spring applications, and you are interested in securing them. Spring Security is most probably the best choice for your cases. It anyway became the de-facto choice in implementing the application-level security for Spring applications. Spring Security, however, doesn't automatically secure your application. It's not some kind of magic that guarantees a vulnerability-free app. Developers need to understand how to configure and customize Spring Security around the needs of the application. 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 very simple. You already implement 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 them based on configurations you specify. And let me refer only to using annotations to make these configurations and leave behind the old-fashioned XML configuration style!

So you use annotations to instruct 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. This is where Spring Security comes into action. So what you want is to use, as comfortable already, annotations, beans, and in general Spring-fashioned configuration style to define your application-level security. If you think of 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 it completely unsecured. Or you can decide not to allow everyone to enter your home.

The way you configure security could be straightforward, like hiding your key under the rug, or it could be more complicated, like choosing a variety of alarm systems, video cameras, and locks. In your applications, you have the option of doing the same. 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 refers to deciding whether an entity 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 ensure whoever makes the requests has the permissions to access protected resources. The developer configures components, so they 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 a fault of the alarm system that it didn't trigger when someone forced a window.

Other responsibilities of these components also relate to the data storing as well as transiting data between different parts of the systems. By intercepting the calls to these different parts, the components can act on the data. When the data is stored, these components may apply encryption or hashing algorithms. The data encodings keep the data accessible only to privileged entities. In the 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 components, communicating can't trust each other. How could 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 side 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. You have to know the part to configure and set it up in your system. This way, Spring Security intercepts the messages and makes sure to validate the communication before the application uses any kind of data sent or received.

Like any framework, one of its primary purposes 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, as well, allows you to configure any of its components, providing great flexibility.

Short recap on this discussion:

- You use Spring Security to bake application-level security in your applications in the “Spring way”. By this, I mean, you'll use annotations, beans, Spring Expression Language (SpEL), and so on.
- Spring Security is a framework that allows you to build application-level security. However, it is up to 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 will provide you the information you need to effectively use Spring Security.

---

### **Alternatives to Spring Security**

This book is about Spring Security. But as for any other 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 configurations and is easy to integrate with Spring and Spring Boot applications. Apache Shiro makes, sometimes, 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 and use. It offers its own annotations and design for web applications based on HTTP filters, which are of great simplicity for web applications. Also, you can secure more than just web applications with Shiro, from smaller command-line applications and mobile applications to large-scale enterprise applications. And even if simpler, 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 you 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.

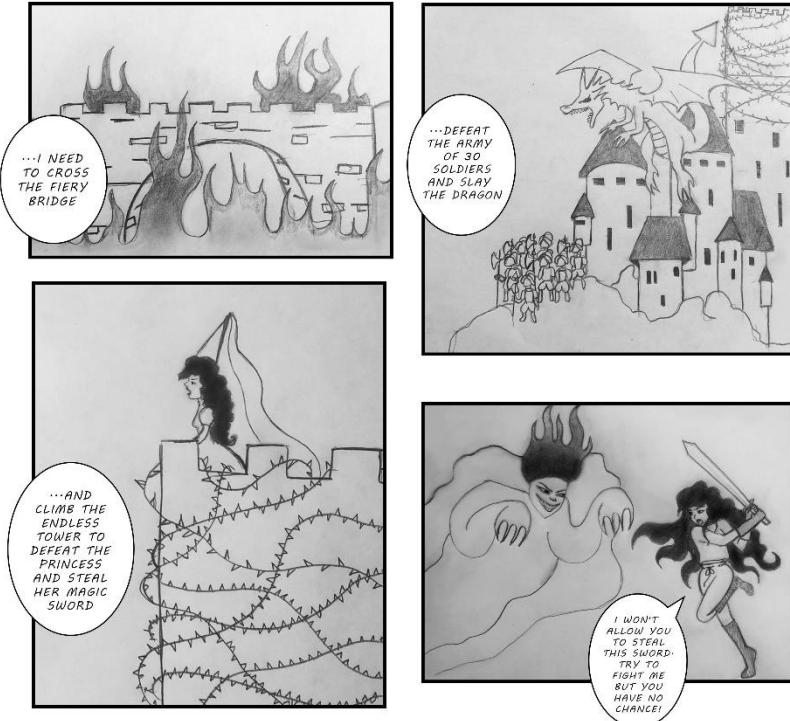
---

## 1.2 What is software security?

Software systems today manage large amounts of data, out of which a significant part can be considered sensitive, especially given the current General Data Protection Regulations (GDPR) requirements. Any information that you, as a user, consider private is sensitive for your software application. Sensitive data could include inoffensive 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 other parties than the users to whom they are intended should be able to interact in any way with it. Loosely defined, this is the meaning of security.

**NOTE** GDPR created a lot of buzz around the world 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 having users in Europe. The owners of such applications risk significant penalties if they don't respect the regulations imposed.

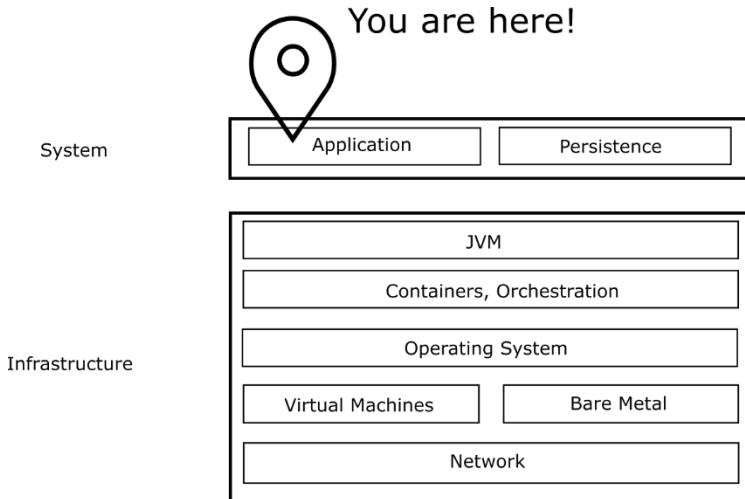
We apply security in layers, with each layer requiring a different approach. Compare these layers to a protected castle (figure 1.2). A hacker needs to bypass several obstacles to obtain the resources managed by the app. The better you secure each layer, the lower the chance a bad-intentioned individual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



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

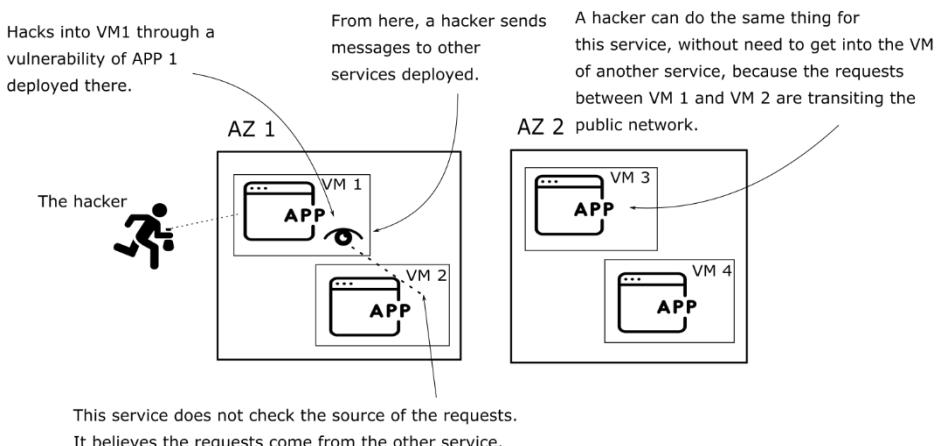
Security is a complex subject. In the case of a software system, security doesn't apply only at the application level. For example, for networking, there are issues to be taken into consideration and specific practices to be used, while for the storage, it's another discussion. Similarly, there's a different philosophy to be known 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.

*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. Each of the layers depends on those below them. In this book, we discuss Spring Security, which is a framework used to implement application-level security.

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.



**Figure 1.4.** If a malicious user manages to get access to the VM, and there's no applied application-level security, a hacker will gain control of the other applications in the system. If the 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 such architectures, we could encounter various vulnerabilities, so you should exercise attention. As mentioned earlier, security is a cross-cutting concern which 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 the 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 "bridge of fire".

With this in mind, let's consider that an individual driven by bad intentions would be able to log in to the virtual machine that's hosting the first application. Let's also assume the second application doesn't validate the requests sent by the first application. The attacker would be able to exploit this vulnerability and control the second application by impersonating the first one.

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

**NOTE** An *availability zone* in terms of cloud deployment is a separate data center. This data center is situated far enough geographically (and has other dependencies) than other data centers of the same region. This way, it's considered that if one availability zone is failing, 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 the public network.

---

### Monolithic and microservices

The discussion on monolithic and microservices architectural styles is a whole different tome. I refer to them from multiple places in this book, so you should at least be aware of the terminology. For an excellent discussion of the two architectural styles, I recommend that you read Chris Richardson's *Microservices Patterns* (Manning, 2018).

By *monolithic architecture*, we refer to an application in which we implement all the responsibilities in the same executable artifact. Consider this as one application that fulfills all the use cases. The responsibilities can sometimes be implemented within different modules to make the application more comfortable to maintain. But you can't separate the logic of one from the logic of others at runtime. Generally, monolithically architectures offer less flexibility for scaling and deployment management.

A *microservice system* has the responsibilities implemented within different executable artifacts. You can see the system as being formed of multiple applications that execute at the same time and communicate between them when needed via the network. While this offers more flexibility for scaling, it introduces other difficulties. We can enumerate here latencies, security concerns, network reliability, distributed persistence, and deployment management.

---

I referred earlier to authentication and authorization. And indeed, these often present in most of the applications. Through authentication, an application identifies a user (a person or another application). The purpose of identifying it is to be able to decide afterward what they should be allowed to do - authorization. I'll detail quite a lot on authentication and authorization, starting with chapter 3 and throughout the book.

In an application, you'll often find the need to implement authorization in different scenarios. Consider another situation: most applications have restrictions regarding who should the user be to access certain functionality. Achieving this implies first the need to identify who creates an access request for a specific feature - authentication. As well, we need to know their privileges to allow them 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 documents of a user. Should you simply share the credentials of the user with the printer? But that allows the printer more rights than it needs! 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 don't only want to answer, but for which we'll also see applications with Spring Security in this book.

Depending on the chosen architecture for the system, you find authentication and authorization at the level of the entire system, as well as for any of the components. And as we'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 16, we'll discuss more on global methods security, which refers to this aspect. The design gets even more complicated as you can have a predefined set of roles and authorities.

I would also like to bring to your attention the 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 either to keep the data encrypted with a private key, or hashed. Secrets, like credentials and private keys, can also be considered data at rest. They 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 for a long time, sensitive data like credentials or private keys. 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, for sure, need to understand (as well as apply) and test solutions for all the layers that collaborate within a system. However, in this book, we focus on presenting you with all the details for what you specifically need to understand in terms of Spring Security. You'll find out from the previous description 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 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 they have access to your data. They 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? As well as the data, some actions are more important than others. You don't care very much about some of them, while others are significant. Maybe for you, it's not that important if one 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 one could read their messages. The application should make sure to protect everything to the desired degree of access. Any leak that would allow the use of data, functionalities, as well as the application to affect other systems, is considered a vulnerability and you should solve it.

Not respecting 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 could lose profitability. It isn't only about directly losing money from a bank account or using a service without paying for it. These are indeed more direct ways which imply costs. The image of a brand or company is also valuable, and losing a good image can be expensive: sometimes even more costly than the direct expenses resulted from the exploit of a vulnerability in the system! The trust that the 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 would you see them as a user? How can these affect the organization responsible for the software?

1. A back-office application should manage the internal data of an organization but, somehow, some information leaks out.
2. Users of a ride-sharing application observe that money is debited from their accounts on behalf of trips that aren't theirs.
3. 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, could be affected. In some instances, the company could be liable, and a significant amount of money could be lost. In this situation, the users don't have the choice to change the application, but the organization could decide to change the provider of the software system.

In the second case, the users will probably choose to change the service provider. The image would be dramatically affected by the company developing the application. The cost lost in terms of money, in this case, is much less than the cost in terms of image. Even if the

payment is returned to the affected users, the application will lose users, which will affect profitability and can even lead to bankruptcy.

The third case could have dramatic consequences on the bank in terms of the 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 or third, it could be a lack of method access control. And for all of them, 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 system was affected? What about systems that control nuclear power? You can reduce the risk by investing early in the security of the application and by allocating enough time for security professionals to develop and test the security mechanism.

**NOTE** The lessons learned from those who failed before you are that the cost of an attack is usually higher than the investment 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 would never be enough written words on 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 Common security vulnerabilities in web applications

Before discussing how to apply security in applications, you should first know what you're protecting the application from. To do something malicious, an attacker identifies and exploits vulnerabilities of your application. We often describe vulnerability as a weakness that could allow the execution of actions that are unwanted and usually done with malicious intentions.

An excellent start to understanding vulnerabilities is being aware of the Open Web Application Security Project, also known as OWASP (<https://www.owasp.org>). At OWASP, you'll also find descriptions of the most common vulnerabilities that you should avoid in your applications. Let's take a few minutes and discuss those theoretically before diving into the next chapters, where you'll start to apply concepts from Spring Security. Among the common vulnerabilities that you should be aware of, you'll find:

- Broken authentication
- Session fixation
- Cross-site scripting (XSS)
- Cross-site request forgery (CSRF)
- Injections
- Sensitive data exposure
- Lack of method access control
- Using dependencies with known vulnerabilities

These items are related to application-level security, and most of them are also directly related to using Spring Security. We'll discuss their relationship with Spring Security and how to protect your application from them in detail in this book, but first, an overview.

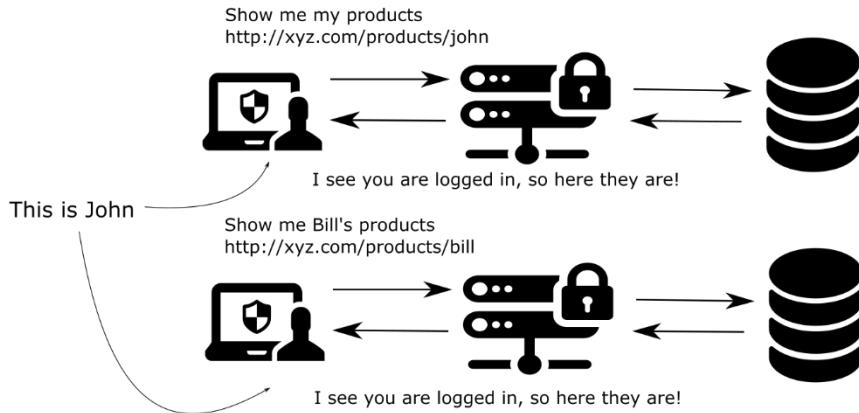
### 1.4.1 Vulnerabilities in authentication and authorization

In this book, we discuss authentication and authorization in-depth, and you'll learn several ways in which you can implement it with Spring Security. *Authentication* represents the process in which an application identifies someone trying to use it. When someone or something uses the app, we want to find their identity so that further access is granted or not. In real-world apps, you'll also find cases in which the access is anonymous, but in most cases, one can use data or do specific actions only when identified. Once we have the identity of the user, we can process the authorization.

*Authorization* is the process of establishing if an authenticated caller has the privileges to use specific functionality and data. For example, in a mobile banking application, most of the authenticated users can transfer money, but only from their account.

We can say that we have a broken authorization if a bad intentioned individual could somehow gain access to functionality or data which doesn't belong to them. Frameworks like Spring Security help in making this vulnerability less possible, but if not used correctly, there's still a chance that this might happen. For example, you could use Spring Security to define the access to specific endpoints for an authenticated individual with a particular role. If there's no restriction at the data level, one might find a way to use data that belongs to another user.

Take a look at figure 1.5. An authenticated user can access the `/products/{name}` endpoint. From the browser, a web app calls this endpoint to retrieve and display the user's products from a database. But what happens if the app doesn't check to whom the products belong when returning them? Some user could find a way to get the details of another user. This situation is just one of the examples that should be taken into consideration from the beginning of the design of the application so that you can avoid them.



**Figure 1.5** A user that is logged in can see their products. But if the application server only checks if the user is logged in, then the user could call the same endpoint to retrieve the products of some other user. This way, John was able to see data that belongs to Bill. The issue that causes the problem is that the application doesn't authenticate the user for data retrieval as well.

Throughout the book, we'll refer to vulnerabilities. We'll discuss vulnerabilities starting with the basic configuration of the authentication and authorization, in chapter 3. Then, we'll discuss how vulnerabilities relate to the integration of Spring Security and Spring Data and how to design the application to avoid those with OAuth2.

#### 1.4.2 What is session fixation?

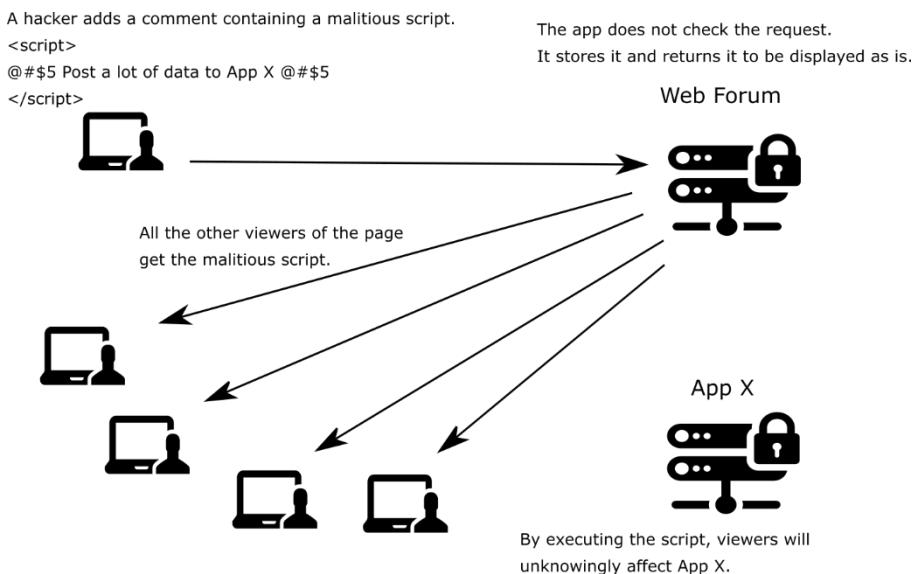
*Session fixation* vulnerability is a more specific, high severity weakness of a web application. If present, it permits an attacker to impersonate a valid user by reusing a previously generated session ID. This vulnerability could happen if, during the authentication process, the web application does not assign a unique session ID, and this could make possible the reuse of existing session IDs. Exploiting this vulnerability consists of obtaining a valid session ID and making the intended victim's browser use it.

Depending on how you implement the web application, there are various ways an individual can use this vulnerability. For example, if the application provides the session ID in the URL, then the victim could be tricked into clicking on a malicious link. If the application uses a hidden attribute, then the attacker can fool the victim into using a foreign form and post the action to the server. If the application stores the value of the session in a cookie, then a script could be injected, and the attacker could force the victim's browser to execute it.

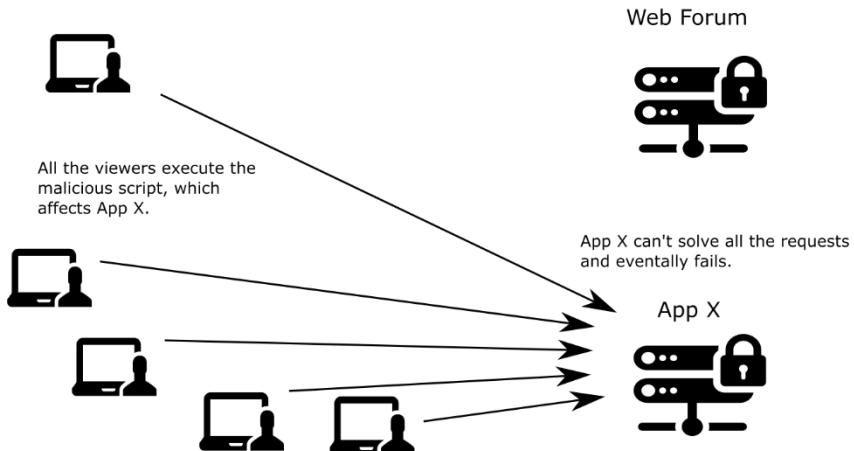
### 1.4.3 What is cross-site scripting (XSS)?

Cross-site scripting, also referred to as XSS, allows the injection of client-side scripts into web services exposed by the server, thereby permitting other users to run them. Before being used, or even stored, the request should be properly “sanitized” to avoid undesired executions of foreign scripts. The potential impact could relate to account impersonation (for example, combined with session fixation) or to participation in distributed attacks like DDoS.

Let's take an example. A user posts a message or a comment in a web application. After posting the message, the site displays it so that everybody visiting the page can see it. Hundreds could visit this page daily, depending on how popular the site. For the sake of our example, we'll consider it a known site, and a significant number of individuals visit its pages. What if this user posts a script that, when found on a web page, the browser executes (figure 1.6 and figure 1.7).



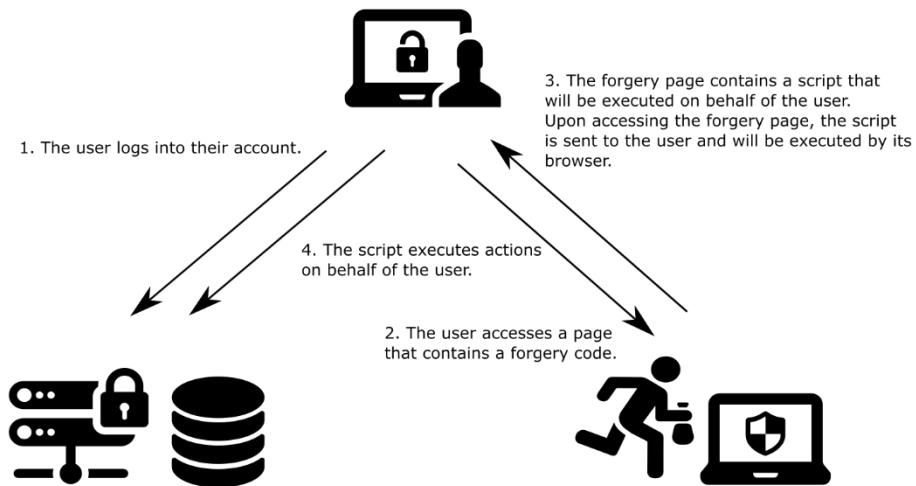
**Figure 1.6** A user posts a comment containing a script on a web forum. The user has defined the script such that it will make requests that try to post or get massive amounts of data from another application (App X), which represents the victim of the attack. If the Web Forum app allows cross-site scripting (XSS), all the users who display the page with the malicious comment will receive it as it is.



**Figure 1.7** The users access a page that displays a malicious script. Their browsers execute the script and then tries to post or get substantial amounts of data from App X.

#### 1.4.4 What is cross-site request forgery (CSRF)?

Cross-Site Request Forgery (CSRF) vulnerabilities are also common in web applications. CSRF attacks assume a URL that calls an action on a specific server can be extracted and reused from outside of the application (figure 1.8). If the server trusts the execution without doing any check on the origin of the request, one could execute it from any other place. Through CSRF, an attacker could make a user execute undesired actions on a server by hiding the actions. Usually, with this vulnerability, the attacker targets actions that change data in the system.



**Figure 1.8 Steps of a cross-site request forgery (CSRF).** After logging into their account, the user accesses a page that contains forgery code. The malicious code could then execute actions on behalf of the unsuspecting user.

One of the ways of mitigating this vulnerability is to use tokens to identify the request or use Cross-Origin Resource Sharing (CORS) limitations. In other words, validate the origin of the request. We look closer at how to deal with CSRF and CORS with Spring Security in chapter 10.

#### 1.4.5 Understanding injection vulnerabilities in web applications

Injection attacks on systems are widespread. In an injection attack, the attacker, employing a vulnerability, introduces specific data into the system. The purpose is to harm the system, change data in an unwanted way, or retrieve data that's not meant to be accessed by them.

There are many types of injection attacks. Even the XSS that we mention in section 1.4.3 can be considered an injection vulnerability. In the end, injection attacks inject a client-side script with the means of harming the system somehow. Other examples could be SQL injection, XPath injection, OS command injection, LDAP injection, and the list continues.

Injection types of vulnerabilities are important, and the results of exploiting them could be change, deletion, or access to data in the systems being compromised. For example, if your application is somehow vulnerable to LDAP injection, an attacker could try to benefit from bypassing the authentication and, from there, control essential parts of the system. The same could happen for XPath injection or OS command injection.

One of the oldest, and maybe the most known type of injection vulnerability, is SQL injection. If your application has an SQL injection vulnerability, an attacker could try to change or run different SQL queries to alter, delete, or extract data from your system. In the most

advanced SQL injection attacks, an individual can run OS commands on the system, and this would lead to a full system compromise.

#### 1.4.6 Dealing with the exposure of sensitive data

Even if, in terms of complexity, the disclosure of confidential data seems to be the easiest to understand and the least complex of the vulnerabilities, it remains one of the most common mistakes. Maybe this happens because the majority of tutorials and examples found online as well as books illustrating different concepts define the credentials directly in the configuration files for simplicity reasons. In case of a hypothetical example that eventually focuses on something else, this makes sense.

**NOTE** Most of the time, developers learn continuously from theoretical examples. Generally, examples are simplified to allow the reader to focus on a specific topic. But a downside of this simplification is that developers get used to wrong approaches. Developers might mistakenly think that everything they read is a good practice.

How is this aspect related to Spring Security? Well, we'll deal with credentials and private keys in the examples in this book. We might use secrets in configuration files, but we'll place a note for these cases to remind us that you should store sensitive data in vaults. Naturally, for a developed system, the developers aren't allowed to see the values for these sensitive keys in all of the environments. Usually, at least for production, only a small group of people should be allowed to access the private data.

By setting such values in the configuration files, such as the `application.properties` or `application.yml` files in a Spring Boot project, you make those private values accessible to anyone who can see the source code. Moreover, you might also find yourself storing all the history of these value changes in your version management system for source code.

Also related to the exposure of sensitive data is the information in logs written by your application to the console or stored in databases such as Splunk or Elasticsearch. I have often seen logs that disclose sensitive data forgotten by the developers.

**NOTE** Never log something that isn't public information. By public, I mean something that can be seen or accessed by anyone. Things like private keys or certificates aren't public and shouldn't be logged together with your error, warnings, or info messages.

Next time you log something from your application, make sure it doesn't look like one of the messages below:

```
[error] The signature of the request is not correct. The correct key to be used should have been X.

[warning] Login failed for username: X and password Y. User with username X has password Z.

[info] A login was performed with success by the user X with password Y.
```

Be careful of what your server returns to the client, especially, but not limited to the cases when the application encounters exceptions. Often, by lack of time or experience, developers

forget to implement all the cases. This way, and usually happening after a wrong request, the application returns too many details, which expose the implementations. This application behavior is also a vulnerability through data exposure. If your app encountered a `NullPointerException` because the request was wrong (part of it was missing, for example), then the exception shouldn't appear in the body of the response. At the same time, the HTTP status should be 400 rather than 500. HTTP status codes of type 4XX are designed to represent problems on the client-side. A wrong request is, in the end, an issue of the client, so the application should represent it accordingly. HTTP status codes of type 5XX are designed to inform that there was a problem on the server. Do you see something wrong in the response presented by the next snippet?

```
{
  "status": 500,
  "error": "Internal Server Error",
  "message": "Connection not found for IP Address 10.2.5.8/8080",
  "path": "/product/add"
}
```

The message of the exception seems to be disclosing an IP address. An attacker could use this address to understand the network configuration and eventually find a way to control the virtual machines in your infrastructure. Of course, with only this piece of data, one could not do any harm. But collecting different disclosed pieces of information and putting them together could provide everything that's needed to adversely affect a system. Having exception stacks in the response is not a good choice either; for example:

```
at
  java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:
  1128) ~[na:na]
at
  java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java
  :628) ~[na:na]
at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
 ~[tomcat-embed-core-9.0.26.jar:9.0.26]
at java.base/java.lang.Thread.run(Thread.java:830) ~[na:na]
```

Not only that, but this approach also discloses the application's internal structure. From the stack of an exception, you could see the naming notations as well as objects used for specific actions and the relationships between them. But even worse than that, logs sometimes can disclose versions of dependencies that your application uses. (Did you spot that Tomcat-core version in the above exception stack?)

We should avoid using vulnerable dependencies. However, if we find ourselves using a vulnerable dependency by mistake, at least we don't want to point this mistake out explicitly. Even if the dependency isn't known as a vulnerable one, this could be because nobody has found the vulnerability yet. Exposures as the previous snippet could motivate an attacker to find vulnerabilities in that specific version because they know now that's what your system uses. It's inviting them to harm your system. And an attacker could use even the smallest detail against a system.

```
Response A:
{
  "status": 401,
```

```

    "error": "Unauthorized",
    "message": "Username is not correct",
    "path": "/login"
}
Response B:
{
    "status": 401,
    "error": " Unauthorized",
    "message": "Password is not correct",
    "path": "/login"
}

```

In this example, the responses A and B are different results of calling the same authentication endpoint. They don't seem to expose any information related to the class design or system infrastructure, but they hide another problem. If the messages disclose context information, then they can as well hide vulnerabilities. The different messages based on different inputs provided to the endpoint can be used to understand the context of execution. In this case, they could be used to know when a username is correct, and only the password is wrong. And this can make the system more liable to a brute force attack. The response provided back to the client shouldn't help in identifying a possible guess of a specific input. In this case, it should have better provided in both situations the same message:

```

{
    "status": 401,
    "error": " Unauthorized",
    "message": "Username or password is not correct",
    "path": "/login"
}

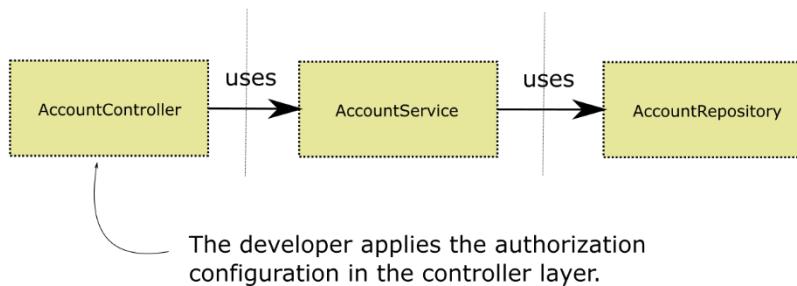
```

It could look small, but if not taken and in the right context, exposing sensitive data could become an excellent tool to be used against the system.

#### **1.4.7 What is the lack of method access control?**

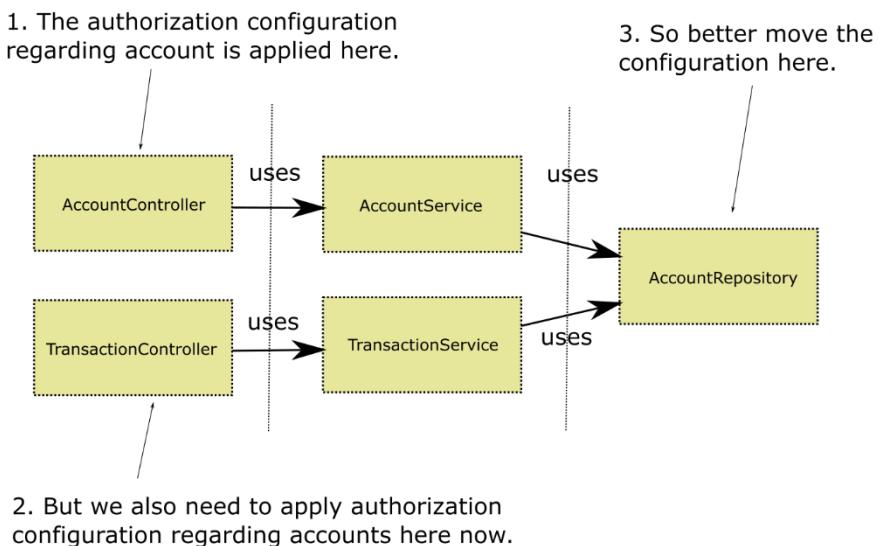
Even at the application level, you don't apply authorization to only one of the tiers. Sometimes it's a must to make sure that a particular use case can't be called at all (for example, if the privileges of the currently authenticated user don't allow it).

Say you have a web application with a straightforward design. The app has a controller exposing endpoints. The controller directly calls a service that implements some logic, and that uses persisted data managed through a repository (figure 1.9). Imagine a situation where the authorization is done only at the endpoint level (assuming that you can access the method through a REST endpoint). A developer might be tempted to apply authorization rules only in the controller layer, as presented in figure 1.9.



**Figure 1.9** A developer applies the authorization rules at the controller layer. But the repository does not know and does not restrict the retrieval of data anyhow. If a service asked for accounts that don't belong to the currently authenticated user, the repository would return them.

While the case presented in figure 1.9 works correctly, applying the authorization rules only at the controller layer might leave room for error. In this case, some future implementation could expose that use case without testing or without testing all the authorization requirements. In figure 1.10, you can see what could happen if a developer adds another functionality that depends on the same repository.



**Figure 1.10** The newly added TransactionController makes use of the AccountRepository in its dependency chain. The developer must reapply the authorization rules in this controller as well. But it would be much

better if the repository itself made sure that data that doesn't belong to the authenticated user is not exposed.

These situations might appear, and you might need to treat them at any layer in your application, not just with the repository. We'll discuss more things related to this subject in chapters 16 and 17. In chapters 16 and 17, you'll learn how we can apply restrictions per application tier when this is needed, as well as the cases when we should avoid doing this.

#### **1.4.8 Using dependencies with known vulnerabilities**

Although not necessarily related directly to Spring Security, but still an essential aspect of the application-level security, we come to paying attention to the dependencies used. Sometimes it's not the application you develop that has vulnerabilities, but the dependencies like libraries or frameworks that you use to build the functionality. You should always be attentive to the dependencies you use and eliminate any version that's known to contain a vulnerability.

Fortunately, we have multiple possibilities for static analyses, quickly done by adding a plugin to your Maven or Gradle configuration. The majority of the applications today are developed based on open source. Even Spring Security is an open-source framework. This development methodology is great, and it allows for fast evolution, but this rush could also make us more error-prone.

When developing any piece of software, we have to take all the needed measures to make sure that we avoid the use of any dependency that was proven to have known vulnerabilities. If we discover we've used such a dependency, then we not only have to correct this fast, we also have to investigate if the vulnerability was already exploited and take the needed measures.

### **1.5 Security applied in various architectures**

In this section, we'll discuss applying security practices depending on the design of your system. It's important to understand that different software architectures imply different possible leaks and vulnerabilities. I want to make you aware of this first chapter about this philosophy to which I'll refer throughout the book. Architecture strongly influences the choices in configuring Spring Security for your applications, so do the functional and non-functional requirements. Even when you think of a real-life situation, to protect something, depending on what you want to protect, you'll use a metal door, bulletproof glass, or a barrier. You couldn't just use a metal door in all the situations. If what you protect is an expensive painting in a museum, you still want people to be able to see it. You don't want them to be able to touch it, damage it, or even take it with them. In this case, the functional requirements are the ones affecting the solution we take for secure systems.

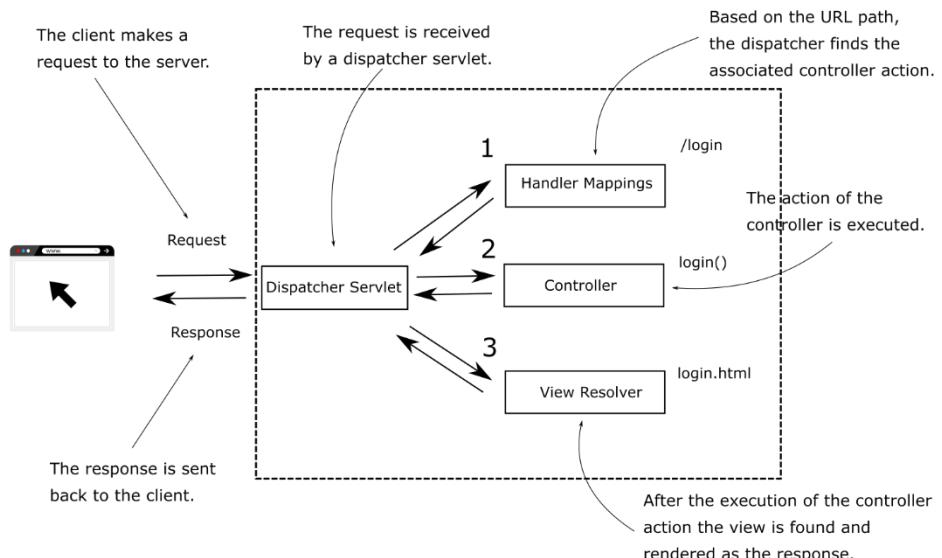
It could be that you need to make a good compromise with other quality attributes like, for example, performance. It's like using your heavy metal door instead of a lightweight barrier at the parking entrance. You could do that, and for sure, the metal door would be more secure, but it takes much more time to open and close it. The time and cost of opening and closing the heavy door aren't worth it. Of course, assuming that this isn't some kind of special parking for expensive cars.

Because the security approach is different depending on the solution we implement, the configuration in Spring Security is also different. In this section, we discuss some examples based on different architectural styles as well as other requirements, how the security approach is affected, and what we should take into consideration. These aspects are linked to all the configurations that we'll work on in the following chapters with Spring Security.

In this section, I present some of the practical scenarios you might have to deal with and with which we'll work over in the rest of the book. For a more detailed discussion on these aspects, I recommend you also read the *Microservices Security in Action* by Prabath Siriwardena and Nuwan Dias (Manning 2019).

### 1.5.1 Designing a one-piece web application

Let's start with the case where you develop a component of the system that represents a web application. In this application, there's no direct separation in development between the backend and the frontend. The way we usually see these kinds of applications is through the general servlet flow: the application receives an HTTP request and sends back an HTTP response to a client. Sometimes we might have a server-side session for each client to store specific details over more HTTP requests. In the examples provided in the book, we use Spring MVC (figure 1.11).



**Figure 1.11** A minimal representation of the Spring MVC flow. The Dispatcher Servlet finds the mapping of the requested path to the controller method (1), executes the controller method (2), and obtains the rendered view (3). The HTTP response is then delivered back to the requester, whereby the browser interprets and displays the response.

You'll find a great discussion about developing web applications and REST services with Spring in Craig Walls's *Spring In Action*, Sixth Edition (Manning, 2020):

<https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-2/>

<https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-6/>

As long as you have a session, you have to take into consideration the session fixation vulnerability as well as the CSRF possibilities previously mentioned. You must also consider what you store in the HTTP session itself.

Server-side sessions are quasi-persistent. They are stateful pieces of data, so their lifetime is longer. The longer they stay in the memory, the more it's statistically probable that they'll be accessed. For example, a person having access to the heap dump could read the information in the app's internal memory. And don't think that the heap dump is challenging to obtain! Especially when developing your applications with Spring Boot, you might find that the actuator is also part of your application. The Spring Boot actuator is a great tool. Depending on how you configure it, the Spring Boot actuator could return a heap dump with only an endpoint call. That is, you don't necessarily need root access to the VM to get your dump.

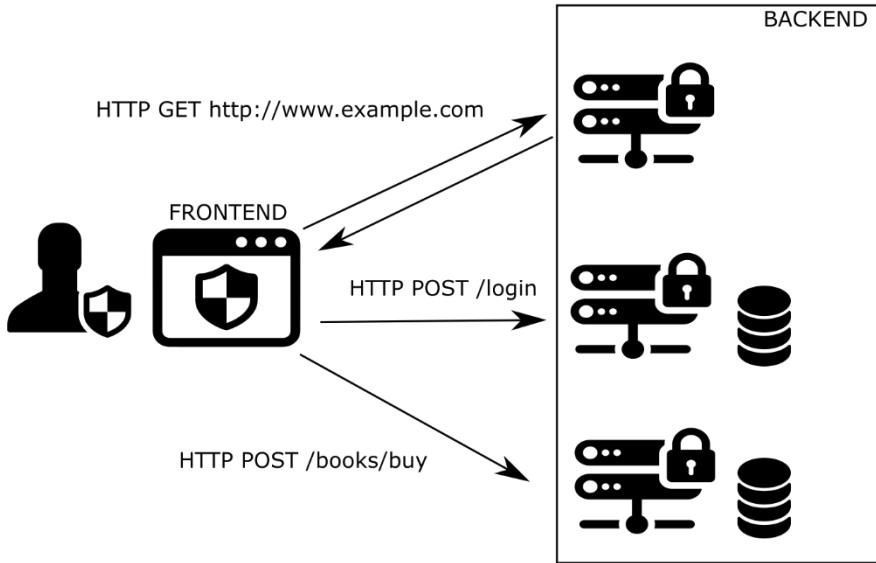
Going back to the vulnerabilities. In terms of CSRF, in this case, the easiest way to mitigate the vulnerability would be to use anti-CSRF tokens. Fortunately, with Spring Security, this capability is available out of the box. CSRF protection, as well as validation of the origin CORS, is enabled by default. You'll have to disable it if you don't want it explicitly. For authentication and authorization, you could choose to use the implicit login form configuration from Spring Security. With this, you'd benefit from only needing to override the look-and-feel of the login and logout and from the default integration with the authentication and authorization configuration. You'd also benefit from the mitigation of the session fixation vulnerability.

If you've authentication and authorization, it also means that you should have some users with valid credentials. Depending on your choice, you could have your application managing the credentials for the users, or you could choose to benefit from another system to do this (for example, you might want to let the user log in with their Facebook, GitHub, or LinkedIn credentials). In any of these cases, Spring Security helps you with a relatively easy way of configuring user management. You can choose to store user information in a database, use a web-service, or connect to another platform. The abstractions used in Spring Security's architecture make it decoupled, which allows you to choose any implementation fit for your application.

### 1.5.2 Designing security for a backend/frontend separation

Nowadays, we more often see in the development of web applications a choice in the segregation of the frontend and the backend (figure 1.12). In these web applications, developers use today a framework like Angular, ReactJS, or Vue.js to develop the frontend. The frontend communicates with the backend through REST endpoints. We'll implement examples to apply Spring Security for these architectures, starting with chapter 11.

We typically avoid using server-side sessions; client-side sessions replace them. This kind of system design is also similar to the one used in the case of mobile applications. Applications that run on Android or iOS operating systems, which can be native or simple progressive web applications, would call a backend through REST endpoints.



**Figure 1.12** The browser executes a frontend application. This application calls REST endpoints exposed by the backend to perform some operations requested by the user.

In terms of security, there are some other aspects to be taken into consideration. First, CSRF and CORS configurations are usually more complicated. You might want to scale the system horizontally and not necessarily to have the frontend with the backend at the same origin. For mobile applications, we can't even talk about an origin. The most straightforward but least desirable approach in a practical solution would be to use HTTP Basic for the endpoint authentication. While this approach is direct to understand and generally used with the first theoretical examples of authentication, it does have leaks that you'd want to avoid. For example, using HTTP Basic implies sending the credentials with each call. As you'll see in chapter 2, credentials aren't encrypted. The browser sends the username and the passwords as a Base64 encoding. This way, the credentials are left available on the network in the header of each endpoint call. Secondly, assuming that the credentials represent the user that's logged in, you don't want the user to type the credentials for every request. You also don't want to have to store the credentials on the client-side. This practice is again, not advisable.

Having the above reasons in mind, you'll learn in chapter 12, an alternative for authentication and authorization that offers a better approach: the OAuth2 flow, but the following section provides an overview for you.

#### A short reminder of application scalability

Scalability refers to the quality of a software application in which it can serve more or fewer requests while adapting the resources used, without a need to change it or its architecture. Mainly we classify scalability in two types: *vertical* and *horizontal*.

When a system is scaled vertically, the resources of the system on which it executes are adapted to the need of the application (for example, when there are more requests, more memory and processing power is added to the system).

We accomplish horizontal scalability by changing the number of instances of the same application that are in execution (for example, if there are more requests, one more instance is started to serve the increased need). If the demand decreases, we can reduce the instances numbers as well. Of course, I assume the newly spun-up application instances consume resources offered by additional hardware, sometimes even in multiple datacenters.

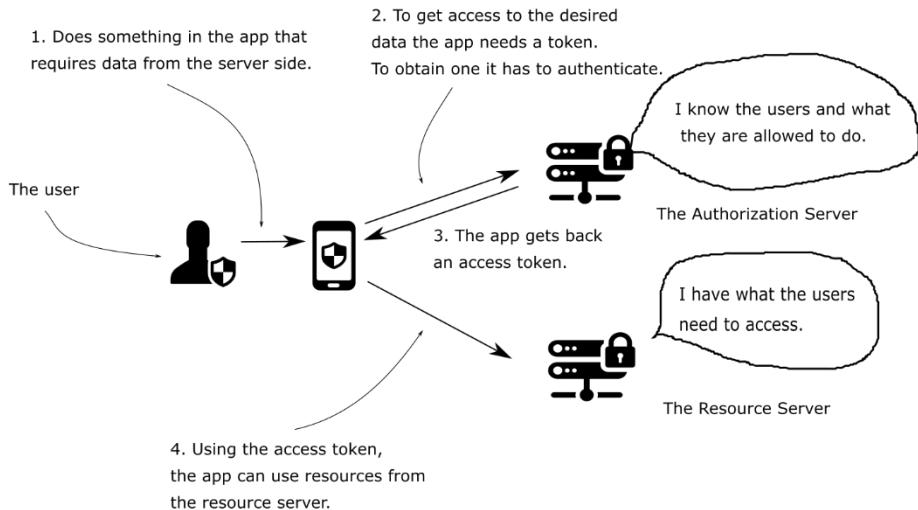
### 1.5.3 Understanding the OAuth2 flow

In this section, we discuss a high-level overview of the OAuth2 flow. I'll focus on the reason for applying OAuth2 and how it relates to what we discussed in section 1.5.2. We'll discuss this topic in detail in chapters 12 to 15.

We certainly want to find a solution to avoid resending credentials for each of the requests to the backend and store them on the client-side. The OAuth2 flow offers a better way to implement authentication and authorization in this case.

The OAuth2 framework defines two separate entities: the authorization server and the resource server. The purpose of the authorization server is to authorize the user and provide them with a token that specifies, among other things, a set of privileges that can be used. The part of the backend implementing the functionality is called the resource server. The endpoints that can be called are considered protected resources. Based on the obtained token, after accomplishing the authorization, a call on a resource will be permitted or rejected. Figure 1.13 presents a general picture of the standard OAuth2 authorization flow. Step by step, the following happens:

1. The user accesses a use case in the application (also known as the client). The application needs to call a resource in the backend.
2. To be able to call the resource, the application first has to obtain an access token, so it calls the authorization server to get the token. In the request, it sends the user's credentials or a refresh token in some cases.
3. If the credentials or the refresh token are correct, the authorization server returns a (new) access token to the client.
4. The access token is used in the header of the request to the resource server when calling the needed resources.



**Figure 1.13 The OAuth 2 authorization flow with password grant type.** To execute an action requested by the user (1), the application requires an access token from the authorization server (2). The application receives a token (3) and accesses a resource from the resource server with the access token (4).

A token is like an access card you use inside an office building. As a visitor, you first visit the front desk, where you receive an access card after identifying yourself. The access card can open some of the doors, but not necessarily all. Based on your identity, you can access precisely the doors that you're allowed to and no more. The same happens with an access token. After the authentication, the caller is provided with a token, and based on that, they can access the resources for which they have privileges.

A token has a fixed lifetime, usually being short-lived. When a token expires, the app needs to obtain a new one. If needed, the token can be disqualified by the server earlier than its expiration time. The following lists some of the advantages of this flow:

- The client doesn't have to store the user's credentials. The access token and, eventually, the refresh token are the only access details needed to be saved.
- The application doesn't expose the user's credentials that are often on the network.
- If someone intercepts a token, you can disqualify the token without needing to invalidate the user's credentials.
- A token can be used by a third entity to access resources on the user's behalf, without having to impersonate the user. Of course, an attacker would be able to steal the token in this case. But, because usually the token has a limited lifespan, the timeframe in which one can use this vulnerability is limited.

**NOTE** To make it simple and only give you an overview I've described you with the OAuth2 flow, which is called the `password grant type`. OAuth 2 defines multiple grant types and, as you'll see in chapters 12 to 15,

it's not always that the client application has the credentials. If we were using the authorization code grant, the application would have redirected the authentication in the browser directly to a login implemented by the authorization server. But more on this later in the book.

Of course, not everything is perfect even with the OAuth2 flow, and you need to adapt it to the application design. One of the questions could be: which is the best way to manage the tokens? In the examples we'll work on in chapters 12 to 15, we cover multiple possibilities that include:

- Persisting the tokens in the app's memory
- Using a database to persist the tokens
- Using cryptographic signatures with JSON Web Tokens (JWT)

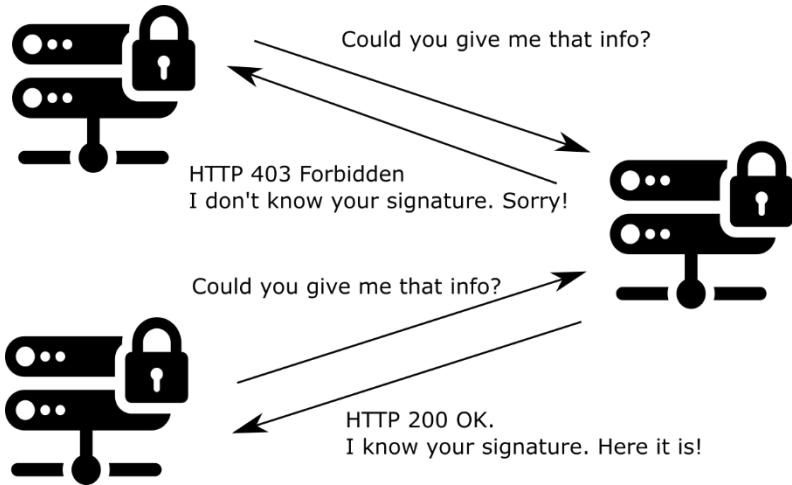
#### **1.5.4 Using API keys, cryptographic signatures, and IP validation to secure requests**

In some cases, you don't need a username and a password to authenticate and authorize a caller, but you still want to make sure that nobody altered the exchanged messages. You might need this approach when the requests are made between two backend components. Sometimes you'd like to make sure that the messages between them are validated somehow (for example, if you deploy your backend as a group of services or you use another backend external to your system). In this direction, a few practices include:

- Using static keys in request and response headers
- Signing the requests and response with cryptographic signatures
- Applying validation for IP addresses

The use of static keys is the weakest approach. In the headers of the request and the response, we use a key. The request and responses aren't accepted if the header value is incorrect. Of course, this assumes that we often exchange the value of the key in the network; if the traffic goes outside the data center, it would be easy to intercept. Someone who gets the value of the key could replay the call on the endpoint. When we use this approach, it's usually done together with IP address whitelisting.

A better approach to test the authenticity of the communication is the use of cryptographic signatures (Figure 1.14). With this approach, a key is used to sign the request and the response. You don't need to send the key on the wire, which is an advantage over static authorization values. The parties can use their key to validate the signature. The implementation can be done using two asymmetric key pairs. This approach assumes that we never exchange the private key. You can also find a simpler version in which we use a symmetric key, which requires a first-time exchange for configuration. The disadvantage is that the computation of a signature consumes more resources.



**Figure 1.14** To make a successful call to another backend, the request should have the correct signature or shared key.

If you know an address or range of addresses from where the request should come from, then together with one of the solutions mentioned previously, IP address validation can be applied. This method implies that the application will reject the requests if coming from other IP addresses than the ones that you configure to be accepted. However, most of the cases, IP validation is not done at the application level, but much earlier, at the networking layer.

## 1.6 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 to read first *Spring Framework In Action*, Sixth Ed., by Craig Walls (Manning 2020). Another great resource is also *Spring Boot In Action* by Craig Walls (Manning 2015).

In the book you're reading now, 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 OAuth2 and OpenID Connect flows and how they apply to a production-ready application
- How to implement security with Spring Security on different layers of your application
- Different configuration styles and the best practices for using those in your project
- Using Spring Security for reactive applications
- Testing your security implementations

In this book, to make the learning process smooth for each described concept, we'll work on multiple simple examples. At the end of each significant subject, we'll review the essential concepts you've learned with a more complex application. You'll find these sections in the book with the name "Hands-On."

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.

## 1.7 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 should use different practices.
- Security is a cross-cutting concern, and you should consider it from the beginning of a software project.
- Usually, the cost of an attack is higher than the investment in avoiding vulnerabilities.
- The Open Web Application Security Project (OWASP) is an excellent place to start, then always refer to that when it comes to vulnerabilities and security concerns.
- 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 that use the basic actors for authentication and authorization
- Understanding the relationship between the main actors involved in the basic authentication process
- Applying the basic contracts to understand how these actors relate to each other
- Writing your implementations for the primary responsibilities
- Overriding the Spring Boot's default configurations

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 something preconfigured, so you override only the configurations that don't match your implementation. We also call this approach "convention-over-configuration".

Before this way of developing applications existed, developers had to write dozens of lines of code and would repeat writing them 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 these configurations once at the beginning, and you'd rarely need to touch them afterward. With software architectures evolving to service-oriented, we've started to feel the pain of boilerplate code we had to write for configuring each service. If you find it amusing, you can check out chapter 3 from *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. You'll understand this way how many configurations you had to write only for a small one-web-page application:

<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 auto-configuration for the 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 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 auto-configured, as well as a brief introduction to overriding these defaults. Considering the default configurations is an excellent introduction, 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 to 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 follows:

1. Create a project with only Spring Security and web dependencies to see how it behaves if we 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 making the same configurations to understand best practices.

## 2.1 Starting with the first project

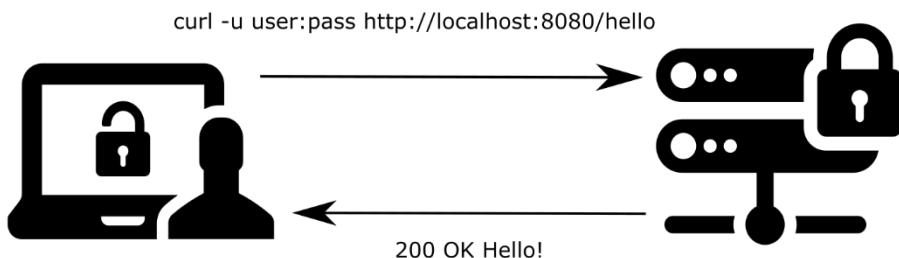
Let's create the first project so that we have something to work on for the first example. This project is a small web application, exposing a REST endpoint. You'll see how, without doing much, Spring Security secures this endpoint using HTTP Basic authentication. Just by creating the project and adding the correct dependencies, Spring Boot applies default configurations. These configurations include 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 an even more detailed discussion, I recommend Craig Walls' *Spring Boot in Action* (Manning, 2016). Chapter 2 from *Spring Boot in Action* accurately describes creating a web app with Spring Boot (<https://livebook.manning.com/book/spring-boot-in-action/chapter-2/>).

The examples in this book refer to the source code. With each example, I'll also specify the dependencies that you need to add to your `pom.xml` file. You can, and I recommend that you do so, download the projects provided with the book and the available source code. The projects will help you if you get stuck with something. You can also use them to validate your final solution.

**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 build 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 presents the basics of the Spring Security architecture for authentication and authorization.



**Figure 2.1.** The 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'll begin learning Spring Security by creating an empty project and naming it `ssia_ch2_ex1`. (You'll also find this example with this name in the example 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 have added these dependencies in 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 added to the project. But we wouldn't be able to learn much about security if we don't have at least an 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 listing 2.2, the 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 returned value from the response body of the HTTP response. The `@GetMapping` annotation maps the `/hello` path to the implemented method. 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-54860f36f7f3
```

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.

**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. The operating system you use might not have any of these tools installed you'll probably need to install them yourself.

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

The response of 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://localhost:8080/hello
```

The response of the call:

```
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 applications 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 it doesn't have the needed privileges for the call 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 returned.

### 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`.

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 makes the call, we'll 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 console (or a Git bash console):

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

The result of running this command is the Base64 encoded string:

```
dXNlcj05M2EwMWNmMC03OTRlTgtdODZlZi01NDg2MGYzNmY3ZjM=
```

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 dXNlcj05M2EwMWNmMC03OTRlTgtdODZlZi01NDg2MGYzNmY3ZjM="
http://localhost:8080/hello
```

```
The result of the call is:  
Hello!
```

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

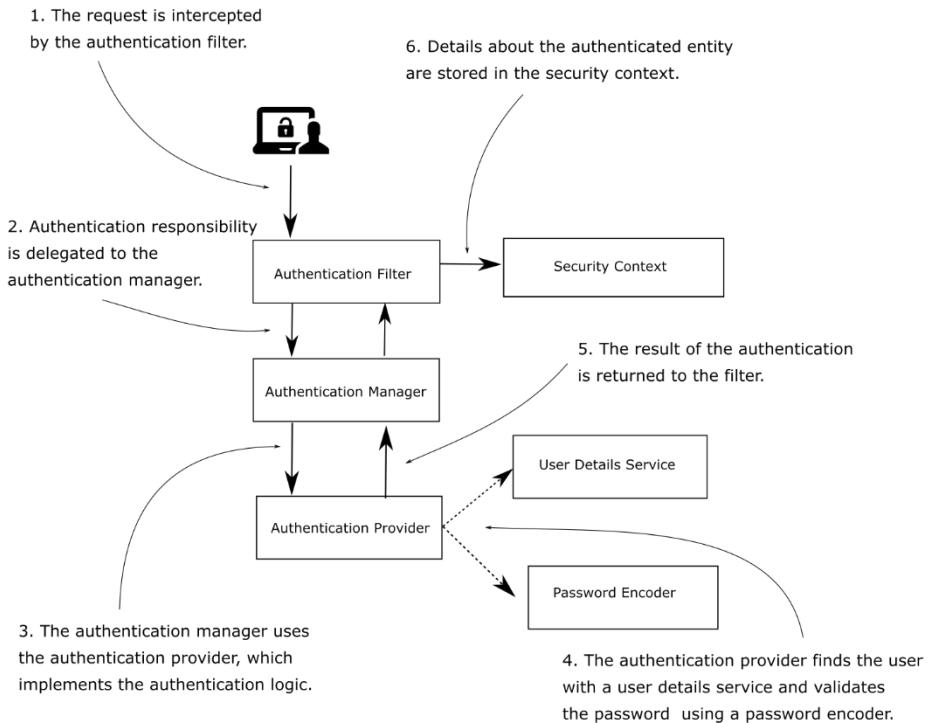
With this first example working, at least we know that Spring Security is in place. The next step is to change the configurations such that it applies to the requirements of your project. First, we'll go more in-depth with what Spring Boot configured in terms of Spring Security does, and then we'll see how we can override the configurations.

## 2.2 Which are the default configurations?

In this section, we'll 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 your applications' needs. To reach this goal, I'll start by describing how Spring Security architecture for authentication and authorization works, and then we'll apply them within projects in this chapter. It would be too much to discuss them all at once, so to minimize your learning efforts, in this chapter, I discuss the high-level picture for each component, and you'll learn details about each in the next chapters.

In section 2.1, you saw some logic executing for authentication and authorization. Mainly, 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 for you some components depending on what dependencies you're using.

In figure 2.2, you can see the big picture of the main actors in Spring Security architecture and the relationships among them. These components have a preconfigured implementation in the first project. In this chapter, I'll make you aware of what Spring Boot is configuring in your application in terms of Spring Security. We'll also discuss the relationship between 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 them. This architecture represents the backbone of implementing authentication with Spring Security. For this reason, we'll refer to it a lot throughout the book when discussing different implementations for authentication and authorization.

In figure 2.2, you can see

- *The Authentication Filter*—Delegates the authentication request to the authentication manager and, based on the response, configures the security context.
- *The Authentication Manager*—Uses the authentication provider to process the authentication.
- *The Authentication Provider*—Implements the authentication logic.
- *The User Details Service*—Implements user management responsibility. The authentication provider uses it in the authentication logic.
- *The Password Encoder*—Implements password management. The authentication provider uses it in the authentication logic.
- *The Security Context*—Keeps the authentication data after the authentication process.

In the following paragraphs, I discuss the following auto-configured beans:

- `UserDetailsService`

- `PasswordEncoder`

You can see these in figure 2.2 as well. The authentication provider uses the beans to find the 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` contract from Spring Security manages the details about the users. Up until now, we've 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 universally unique identifier (UUID) default password, randomly generated when the Spring context is loaded. At this time, the application also writes the password to the console where you can see it. Thus, you were able to take it and use it in the example we've just worked on in this chapter.

These default implementations serve only as a proof of concept and allow us to see that the dependency is in place. This default 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'll avoid it with a production-ready application.

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

- Encodes a password
- 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 them. We'll discuss more details on 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. This authentication method Spring chooses by default is the HTTP Basic access authentication and is 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, not an encryption or hashing method. While in transit, if intercepted, the credentials can be seen. Generally, we don't use HTTP Basic authentication without at least HTTPS for confidentiality. You can read the detailed definition of the HTTP Basic method 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, your applications will, however, communicate only over HTTPS. For the examples we discuss, the configurations related to Spring Security aren't different if we use HTTP or HTTPS. So you can focus on the examples related to Spring Security, I won't configure HTTPS for the endpoints in the examples of this book. But you can enable HTTPS for any of the endpoints if you wish in the way I present in this sidebar.

There are several patterns to configure HTTPS in a system. In some cases, the developers configure HTTPS at the application level; in others, they could 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 any of these configuration scenarios, you need a certificate signed by a certification authority. 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 buy such a certificate and have to renew it at a specific period of time. If you only need to configure HTTPS to test your application, you could generate a self-signed certificate using a tool like `openssl`. 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 -days 365
```

After running the command in a terminal, you'll be asked for a password and details about your "certificate authority". Because it is only a self-signed certificate for a test, you can input any data here, just make sure to remember the password. The command outputs two files: `key.pem`, which is the private key `cert.pem`, which is a public certificate. We'll use these further to generate our self-signed certificate to use for enabling HTTPS. In most of the cases, the certificate is a Public Key Cryptography Standard #12 (PKCS12). Less frequently, we use a Java Key Store (JKS) format. Let's continue our example with a PKCS12 format. For an excellent discussion on cryptography, I also recommend "Real-World Cryptography" by David Wong (Manning, 2020).

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out certificate.p12 -name "certificate"
```

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 before them `winppty` as shown in the next code snippet:

```
winppty openssl req -newkey rsa:2048 -x509 -keyout key.pem -out cert.pem -days 365
```

```
winppty openssl pkcs12 -export -in cert.pem -inkey key.pem -out certificate.p12 -name "certificate"
```

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
```

#A The value of the password is the one you have specified when running the second command to generate the pkcs12 certificate file.

Mind, 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.

Add a test endpoint to your 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 the tool 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 https://localhost:8080/hello
```

The response of the call is:

```
Hello
```

However, 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 very helpful in protecting communication, HTTPS is 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 the default configurations

Now that you know the defaults of your first project, it's time to see how you can replace them. You need to understand the options you have for overriding the default components because this is the way you'll plug in your custom implementations and apply security as it fits into 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 the same configuration. This flexibility can create confusion. I've frequently seen a mix of different styles of configuring different parts of Spring Security in the same application. This flexibility is something for which you must exercise caution, mainly because you'll find a lot of examples applying different styles. You need to learn how to choose from them; therefore, this section is also about knowing what are your options for this.

In some cases, developers choose to use beans in the Spring context for the configuration. In other examples, you'll see them overriding 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 will help you better understand how you should configure the application-level security in a project.

You'll learn in this section how to configure a `UserDetailsService` and `PasswordEncoder`. These two components both take part in authentication, and applications customize these depending on its 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'll use in this chapter are all provided by Spring Security.

### 2.3.1 Overriding the `UserDetailsService` component

The first component we talked about in this chapter was `UserDetailsService`. As you saw, the application uses this component in the process of authentication. You can define your bean of type `UserDetailsService`. As you'll see in more detail in chapter 3, you have the option to create your implementation or 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 in this kind of object into the 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'll 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 will allow us to have our own managed credentials for authentication. For this example, we aren't implementing our class, but we're using an implementation provided by Spring Security.

In this example, we'll use the `InMemoryUserDetailsManager` implementation. Even if this implementation is a bit more than just a `UserDetailsService`, for now, we'll 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'll 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 also find the example applied in project `ssia_ch2_ex2`.

**NOTE** The examples in this book are designed for Java 11, which is the latest long-term supported Java version. For this reason, I expect more and more applications in production to use Java 11. So it makes a lot of sense to keep it this version also for the examples in this book. You'll sometimes see that I use `var` in the code. The reserved type name `var` was introduced in Java 10, and you can only use it for local declarations. In this book, I use it to make the syntax shorter as well as sometimes 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.

#### Listing 2.3 The configuration class for the `UserDetailsService` bean

```

@Configuration #A
public class ProjectConfig {

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

        return userDetailsService;
    }
}

```

#A The `@Configuration` annotation marks the class as a configuration class.

#B The `@Bean` annotation instructs Spring to add the returned value as a bean in the Spring context.

#C The `var` word makes the syntax shorter and hides some details.

We annotate the class with `@Configuration`. The purpose of the `@Bean` annotation is to instruct 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 auto-generated password in the console. The application now uses the instance of type `UserDetailsService` you've added to the context instead of the default auto-configured one. But, at the same time, you won't be able to access the endpoint any more for two reasons:

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

In figure 2.2, you can see that the 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 to the one stored and managed by `UserDetailsService`

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

When building the instance, we have to provide the username, the password, and at least one authority. The `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
    public UserDetailsService userDetailsService() {
        var userDetailsService =
            new InMemoryUserDetailsManager();

```

```

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

userDetailsService.createUser(user);      #B

return userDetailsService;
}
}

```

#A Builds the user with a given username, password, and authorities list

#B Adds the user to be managed by UserDetailsService

**NOTE** You'll find the class `User` in the `org.springframework.security.core.userdetails` package. It's the builder implementation we used 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, I had to provide a value for the username, one for the password, and at least one for the 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` was 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 will get back an HTTP 401 Unauthorized and an empty response body.

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

The result of the call (in the app's console):

```

java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
at
    org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdPas-
swordEncoder.matches(DelegatingPasswordEncoder.java:244) ~[spring-security-core-
5.1.6.RELEASE.jar:5.1.6.RELEASE]

```

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'll use an existing implementation of `PasswordEncoder`:

```

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

```

**NOTE** The `NoOpPasswordEncoder` instance treats the passwords as plain text. The `NoOpPasswordEncoder` doesn't encrypt or hash the password. For matching, the

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 application. 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 strikeout through it.

You can see the full code of the configuration class in listing 2.5.

#### **Listing 2.5 The full definition of the configuration class**

```
@Configuration
public class ProjectConfig {

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

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

        userDetailsService.createUser(user);

        return userDetailsService;
    }

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

#A A new method annotated with @Bean to add a PasswordEncoder to the context
```

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 the book. However, to help you focus on the presented topics for each chapter, I have separated the discussion about testing Spring Security integrations in chapter 20.

### **2.3.2 Overriding the endpoint authorization configuration**

With the 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 the authorization configuration in chapters 7, 8, and 9. But before diving into details, you must understand the big picture. And the best way to achieve this is within our first

example. With the default configuration, all the endpoints assume you have a valid user managed by the application to be able to call them. Also, by default, your app uses HTTP Basic authentication as the authorization method. You can easily override these configurations.

As you'll learn in the next chapters, HTTP Basic authentication doesn't always fit 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 are, we might need to choose different authorization rules. To make such changes, we start by extending the `WebSecurityConfigurerAdapter` class. Extending this class allows us to override the `configure(HttpSecurity http)` method, as presented in this listing. For this example, I'll continue writing the code in the project `ssia_ch2_ex2`.

#### **Listing 2.6 Extending WebSecurityConfigurerAdapter**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // ...
    }
}
```

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

#### **Listing 2.7 Using the HttpSecurity parameter to alter the configuration**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()      #A
            .anyRequest().authenticated();   #A
    }
}
```

#A All the requests require authentication.

The code in listing 2.7 configures the endpoints 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 can see how to do this in the next listing.

#### **Listing 2.8 Using permitAll() to change the authorization configuration**

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()      #A
            .anyRequest().permitAll();   #A
    }
}

```

#A None of the requests need to be authenticated.

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
```

The response body of the call is:

```
Hello!
```

The purpose of this example is to give you a feeling of how to override the default configurations. We'll get into the details about authorization in chapters 7 and 8.

### 2.3.3 Setting the configuration 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 the `UserDetailsService` and `PasswordEncoder`. It's essential to know the options you have so that you'll recognize them in examples that you find in this book or other sources like blogs or articles. It's also important that you understand how and when to use these in your application. This is why I'd like to make you aware of them. In further chapters, you'll see different examples that extend the information in this section.

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

In the configuration class, instead of defining these two objects as beans, we'll set them up through the `configure(AuthenticationManagerBuilder auth)` method. We override this method from the `WebSecurityConfigurerAdapter` class and use its parameter of type `AuthenticationManagerBuilder` to set both the `UserDetailsService` and the `PasswordEncoder` as shown in this listing. You find this example applied in the project `ssia_ch2_ex3`.

**Listing 2.9 Setting `UserDetailsService` and `PasswordEncoder` in the `configure()` method**

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {
    // Omitted code

    @Override
    protected void configure(
        AuthenticationManagerBuilder auth)
        throws Exception {
        var userDetailsService =
            new InMemoryUserDetailsManager();      #A

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

        userDetailsService.createUser(user);      #C

        auth.userDetailsService(userDetailsService)      #D
            .passwordEncoder(NoOpPasswordEncoder.getInstance());
    }
}

```

#A We declare a UserDetailsService to store the users in-memory.

#B We define a user with all its details.

#C We add the user to be managed by our UserDetailsService defined above.

#D The UserDetailsService and PasswordEncoder are now set up within the configure() method.

In listing 2.9, you can observe that we declared the `UserDetailsService` in the same way as in listing 2.5. The difference is that now, this is done locally inside the second overridden method. We called the `userDetailsService()` method from the `AuthenticationManagerBuilder` to register the `UserDetailsService` instance. Furthermore, we called the `passwordEncoder()` method to register the `PasswordEncoder`.

**NOTE** The `WebSecurityConfigurerAdapter` class contains three different overloaded `configure()` methods. In listing 2.9, we overrode a different one than in listing 2.8. In the next chapters, we'll discuss all three in more detail.

In listing 2.10, you can see the full contents of the configuration class.

#### Listing 2.10 Full definition of the configuration class

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure[CA]
        (AuthenticationManagerBuilder auth) throws Exception {
        var userDetailsService =
            new InMemoryUserDetailsManager();      #A

        var user = User.withUsername("john")      #B

```

```

        .password("12345")      #B
        .authorities("read")    #B
        .build();      #B

        userDetailsService.createUser(user);      #C

        auth.userDetailsService(userDetailsService)  #D
            .passwordEncoder(      #D
                NoOpPasswordEncoder.getInstance());      #D
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()      #E
            .anyRequest().authenticated();      #E
    }
}

#A Creates an instance of InMemoryUserDetailsManager()
#B Creates a new user
#C Adds the user to be managed by our UserDetailsService
#D Configures UserDetailsService and PasswordEncoder
#E Specifies that all the requests require authentication

```

Any of these configuration options are correct. The first option, where we added 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. However, I recommend you to avoid mixing the configurations because it might create confusion. For example, the code in listing 2.11 could make you wonder about where the link between the `UserDetailsService` and the `PasswordEncoder` is.

### Listing 2.11 Mixing configuration styles

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

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

    @Override
    protected void configure[CA]
        (AuthenticationManagerBuilder auth) throws Exception {
        var userDetailsService = new InMemoryUserDetailsManager();

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

        userDetailsService.createUser(user);

        auth.userDetailsService(userDetailsService);      #B
    }
}

```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic();
    http.authorizeRequests()
        .anyRequest().authenticated();
}
}

```

#A The PasswordEncoder is designed as a bean.

#B But the UserService is configured directly in the configure() method.

Functionally, the code in listing 2.11 works just fine, but I recommend you avoid mixing the two approaches to keep the code clean and easier to understand. Using the AuthenticationManagerBuilder, you can configure the users for authentication directly. It creates the UserService for you in this case. The syntax, however, becomes even more complex and could be considered difficult to read. I've seen this choice more than once, even with production-ready systems.

It could be that this example looks fine because we use an in-memory approach to configure users. But in a production application, this isn't the case. There, you'll probably store your users in a database or access them from another system. As in this case, the configuration could become pretty long and ugly. Listing 2.12 shows the way that you can write the configuration for in-memory users. You find this example applied in project ssia\_ch2\_ex4.

#### **Listing 2.12 Configuring user management in-memory**

```

@Override
protected void configure[CA]
(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("john")
        .password("12345")
        .authorities("read")
        .and()
        .passwordEncoder(NoOpPasswordEncoder.getInstance());
}

```

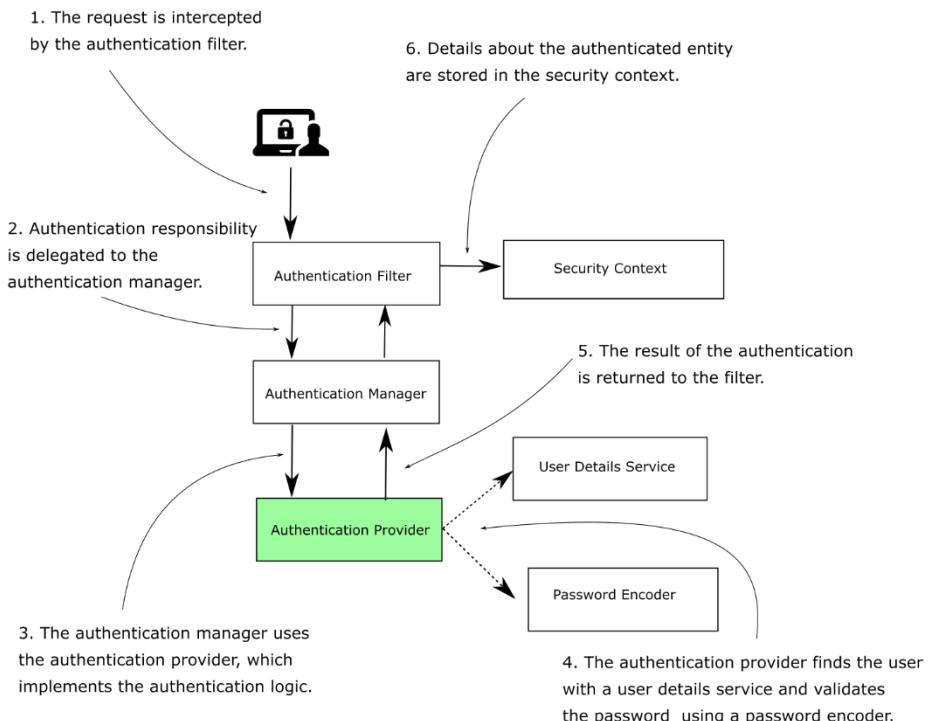
Generally, I don't recommend this approach as I find it better to separate and write the responsibilities as decoupled as possible in an application.

#### **2.3.4 Overriding the AuthenticationProvider implementation**

As you already observe, the 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 UserService and PasswordEncoder in the Spring Security architecture. And you've seen a few ways to configure those. It's time to learn that the component delegating to them, the AuthenticationProvider, can also be customized.

The AuthenticationProvider, as you find figure 2.3, implements the authentication logic and delegates to the UserService and PasswordEncoder for user and password management. So we could say that with this section, we go one step deeper in the

authentication and authorization architecture to learn how to implement a custom authentication logic with 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.**

Because this is the first example, I'll only show you the brief picture so that you understand better the relationship between the components in the architecture. But we'll detail more in chapters 3, 4, and 5. In these chapters, you'll find it implemented as well in a more significant exercise - the first "Hands-On" section of the book, which is chapter 6.

I recommend you respect the responsibilities as they were designed in the Spring Security architecture. This architecture is loosely-coupled with fine-grained responsibilities. The design is one of the things that makes Spring Security flexible and easy to integrate into 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 could 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, listing 2.13 shows

how to create a custom authentication provider. You find this example applied in project `ssia_ch2_ex5`.

### **Listing 2.13 Implementing the AuthenticationProvider interface**

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

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

        // 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, you can add an implementation like that in listing 2.14. I will explain the usage of the `supports()` method in detail in chapter 5. For the moment, I recommend you take its implementation for granted. It's not essential for the current example.

### **Listing 2.14 Implementing the authentication logic**

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

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

    if ("john".equals(username) &&
        "12345".equals(password)) {      #B

        return new UsernamePasswordAuthenticationToken[CA]
            (username, password, Arrays.asList());
    } else {
        throw new AuthenticationCredentialsNotFoundException[CA]
            ("Error in authentication!");
    }
}
```

#A The `getName()` method is inherited by `Authentication` from the `Principal` interface.

#B This condition would generally call `UserDetailsService` and `PasswordEncoder` to test the username and password.

As you see, here the condition of the `if-else` clause is replacing the responsibilities of the `UserDetailsService` and `PasswordEncoder`. You are, though, not forced to use the two beans. But if you work with users and passwords for authentication, I strongly suggest you separate

the logic of users and passwords management. Apply it as the Spring Security architecture designed it, even when you are overriding 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 a custom authentication logic. The full `AuthenticationProvider` implementation looks like the one in listing 2.15.

#### **Listing 2.15 The full implementation of the authentication provider**

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

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

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

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

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

In the configuration class, you can now register `AuthenticationProvider` in the `configure(AuthenticationManagerBuilder auth)` method.

#### **Listing 2.16 Registering the new implementation of `AuthenticationProvider`**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomAuthenticationProvider authenticationProvider;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(authenticationProvider);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
```

```

        http.authorizeRequests().anyRequest().authenticated();
    }
}

```

You can now call the endpoint, which will be accessible by the only user recognized as that defined by the authentication logic: john with the password 12345.

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

The response body will be:

```
Hello
```

In chapter 5, you'll learn more details about the `AuthenticationProvider` and how to override its behavior in the authentication process. Still, in chapter 5, we'll discuss the `Authentication` interface and its implementations like the `UserPasswordAuthenticationToken`.

### 2.3.5 Using multiple configuration classes in your project

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. This separation is needed because the configuration starts to become more complex. In a production-ready application, you'll most probably have more declarations than in our first examples. You'll 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 the user management configuration from the authorization configuration. In the next example, we'll do that by defining two configuration classes: `UserManagementConfig` and `WebAuthorizationConfig`. You find this example applied in project `ssia_ch2_ex6`.

#### **Listing 2.17 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 the user management: the `UserDetailsService` and the `PasswordEncoder`. Also, in this case, we choose to configure the two objects as beans because this class can't extend `WebSecurityConfigurerAdapter`.

#### **Listing 2.18 Defining the configuration class for authorization management**

```
@Configuration
public class WebAuthorizationConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

Here the `WebAuthorizationConfig` class needs to extend `WebSecurityConfigurerAdapter` and override the `configure(HttpSecurity http)` method.

**NOTE** You can't have both classes extending `WebSecurityConfigurerAdapter` in this case. If you do so, the dependency injection will fail. You might solve the dependency injection by setting the priority for injection using the `@Order` annotation. But functionally, this wouldn't work as the configurations will exclude each other instead of merge.

## **2.4 Summary**

- Spring Boot makes some default configurations when you add Spring Security to the dependencies of the application.
- In this chapter, you implement the basic components—`UserDetailsService`, `PasswordEncoder`, and `AuthenticationProvider`—for authentication and authorization.
- 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 very 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 a production-ready application.
- 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.

# Part 2.

## *Implementation*

In part 1, we discussed the importance of security and how to create the Spring Boot project using Spring Security as a dependency. This way, we've gone through the essential components for authentication, and we have a point to start from.

Part 2 makes up the bulk of this book. In this part, we dive into using Spring Security in application development. We detail each of the Spring Security components, and discuss different approaches you need to know when developing any real-world app. You'll find all the various subjects you need to learn on developing security features in apps with Spring Security. I'll drive you through a path of knowledge with multiple subjects, from the basics to using OAuth 2 and from securing apps using imperative programming to applying security in reactive applications. And I'll make sure what we discuss is well spiced with lessons I've learned in my experience with using Spring Security.

In chapters 3 and 4, you'll learn to customize apps' user management and the way they deal with the passwords. In many cases, applications rely on credentials to authenticate users. For this reason, discussing here the management of users' credentials opens the gate to further discussing authentication and authorization. We'll then continue with customizing the authentication logic in chapter 5. Once we finish talking about authentication, in chapters 6 to 11, we discuss the components related to authorization. Throughout all these chapters, you'll learn how to deal with basic elements like user details managers, password encoders, authentication providers, and filters. Knowing how to apply these components and properly understanding them enables you to solve the security requirements you'll face in real-world scenarios you'll implement.

Nowadays, many apps, and especially systems deployed in the cloud, implement the authentication and authorization over the OAuth 2 specification. In chapters 12 to 15, you'll learn how to implement the authentication and authorization in your apps using Spring Security.

In chapters 16 and 17, we'll discuss applying authorization rules at the method level. This approach enables you to use also in non-web applications what you've learned already about

Spring Security. It also gives you incredible more flexibility to apply restrictions in web apps as well.

In chapter 19, you'll learn to apply Spring Security to secure reactive apps. And, because there's no development process without testing, in chapter 20, you'll learn how to write integrations tests for your security implementations.

Throughout the section, you'll find chapters where we'll use a different way to address the subjects. We'll work during each of these chapters on a requirement that helps you refresh what you've learned, understand how more of the subjects we discussed fit together but also learn to apply new things. I've called these, the "Hands-On" chapters.

# 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 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 could and what you should not mix. Then for each recipe, you only remember some tricks".

This analogy is very 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 the "implementation". When using a framework, the programmer knows the framework's abstractions and uses them to integrate with it.

This chapter is about understanding in detail one of the fundamental roles you've encountered in the first example we've worked on in chapter 2: the `UserDetailsService`.

Along with the `UserDetailsService`, we'll discuss:

- The `UserDetails`, which describes the user for Spring Security.
- The `GrantedAuthority`, which allows us to define actions that the user can execute.
- The `UserDetailsManager`, which extends the `UserDetailsService` contract. Above the inherited behavior, it also describes actions like creating a user, modifying the user's

password, or deleting it.

From chapter 2, you already have an idea of the roles of the `UserDetailsService` and `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 like:

- which are the implementations provided by Spring Security and how to use them
- how to define a custom implementation for these contracts and when would you do so
- different ways to implement the interfaces that you'll find in real-world applications
- best practices for using them

The plan is to start with how Spring Security understands the user definition. For this, we discuss the `UserDetails` and `GrantedAuthority` contracts.

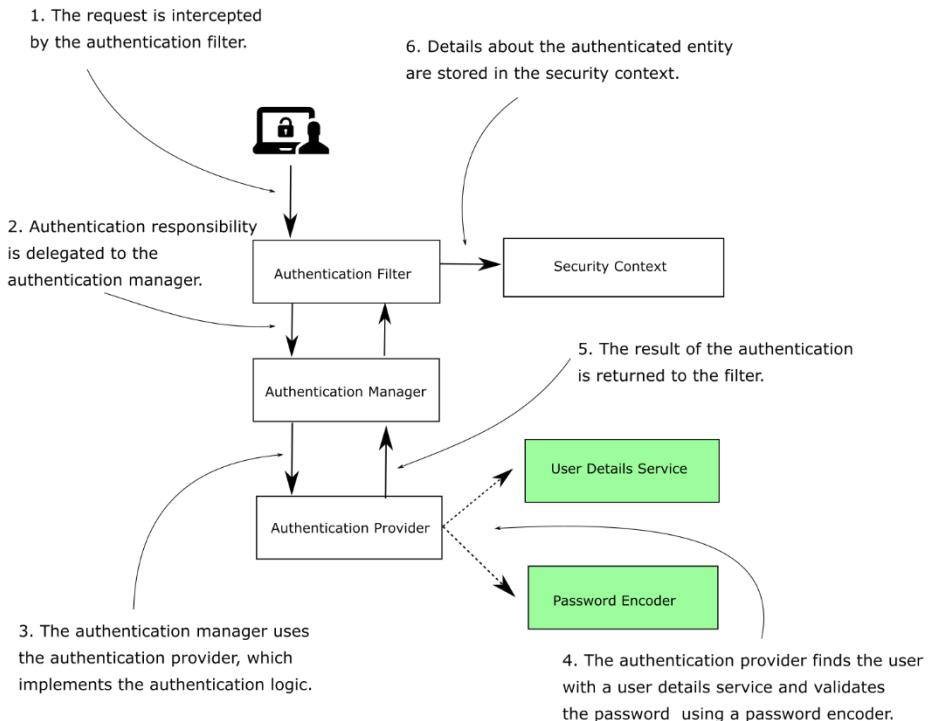
Further, we detail the `UserDetailsService` and how `UserDetailsManager` extends this contract. You'll apply provided implementations for these interfaces like the `InMemoryUserDetailsService`, `JdbcUserDetailsService`, and `LdapUserDetailsService`. For when these provided implementations aren't a good 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. These defaults 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 very surface of them so that you have an idea of what we'll be doing. In this chapter and chapters 4 and 5, we discuss these interfaces in more detail, together with different provided implementations for them and ways you might find them implemented in real 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 we'll 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. Because if you know this architecture, you're like a chef knowing their ingredients to be able to cook any recipe.

With a different shade, I represented the components that we'll start with: the `UserDetailsService` and the `PasswordEncoder`. These two components focus on the part of the flow, which I'll often refer to as "the user management part". The `UserDetailsService` and the `PasswordEncoder` are the components dealing directly with the user details and their credentials, as you'll find in this chapter. We'll discuss the `PasswordEncoder` in detail in chapter 4. Further, in this book, I'll detail the other components you could customize in the authentication flow: the `AuthenticationProvider`, the `SecurityContext`, and the filters.

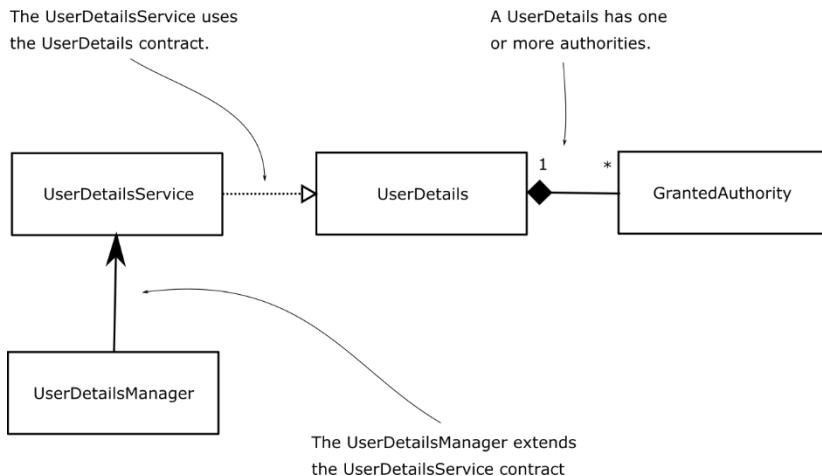


**Figure 3.1 The authentication flow:** The `AuthenticationFilter` intercepts the request and delegates the authentication responsibility to the `AuthenticationManager`. The `AuthenticationManager` uses further an authentication provider, which implements the authentication logic. To check the username and the password, the `AuthenticationProvider` uses a `UserDetailsService` and a `PasswordEncoder`.

As part of user management, we'll use the `UserDetailsService` and `UserDetailsManager` interfaces. The `UserDetailsService` is only responsible for retrieving the user by its 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 required functionality in most of the applications. The separation between the two contracts is an excellent example of the "interface segregation" principle. Separating the interfaces allows for better flexibility as 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, the `UserDetailsService` and the `UserDetailsManager` components need a way to represent them.

The `UserDetails` is the contract offered by Spring Security, which you have to implement to describe a user in the way the framework understands it. As you'll learn further in this chapter, in Spring Security, a user has a set of privileges which are actions the user is allowed

to do. We'll work a lot with these privileges in chapters 7 and 8 when discussing authorization. Spring Security represents the actions that a user can do with the `GrantedAuthority` interface. We often call them shortly "authorities". A user has one or more authorities. In figure 3.2, you find a representation of the relation 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). The `UserDetailsManager` contract extends `UserDetailsService` to add operations with the user, such as create, delete or change of its password.**

Understanding the links between these objects in the Spring Security architecture and ways to implement them will give you a wide range of options to choose from when working on applications. Each of these options could be the right puzzle piece in the app that you are working on, and you will have 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 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 certain functionality is or isn't allowed. To work with users, first, you have to understand how to define the prototype of the user in your application. In this section, I will 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 will have to implement this interface, and this way, the framework will understand it.

### 3.2.1 Demystifying the definition of the UserDetails contract

In this section, you'll learn how to implement the `UserDetails` interface to describe the users in your application. We discuss the methods declared by the `UserDetails` contract to understand how and why each of them implemented. Let's start first by looking at the interface, as presented in the listing 3.1.

#### **Listing 3.1 The UserDetails interface**

```
public interface UserDetails extends Serializable {
    String getUsername();      #A
    String getPassword();
    Collection<? extends GrantedAuthority> getAuthorities();      #B
    boolean isAccountNonExpired();      #C
    boolean isAccountNonLocked();
    boolean isCredentialsNonExpired();
    boolean isEnabled();
}
```

#A These methods return the credentials of the user

#B This method returns the actions that the application allows the user to do, as a collection of `GrantedAuthority` instances

#C These four methods refer to making the account enabled or disabled based on different reasons

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

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 do such an action. An authority represents such a privilege a user has. The `getAuthorities()` method should return the group of authorities granted for a user.

**NOTE** As you'll learn in chapter 7, Spring Security uses authorities to refer either to fine-grained privileges or to roles (which are groups of privileges). To make your reading more comfortable, in this book, I'll refer to the fine-grained privileges as "authorities".

Furthermore, as seen in the `UserDetails` contract, a user could:

- have the account expired
- have the account locked
- have the credentials expired
- be disabled

If you choose to implement these user restrictions in your application's logic, you will override the following methods: `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()`, `isEnabled()` such that those needed to be enabled return true.

Not all the applications have accounts that expire or get locked in certain conditions. If you do not need to implement these functionalities in your application, you can simply make all these four methods return true.

**NOTE** The names of the last four methods in the `UserDetails` interface may sound strange. One could argue that they 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. They have been named such that they all return false for the case in which the authorization should fail and true otherwise. This approach is right because the human mind tends to associate the word “false” with negativity, while people usually associate the word “true” to positive scenarios.

### 3.2.2 Detailing on the `GrantedAuthority` contract

As you have observed in section 3.2.1, in the definition of the `UserDetails` interface, the actions granted for a user are called authorities. In chapters 7 and 8, we’ll write authorization configurations based on these authorities of the user. So it’s essential to know how to define them. The authorities represent what the user can do in your application. Without authorities, each user would be equal to another. While there are simple applications in which the users are equal one to the other, 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 can differentiate between them. Of course, it depends 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 going further to discuss implementing `UserDetails`, let’s understand the `GrantedAuthority` interface. As presented in chapter 2, we use this interface in the definition of the user details. It represents a privilege granted to the user. A user could have none to any number of authorities. Usually, they have at least one.

```
public interface GrantedAuthority extends Serializable {
    String getAuthority();
}
```

To create an authority, you only need to find a name for it. Find a name for that privilege so that you refer to it later when writing the authorization rules. For example, a user can “read” the records managed by the application or “delete” them. Based on the names you give to these actions, you will write the authorization rules. In chapters 7 and 8, you’ll learn about writing the authorization rules based on the user’s authorities. You’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 will 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 find two examples of implementing a `GrantedAuthority`. First, we make use of a lambda expression, and the second uses the `SimpleGrantedAuthority` class.

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

**NOTE** It is good practice to verify that the interface is marked as functional with the `@FunctionalInterface` annotation before implementing it with lambda expressions. The reason for this practice is that if the interface is not marked as functional, it could mean that its developers reserve the right to add more abstract methods to it in future versions. The `GrantedAuthority` interface is not marked as functional. In this book, we will implement it with lambda expressions often to make the code snippets shorter and easier to understand. However, I recommend you avoid doing so in your applications until it is explicitly marked as functional.

### 3.2.3 Writing the minimal implementation of the `UserDetails`

In this section, you'll write your first implementation of the `UserDetails` contract. We'll start with a basic implementation in which each method returns a static value. Once you've done this, we change it to a version that is more probable to find in a practical scenario and which allows you to have multiple different instances of users. Now that you know how to implement the `UserDetails` and the `GrantedAuthority` interfaces, we can already write the simplest definition of a user for an application. With a class that I would name a `DummyUser`, let's implement a minimal description of a user. I use this class mainly as a demonstration of implementing the methods for the contract. Instances of this class always refer to only one user: "bill" which has the password "12345" and an authority named "READ".

#### Listing 3.2 The `DummyUser` class is a straightforward implementation of the `UserDetails` contract

```
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 will have to implement all its methods. You find here the implementation of the `getUsername()` and `getPassword()`. In this example, these methods only return a fixed value for each of the properties. Next, we will add a definition for the list of authorities. Listing 3.3 shows the

implementation of the `getAuthorities()` method. This method returns here 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 would have to add an implementation for the last four methods. For the `DummyUser` class, they will always return true, which means that the user is forever active and usable. You find the example in the listing 3.4.

#### **Listing 3.4 The implementation for the last four methods from the `UserDetails` interface**

```
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 means that all the instances of the class represent the same user. The previous minimal implementation is a good start with understanding the contract, but not something you would do in a real application. You should create a class that you could use to create instances that can represent different users. In this case, probably your definition would

at least have the username, and the password as attributes in the class, as presented by listing 3.5.

#### **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` that brings something out of the ordinary. 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 provided by Spring Security. The `User` class from the `org.springframework.security.core.userdetails` package is a simple way to build instances of the `UserDetails` type. Using the class, you create immutable instances of `UserDetails`. You need to provide at least a username, and a password, and the username shouldn't be the empty string.

Listing 3.6 shows a demo for using this builder. Building the user as presented in listing 3.6, you wouldn't even need to have your implementation of this 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 code snippet 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 from the username given as a string. Afterward, we demonstrate how to create a builder starting from an already existing instance of `UserDetails`.

#### **Listing 3.7 Creating the User.UserBuilder instance**

```
User.UserBuilder builder1 = 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();
```

#A You can start building a user with their username.

#B Here, the password encoder is only a function that does an encoding

#C At the end of the build pipeline, you will call the `build()` method

#D You can also start building a user from an existing `UserDetails` instance.

With any of the builders defined in listing 3.7, you observe you can use the builder to obtain a user represented with the `UserDetails` contract. At the end of the build pipeline, you will 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** Here, the password encoder is not the same as the bean we have shortly discussed in chapter 2. The name might be confusing, but here we only have a `Function<String, String>`. This function's only responsibility is to transform a password in a given sort of encoding. In the next section of this chapter, we discuss in detail the `PasswordEncoder` contract from Spring Security that you used in chapter 2.

#### **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, there'll be multiple responsibilities to which the user relates. For example, if you store the users in a database, then, in the application, you would need a class to represent the persistence entity as well. Or, if you retrieve the 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 a SQL database in which we store the users. To make the example shorter, each user will have only one authority. The entity class which maps the table would then look in the listing 3.8.

#### **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 like in listing 3.9? I don't know what you think about this code, but from my point of view, it is a mess. For sure, 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(() -> this.authority);
    }

    // Omitted code
}

```

Now, the class contains JPA annotations, getters, and setters, out of which both `getUsername()` and `getPassword()` also override the methods in the `UserDetails` contract. It has a `getAuthority()` method, which returns a `String`, but also a `getAuthorities()` method that returns a `Collection`. The `getAuthority()` method is just a getter in the class, while the `getAuthorities()` implements the method in the `UserDetails` interface. And things

would get even more complicated when adding relationships to other entities. Again, this code isn't friendly at all!

How could 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 them into the same class. Let's try to separate them below by defining a separate class called `SecurityUser`, which will decorate the `User` class. The `SecurityUser` class implements the `UserDetails` contract and will be used to plug in our user into the Spring Security architecture. The `User` class only remains with its JPA entity responsibility.

#### **Listing 3.10 The User class only remains with the responsibility of being a JPA entity**

```
@Entity
public class User {

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

    // Omitted getters and setters
}
```

As presented by listing 3.10, the `User` class remains only with its entity responsibility and becomes more readable. If you read this code, you can now focus exclusively on details related to persistence. These are anyway not important from the Spring Security perspective.

In listing 3.11, we have the implementation of the `SecurityUser` class.

#### **Listing 3.11 SecurityUser class implement the UserDetails contract and wraps the User entity**

```
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. Now, you have to provide the user through the constructor. This class decorates the other one and adds the needed code related to the Spring Security contract without mixing the code into the JPA entity and adding in the same place elements of different purposes.

**NOTE** You can find different other approaches to separate the two responsibilities. I don't want to say that the approach I presented in this section is the best or the only one. Usually, the way you choose to implement the class design depends 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 the users

You implemented the `UserDetails` contract to describe users such that Spring Security understands them. But how does Spring Security manage the users? Where are they taken from when comparing the credentials and how to add new users or change them? In chapter 2, you learned that the framework defines a specific component to which the authentication process delegates the user management. And this was the `UserDetailsService` instance. We even defined a `UserDetailsService` to override the default implementation initially provided by Spring Boot.

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

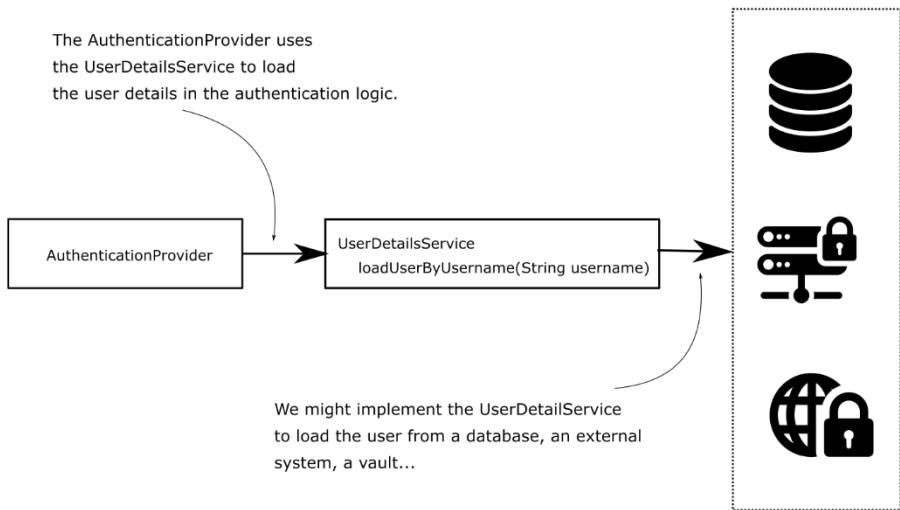
#### 3.3.1 Understanding the `UserDetailsService` contract

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 the `UserDetailsService` and how to work with implementations of this component. The `UserDetailsService` interface contains only one method.

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
```

```
    throws UsernameNotFoundException;
}
```

This method, `loadUserByUsername(String username)`, is called by the authentication implementation 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 should throw a `UsernameNotFoundException`.



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

**NOTE** The `UsernameNotFoundException` is a `RuntimeException`. The `throws` clause in the context of the `UserDetailsService` interface is only for documentation purposes. `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 will work on a practical example to demonstrate the implementation of the `UserDetailsService`. Your application manages details about credentials and other user details. It could be that they are stored in the database or handled by another system that you access through a web service or 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 its username.

In the next example, we will 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 find this example as project `ssia-ch3-ex1`.

In the package named `model`, we define the `UserDetails` as presented by listing 3.12.

### **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;
    }
}

#A The User class is immutable. You give the values for the three attributes when you build the instance, and these
values cannot be changed afterward.
#B To make the example simple, a user has only one authority.
#C The method returns a list containing only the GrantedAuthority object with the name provided when you build the
instance.
#D The account does not expire or get locked.

In the package named services, we will create a class called
InMemoryUserDetailsService. We implement this class, as presented by listing 3.13.

```

### **Listing 3.13 The implementation of the UserDetailsService interface**

```

public class InMemoryUserDetailsService implements UserDetailsService {

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

```

#A The list of users managed in-memory by the UserDetailsService.  
#B From the list of users we filter the one that has the requested username.  
#C If there is such a user, the method returns it.  
#D If a user with this username does not exist, the method throws an exception.

The `loadUserByUsername(String username)` method searches in the list of users for the given `username` and returns the desired `UserDetails` instance. If there is no instance with that `username`, it throws `UsernameNotFoundException`.

We can now use this implementation as our `UserDetailsService`. We add it as a bean in the configuration class and register one user within it.

### **Listing 3.14 The UserDetailsService implementation is 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.

#### **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 the user “john” with the password “12345”, we get back an HTTP 200 OK. If we use something else, the application will return a 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. The `UserDetailsService` is the contract needed by Spring Security to be able to do the authentication. But generally, in applications, there is also a need for managing these 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, the `UserDetailsManager`. The `UserDetailsManager` extends `UserDetailsService` and adds some 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);
}

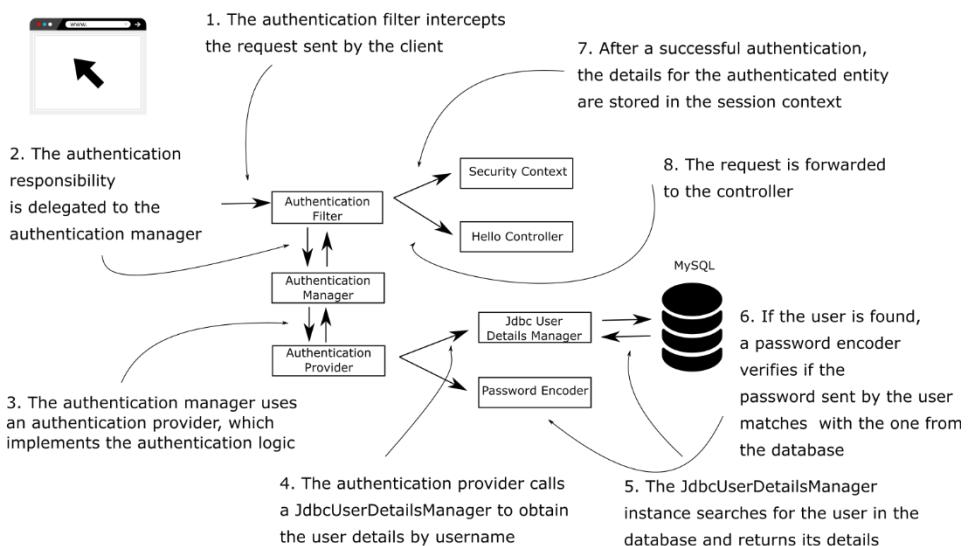
```

The `InMemoryUserDetailsManager` object that we have used in chapter 2 is actually a `UserDetailsManager`. At that time, we have only considered its `UserDetailsService` characteristics, but now you understand better why we were able to call a `createUser()` method on the instance.

### USING A JDBCUSERDETAILSMANAGER FOR USER MANAGEMENT

Beside the `InMemoryUserDetailsManager`, another `UserDetailManager` often used is the `JdbcUserDetailsManager`. The `JdbcUserDetailsManager` manages the users in a 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 in action with an example. In the following example, you'll implement an application that manages the users in a MySQL database using the `JdbcUserDetailsManager`. In figure 3.4, you have an overview of the place taken by the `JdbcUserDetailsManager` implementation in the flow.



**Figure 3.4 The authentication flow. In this example we use a `JdbcUserDetailsManager` as our `UserDetailsService` component. The `JdbcUserDetailsManager` uses a database to manage the 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, the database name is `spring`. 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 allows you to override these default names if you would like so. The purpose of the `users` table is to keep the records of the users. 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 or a client application. For example, for MySQL, you could choose to use MySQL Workbench to do this. But the easiest would be to leave 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 queries. Spring Boot will automatically run them 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 wouldn't need to install a separate DBMS solution. If you prefer, you could go with H2 as well when developing the applications presented with this book. I chose to implement the examples having an external DBMS to make it clear it's an external component of the system and, this way, avoid confusion.

You use the code in listing 3.16 to create the `users` table with a MySQL server. You 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 the 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** You observe that, for simplicity, in the examples provided with this book, I skip the definitions of indexes or foreign keys.

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

```
INSERT IGNORE INTO `spring`.`authorities` VALUES (NULL, 'john', 'write');
INSERT IGNORE INTO `spring`.`users` VALUES (NULL, 'john', '12345', '1');
```

For your project, you will have to add at least the dependencies stated in listing 3.18. Check your `pom.xml` file to make sure you have 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>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

**NOTE** You can use in your examples any SQL database technology as long as you add the correct JDBC driver to the dependencies.

You should 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'll have to add the following lines.

```
spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=<your user>
spring.datasource.password=<your password>
spring.datasource.initialization-mode=always
```

In the configuration class of the project, you'll define the `UserDetailsService` and the `PasswordEncoder`. The `JdbcUserDetailsService` needs the `DataSource` to connect to the database. The data source can be autowired through a parameter of the method, as presented by listing 3.19 or through an attribute of the class.

**Listing 3.19 Registering the JdbcUserDetailsService in the configuration class**

```
@Configuration
public class ProjectConfig {

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        return new JdbcUserDetailsService(dataSource);
    }

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

To access any endpoint of the application, now you will have to use HTTP basic authentication with one of the users stored in the database. To prove this, we create a new endpoint and call it with curl.

#### **Listing 3.20 The test endpoint we will use 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 of the call is:

```
Hello
```

The JdbcUserDetailsManager also allows you to configure the queries used. In the previous example, we made sure we use the exact names for the tables and columns as the JdbcUserDetailsManager implementation expected them. But it could be that for your application, these names are not the best choice. JdbcUserDetailsManager offers the possibility to override the queries, as presented by listing 3.21.

#### **Listing 3.21 Changing the SQL query used by JdbcUserDetailsManager to find the user in the database**

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    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(authsByUserQuery);
    return userDetailsManager;
}
```

In the same way, we could 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, such as the authentication works with the new table structure. Project `ssia-ch3-ex2` features a possible solution.

#### USING A `LDAPUserDetailsManager` FOR USER MANAGEMENT

Spring Security also offers an implementation of `UserDetailsService` for LDAP. Even if it is less popular than the `JdbcUserDetailsService`, you could count on it if you need to integrate with an LDAP system for your user's management. In project `ssia-ch3-ex3` you find a very simple demonstration of using the `LdapUserDetailsService`. Because I can't use a real LDAP server for the demonstration, for this example, I have set up an embedded one in my Spring Boot application. To set up the embedded LDAP server, I defined a very simple LDAP Data Interchange Format (LDIF) file. In listing 3.22, you find 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

#A The definition of the base entity
#B The definition of a group entity
#C The definition of a user
```

In the LDIF file, I have added only one user, which we need to test the app's behavior at the end. We can add the LDIF file directly to the `resources` folder. This way, it'll be automatically in the classpath so we can easily refer to it later. I'll name the LDIF file `server.ldif`. To work with LDAP and to allow Spring Boot to start an embedded LDAP server, you need to add in `pom.xml` the dependencies presented 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 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 are 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 configure your application to use it. Listing 3.23 shows you how to configure the `LdapUserDetailsManager` to enable your app to authenticate the users through the LDAP server.

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

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

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

        var manager = new LdapUserDetailsManager(cs);      #C
        manager.setUsernameMapper(      #D
            new DefaultLdapUsernameToDnMapper("ou=groups", "uid"));

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

        return manager;
    }

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

#A Adding a `UserDetailsService` implementation to Spring context.

#B Creating a context source to specify the address of the LDAP server.

#C Creating the `LdapUserDetailsManager` instance.

#D Setting a username mapper to instruct the `LdapUserDetailsManager` on how to search for the users.

#E Setting the group search base, which the app needs as well for searching the users.

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 observe you need to authenticate with user john if you want to app to allow you to call the endpoint. The next code snippet shows you the result of calling the endpoint with `curl`.

### 3.4 Summary

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

# 4

## *Dealing with passwords*

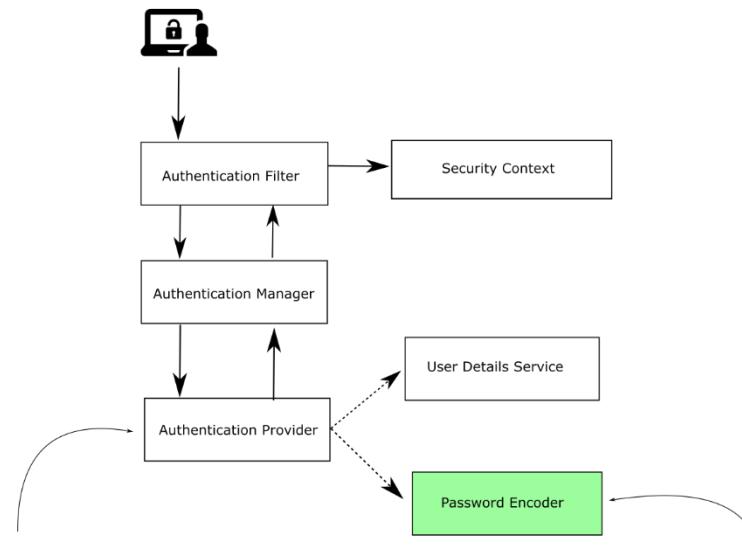
### This chapter covers

- 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 the passwords? They're certainly an essential piece in the authorization 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 for the management of passwords.

### 4.1 Understanding the `PasswordEncoder` contract

From chapter 3, you should now have a clear image of what `UserDetails` interface is, and multiple ways to use its implementations. But as you learned in chapter 2, the user representation is managed during the authentication and authorization processes by different actors. You also learned that some of them have defaults, like the `UserDetailsService` and the `PasswordEncoder`. You noticed that you could override the defaults. We will continue with a deep understanding of these beans and ways to implement them, and in this section, we will analyze the `PasswordEncoder`. Figure 4.1 reminds you of the place taken by the `PasswordEncoder` in the authentication process.



**Figure 4.1 The authentication process: The AuthenticationProvider uses the PasswordEncoder to validate the password of the user in the authentication process.**

Because in general, a system doesn't manage the passwords in plain text, they usually suffer a sort of transformation that makes it more challenging to be read and stolen. For this responsibility, Spring Security defines a separate contract. To explain it easily, I'll provide in this section 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. Next, I'll provide you a list of the most known and widely used implementations of the `PasswordEncoder` provided by Spring Security.

#### 4.1.1 The definition of the `PasswordEncoder` contract

In this section, we discuss the definition of the `PasswordEncoder` contract. You'll 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. Any system stores the passwords encoded somehow. You preferably store them hashed so that there are no chances that someone would be able to read the passwords. The `PasswordEncoder` can also encode the password. These couple of methods, the `encode()` and `matches()`, that the contract

declares are actually the definition of its responsibility. Both of them are parts of the same contract as they are strongly linked one to the other. The way the application encodes a password is strongly related to the way the password is validated.

Let's first review the content of the interface:

```
public interface PasswordEncoder {

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

    default boolean upgradeEncoding(String encodedPassword) {
        return false;
    }
}
```

As presented in the previous code snippet, the interface defines two abstract methods and one with a default implementation. The `encode()` and the `matches()` methods that are abstract are also the ones that you will mostly hear about when dealing with a `PasswordEncoder` implementation.

The purpose of the `encode(CharSequence rawPassword)` method is to return a transformation of a string provided. In terms of Spring Security functionality, it will be used to provide encryption or hash for a given password. You can use the `matches(CharSequence rawPassword, String encodedPassword)` 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 from the contract. If overridden 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 the PasswordEncoder contract

As you observe, the two methods `matches()` and `encode()` have a strong relationship one to the other. Whenever 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 to implement the `PasswordEncoder` you'll be able to choose how the application manages the passwords for the authentication process.

The most straightforward implementation would be a password encoder that considers passwords in plain text. That is, it doesn't do any encoding on the password. Managing passwords in clear text is what the instance of `NoOpPasswordEncoder` does precisely. We have used this class in our first example in chapter 2. If you were to write your own, it would look like in listing 4.1.

##### **Listing 4.1 The simplest implementation of a PasswordEncoder**

```

public class PlainTextPasswordEncoder
    implements PasswordEncoder {

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

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

```

The result of the encoding is always the same as the password. So to check if it matches, you only need to compare the strings with `equals()`. If we consider having an encoding, a simple implementation of `PasswordEncoder` that uses as encoding the hashing algorithm SHA-512 will look like the code in the listing 4.2.

#### **Listing 4.2 An implementation of the `PasswordEncoder` that uses SHA-512 hashing algorithm**

```

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 have omitted 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 that 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.

For your curiosity, listing 4.3 presents the method that I have omitted in listing 4.2.

#### **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();
}

```

But 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 implementations of PasswordEncoder

While knowing to implement your `PasswordEncoder` is powerful, you also have to be aware that Spring Security already provides you with some advantageous implementations. If one of them matches your application, you don't need to rewrite it. In this section, we discuss the `PasswordEncoder` implementation options which Spring Security provides:

- `NoOpPasswordEncoder`, which doesn't encode the password keeping it in cleartext – we use this implementation only for examples. Being that it doesn't hash the password, you **should never use it in a real-world scenario**.
- `StandardPasswordEncoder`, which uses SHA-256 to hash the password. This implementation **is now deprecated, and 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.
- `Pbkdf2PasswordEncoder`, which uses Password-Based Key Derivation Function 2 (PBKDF2).
- `BCryptPasswordEncoder`, which uses a BCrypt strong hashing function to encode the password.

`SCryptPasswordEncoder`, which uses a SCrypt hashing function to encode the password.

About hashing and also referring to the mentioned algorithms, you'll also find a good discussion in chapter 2 of *Real-World Cryptography* by David Wong (Manning, 2020):

<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 previous example described in listing 4.1. For this reason we only use this password encoder with theoretical examples. The `NoOpPasswordEncoder` class is designed as a singleton. You can't call its constructor directly from outside, but you can use the `NoOpPasswordEncoder.getInstance()` method to obtain the instance of the class.

```
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 can set the value of this secret by the constructor's parameter. If you choose to call the no-arguments constructor, the implementation will use the empty string as a value for the key. However, the `StandardPasswordEncoder` is deprecated now, and I don't recommend that you use it within 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 Password-Based Key Derivation Function 2 (PBKDF2) for the password encoding. To create instances of the `Pbkdf2PasswordEncoder`, you have the following options:

```
PasswordEncoder p = new Pbkdf2PasswordEncoder();
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret");
PasswordEncoder p = new Pbkdf2PasswordEncoder("secret", 185000, 256);
```

The PBKDF2 is a pretty easy slow-hashing function that performs an HMAC as many times as specified by an iterations argument. The 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. You can choose more or fewer iterations as well as the length of the result. The longer the hash, the more powerful the password is. However, mind that the performance is affected by these values: the more iterations, the more resources your application consumes as well. You should make a wise compromise between the resources consumed for generating the hash and the needed strength of the encoding. In this book, I refer to several cryptography concepts that you might like to know more in detail. For relevant information on HMACs and other cryptography details, I recommend *Real-World Cryptography* by David Wong (Manning, 2020). Chapter 3 of *Real-World Cryptography* teaches you more on HMAC:

<https://livebook.manning.com/book/real-world-cryptography/chapter-3/>

If you do not specify one of the second or the third values, the default is 185000 for the number of iterations and 256 for the length of the result. You could do this by choosing one of the other two overloaded constructors: the one without parameters `Pbkdf2PasswordEncoder()` and the one that receives only the secret value as a parameter `Pbkdf2PasswordEncoder("secret")`.

Another excellent option offered by Spring Security is the `BCryptPasswordEncoder`, which uses a BCrypt strong hashing function to encode the password. You could instantiate the `BCryptPasswordEncoder` by calling the no-arguments constructor. But you also have the option to specify a strength coefficient representing the log rounds used in the encoding process. Moreover, you can as well 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 (logarithmic rounds) value provided affect the number of iterations used in the hashing operation. 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 it by calling one of the second or third overloaded constructors, as shown in the previous code snippet.

The last option I'll present you is `SCryptPasswordEncoder`. This password encoder uses a SCrypt hashing function to encode the password. For the `SCryptPasswordEncoder`, you have two options to create its instances:

```
PasswordEncoder p = new SCryptPasswordEncoder();
PasswordEncoder p = new SCryptPasswordEncoder(16384, 8, 1, 32, 64);
```

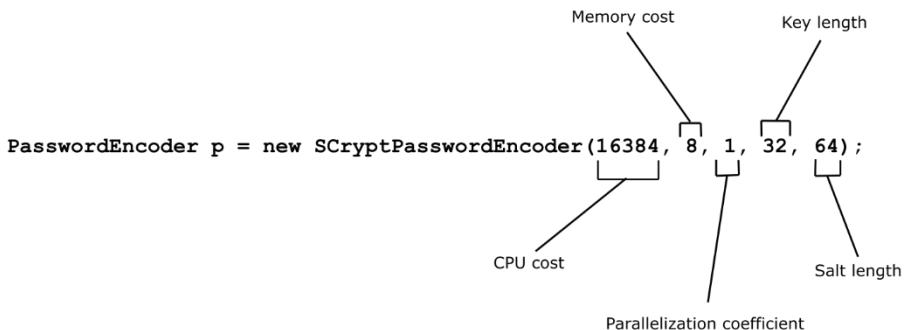


Figure 4.2 The `SCryptPasswordEncoder` constructor that takes five parameters allows you to configure the CPU cost, Memory cost, Key length, and Salt length.

The values in the examples are the ones used if you create the instance by calling the no-arguments constructor.

#### 4.1.4 Having multiple encoding strategies with `DelegatingPasswordEncoder`

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

In some applications, you might find it useful to have various password encoders and choose from them depending on some specific configuration. A most common scenario in which

I found 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 the newly registered users. You do not want to change it for the existing credentials. So you will end up having multiple kinds of hashes. How to 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, will delegate 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` will delegate to the correct implementation of the `PasswordEncoder` based on the prefix of the password.

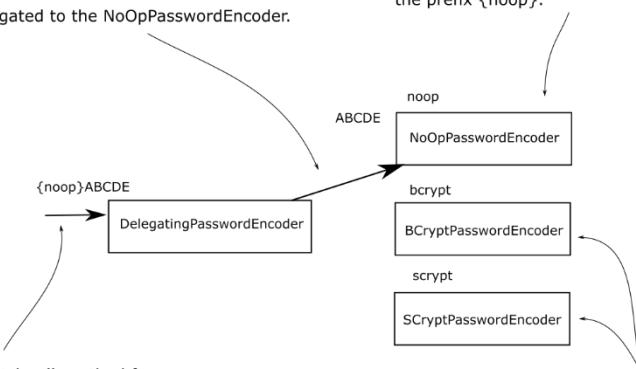
It sounds complicated, but with an example, you'll observe that it is pretty easy. Figure 4.3 presents the relationship between the `PasswordEncoder` instances. The `DelegatingPasswordEncoder` has a list of `PasswordEncoder` implementations to which it delegates. The `DelegatingPasswordEncoder` stores 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` will delegate 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.

When you call the `matches()` method with a password with the prefix {noop} the call is delegated to the `NoOpPasswordEncoder`.

The `NoOpPasswordEncoder` is registered for the prefix {noop}.

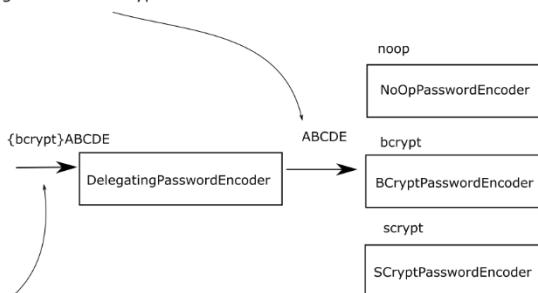
Call to the `matches()` method for a password having the prefix {noop}.

The `DelegatingPasswordEncoder` also contains references for password encoders under different prefixes.



**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}". Because the password has the prefix "{noop}", the `DelegatingPasswordEncoder` will forward the operation to the `NoOpPasswordEncoder` implementation.

When you call the matches() method  
with a password with the prefix {bcrypt}  
the call is delegated to the BCryptPasswordEncoder



Call to the matches() method for  
a password having the prefix {bcrypt}

**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 will forward 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 them together in a DelegatingPasswordEncoder like in listing 4.4.

#### **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 a `SCryptPasswordEncoder` and delegates default to the `BCryptPasswordEncoder` implementation. Based on the prefix of the hash, the `DelegatingPasswordEncoder` uses the right `PasswordEncoder` implementation for the operation of matching the password. This prefix has the key that identifies the password encoder to be used from the map of encoders. You have to specify the key between curly braces. If there is no prefix, it will use 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` used is "bcrypt".

**NOTE** In the prefix of the hash, the key is between curly braces. The curly braces are part of the prefix, and they should surround the name of the key. E.g. For the key `noop`, the prefix is `{noop}`.

For example, if the provided hash is the following: `{noop}12345`, the `DelegatingPasswordEncoder` will delegate to the `NoOpPasswordEncoder` that we have registered for the prefix "noop". Again, don't forget that the curly braces are mandatory to be part of the prefix.

If the hash looks like in the next code snippet, the password encoder used will be the one we have assigned to the prefix "bcrypt" which is the `BCryptPasswordEncoder`. The `BCryptPasswordEncoder` is also the one to which the application will delegate if there is no prefix at all, because we have defined it as the default implementation.

```
{bcrypt}$2a$10$xn3LI/AjqicFYzFruSwve.681477XaVNaUQbr1gioaWPn4t1KsnmG
```

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 the implementation of the `DelegatingPasswordEncoder` with `BCrypt` as a default encoder.

```
PasswordEncoder passwordEncoder =
    PasswordEncoderFactories.createDelegatingPasswordEncoder();
```

## 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  $x \rightarrow y$  applied to "ABCD" will produce "DCBA".

**Encryption** is a particular type of encoding, where, to obtain the output, both the input value and a key are provided. 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 would look like this:  $(x, k) \rightarrow y$  where  $x$  is the input,  $k$  is the key, and  $y$  is the result of the encryption. This way, an individual knowing the key can use a known function to obtain the input from the output:  $(y, k) \rightarrow x$ . 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 as well 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  $x \rightarrow y$  then we should also have a matching function  $(x, y) \rightarrow \text{boolean}$

Sometimes the hashing function could also use a random value added to the input:  $(x, k) \rightarrow y$ . 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 sum up the contracts we have discussed and applied up to now in this book, table 4.1 shortly describes each of the components.

**Table 4.1 The interfaces that represent the main contracts for the 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 will be allowed to the user, like read, write, delete, etc.
UserDetailsService	Represents the object used to retrieve the user details by making use of its username.
UserDetailsManager	A more particular contract for UserDetailsService. Besides retrieving the user by username it could also be used to mutate the collection of users or a specific user.
PasswordEncoder	A password encoder specifies how the password is encrypted or hashed and how to check if a given encoded string matches a plain text password.

## 4.2 More about the Spring Security Crypto Module

In this section, we discuss the Spring Security Crypto Module (SSCM), which is the part of Spring Security that is dealing with cryptography. Using encryption and decryption functions and generating keys isn't offered out-of-the-box with the Java language. An this constrains developers to add dependencies that provide a more accessible approach to these features. To make our lives easier, Spring Security also provides its own solution. This solution 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 above. In this section, we will discuss what other options the SSCM offers related to cryptography. I will present you by example, how to use two essential features from the Spring Security Crypto Module:

- Key generators - objects used to generate keys used in hashing and encryption algorithms
- Encryptors - objects used to encrypt and decrypt data

### 4.2.1 Using key generators

In this section, we discuss key generators. A 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 you to be 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`.

A string key generator, represented by the `StringKeyGenerator` contract, can be used to obtain a key as a `String`. Usually, we use this key as a salt value for a hashing or encryption algorithm. Below you can find the definition of the `StringKeyGenerator` contract.

```
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 then 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();
}
```

Beside the `generateKey()` method, which returns the key as a `byte[]`, the interface defines as well a method that returns the key length in number of bytes. A default `BytesKeyGenerator` generates keys of 8 bytes length, but you can specify the length of the keys that will be generating when obtaining the generator instance.

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

In the previous code snippet, the key generator generates keys of 8-bytes long. 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 key generated by the `BytesKeyGenerator` created with the `KeyGenerators.secureRandom()` method is unique per each call of the `generateKey()` method. In some cases, we prefer an implementation that will return the same key value per each call of the same key generator. In this case, we can create a `BytesKeyGenerator` with the `KeyGenerators.shared(int length)` method.

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

In the above code snippet, `key1`, and `key2` have the same value.

#### 4.2.2 Using encryptors for encryption and decryption operations

In this section, we'll apply the implementations of encryptors, which Spring Security offers, with code examples. An "encryptor" is an object that implements an encryption algorithm. When talking about security, encryptions, and decryptions are common operations, so expect to need them 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 the encryption and the decryption. There are two types of encryptors defined by the Spring Security Crypto Module: `BytesEncryptor` and `TextEncryptor`. While they have similar responsibilities, the two kinds treat different data types. The `TextEncryptor` manages data as `String`. Its methods receive strings as inputs and return strings as outputs as can be seen from the definition of its interface:

```
public interface TextEncryptor {

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

}
```

The `ByteEncryptor` 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 `ByteEncryptor` we could use the `Encryptors.standard()` or the `Encryptors.stronger()` methods.

```
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 the input.

To build a stronger instance of 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 GCM (Galois Counter Mode) as the mode of operation. The standard mode uses CBC (Cipher Block Chaining), 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()`. Besides these methods to create the encryptors, there is also a method returning a dummy `TextEncryptor` that 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 taking into consideration the time spent for encryption. The method that returns this no-op encryptor is called `Encryptors.noOpText()`. In the code snippet, you find an example of using a `TextEncryptor`.

```

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

```

Even if it is a call to an encryptor, in the above example, `encrypted` and `valueToEncrypt` are the same.

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

```

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);

```

#A Creating a `TextEncryptor` object that is using a salt and a password.

For `Encryptors.text()` and `Encryptors.delux()`, the `encrypt()` method called on the same input repetitively will generate different outputs. The different outputs occur because of the randomly generated initialization vectors used in the encryption process. In the real world, you'll find cases in which you don't want this to happen, as in the case of the OAuth API key, for example. We will discuss more on the OAuth 2 subject in chapters 12 to 15. This kind of input is called a "queryable text", and for this situation, you would make use of an

`Encryptors.queryableText()` instance. The use of this encryptor guarantees you that sequential encryption operations will generate the same output for the same input.

In the following example, the value of the “`encrypted1`” variable equals the value of the “`encrypted2`”.

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

TextEncryptor e = Encryptors.queryableText(password, salt);      #A
String encrypted1 = e.encrypt(valueToEncrypt);
String encrypted2 = e.encrypt(valueToEncrypt);
```

#A Creating a queryable text encryptor.

### 4.3 Summary

- The `PasswordEncoder` has one of the most critical responsibilities in the authentication logic: dealing with passwords. Spring Security offers a bunch of alternatives in terms of hashing algorithms, which makes the implementation only a matter of choice.
- Spring Security Crypto Module 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 which help you apply encryption and decryption of data.

# 5

## *Implementing authentication*

### This chapter covers

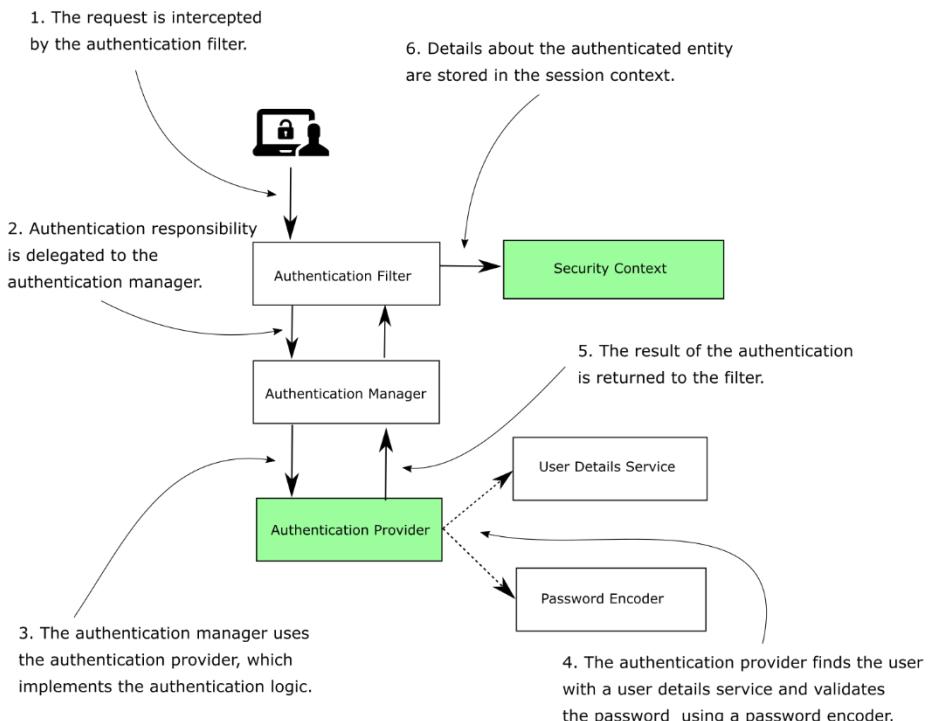
- **Implementing the authentication logic using a custom `AuthenticationProvider`**
- **Using the HTTP Basic authentication method and Form Login authentication method**
- **Understanding and managing the `SecurityContext`**

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

However, the `AuthenticationProvider` layer is the one responsible for the logic of authentication. The `AuthenticationProvider` is the place where you find the conditions and instructions that decide whether to authenticate a request. The component that delegates this responsibility to the `AuthenticationProvider` is the `AuthenticationManager`, which receives the request from the HTTP filter layer. We'll discuss the filters layer in detail in chapter 9. The authentication process has only two possible results:

- 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 an HTTP 401 Unauthorized.
- The entity making the request is authenticated. The details about the requester are stored such that the application can use them for authorization. As you'll find out in this chapter, the `SecurityContext` is the instance that stores the details about the current authenticated request.

To remind you of the actors and the links between them, in figure 5.1, you have the diagram that you've also seen in chapter 2.



**Figure 5.1** The authentication flow in Spring Security. This process defines how the application identifies someone making a request. The components discussed in this chapter are shaded differently. 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 5.1). Then, in chapters 7 and 8, you'll learn how authorization works, which is the process that follows authentication in the HTTP request. We have to discuss how to implement the `AuthenticationProvider`. To be able to do this, you first have to know how Spring Security understands a request in the authentication process. To give a clear description of how to represent a request, we start with the `Authentication` interface. Once we have discussed this, we go further and observe what happens with the details of a request after successful authentication. After successful authentication, we discuss the `SecurityContext` and the way Spring Security manages it. Close to the end of the chapter, you will learn how we could customize the HTTP Basic authentication method. Together with this, we'll also

discuss another option of authentication method we could use for our applications, the Form Login.

## 5.1 Understanding the AuthenticationProvider

In enterprise applications, sometimes you might find yourself in the situation in which the default implementation of authentication based on username and password does not apply. Several scenarios could be required to be implemented by your application when it comes to authentication (figure 5.2). 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. 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 requested scenarios.

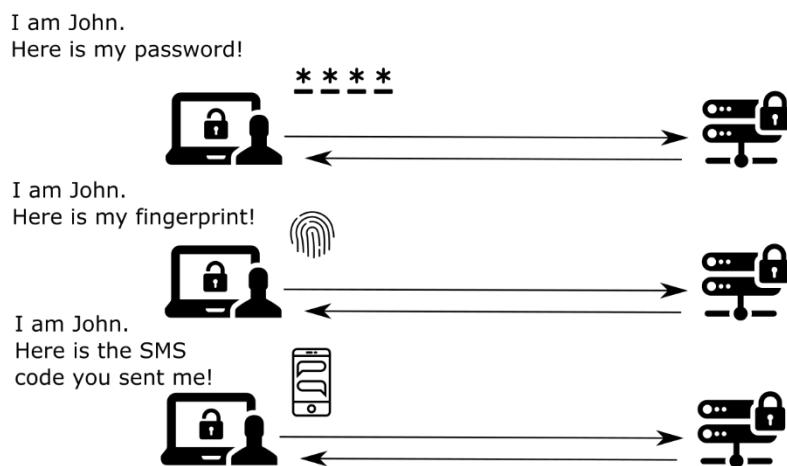


Figure 5.2 For an application, you might need to implement authentication in different fashions. While in most cases, a username and a password are enough, in some cases, the scenario of the user's authentication might be more complicated.

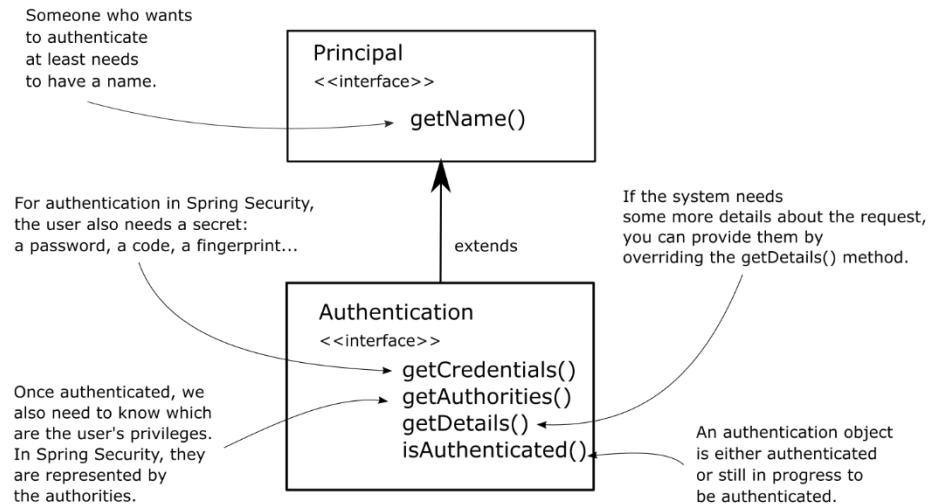
Usually, a framework also provides a set of the most commonly used implementations, but it cannot, of course, cover all the possible options. In terms of Spring Security, you will use the `AuthenticationProvider` contract to define any custom authentication logic. In this section, you will learn to represent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

- In section 5.1.1, we analyze how Spring Security represents the authentication event.
- In section 5.1.2, we discuss the `AuthenticationProvider` contract, which is responsible for the authentication logic.
- In section 5.1.3, you'll write custom authentication logic by implementing the `AuthenticationProvider` contract in an example.

### 5.1.1 Representing the request during authentication

In this section, we discuss how Spring Security represents a request during the authentication process. It is important to touch this aspect before diving into implementing custom authentication logic. As you'll learn in section 5.1.2, to implement a custom `AuthenticationProvider`, you first need to understand how to represent the authentication event itself. In this section, we take a look at the contract representing the 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 details of the entity that requested access to the application. You can use the information related to the authentication request event during the authentication process and after its end. The details of the entity that requested access to the application are also called a principal. If you ever used the Java Security API in any app, you learned that, in the Java Security API, an interface named `Principal` represents the same concept. The `Authentication` interface of Spring Security extends this contract. The `Authentication` contract in Spring Security represents a principal and also adds information on whether the authentication process finished, as well as the collection of authorities. The fact that this contract was designed to extend the `Principal` contract from Java Security API is a good aspect in terms of compatibility with implementations of other frameworks or applications. This flexibility allows for more facile migrations to Spring Security from applications that already implement authentication in another fashion.



**Figure 5.3** The `Authentication` contract inherits from the `Principal` contract. It adds the requirements from above, like the need for the password or possibility to specify more details about the authentication request. Some of these details are Spring Security specific, like the list of authorities.

Let's find out more about the design of the `Authentication` interface in listing 5.1.

#### **Listing 5.1** 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

- `isAuthenticated()` - Returns true if the authentication process ended 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 the collection of granted authorities for the authenticated request.

We discuss the other methods in later chapters, where appropriate to the implementation we work on.

### 5.1.2 Implementing the custom authentication logic

In this section, we discuss implementing custom authentication logic. We'll analyze the contract of Spring Security related to this responsibility to understand its definition. With these details, you'll be able to implement custom authentication logic with a code example in section 5.1.3.

The `AuthenticationProvider` in Spring Security takes care of the authentication logic. The default implementation of the `AuthenticationProvider` delegates the responsibility of finding the user in the system to a `UserDetailsService`. It uses as well the `PasswordEncoder` for password management in the process of authentication. Listing 5.2, gives the definition of the `AuthenticationProvider` interface, which you need to implement for defining a custom authentication provider for your application.

#### **Listing 5.2 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 as well. 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 `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 could have the possibility of using multiple `Authentication` types separated at the HTTP filter level. We discuss this aspect more in chapter 9. You'll also find an applied example for having multiple `AuthorizationProvider` classes in chapter 11, which is the second hands-on chapter of this book.
- 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 needed details about the authenticated entity. Usually, the application also removes from this instance sensitive data, like the password. The password is not required anymore, and keeping these details could potentially expose them to unwanted eyes.

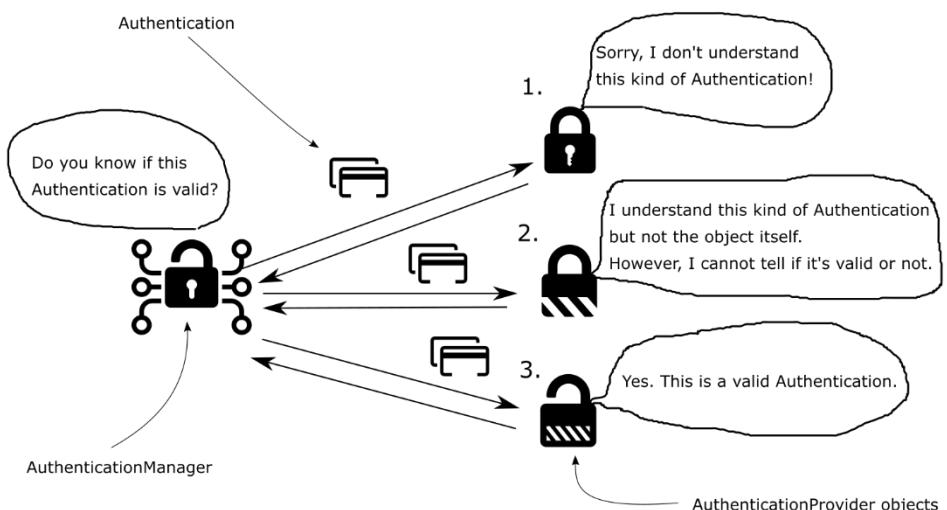
The second method in the `AuthenticationProvider` interface is `supports(Class<?> authentication)`. You'll implement this method to return true if the current `AuthenticationProvider` supports the type provided as the `Authentication` object. Observe that even if this method returns true for an object, there is still a chance that the `authenticate()` method will reject 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 and not only its type.

An analogy of how the authentication manager and authentication providers work together to validate or invalidate an authentication request is a more complex door lock. You can open this door lock either by using a card or an old fashioned physical key (figure 5.4). The lock itself is the authentication manager who decides whether to open the door. To make the 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. When you present a card to open the door, the authentication provider that works only with physical keys will complain that it doesn't know this kind of authentication. But the other one will support this kind of authentication and will verify whether the card is valid for that door. This is actually the purpose of the `supports()` methods.

Besides testing the authentication type, Spring Security adds one more layer for the flexibility of the implementation. The door lock can have the possibility of recognizing 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 5.4** The authentication manager delegates to one of the available authentication providers. The

authentication provider may not support the provided type of authentication. If it does support the type of object, it might not know for sure how to authenticate that specific object. The authentication is evaluated, and an authentication provider that could say if the request is correct or not will respond to the authentication manager.

### 5.1.3 Applying the custom authentication logic

In this section, we write an example in which we implement custom authentication logic. You can find this example in project `ssia-ch5-ex1`. With this example, you'll apply what you've learned about the `Authentication` and `AuthenticationProvider` interfaces in sections 5.1.1 and 5.1.2.

In listings 5.3 and 5.4, we build step by step, an example of how to implement a custom `AuthenticationProvider`. The steps, as also presented in figure 5.5, are :

1. Declare a class that implements the `AuthenticationProvider` contract.
2. Decide which kinds of `Authentication` objects will the new `AuthenticationProvider` support.
3. Override the `supports(Class<?> c)` method to specify which type of `Authentication` is supported by the current `AuthenticationProvider` we define.
4. Override the `authenticate(Authentication a)` method to implement the `Authentication` logic.
5. Register an instance of the new `AuthenticationProvider` implementation with Spring Security.

**Listing 5.3 Overriding the `supports()` method of the `AuthenticationProvider`**

```
@Component
public class CustomAuthenticationProvider
    implements AuthenticationProvider {
    // Ommitted code
    @Override
    public boolean supports(Class<?> authenticationType) {
        return authenticationType
            .equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

In listing 5.3, we defined a new class that implements the `AuthenticationProvider` interface. We marked 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 will this `AuthenticationProvider` support. This aspect depends on what type we expect to be provided as a parameter to the `authenticate()` method. If you didn't customize anything at the authentication filter level (which is our case, but we'll do that when reaching chapter 9), then the type will be defined by the class `UsernamePasswordAuthenticationToken`. This class is an implementation of the

Authentication interface and is used to represent a standard authentication request with username and password. So, now, with this definition, we have 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.

#### **Listing 5.4 Implementing the authentication logic**

```
@Component
public class CustomAuthenticationProvider
    implements AuthenticationProvider {

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private PasswordEncoder passwordEncoder;

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

        UserDetails u = userDetailsService.loadUserByUsername(username);

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

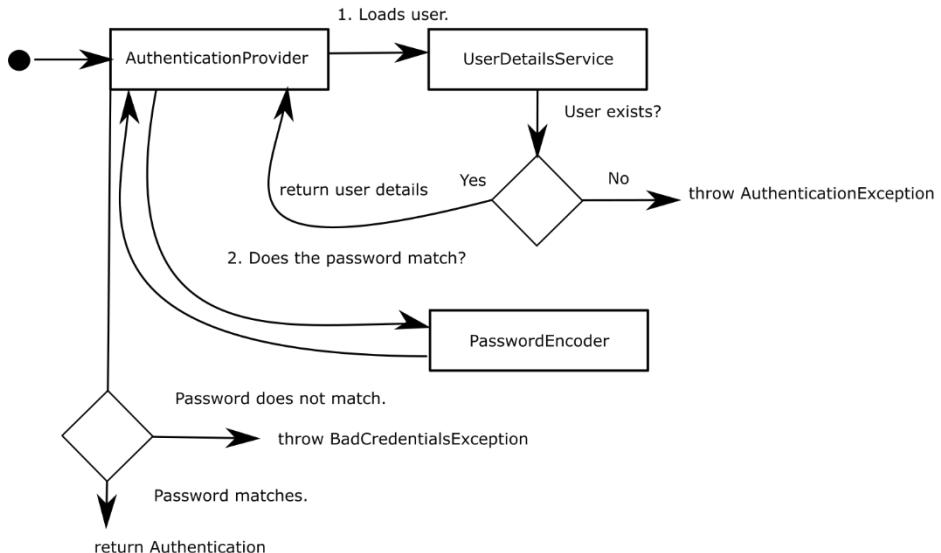
    // Omitted code
}
```

#A If the password matches, the method returns an implementation of the Authentication contract with the needed details

#B If the password does not match, the method throws an exception of type AuthenticationException.

BadCredentialsException inherits from AuthenticationException

The logic in listing 5.4 is simple. Figure 5.5 shows this logic in a visual. 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 will stop, and the HTTP filter will set 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, an instance of `Authentication` marked as "authenticated" and containing the details about the request is returned.



**Figure 5.5** 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 validates if the password matches 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`, override the `configure(AuthenticationManagerBuilder auth)` method of the `WebSecurityConfigurerAdapter` class in the configuration class of the project.

#### Listing 5.5 Registering the `AuthenticationProvider` in the configuration class

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationProvider authenticationProvider;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(authenticationProvider);
    }

    // Omitted code
}

```

**NOTE** In listing 5.5, I used the `@Autowired` annotation over a field declared as `AuthenticationProvider`. Being an interface, 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 have declared and added to the Spring context using the `@Component` annotation.

This is it! You successfully defined your custom implementation of `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 that sometimes those who fail in using the framework sometimes believe it's the framework's fault. Let me tell you a story.

One of the previous winters, the head of development of 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 a long time ago, in Spring's early days. Unfortunately, when implementing the application's class design, the developers at that time didn't rely properly on the Spring Security's backbone architecture. They've only relied on the filter chain, almost reimplementing entire features from Spring Security as custom code.

Developers observed that with time, customizations became more and more difficult. But nobody took action in redesigning the component to use the contracts as intended in Spring Security properly. Much of the difficulty was 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'd 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 they only used maybe ten percent of what Spring Security offers in their application. 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 re-write a big bunch of custom code to rely correctly on Spring Security and, thus, make the application easier to extend in what concerns security implementations. We also discovered some other issues unrelated to Spring Security, but that's another story.

A few lessons to take from this story:

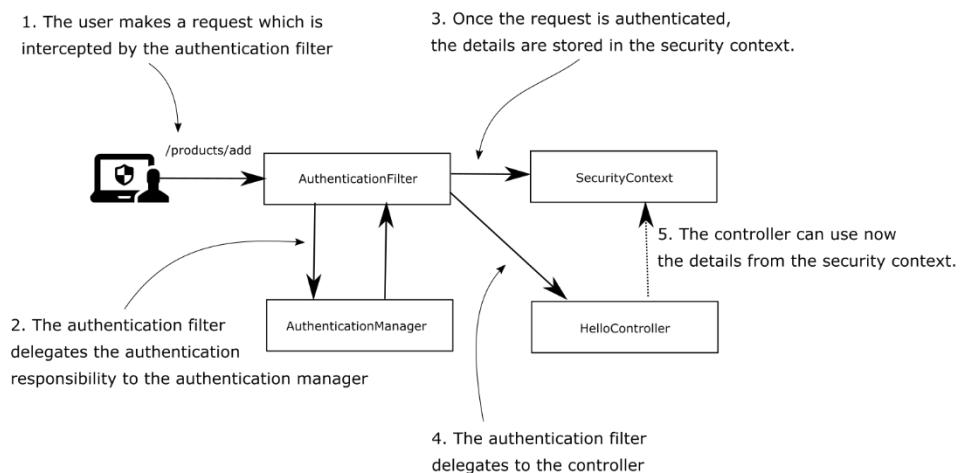
- 1) A framework, and especially one widely-used in applications, was written with the participation of many smart individuals. It's hard to believe it can be that badly implemented. Always analyze your application before concluding that any problems are the framework's fault.
- 2) When deciding to use a framework, make sure you understand at least its basics well. Be careful with the resources you use to learn about the framework. Sometimes, articles you find around the web show you how to do quick workarounds and not necessarily how to correctly implement a class design. Use multiple sources for your research. To clarify your misunderstandings, write proof-of-concepts when unsure how to use something.
- 3) If you decide to use a framework, use it as much as possible for its purpose. 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 have several benefits. We know they're tested and, there are fewer changes they include vulnerabilities. Also, a good framework relies on abstractions, which helps you create maintainable applications. When you write your own implementations, you're rather more susceptible to including vulnerabilities.

## 5.2 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've finished this section, you'll know how to configure the security context for various situations. This way, you'll also be able to use the details about the authenticated user, stored by the security context, in configuring authorization in chapters 7 and 8.

There is a great chance that you will need details about the authenticated entity after the authentication process ends. You might need, for example, to refer to the username or the authorities of the currently authenticated user. Is this information still accessible after the authentication process has finished? 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 security context.



**Figure 5.6** 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, as shown in listing 5.6.

### Listing 5.6 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` managed itself? Spring Security offers three strategies to manage the `SecurityContext` with an object with the role of a manager named `SecurityContextHolder`:

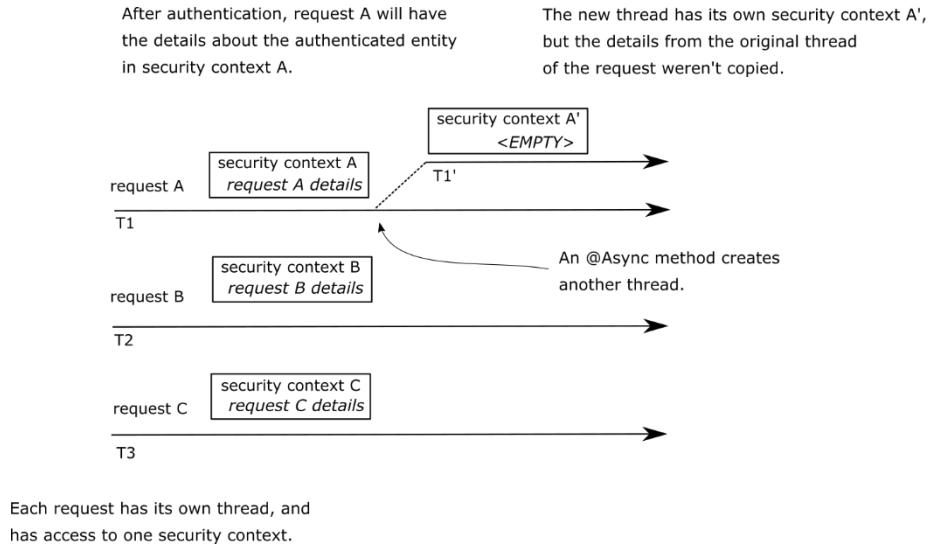
- `MODE_THREADLOCAL`, which allows each thread to store its own details in the security context. In a thread-per-request web application, this is a common approach as each request has an individual thread.
- `MODE_INHERITABLETHREADLOCAL`, which is similar to the `MODE_THREADLOCAL` but also instructs 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.
- `MODE_GLOBAL`, which 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 will have 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.

### 5.2.1 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 one for managing the security context used by Spring Security. With this strategy, Spring Security uses a `ThreadLocal` to manage the context. The `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 they have stored in the collection. This way, each request will have access to its security context. Any other thread will not have access to other's thread-local. 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 would generally want to have for a web backend application.

Figure 5.7 offers an overview of this functionality. Each request (A, B, and C) has its own thread allocated (T1, T2, and T3). This way, each request will only see the details stored in their 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. The details from the parent thread (the original thread of the request) are not copied to the security context of the new thread.



**Figure 5.7** 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.

**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 will discuss the security for reactive approaches in detail in chapter 19.

Being the default strategy for managing the security context, you do not need to configure anything explicitly to use this mode. 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 5.7, 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-ch5-ex2`.

#### **Listing 5.7 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. So you don't need to

refer every time to the `SecurityContextHolder` class explicitly. This approach is better, as presented in listing 5.8.

#### **Listing 5.8 Spring injects the value of the Authentication in the parameter of the method**

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

#A Spring Boot injects the current Authentication in the method parameter.

When calling the endpoint with a correct user, the response body will contain the username.

```
curl -u user:99ff79e3-8ca0-401c-a396-0a8625ab3bad http://localhost:8080/hello
Hello, user!
```

### **5.2.2 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. The `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 listing 5.9:

#### **Listing 5.9 An @Async method is 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
}
```

#A Being @Async, the method is executed on a separate thread

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

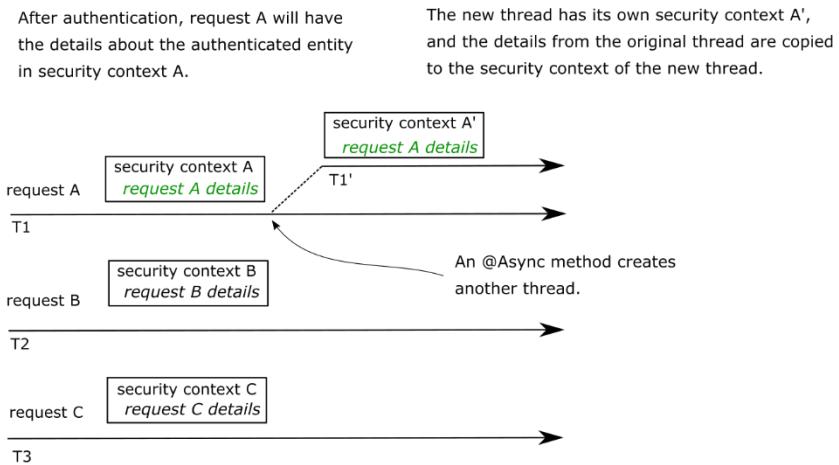
```
@Configuration
@EnableAsync
public class ProjectConfig {
```

**NOTE** Sometimes, in articles or forums, you will 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, as 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 also as a configuration class. For the examples of this book, to make things as clear as possible, I prefer to keep these annotations over the `@Configuration` class similarly to how you'll find them in practical scenarios.

If you try the code as it is now, you will get a `NullPointerException` on the line that gets the name from the authentication 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 using the "spring.security.strategy" system property. By setting this strategy, the framework will know to copy the details of the original thread of the request, to the newly created thread of the asynchronous method (figure 5.8).



**Figure 5.8** 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.

Listing 5.10, presents a way to set it calling the `setStrategyName()` method.

**Listing 5.10 Using an InitializingBean to set the mode of the SecurityContextHolder**

```
@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, and the `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 the framework would not know in this case about the thread that your code created. We discuss how to solve the issues of these cases in sections 5.2.4 and 5.2.5.

### 5.2.3 Using a holding strategy for standalone applications

If what you need is a security context shared by all the threads of the application (figure 5.9), you would change the strategy to `MODE_GLOBAL`. You would not desire 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 it could be a good use for a standalone application.

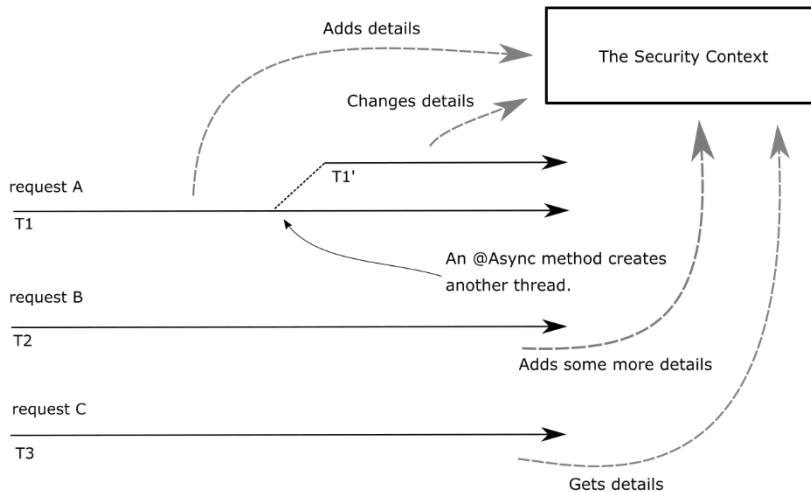


Figure 5.9 With `MODE_GLOBAL` used as the security context management strategy, all the threads access the same security context. This implies that they all have access to the same data, and they can change that

information. Because of this, race conditions can occur, and you have to take care of the synchronization.

You can change the strategy in the same way we've done with the `MODE_INHERITABLETHREADLOCAL`, by using the `SecurityContextHolder.setStrategyName()` method or the system property "spring.security.strategy".

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

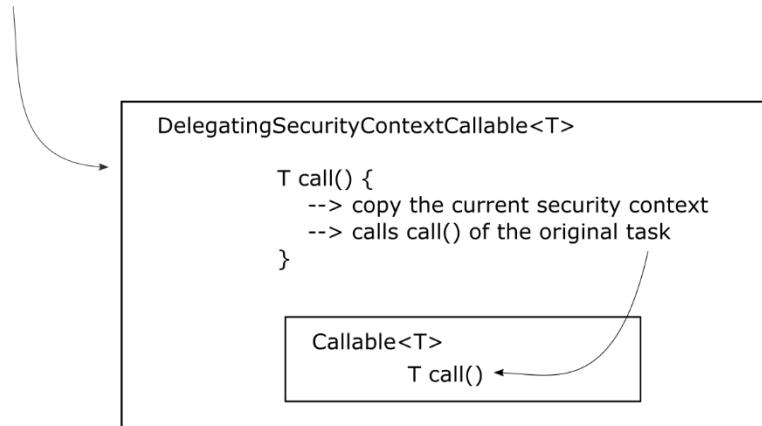
Be also aware that the `SecurityContext` is not thread-safe. So with this strategy where all the threads of the application can access the `SecurityContext` object, you will have to take care also of the concurrent access.

#### 5.2.4 Forwarding the security context with a `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 have 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 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 the newly created threads in this situation.

A specific strategy of the `SecurityContextHolder` doesn't offer you a solution to the self-managed threads. In this case, you should 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 when there is no value expected following the execution of the task. If there is any, then you can use the `Callable<T>` alternative, which is the `DelegatingSecurityContextCallable<T>`. Both classes represent tasks executed asynchronously as any other `Runnable` or `Callable`. Moreover, they make sure to copy the current security context for the thread that executes the task. As figure 5.10 presents, these objects decorate the original tasks and make sure to copy the security context to the new thread.

The DelegatingSecurityContextCallable decorates the Callable task that will be executed on a separate thread.



**Figure 5.10** The `DelegatingSecurityContextCallable` is designed as a decorator of the `Callable` object. When building such an object, you will provide the callable task that the application executes asynchronously. The `DelegatingSecurityContextCallable` makes sure to copy the details from the security context to the new thread and then execute the task.

Listing 5.11 presents the use of a `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 5.11 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();
    };
    ...
}
    
```

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

#### **Listing 5.12 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();
}
}

```

But if you run the application as is, you will 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 the problem, we decorate the task in a `DelegatingSecurityContextCallable`, which will provide the current context to the new thread.

#### **Listing 5.13 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<>(task);
        return "Ciao, " + e.submit(contextTask).get() + "!";
    } finally {
        e.shutdown();
    }
}

```

Calling now the endpoint, you will 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:

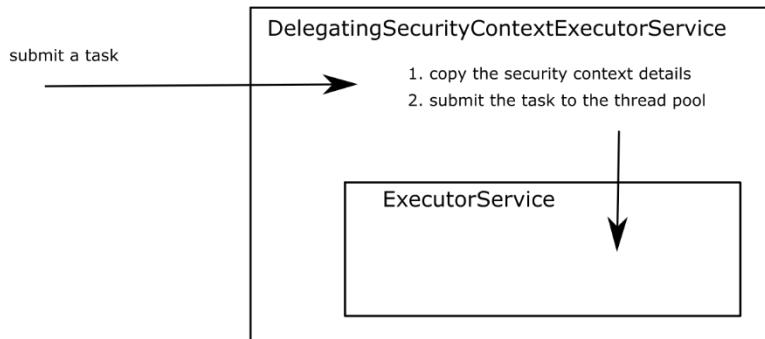
```
Ciao, user!
```

#### **5.2.5 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 the propagation of the details from the security context to the next thread. In section 5.2.4, you have 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 the `DelegatingSecurityContextRunnable` and `DelegatingSecurityContextCallable`, which decorate the tasks you execute asynchronously and also take the responsibility to copy the details from the security context such that your implementation can access them from the

newly created thread. But we have a second fashion to deal with the security context propagation to a new thread. And this is, to manage the propagation from the thread pool instead of from the task itself. In this section, you learn how to apply this second technique by using some great utility classes provided by Spring Security.

An alternative to decorating the tasks is to use a particular type of `Executor`. In the next example, you 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 5.11.



**Figure 5.11** The `DelegatingSecurityContextExecutorService` decorates an `ExecutorService` and makes sure to propagate the security context details to the next thread before submitting the task.

Code in listing 5.14 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 5.14 Using a `DelegatingSecurityContextExecutorService` to propagate 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://localhost:8080/hola
```

The response body for this call is:

```
Hola, user!
```

Out of the classes that are related to the concurrency support for the security context, I recommend you be aware of the ones presented in table 5.1. Spring offers various implementations of the utility classes that you could use in your application to manage the security context when creating your own threads. In section 5.2.4, you implemented in the examples the `DelegatingSecurityContextCallable`. In this section, we used a `DelegatingSecurityContextExecutorService`. If you need to implement security context propagation for the case of a scheduled task, then you will be happy to hear that Spring Security also offers you a decorator named `DelegatingSecurityContextScheduledExecutorService`. The 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. In addition, for more flexibility, Spring Security offers you a more abstract version of a decorator called `DelegatingSecurityContextExecutor`. `DelegatingSecurityContextExecutor` directly decorates an `Executor`, which is the most abstract contract of this hierarchy of thread pools. You will choose it for the design of your application whenever you want to be able to replace the implementation of the thread pool with any of the choices the language provides you with.

**Table 5.1 Main objects responsible for delegating the security context to a separate thread.**

Class	Description
<code>DelegatingSecurityContextExecutor</code>	Implements the <code>Executor</code> interface and is designed to decorate an <code>Executor</code> object with the capability of forwarding a security context to the threads created by its pool.
<code>DelegatingSecurityContextExecutorService</code>	Implements the <code>ExecutorService</code> interface and is designed to decorate an <code>ExecutorService</code> object with the capability

	of forwarding a security context to the threads created by its pool.
DelegatingSecurityContextScheduledExecutorService	Implements the ScheduledExecutorService interface and is designed to decorate a ScheduledExecutorService object with the capability of forwarding a security context to the threads created by its pool.
DelegatingSecurityContextRunnable	Implements the Runnable interface and represents a task that is executed on a different thread without returning a response. Above a normal Runnable, it is also able to propagate a security context to be used on the new thread.
DelegatingSecurityContextCallable	Implements the Callable interface and represents a task that is executed on a different thread, and that will eventually return a response. Above a normal Callable, it is also able to propagate a security context to be used on the new thread.

## 5.3 Understanding HTTP Basic and Form Login authentication methods

Up to now, we have 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-concepts. But for the same reason, it might not fit in all of the real-world scenarios that you'll need to implement. In this section, you will learn more configurations related to HTTP Basic. As well, we'll discover a new authentication method called the Form Login. Within the rest of this book, we'll discuss other methods for authentication, which match well with different kinds of architectures. We'll as well compare them such that you understand the best practices as well as the anti-patterns.

### 5.3.1 Using and configuring HTTP Basic

You are already aware that HTTP Basic is the default authentication method, and we have already observed the way it works with 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 the aspects. 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 apply these cases with practical examples to understand how you could implement them. Let's point out again how you can set this method explicitly (listing 5.15). You can find this example in project `ssia-ch5-ex3`.

#### **Listing 5.15 Setting the HTTP Basic authentication method**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.httpBasic();
    }
}
```

You can also 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. One of the configurations is setting the realm name. You could think about the realm as the protection space that uses a specific authentication method. For a complete description, refer to RFC 2617 <https://tools.ietf.org/html/rfc2617>.

#### **Listing 5.16 Configuring the realm name for the response of the failed authentication**

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic(c -> {
        c.realmName("OTHER");
    });

    http.authorizeRequests().anyRequest().authenticated();
}
```

Listing 5.16 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 change the name of the realm. You can use `curl` with the `-v` flag to get a verbose HTTP response in which the realm name was 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.

```
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 will need to do this if the client of your system expects something specific in the response for the case of a failed authentication. Maybe you'll need to add or remove one or more headers. Or you could 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 Open Web Application Security Project [OWASP] top ten vulnerabilities) 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 method `commence()` receives the `HttpServletRequest`, the `HttpServletResponse`, and the `AuthenticationException` that caused the authentication to fail. Listing 5.17 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 upon authentication failure. In the Spring Security architecture, this is used directly by a component called `ExceptionTranslationManager`. The `ExceptionTranslationManager` handles any `AccessDeniedException` and `AuthenticationException` thrown within the filter chain. You can view the `ExceptionTranslationManager` as a bridge between the Java exceptions and the HTTP responses.

#### Listing 5.17 Implementing an `AuthenticationEntryPoint` to alter the HTTP response

```
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` to the HTTP Basic method in the configuration class, as presented by listing 5.18.

#### Listing 5.18 Setting the custom `AuthenticationEntryPoint` in the configuration class

```

@Override
protected void configure(HttpSecurity http)
    throws Exception {
    http.httpBasic(c -> {
        c.realmName("OTHER");
        c.authenticationEntryPoint(new CustomEntryPoint());
    });
    http.authorizeRequests()
        .anyRequest()
        .authenticated();
}

```

If you make now 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:

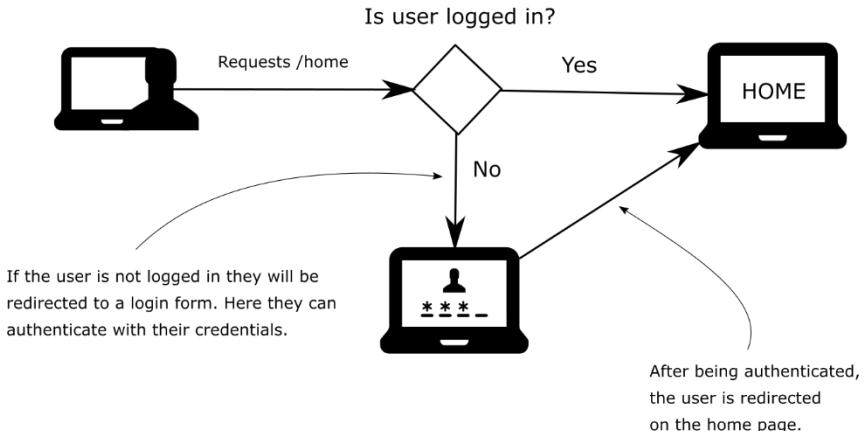
```

...
< HTTP/1.1 401
< Set-Cookie: JSESSIONID=459BAFA7E0E6246A463AD19B07569C7B; Path=/; HttpOnly
< message: Luke, I am your father!
...

```

### 5.3.2 Implementing the authentication with the Form Login method

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 would like your user to be able to surf authenticated through the web pages after they logged in and be able to log out. For a small web application, you can take advantage of the form login method. In this section, you'll learn to apply and configure the Form Login authentication method for your application. To achieve this, we'll write a small web application that uses the form login. Figure 5.12 describes the flow we'll implement. The examples in this section are part of the project `ssia-ch5-ex4`.



**Figure 5.12** Using the Form Login authentication method. 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 home page of the application.

**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 not great. We will discuss in more detail these aspects in chapters 12 to 15 when dealing with OAuth 2.

To change the authentication method to form login, in the `configure(HttpSecurity http)` method of the configuration class, instead of `httpBasic()`, call the `formLogin()` method of the `HttpSecurity` parameter. Listing 5.19 presents this change.

#### Listing 5.19 Changing the authentication method to form login

```

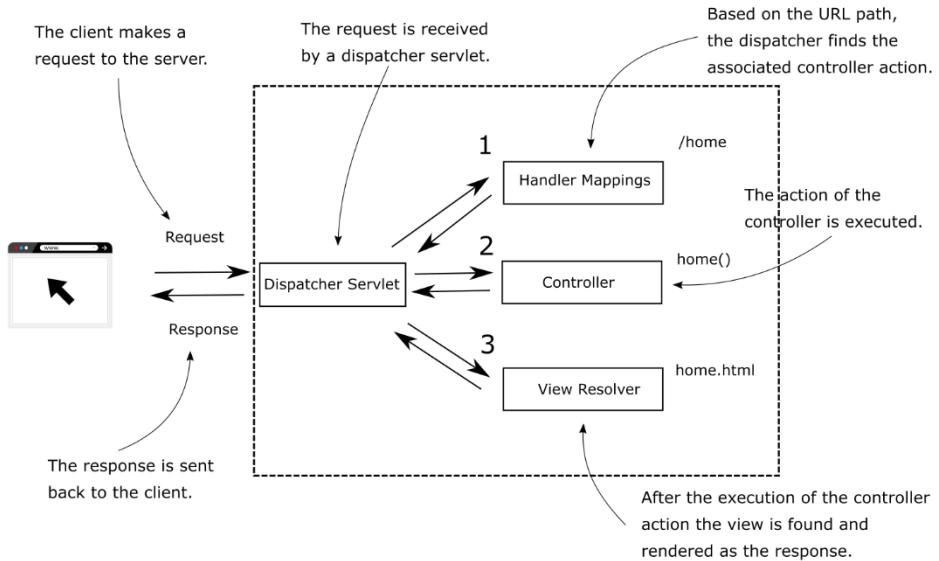
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.formLogin();
        http.authorizeRequests().anyRequest().authenticated();
    }
}
  
```

Even with this minimal configuration, Spring Security has already configured a login form as well as a logout page for your project. Starting the application and accessing it with the browser should redirect you to a login page.



Figure 5.13 The default login page auto-configured by Spring Security when using the form login method.

You can log in using the default provided credentials as long as you did not register your `UserDetailsService`. These are, as we have learned in chapter 2, username “user” and a UUID password, which is printed in the console when the application starts. After a successful login, because there is no other page defined, you will be redirected to a default error page. The application relies on the same architecture for authentication that we have encountered in previous examples, as well. So like figure 5.14 presents, you will need to implement a controller for the home page of the application. The difference is that instead of having a simple JSON formatted response, we want the endpoint to return HTML that will be interpreted by the browser as our web page. Because of this, we will 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 5.14 presents the Spring MVC flow for rendering the home page of the application.



**Figure 5.14** A simple representation of the Spring MVC flow. The dispatcher finds the controller action associated with the given path /home. 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 will 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 `<h1>Welcome</h1>`.

After creating the HTML page, a controller needs to define the mapping from the path to the view. Listing 5.20 presents the definition of the action method for the `home.html` page in the controller class.

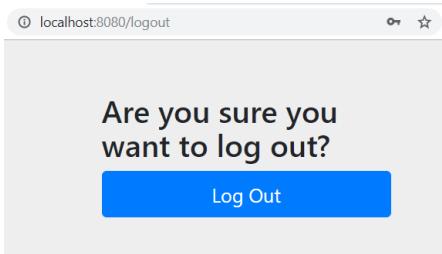
#### **Listing 5.20 Defining the action method of the controller for the `home.html` page**

```
@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 will not send the value returned by the method in the HTTP response. Instead, it will find and render the view with the name `home.html`.

Trying to access the `/home` page will now first ask you to log in. After a successful login, you will be redirected to the home page where the welcome message appears now. You can now access the `/logout` path, and this should redirect you to a logout page.



**Figure 5.15** The logout page configured by Spring Security for the form login authentication method

After an attempt to access a path without being logged in, the application automatically redirects the user in the browser to the login page. After a successful login, the application redirects back the user to the path they tried to 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.

#### **Listing 5.21** Setting a default success URL for the login form

```
@Override
protected void configure(HttpSecurity http)
    throws Exception {
    http.formLogin()
        .defaultSuccessUrl("/home", true);

    http.authorizeRequests()
        .anyRequest().authenticated();
}
```

If you need to go into even more in-depth with this, a more detailed customization approach is offered using the `AuthenticationSuccessHandler` and `AuthenticationFailureHandler` objects. These interfaces allow you to 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 5.22, you find an example of implementing the `onAuthenticationSuccess()` method to make different redirects depending on the granted authorities of the logged-in user.

#### **Listing 5.22** 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();

        Optional<GrantedAuthority> auth =
            authorities.stream()
                .filter(a -> a.getAuthority().equals("read"))
                .findFirst();      #A

        if (auth.isPresent()) {      #B
            httpServletResponse
                .sendRedirect("/home");
        } else {
            httpServletResponse
                .sendRedirect("/error");
        }
    }
}

```

#A Returning an empty Optional object if the “read” authority doesn’t exist.

#B If the “read” authority exists redirecting to /home page

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 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 the one in which 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 for logging the event of further investigation.

If you would like to customize the logic that the application executes when the authentication fails, you can do this similarly with an `AuthenticationFailureHandler` implementation. For example, if you would like to add a specific header for any failed authentication, you could do like in listing 5.23. You could, of course, implement any logic here, as well. For the `AuthenticationFailureHandler`, the `onAuthenticationFailure()` receives the request, response, and the `Authentication` object.

#### **Listing 5.23 Implementing an `AuthenticationFailureHandler`**

```

@Component
public class CustomAuthenticationFailureHandler
    implements AuthenticationFailureHandler {

    @Override
    public void onAuthenticationFailure(
        HttpServletRequest httpServletRequest,

```

```

        HttpServletResponse httpServletResponse,
        AuthenticationException e) {
    httpServletResponse
        .setHeader("failed", LocalDateTime.now().toString());
}
}

```

To use the two objects, you have to register them within the `configure()` method on the `FormLoginConfigurer` object returned by the `formLogin()` method.

#### **Listing 5.24 Registering the handler objects in the configuration class**

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomAuthenticationSuccessHandler authenticationSuccessHandler;

    @Autowired
    private CustomAuthenticationFailureHandler authenticationFailureHandler;

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.formLogin()
            .successHandler(authenticationSuccessHandler)
            .failureHandler(authenticationFailureHandler);

        http.authorizeRequests()
            .anyRequest().authenticated();
    }
}

```

You could choose to use both the HTTP basic and the form login together. For now, if you try to access the `/home` path using HTTP basic with the proper username and password, you would be 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. So even if you have provided the right username and password, it won't consider them and would instead try to send you to the login form as requested by the Form Login method. You could change the configuration to support both methods, like in listing 5.25.

#### **Listing 5.25 Using form login and HTTP basic together**

```

@Override
protected void configure(HttpSecurity http)
    throws Exception {

    http.formLogin()
        .successHandler(authenticationSuccessHandler)
        .failureHandler(authenticationFailureHandler)
        .and()
        .httpBasic();
}

```

```

    http.authorizeRequests()
        .anyRequest().authenticated();
}

```

Accessing `/home` path will now work with both Form Login and HTTP Basic authentication methods.

```
curl -u user:cdd430f6-8ebc-49a6-9769-b0f3ce571d19 http://localhost:8080/home
```

The response of the call is:

```
<h1>Welcome</h1>
```

## 5.4 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. So, for user management, the `AuthenticationProvider` delegates to a `UserDetailsService`, and for the responsibility of the 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_SHAREDTHREADLOCAL`, and `MODE_GLOBAL`. The 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, this mode is applied only 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. You can use
  - `DelegatingSecurityContextRunnable`,
  - `DelegatingSecurityContextCallable`, and
  - `DelegatingSecurityContextExecutor`
 to manage the `SecurityContext` for the threads that you create.
- Form Login is an authentication method where Spring Security autoconfigures a form for login and an option to logout. It is very comfortable to use when developing small web applications.
- The Form Login authentication method is highly customizable. Moreover, you can use this method together with the HTTP Basic method.

# 6

## *Hands-On: A small secured web application*

### This chapter covers

- Applying authentication in a hands-on example.
- Defining the user with the `UserDetails` interface.
- Defining a custom `UserDetailsService`.
- Using a provided implementation of `PasswordEncoder`.
- Defining your authentication logic by implementing an `AuthenticationProvider`.
- Setting the Form Login authentication method.

We've come a long way and have already discussed plenty of details on authentication. And we have applied each of the new details individually. It is time to put what we have learned until now in action together in a more complex example. This hands-on example will help you have a better overview of how all the components we've discussed work together in a real application.

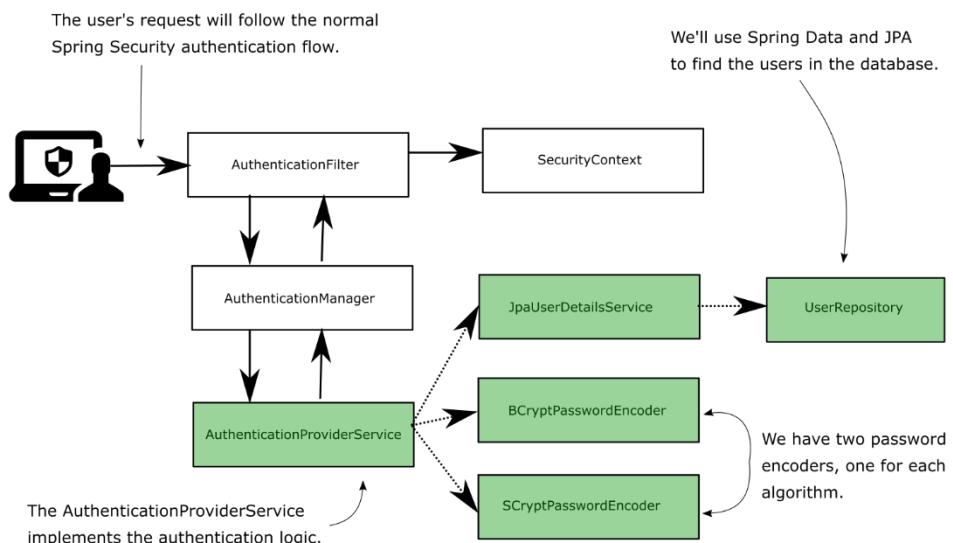
### 6.1 Requirements and setup of the project

In this section, we implement a small web application where the user, after successful authentication, can see a list of products on the main page. You find the complete implementation with the provided projects, in example `ssia-ch6-ex1`. The products, as well as the users, are stored in a database. The passwords for the users will be either hashed with BCrypt or with SCrypt, for each user. I've chosen two hashing algorithms to have a reason in the example to customize the authentication logic. A column in the "users" table will store the encryption type. A third table will store the authorities for the users.

The authentication flow for this application, as we want to implement it is described in figure 6.1. I have shaded differently the components that we'll customize. For the others, we'll use the defaults provided by Spring Security. The request follows the standard authentication flow that we've discussed in chapters 2 to 5. I have represented the request in the diagram with the arrows having a continuous line. The `AuthenticationFilter` intercepts the request and then delegates the authentication responsibility to the `AuthenticationManager`. The `AuthenticationManager` uses an `AuthenticationProvider` to authenticate the request and returns the details of a successfully authenticated call so that the `AuthenticationFilter` can store them in the `SecurityContext`.

What we implement in this example is the `AuthenticationProvider` and everything related to the authentication logic. As presented in figure 6.1, we'll create the `AuthenticationProviderService` class, which implements the `AuthenticationProvider` interface. This implementation defines the authentication logic where it needs to call a `UserDetailsService` to find the user details from a database and the `PasswordEncoder` to validate if a password is correct. For this application, we create a `JpaUserDetailsService` that will use Spring Data JPA to work with the database.

For this reason, it depends on a Spring Data `JpaRepository`, which, in our case, I've named `UserRepository`. We need two password encoders as the application scenario validates passwords hashed with BCrypt as well as passwords hashed with SCrypt. Being a simple web application, it needs a standard login form to allow the user to authenticate. For this, we'll configure Form Login as the authentication method in this application.



**Figure 6.1. The authentication flow in the hands-on application. The custom authentication provider will implement the authentication logic. For this, the authentication provider uses a `UserDetailsService` implementation and two `PasswordEncoder` implementations, one for each requested hashing algorithm. The**

UserDetailsService implementation, called JpaUserDetailsService uses Spring Data and JPA to work with the database and obtain the UserDetails.

**NOTE** In some of the examples in the book, I'll use Spring Data JPA. This approach brings you closer to the applications you'll find when working with Spring Security. You don't need to be an expert in JPA to understand the examples. From Spring Data and JPA point of view, I'll limit the use cases to simple syntaxes and focus on Spring Security. However, if you want to learn more on JPA and JPA implementations like Hibernate, I strongly recommend you to read Java Persistence with Hibernate, Second Edition, written by Christian Bauer, Gavin King and Gary Gregory (Manning 2015). For a great discussion on Spring Data, you can read Craig Walls's Spring in Action, Sixth Edition (Manning 2018).

The application will also have a main page that the user can access after a successful login. This page displays details about products stored in the database. In figure 6.2, I have shaded the components that we'll create differently. We'll need a MainPageController that defines the action which the application will execute upon the request for the main page. The MainPageController displays the name of the user on the main page, so this is why it depends on the SecurityContext. It will obtain the username from the security context and the list of products to display from a service that I have called ProductService. The ProductService will get the list of products from the database using a ProductRepository, which is a standard Spring Data JPA repository.

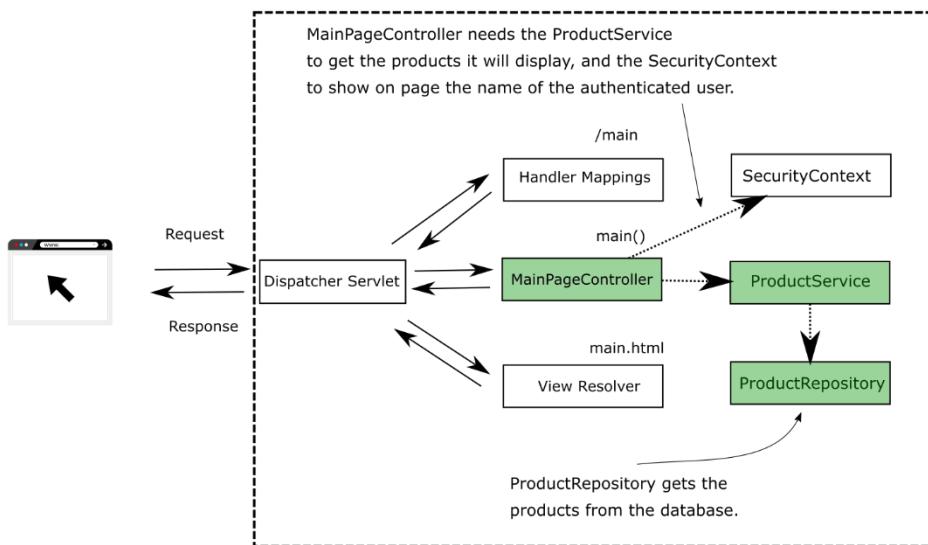
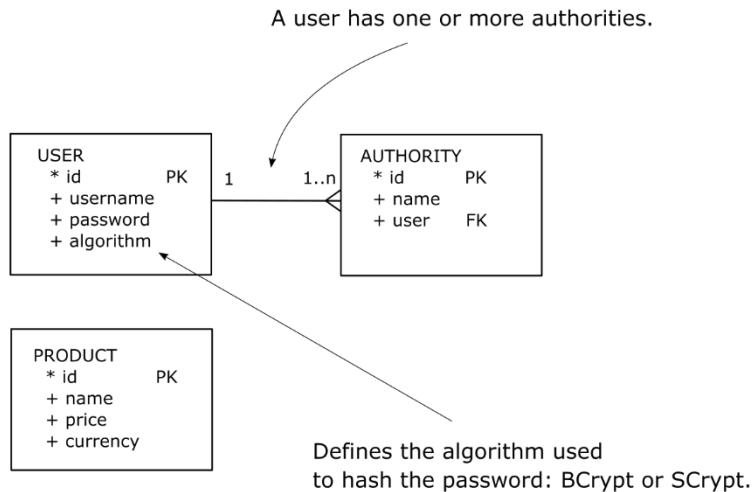


Figure 6.2 The MainPageController will serve the requests for the main page of the application. To display the products from the database, it will use a ProductService, which further obtains the products through a JpaRepository named ProductRepository. The MainPageController also takes the name of the authenticated user from the SecurityContext.

The database contains three tables: user, authority, and product. Figure 6.3 presents the entity relationship diagram (ERD).



**Figure 6.3 The Entity Relationship Diagram (ERD) of the database for the current example.** The user table stores the username, password, and the algorithm used to hash the password. Also, an user has one or more authorities. The authority table stores the users' authorities. A third table, named product, stores the details of the product records: a name, a price, and a currency. The main page displayed the details of all the products stored in this table.

The main steps we take to implement this project are:

1. Set up the database
2. Define user management
3. Implement the authentication logic
4. Implement the main page
5. Run and test the application

Let's get started now with the implementation. We first have to create the tables. The name of the database I use is "spring". You should first create the database either by using the command-line tool or a client. If you are using MySQL like in the examples of this book, you could use MySQL Workbench to create the database and eventually to run the scripts. I prefer, however, to let Spring Boot run the scripts that create the database structure and add data to it. To do this, you have to create the `schema.sql` and the `data.sql` files in the resources folder of your project. The `schema.sql` file will contain all the queries that create or alter the structure

of the database, while in the `data.sql` you put all the queries that work with data. The next code snippets define the three tables used by the application.

The fields of the “user” table are:

- **id** - primary key of the table, defined as auto-increment
- **username** - to store the username
- **password** - to save the password hash which will be BCrypt or SCrypt
- **algorithm** - will store the values BCRYPT or SCRYPT and decides which is the hashing method of the password for the current record

In listing 6.1, you find the definition of the “user” table. You can choose to run this script manually or add it to the `schema.sql` file to let Spring Boot run it when the project starts.

#### **Listing 6.1 Script for creating the “user” table**

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

The fields of the “authority” table are:

- **id** - primary key of the table, defined as auto-increment
- **name** - the name of the authority
- **user** - the foreign key to the “user” table

In listing 6.2, you find the definition of the “authority” table. You can choose to run this script manually or add it to the `schema.sql` file to let Spring Boot run it when the project starts.

#### **Listing 6.2 Script for creating the “authority” table**

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

The third table is named “product”. It will store the data that will be displayed after the user successfully logs in.

The fields of the “product” table are:

- **id** - primary key of the table, defined as auto-increment
- **name** - a string representing the name of the product
- **price** - a double representing the price of the product
- **currency** - a string representing the currency (E.g., USD, EUR)

In listing 6.3, you find the definition of the “product” table. You can choose to run this script manually or add it to the `schema.sql` file to let Spring Boot run it when the project starts.

#### **Listing 6.3 Script for creating the “product” table**

```
CREATE TABLE IF NOT EXISTS `spring`.`product` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `price` VARCHAR(45) NOT NULL,
  `currency` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`id`));
```

**NOTE** It is more advisable to have a many-to-many relationship between the authorities and the users. To keep the example simpler from the persistence layer point of view and focus on the essential aspects of Spring Security, I have decided to make it one-to-many.

Let's add some data which we can use to test our application. You can run these INSERT queries manually or add them to the `data.sql` file in the resources folder of your project to allow Spring Boot to run them when you start the application.

```
INSERT IGNORE INTO `spring`.`user` (`id`, `username`, `password`, `algorithm`) VALUES ('1',
  'john', '$2a$10$xn3LI/AjqicFYZFruSwve.681477XaVNaUQbr1gioaWPn4tKsnmG', 'BCRYPT');

INSERT IGNORE INTO `spring`.`authority` (`id`, `name`, `user`) VALUES ('1', 'READ', '1');
INSERT IGNORE INTO `spring`.`authority` (`id`, `name`, `user`) VALUES ('2', 'WRITE', '1');
```

For user "john", the password is hashed using BCrypt. The raw password is "12345".

```
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `price`, `currency`) VALUES ('1',
  'Chocolate', '10', 'USD');
```

**NOTE** It is common to use `schema.sql` and `data.sql` files in examples. In a real application, you would rather choose a solution that allows you also to version the SQL scripts. You'll find this very often done using a dependency like Flyway (more details here <https://flywaydb.org/>) or Liquibase (more details here <https://www.liquibase.org/>).

Now that we have a database and some test data, let's start with the implementation. We create a new project, and the dependencies to add are the following (listing 6.4):

- **spring-boot-starter-data-jpa**, used to connect to the database using Spring Data
- **spring-boot-starter-security**, the Spring Security dependencies
- **spring-boot-starter-thymeleaf**, adds Thymeleaf as a template engine to simplify the definition of the web page
- **spring-boot-starter-web**, the standard web dependencies
- **mysql-connector-java**, the MySQL JDBC driver

#### Listing 6.4 Dependencies needed for the development of the example project

```
<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-thymeleaf</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>

```

The `application.properties` file will have to declare the database connectivity parameters.

```

spring.datasource.url=jdbc:mysql://localhost/spring?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=<your_username>
spring.datasource.password=<your_password>
spring.datasource.initialization-mode=always

```

**NOTE** I might repeat myself a lot saying this here and there, but make sure you never expose passwords. In examples is fine, but in a real-world scenario you should never write sensitive data as credentials or private keys in the `application.properties` file. Instead, use a secrets vault for this purpose.

## 6.2 Implementing user management

In this section, we discuss implementing the user management part of the application. The representative component of the user management part of the application in regards to Spring Security is the `UserDetailsService`. You need to implement at least this contract to instruct Spring Security on how to retrieve the details of your users. Now that we have a project in place and the database connection configured, it is time to think about the implementations related to application security. The steps we need to take to build the part of the application taking care of the user management are:

1. Define the password encoder objects for the two hashing algorithms.
2. Define the JPA entities to represent the tables storing details needed in the authentication process: `user` and `authority`.
3. Declare the `JpaRepository` contracts for Spring Data. In this example, we only need to refer directly to the users, so we will only need to declare a repository which I'll name  `UserRepository`.
4. Create a decorator that implements the `UserDetailsService` contract over the `User` JPA entity. We use here the approach to separate responsibilities discussed in section 3.2.5.
5. Implement the `UserDetailsService` contract. I'll create a class named `JpaUserDetailsService`, which implements the `UserDetailsService` interface. The `JpaUserDetailsService` uses the  `UserRepository` we have created at step 3, to

obtain the details about the users from the database. If the `JpaUserDetailsService` finds the user, it returns it as an implementation of the decorator we have defined at step 4.

We first consider the users and passwords management. We know from the requirements of the example that the algorithms that our app will use to hash the passwords are BCrypt and SCrypt. We can start by creating a configuration class and declare these two password encoders as beans. You find this in listing 6.5.

#### **Listing 6.5 Registering a bean for each PasswordEncoder**

```
@Configuration
public class ProjectConfig {

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SCryptPasswordEncoder sCryptPasswordEncoder() {
        return new SCryptPasswordEncoder();
    }

}
```

For user management, we need to declare a `UserDetailsService` implementation. The `UserDetailsService` implementation will retrieve the user by its name from the database. It will have to return the user as an implementation of the `UserDetails` interface.

We need to implement two JPA entities for the authentication: the `User` and the `Authority`.

Listing 6.6 shows how to define the `User`. It has a one-to-many relationship with the `Authority` entity.

#### **Listing 6.6 The User entity class**

```
@Entity
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String username;
    private String password;

    @Enumerated(EnumType.STRING)
    private EncryptionAlgorithm algorithm;

    @OneToMany(mappedBy = "user", fetch = FetchType.EAGER)
    private List<Authority> authorities;

    // Omitted getters and setters
}
```

The `EncryptionAlgorithm` is an enum defining the two supported hashing algorithms as specified in the request.

```
public enum EncryptionAlgorithm {
    BCRYPT, SCRYPT
}
```

Listing 6.7 shows how to implement the `Authority` entity.

#### **Listing 6.7 The Authority entity class**

```
@Entity
public class Authority {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @JoinColumn(name = "user")
    @ManyToOne
    private User user;

    // Omitted getters and setters
}
```

A repository must be declared to retrieve the users by their names from the database.

#### **Listing 6.8 The definition of the Spring Data repository for the User entity**

```
public interface UserRepository extends JpaRepository<User, Integer> {

    Optional<User> findUserByUsername(String username);      #A
}
```

#A It is not mandatory to write the query. Spring Data translates the name of the method in the needed query.

I use here a Spring Data JPA repository. The method declared in the interface will be implemented by Spring Data, and it will execute a query based on its name. In our case, the method returns an `Optional` instance containing the `User` entity with the name provided as a parameter. If no such user exists in the database, the method will return an empty `Optional`.

To return the user from a `UserDetailsService`, we need to represent it as a `UserDetails`. In listing 6.9, the class `CustomUserDetails` implements the `UserDetails` interface and wraps the `User` entity.

#### **Listing 6.9 The implementation of the UserDetails contract**

```
public class CustomUserDetails implements UserDetails {

    private final User user;

    public CustomUserDetails(User user) {
        this.user = user;
    }
}
```

```
// Omitted code

public final User getUser() {
    return user;
}
```

The `CustomUserDetails` class implements the methods of the `UserDetails` interface.

#### **Listing 6.10 Implementing the rest of the methods of the `UserDetails` interface**

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    return user.getAuthorities().stream()
        .map(a -> new SimpleGrantedAuthority(a.getName()))      #A
        .collect(Collectors.toList());    #B
}

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

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

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

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

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

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

#A Each authority name found in the database for the user is mapped to a `SimpleGrantedAuthority`.

#B All the instances of `SimpleGrantedAuthority` are collected in a list and returned.

**NOTE** In listing 6.10, I use `SimpleGrantedAuthority`, which is a straightforward implementation of the `GrantedAuthority` interface. Spring Security provides this implementation.

You can now implement the `UserDetailsService` to look like in listing 6.11. If the application finds the user by its username, the instance of type `User` is wrapped in a

`CustomUserDetails` instance and returned. The service should throw an exception of type `UsernameNotFoundException` if the user doesn't exist.

#### **Listing 6.11 The implementation of the `UserDetailsService` contract**

```
@Service
public class JpaUserDetailsService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public CustomUserDetails loadUserByUsername(String username) {
        Supplier<UsernameNotFoundException> s = #A
            () -> new UsernameNotFoundException(
                "Problem during authentication!");

        User u = userRepository
            .findUserByUsername(username)      #B
            .orElseThrow(s);      #C

        return new CustomUserDetails(u);      #D
    }
}
```

#A We declare a supplier which creates exception instances.

#B This method returns an `Optional` instance containing the user or an empty `Optional` if the user does not exist.

#C If the `Optional` is empty, we throw an exception created by the above-defined `Supplier`. Else, it returns the `User` instance.

#D Finally, the method wraps the `User` instance with the `CustomUserDetails` decorator and returns it.

### **6.3 Implementing the custom authentication logic**

Having the users and passwords management, we can begin writing the custom authentication logic. To do this, we have to implement an `AuthenticationProvider` (listing 6.12) and register it in the Spring Security authentication architecture. The dependencies needed for writing the authentication logic are the `UserDetailsService` implementation and the two password encoders. Beside auto-wiring them, we also override the `authenticate()` and `supports()` methods. We implement the `supports()` to specify that the `Authentication` implementation type supported is `UsernamePasswordAuthenticationToken`.

#### **Listing 6.12 Implementation of the `AuthenticationProvider`**

```
@Service
public class AuthenticationProviderService
    implements AuthenticationProvider {

    @Autowired #A
    private JpaUserDetailsService userDetailsService;

    @Autowired
    private BCryptPasswordEncoder bCryptPasswordEncoder;

    @Autowired
```

```

private SCryptPasswordEncoder sCryptPasswordEncoder;

@Override
public Authentication authenticate(
    Authentication authentication)
    throws AuthenticationException {
    // ...
}

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

```

#A We inject the needed dependencies, which are the `UserDetailsService` and the two `PasswordEncoder` implementations.

The `authenticate()` method first loads the user by its username and then verifies if the password given matches the hash stored in the database. The verification depends on the algorithm used to hash the user's password.

#### **Listing 6.13 Defining the authentication logic by overriding the `authenticate()` method**

```

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

    String username = authentication.getName();
    String password = authentication
        .getCredentials()
        .toString();

    CustomUserDetails user =      #A
        userDetailsService.loadUserByUsername(username);

    switch (user.getUser().getAlgorithm()) {      #B
        case BCRYPT:      #C
            return checkPassword(user, password, bCryptPasswordEncoder);
        case SCRYPT:      #D
            return checkPassword(user, password, sCryptPasswordEncoder);
    }

    throw new BadCredentialsException("Bad credentials");
}

```

#A With the `UserDetailsService`, we find the user details from the database.

#B We validate the password depending on the hashing algorithm specific to the user.

#C If the password of the user is hashed using BCrypt, we use the `BCryptPasswordEncoder`.

#D Otherwise, we use the `SCryptPasswordEncoder`.

We choose the `PasswordEncoder`, which we use to validate the password based on the value of the "algorithm" attribute of the user. In listing 6.14, you find the definition of the `checkPassword()` method. This method uses the password encoder to validate that the raw password received from the user input matches the encoding in the database. The method uses the password encoder sent as a parameter to verify the password with the hash in the

database. If the password is valid, it returns an instance of an implementation of the Authentication contract. The UsernamePasswordAuthenticationToken class is an implementation of the Authentication interface. The constructor that I have called in listing 6.14 also sets the value “authenticated” to true. This detail is important because you know that the authenticate() method of the AuthenticationProvider has to return an authenticated instance.

#### **Listing 6.14 The checkPassword() method used in the authentication logic**

```
private Authentication checkPassword(CustomUserDetails user,
                                    String rawPassword,
                                    PasswordEncoder encoder) {

    if (encoder.matches(rawPassword, user.getPassword())) {
        return new UsernamePasswordAuthenticationToken(
            user.getUsername(),
            user.getPassword(),
            user.getAuthorities());
    } else {
        throw new BadCredentialsException("Bad credentials");
    }
}
```

We need to register the AuthenticationProvider within the configuration class, as shown in listing 6.15.

#### **Listing 6.15 Registering the AuthenticationProvider in the configuration class**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired    #A
    private AuthenticationService authenticationProvider;

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SCryptPasswordEncoder sCryptPasswordEncoder() {
        return new SCryptPasswordEncoder();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) {
        auth.authenticationProvider(authenticationProvider);    #B
    }
}
```

#A We get the instance of AuthenticationService from the context.

#B By overriding the configure() method, we register the authentication provider for Spring Security.

Besides this, in the configuration class, we set the authentication method to Form Login and set as the default success URL the path `/main` (listing 6.16). We intend to implement this path to be the main page of the web application.

#### **Listing 6.16 Configuring Form Login as the authentication method**

```
@Override
protected void configure(HttpSecurity http)
    throws Exception {
    http.formLogin()
        .defaultSuccessUrl("/main", true);

    http.authorizeRequests()
        .anyRequest().authenticated();
}
```

## **6.4 Implementing the main page**

Finally, as we have the security part in place, we can implement the main page of the app. It is a simple page that displays all the records of the product table. The page is accessible only after the user logs in.

To get the product records from the database, we have to add a `Product` entity class and a `ProductRepository` interface. The `Product` class is defined as shown in listing 6.17.

#### **Listing 6.17 Definining the Product JPA entity**

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private double price;

    @Enumerated(EnumType.STRING)
    private Currency currency;

    // Omitted code
}
```

The `Currency` enumeration declares the types allowed as currencies in the application.

```
public enum Currency {
    USD, GBP, EUR
}
```

The `ProductRepository` interface only has to inherit from `JpaRepository`. The application scenario only asks to display all the products. For this, we only need to use the `findAll()` method, which we inherit from the `JpaRepository` interface.

#### **Listing 6.18 Definition of the ProductRepository interface**

```

public interface ProductRepository
    extends JpaRepository<Product, Integer> {
} #A

#A The interface doesn't need to declare any method. We only use methods inherited from the JpaRepository
interface implemented by Spring Data.

The ProductService class uses the ProductRepository to retrieve all the products from
the database.

```

#### **Listing 6.19 Implementation of the ProductService class**

```

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public List<Product> findAll() {
        return productRepository.findAll();
    }
}

```

In the end, a MainPageController will define the path for the page and fill the Model with what the page will display.

#### **Listing 6.20 The definition of the controller class**

```

@Controller
public class MainPageController {

    @Autowired
    private ProductService productService;

    @GetMapping("/main")
    public String main(Authentication a, Model model) {
        model.addAttribute("username", a.getName());
        model.addAttribute("products", productService.findAll());
        return "main.html";
    }
}

```

The main.html page is stored in the resources/templates folder and displays the products and the name of the logged-in user.

#### **Listing 6.21 The definition of the main page**

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">      #A
    <head>
        <meta charset="UTF-8">
        <title>Products</title>
    </head>
    <body>
        <h2 th:text="'Hello, ' + ${username} + '!' />      #B
        <p><a href="/logout">Sign out here</a></p>
    </body>
</html>

```

```

<h2>These are all the products:</h2>
<table>
  <thead>
    <tr>
      <th> Name </th>
      <th> Price </th>
    </tr>
  </thead>
  <tbody>
    <tr th:if="${products.empty}">      #C
      <td colspan="2"> No Products Available </td>
    </tr>
    <tr th:each="book : ${products}">    #D
      <td><span th:text="${book.name}"> Name </span></td>
      <td><span th:text="${book.price}"> Price </span></td>
    </tr>
  </tbody>
</table>
</body>
</html>

```

#A The prefix th is declared so that we can use the Thymeleaf components in the page

#B The message is displayed on the page. \${username} is the variable which will be injected from the model after the execution of the controller action

#C If there are no products in the list from the model a message is displayed

#D For each product found in the list from the model a row in the table is created

## 6.5 Running and testing the application

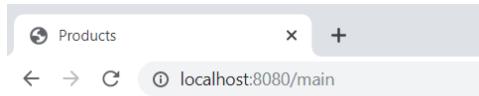
We have finished writing the code for the first Hands-On project of the book. It is time to verify that it is working according to the specifications. So let's run the application and try to log in. After running the application, we can access it in the browser by typing the address <http://localhost:8080>. The standard login form appears as presented in figure 6.4. The user I have stored in the database (and the one in the script given at the beginning of this chapter) is "john" with the password "12345" hashed using BCrypt. We use these credentials to log in.

**NOTE** In a real-world application, you should never allow your users to define simple passwords like "12345".  
Passwords so simple are easy to guess, and they represent a security risk.

The screenshot shows a web browser window with a single tab open. The URL in the address bar is 'localhost:8080/login'. The main content of the page is a login form with a light gray background. At the top, it says 'Please sign in'. Below that is a text input field containing the word 'john'. Below that is a password input field containing five asterisks ('\*\*\*\*\*'). At the bottom of the form is a large blue rectangular button with the white text 'Sign in'.

Figure 6.4 The login form of the application.

Once logged in, the application redirects you to the main page (figure 6.5). Here, the username taken from the security context appears on the page, together with the list of the products from the database.



## Hello, john!

[Sign out here](#)

### These are all the products:

Name	Price
Chocolate	10.0

Figure 6.5 The main page of the application

When you click the “Sign out here” link, the application redirects you to the standard sign out confirmation page (figure 6.6) as it was auto defined by Spring Security because we are using the Form Login authentication method.

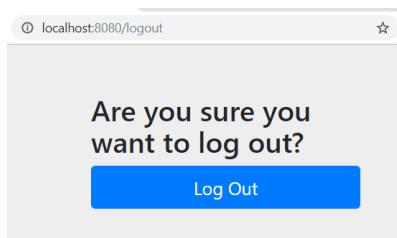


Figure 6.6 The standard log out confirmation page

When clicking on the “Log Out” button, you are redirected back to the login page (figure 6.7).

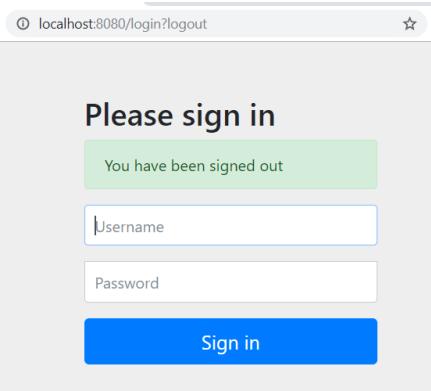


Figure 6.7 The login page appears after logging out from the application.

Congratulations! You've just implemented the first hands-on example and managed to put together some of the essential things already discussed in this book. With this example, you managed to develop a small web application that has the authentication managed with Spring Security. You used the Form Login authentication method, and we have stored the user details in the database. You have also implemented custom authentication logic.

Before closing this chapter, I'd like to make one more observation. Like any other software requirement, you can implement the same application in different ways. I have chosen this implementation to touch as many of the things we have earlier discussed as possible. Mainly, I wanted to have a reason to implement a custom `AuthenticationProvider`. I will leave you as an exercise to simplify the implementation by using a `DelegatingPasswordEncoder`, as discussed in chapter 4.

## 6.6 Summary

- It is common in a real application to have dependencies that require a different implementation of the same concept in your application. In our case, the `UserDetails` of Spring Security and the `User` entity of the JPA implementation. A good recommendation is to decouple the responsibilities in different classes to enhance the readability.
- In most cases, in practice, you have multiple ways to implement the same functionality. You should generally choose the most simple of solutions. Making your code easier to understand leaves less room for errors and, thus, security breaches.

## 7

# *Configuring 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 to get into the cabin to go at 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. "The 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.

In most of the cases in life, it doesn't only matter who you are. We can say the same about software applications. It doesn't only matter who you are when trying to access a specific functionality or data.

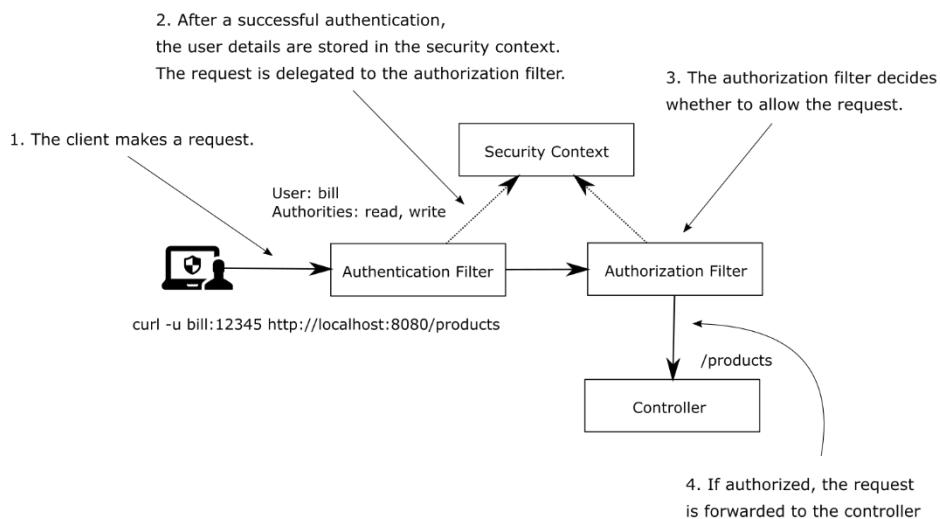
Up to now, we only discussed authentication, which is, as you learned, the process in which the application identifies the caller of a resource. In the examples we've worked on in the previous chapters, we didn't implement any rule to decide whether to approve the request. We only cared if the system knows them or not. In most applications, it doesn't happen that all the users identified by the system can access every resource of the system.

In this chapter, we discuss authorization. Authorization is the process during which the system decides if an identified client has permission to access the requested resource.



**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 take the authorization decision, the authorization filter uses the details from the security context.

So, to cover all the essential details on this aspect, in this chapter, we will follow the next steps:

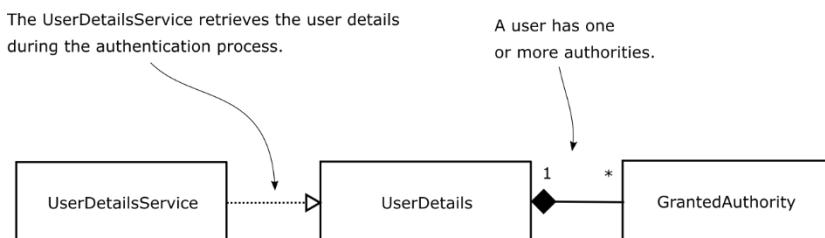
1. You'll start by understanding what an "authority" is, and you'll learn to apply access rules on all the endpoints based on the users' authorities.
2. We discuss how to group authorities in roles and how to apply authorization rules based on roles.

In chapter 8, we continue with selecting the endpoints to which we apply the authorization rules.

## 7.1 Restricting access based on authorities and roles

In this section, you'll understand the concepts of authorization and role, and you will apply them to secure all the endpoints of the application. You'll need to learn this for applying it in real-world scenarios where different users have different permissions. Based on what privileges the users have, they will be allowed to execute a specific action. And the application provides these 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 now return to `GrantedAuthority` to also examine its purpose. Figure 7.3 presents the relationship between the `UserDetails` contract and `GrantedAuthority`. Once we finish discussing this contract, in the next section, you will learn how to use these rules individually or per specific requests.



**Figure 7.3** A user has one or more authorities (actions a user can do). The `UserDetailsService` obtains all the details about the user, including the authorities, during the authentication process. The application uses the authorities, represented by the `GrantedAuthority` interface, for authorization after it successfully authenticates the user.

Listing 7.1 shows you the definition of the `GrantedAuthority` contract. The authority is an action that a user can do with a resource of the system. An authority has a name that the

`getAuthority()` behavior of the object returns as a `String`. We'll use the name of the authority when defining the custom authorization rule. Often an authorization rule can look like this: "Jane is allowed to *delete* the product records" or "John is allowed to *read* the document records". In these cases, "delete" and "read" are the granted authorities. The application allows the users "Jane" and "John" to perform these actions. You'll often encounter these actions having 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. This way, a user may be allowed 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 will implement this method such that it returns all the authorities granted for the user. After the authentication ends, the authorities are part of the details about the user that logged in, which the application can use to grant permissions.

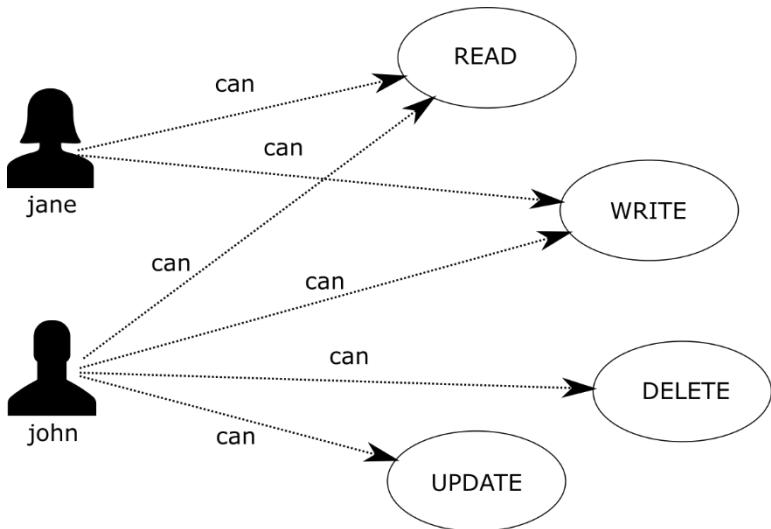
#### **Listing 7.2 The `getAuthorities()` method from the `UserDetails` contract**

```
public interface UserDetails extends Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();

    // Omitted code
}
```

### **7.1.1 Restricting access for all the endpoints based on the user authorities**

In this section, we discuss limiting access to endpoints for specific users. If, up to now in our examples, any authenticated user could have called any endpoint of the application, from now on, you'll learn to customize this access. In 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. We will write several examples to prove the various ways in which you can apply these restrictions with Spring Security.



**Figure 7.4 Authorities are actions that users can do in the application. Based on these actions, you will implement the authorization rules. For example, only users having specific authorities can make a particular request to an endpoint.**

Now that you've remembered 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'll learn a few alternatives to configure the access to endpoints based on the user authorities. We start a new project that I will name `ssia-ch7-ex1`.

I'll show you three ways in which you can configure access to the endpoint based on the authorities of the user:

- The `hasAuthority()` method: It receives as parameters only one authority for which the application configures the restrictions. Only users having that authority can call the endpoint.
- The `hasAnyAuthority()` method: It could receive more than one authority for which the application configures the restrictions. I usually like to remember this method as "has any of the given authorities". The user should have at least one of the specified authorities to make the request. I recommend using this method or the `hasAuthority()`, depending on the number of privileges you assign for their simplicity. They are easy to read in configurations and make your code easier to understand.
- The `access()` method: The application configures the authorization rules based on a Spring Expression Language (SpEL). Because it uses SpEL, the `access()` method offers you unlimited possibilities for configuring the access. It makes, however, the code more difficult to read and debug. For this reason, I recommend it as the least solution, and only if you cannot apply the `hasAnyAuthority()` or `hasAuthority()` methods.

The only dependencies needed in your pom.xml are the spring-boot-starter-web and spring-boot-starter-security. These dependencies are enough to approach all the three solutions previously enumerated. You find this example in 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'll 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 to be managed by this instance. Each of the two users will have a different authority. You can see how to do this in listing 7.3.

### **Listing 7.3 Declaring the UserDetailsService and assigning the 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();
    }
}
```

```
}
```

```
#A The UserDetailsService returned by the method is added in SpringContext
#B We declare an InMemoryUserDetailsManager which stores a couple of users
#C First user "john" has the authority "READ"
#D Second user "jane" has the authority "WRITE"
#E The users are added to be managed by the UserDetailsService
#F Don't forget that a PasswordEncoder is also needed
```

The next thing we'll do is adding the authorization configuration. In chapter 2, when we've worked on the first example, you saw how we could make all the endpoints accessible for everyone. To do that, you extended the `WebSecurityConfigurerAdapter` class and overrode the `configure()` method similar to what you see in listing 7.4.

#### **Listing 7.4 Making all the endpoints accessible for everyone without authentication**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .anyRequest().permitAll();      #A
    }
}
```

**#A Permit the access for all the requests**

In listing 7.4, the `authorizeRequests()` method states that we continue with specifying the 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()` allows access to all requests - authenticated or not.

Let's say we want to make sure that only the users having the authority "WRITE" can access all the endpoints. For our example, this means only "jane". In the same way, we can achieve our goal and restrict access, this time based on the authorities of users. Take a look at the code in listing 7.5.

#### **Listing 7.5 Restricting access to only users having the "WRITE" authority**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .anyRequest()
            .hasAuthority("WRITE");      #A
    }
}
```

```
}
```

#A We use the hasAuthority() method to specify which is the condition in which the user has access to the endpoints.

In listing 7.5, you observe that I've 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 authorities, the app decides whether the call is allowed.

We can now start to test the application by calling the endpoint with each of the two users. When we call the endpoint with the user "jane", the HTTP response status will be 200 OK, and we will see the response body "Hello!". When we call it with the user "john" the HTTP response status is 403 Forbidden, and we get an empty response body back.

Calling the endpoint with user "jane":

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

The response body of the call is:

```
Hello
```

Calling the endpoint with user "john":

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

The response body of the call is:

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

In a very similar way, you could have used the `hasAnyAuthority()` method. This method has a varargs as a parameter; 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 try replacing it in the current examples with `hasAnyAuthority("WRITE")`, case in which the application will work precisely in the same way. If you replace it with `hasAnyAuthority("WRITE", "READ")`, then requests from users having either authority will be accepted. For our specific case, the application will allow the requests from both "john" and "jane". In listing 7.6, you can see how you could apply the `hasAnyAuthority()` method.

#### **Listing 7.6 Applying the `hasAnyAuthority()` method**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
    }
}
```

```

        http.authorizeRequests()
            .anyRequest()
                .hasAnyAuthority("WRITE", "READ");      #A
    }
}

```

#A Requests from users with both "WRITE" and "READ" authorities are permitted.

You can successfully call the endpoint now with any of our two users.

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

The response body is:

```
Hello!
```

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

The response body is:

```
Hello!
```

The third way you'll find used in practice to specify the access based on authorities of the user is the `access()` method. The `access()` method is more general. It receives as parameter a Spring expression (SpEL) that specifies the authorization condition. This method is very powerful, and it doesn't only refer 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'll first present it as an alternative to specifying the authorities with the `hasAuthority()` and `hasAnyAuthority()` methods. As you'll learn in this example, you have to provide a Spring expression as a parameter to the method. The authorization rule we define becomes more challenging to read, and this is why I don't recommend this approach for simple rules for which you could avoid it. However, the `access()` method has the advantage of allowing you to customize rules through the expression you provide as a parameter. And this is really powerful, as with SpEL expressions, you can basically define any condition. But in most situations, you could implement the required restrictions with the `hasAuthority()` and `hasAnyAuthority()` methods, and I recommend you use these in the first place. Use the `access()` method only if the other two options do not fit because you have some more generic authorization rules to implement.

I'll 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 would be one of the following:

- `hasAuthority('WRITE')`, which stipulates that the user needs the 'WRITE' authority to call the endpoint
- `hasAnyAuthority('READ', 'WRITE')`, which specifies that the user needs one of the 'READ' or 'WRITE' authorities. 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. Listing 7.7 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 extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .anyRequest()
            .access("hasAuthority('WRITE')");      #A
    }
}
```

#A Requests from users with the “WRITE” authority are authorized

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 `hasAnyAuthority()` method directly. But the `access()` method is not all evil. As I’ve 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 wouldn’t be easy to write otherwise. Precisely, the configuration presented in listing 7.8 defines two users: “john” and “jane” who have different authorities. “john” has only the “read” authority, while “jane” has the “read”, “write” and “delete” authorities. The endpoint should be accessible to those users who have the “read” authority but don’t have the “delete” authority.

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 use an expression that reflects the requirement: `"hasAuthority('read') and !hasAuthority('delete')"`. You 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 extends WebSecurityConfigurerAdapter {

    @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();
}

@Override
protected void configure(HttpSecurity http)
    throws Exception {

    http.httpBasic();

    String expression =
        "hasAuthority('read') and[CA]      #A
        !hasAuthority('delete')";      #A

    http.authorizeRequests()
        .anyRequest()
        .access(expression);
}
}

```

#A The expression states that the user must have the authority 'read' but not the authority 'delete'  
Let's test it now by calling the /hello endpoint.

Calling the endpoint with the user "john":

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

The body of the response is:

```
Hello!
```

Calling the endpoint with the 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 the “read” authority and can call the endpoint successfully. But “jane” also has the “delete” authority and is not authorized to call the endpoint. The HTTP status for its call will be 403 Forbidden.

With these examples, you’ve seen how to set the constraints regarding the authorities that a user needs to access some specified endpoints. Of course, we haven’t yet discussed selecting which requests to be secured based on path or HTTP method, so we have applied the rules for all the requests regardless of the endpoint exposed by the application. Once we finish doing the same configuration for user roles, we’ll discuss how to select the endpoints to which you apply the authorization configurations.

### 7.1.2 Restricting access for all the endpoints based on the 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. You’ll find them as well used in real applications, so this is why it is important to understand roles and the difference between them and authorities. In this section, we’ll apply several examples using roles such that you know all the essential practical scenarios in which the application uses roles and how to write the configurations for these cases.

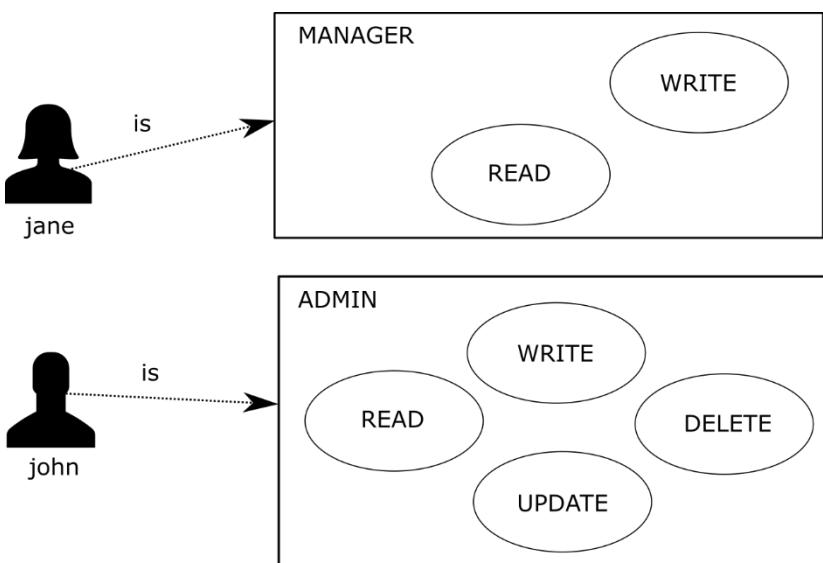


Figure 7.5 Roles are coarse-grained. Each user with a specific role can do the actions granted by that role. When applying this philosophy in authorization, a request is allowed based on the purpose of the user in the system. For example, only users who have a specific role can call a certain endpoint.

Spring Security understands authorities as fine-grained privileges on which we apply restrictions (figure 7.5). Roles are like badges for the users, which give them privileges for a group of actions. Some applications always provide the same groups of authorities to specific

users. Imagine in your application a user could either only have the “read” authority, or have all “read”, “write” and “delete”. In this case, it may be more comfortable to think that those users who can only “read” have a role named “READER”, while the others have the role “ADMIN”. So having the “ADMIN” role means that the application grants you all the “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 the authorities anymore. The authorities exist, in this case, as a concept and may appear in the implementation requirements. But in the application, you’ll 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 the roles are, like in the case of the authorities, at your own choice. We could say that roles are coarse-grained when compared with authorities. Behind the scenes, anyway, they are represented using the same contract in Spring Security: `GrantedAuthority`. Whenever defining a role, its name should start with the “ROLE\_” prefix. At the implementation level, this prefix makes the difference between a role and an authority. You find the example we work on in this section in project `ssia-ch7-ex3`.

Take a look at the change I’ve done to the previous example in listing 7.9:

### Listing 7.9 Setting roles for users

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @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
}

```

#A Having the ROLE\_ prefix, the GrantedAuthority now represents a role

To set the constraints, you can now use one of the following methods:

- `hasRole()`, receives as a parameter the role name for which the application will authorize the request.
- `hasAnyRole()`, receives as parameters the role names for which the application will approve the request.
- `access()`, uses a Spring expression to specify the role or roles for which the application will authorize the requests. In terms of roles, you could use as SpEL the `hasRole()` or `hasAnyRole()` expressions.

As you observe, the names are very similar to the methods presented in section 7.1.1. We use them 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 fallback to using `access()` only when the previous two don't apply.

**In listing 7.10, you can see how the `configure()` method looks like now**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .anyRequest().hasRole("ADMIN");      #A
    }
}
```

#A The `hasRole()` method is used now to specify the roles for which the access to the method is permitted. Mind that the prefix does not appear here.

**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 the user “john” can access the endpoint, while “jane” receives an HTTP 403 Forbidden.

Calling the endpoint with user “john”:

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

The response body is:

```
Hello
```

Calling the endpoint with user “jane”:

```
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've also done in the example of 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 the "ROLE\_" prefix is inadvertently included in the `role()` parameter, the `role` method will throw 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 listing 7.11, you can see the correct way to use the `roles()` method instead of `authorities()` when you design the access based on roles.

#### Listing 7.11 Using the `roles()` method to build `UserDetails` instances with the `User` class builder

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @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
}

```

#A The `roles()` method is used to specify the roles of the user.

## More on the access() method

In sections 7.1.1 and 7.1.2, you have 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 very generic. With the examples I presented, I focused on teaching you how to apply it for authorities and roles, but in general, it receives any SpEL expression. It doesn't need to be related to authorities and roles.

A straightforward example would be: configure the `access` to the method 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, 0))
```

More about SpEL expressions you can also find in the Spring Framework documentation:

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions>

So we could say that with the `access()` method, you could basically implement any kind of rule. The possibilities are endless. Just don't forget that in applications, we always strive to keep the syntaxes as simple as possible.

Complicate your configurations only when you don't have any other choice. You find this example applied in project ssia-ch7-ex4.

### 7.1.3 Restricting all the access to endpoints

In this section, we discuss restricting access to all requests. You learned in chapter 5 that, using the `permitAll()` method, you can permit access for all the requests. You learned as well that you apply access rules based on authorities and roles. But what you could also do is to deny all the requests. The `denyAll()` method is just the opposite of `permitAll()`. In the listing 7.12, you can see how to use the `denyAll()` method.

#### Listing 7.12 Using the `denyAll()` method to restrict the access to endpoints

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

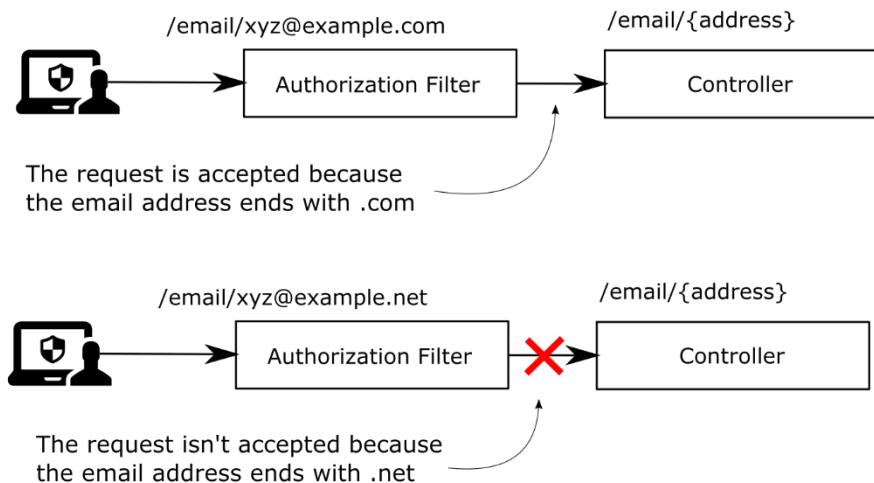
        http.authorizeRequests()
            .anyRequest().denyAll();      #A
    }
}
```

#A Using the `denyAll()` method to restrict access for everyone

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. So let me show you a couple of cases to make an idea.

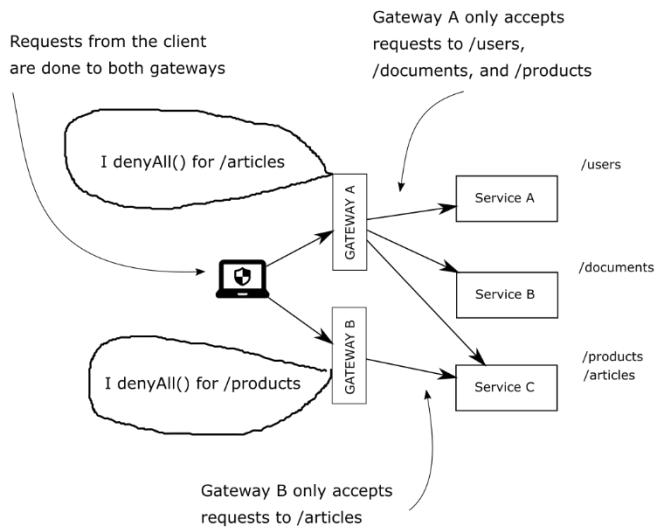
Let's assume that you have an endpoint receiving as path variable an email address. What you want is to allow the requests that have the value of the variable addresses ending in `.com`. You don't want the application to accept any other format for the email address. You'll learn in the next session how to apply restrictions for a group of requests based on the path and HTTP

method and even the path variables. For this requirement, you would use a regular expression to group the requests that match your rule and 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 a behavior, you can use the `denyAll()` method for all the 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 7.5, I've called them Gateway A and Gateway B. The client requests Gateway A if they want to access the `/products` path. But for the `/articles` path, it has to request Gateway B. Each of the gateway services is designed to deny all the requests to the other paths that they 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 to the use cases is done through Gateway A and Gateway B. Each of the gateways only delivers requests for specific paths and denies all the others.**

Applications out there in production face various architectural requirements, which could look strange sometimes. A framework must allow you the needed flexibility for any situation you might be in. For this reason, the `denyAll()` is as important as all the other options you learned in this 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 the requests based on the authorities and roles of the authenticated user.
- You can configure that certain requests are also possible for unauthenticated users.

# 8

## *Configuring authorization: applying restrictions*

### This chapter covers

- Selecting requests for applying restrictions using matcher methods.

In chapter 7, you learned how to configure access based on authorities and roles. But we have only applied the configurations to 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 the requests. You'll 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 custom authorization configuration. Let's discuss the options you have to refer to different requests when you write the access configurations.

Even if we didn't call it this way, the first matcher method you have used is the `anyRequest()` method. As you have used it in the previous examples, you know now that it refers to all the requests, regardless of the path, or HTTP method. It is the way you say "any request" or, sometimes, "any other request".

First of all, let's talk about selecting requests by path, then we will also add the HTTP method to the scenario. To choose the requests to which we apply the authorization configuration, we'll use matcher methods. Spring Security offers you three types of matcher methods:

- MVC matchers, for which you can use MVC expressions for paths to select the endpoints.
- ANT matchers, for which you can use ANT expressions for paths to select the endpoints.
- REGEX matchers, for which you can use regex expressions for paths to select the endpoints.

## 8.1 Using matcher methods to select endpoints

In this section, you'll learn how to use the matcher methods in general so that in sections 8.2, 8.3, and 8.4, we can continue with describing each of the three options you have: MVC, ANT, and regex. By the end of this chapter, you'll be able to apply matcher methods for any of the authorization configurations you might need to write for your applications' requirements. Let's start with a straightforward example so that you learn to apply the matcher methods.

We create an application that exposes two endpoints: `/hello` and `/ciao`. We want to make sure that only the users having the role "ADMIN" can call the `/hello` endpoint. Similarly, we want to make sure that only the users having the role "MANAGER" can call the `/ciao` endpoint. You find this example in project `ssia-ch8-ex1`.

In listing 8.1, you can see 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" will have the role "ADMIN", while "jane" will have the role "MANAGER".

To specify that only users having the role of an "ADMIN" can call the endpoint `/hello`, when authorizing requests, we use the `mvcMatchers()` method. In listing 8.2, you find the definition of the configuration class.

### Listing 8.2 The definition of the configuration class

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @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();
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic();

    http.authorizeRequests()
        .mvcMatchers("/hello").hasRole("ADMIN")      #A
        .mvcMatchers("/ciao").hasRole("MANAGER");     #B
}
}

```

#A Path /hello can only be called if the user has the role “ADMIN”

#B Path /ciao can only be called if the user has the role “MANAGER”

You can run and test the application. When you call the endpoint /hello with the user “john”, you will get a successful response. But if you call the endpoint with the user “jane”, the response status will be HTTP 403 Forbidden. Similarly, for the endpoint /ciao, you can only use “jane” to get a successful result. For the user “john”, the response status will be HTTP 403 Forbidden. You can see the example calls using curl in the next code snippets.

Calling the endpoint /hello with user “john”:

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

The response body is:

```
Hello!
```

Calling the endpoint /hello with user “jane”:

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

The response body is:

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

Calling the endpoint /ciao with user “jane”:

```
curl -u jane:12345 http://localhost:8080/ciao
```

The response body is:

```
Hello!
```

Calling the endpoint `/ciao` with user "john":

```
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 will be accessible by default to anyone, even unauthenticated. Let's assume you add a new endpoint `/hola` as presented in listing 8.3.

### **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!";
    }
}
```

You can now try to access this new endpoint. You will see that it is accessible with or without having a valid user, as presented in the next code snippets.

Calling the endpoint `/hola` without authenticating:

```
curl http://localhost:8080/hola
```

The response body is:

```
Hola!
```

Calling the endpoint `/hola` with user "john":

```
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 for any other request. You can do this by using the `anyRequest()` matcher method at the end, as presented in listing 8.4.

**NOTE** It is good practice to make all your rules explicit. Listing 8.6 clearly and unambiguously indicates the intention to permit requests to everyone to endpoints except for `/hello` and `/ciao`.

#### Listing 8.4 Marking all the other requests explicitly as being accessible without authentication

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .mvcMatchers("/hello").hasRole("ADMIN")
            .mvcMatchers("/ciao").hasRole("MANAGER")
            .anyRequest().permitAll();      #A
    }
}
```

#A The `permitAll()` method clearly states that all the other requests are allowed without authentication

**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 matcher method like `mvcMatchers()`.

---

#### 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 anymore. 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), the authentication will fail, and the response status will be 401 Unauthorized. To be more precise, in the previous example, if you call the `/hola` endpoint without a user, it will return the body "Hola!" as expected, and the response status will be 200 OK.

curl <http://localhost:8080/hola>

The response body is:

Hola!

But if you call the endpoint with non-valid 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:

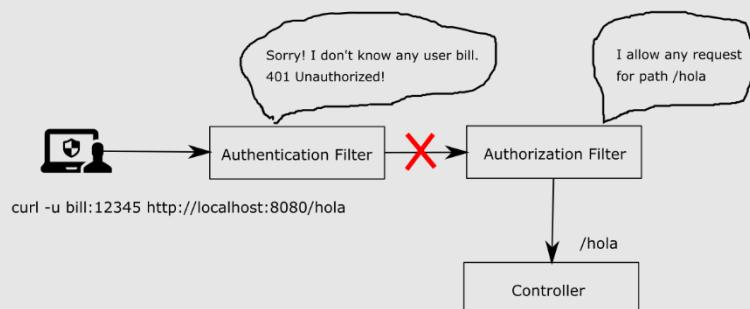
```
{
    "status":401,
    "error":"Unauthorized",
```

```

    "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 have already learned at the beginning of this chapter, the application always does the authentication before authorization (figure 8.1).



**Figure 8.1** The authorization filter would allow 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 the authentication fails will generate a response with status 401 Unauthorized, and the application won't forward the call to the endpoint. The `permitAll()` method refers to the authorization configuration only, and if the 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 only change the `permitAll()` method with `authenticated()`, as presented in listing 8.5.

#### **Listing 8.5 Making other requests accessible for all authenticated users**

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.httpBasic();
}

```

```

http.authorizeRequests()
    .mvcMatchers("/hello").hasRole("ADMIN")
    .mvcMatchers("/ciao").hasRole("MANAGER")
    .anyRequest().authenticated();      #A
}
}

```

#A All the other requests are accessible only by authenticated users

Similarly, you could even deny all the other requests by using the `denyAll()` method.

By the end of this section, you've become familiar with how you should use the matcher methods to refer to requests for which you want to configure the authorization restrictions. Now we must go a little bit more in detail with the syntaxes you can use. In most of the practical scenarios, more endpoints together will have the same authorization rules. So you will not have to set them up endpoint by endpoint. As well, you will sometimes need to specify the HTTP method, not only the path as we've done until now. You'll find requirements for which, for some path, you need to configure rules only for when somebody calls it with the HTTP GET method. But for this path, you will have to define other rules for when someone calls it with HTTP POST or HTTP DELETE, for example.

In the next sections, we will take each type of matcher method and discuss in detail these aspects.

## 8.2 Selecting requests for authorization using MVC matchers

In this section, we discuss the MVC matchers. Using MVC expressions is a commonly used approach to refer to requests for applying the authorization configuration. So I expect you have big chances to find this method to refer to requests in the applications you develop. And for this, I consider it essential for you to know it. This method uses the standard MVC syntax for referring to paths. This syntax is the same one you are using when writing the endpoints mappings with annotations like `@RequestMapping`, `@GetMapping`, `@PostMapping`, etc.

The two methods you can use to declare MVC matchers are:

- `mvcMatchers(HttpMethod method, String... patterns)`, which allows you to specify both the HTTP method to which the restrictions will apply and the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same path.
- `mvcMatchers(String... patterns)`, which is simpler and easier to use if you only need to apply the authorization restrictions based on paths. The restrictions will automatically apply for any HTTP method used with that path.

In this section, we approach multiple ways of using the `mvcMatchers()` methods. To demonstrate these different fashions of using them, we start by writing an application that exposes multiple endpoints.

For the first time, we'll write endpoints that can be called with another HTTP method than GET. You might have observed that until now, I've avoided using other HTTP methods than GET. The reason why I have avoided doing this is that Spring Security applies by default protection against Cross-Site Request Forgery (CSRF). In chapter 1, I have described CSRF, which is one of the most common vulnerabilities for web applications. Cross-Site Request

Forgery was for a long time present in the OWASP Top 10. In chapter 10, we'll discuss how Spring Security mitigates this vulnerability by using CSRF tokens. But to make things simpler for the current example and be able to call all the endpoints, including those exposed with POST, PUT or DELETE, we will have to disable the CSRF protection.

In our `configure()` method, you will also have to tell Spring Security to disable CSRF protection:

```
http.csrf().disable();
```

**NOTE** We disable the CSRF protection now only to enable you to focus for the moment on the discussed subject: matcher methods. Don't rush to consider this is a good approach. In chapter 10, we discuss in detail the CSRF protection provided by Spring Security.

We start by defining four endpoints to be used in our tests:

- /a, using HTTP method GET
- /a, using HTTP method POST
- /a/b, using HTTP method GET
- /a/b/c, using HTTP method GET

With them, we will consider different scenarios for the authorization configuration. In listing 8.6, you can see the definitions of these endpoints. You find this example in project `ssia-ch8-ex2`.

#### Listing 8.6 Definition of the four endpoints for which we configure the authorization

```
@RestController
public class TestController {

    @PostMapping("/a")
    public String postEndpointA() {
        return "Works!";
    }

    @GetMapping("/a")
    public String getEndpointA() {
        return "Works!";
    }

    @GetMapping("/a/b")
    public String getEndpointB() {
        return "Works!";
    }

    @GetMapping("/a/b/c")
    public String getEndpointC() {
        return "Works!";
    }
}
```

We will also need a couple of users with different roles. To keep things simple, we will continue using an `InMemoryUserDetailsService`. In listing 8.7, you can see the definition of the `UserDetailsService` in the configuration class.

#### **Listing 8.7 The definition of the UserDetailsService**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {
        var manager = new InMemoryUserDetailsService();      #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
    }
}
```

#A We define an `InMemoryUserDetailsService` to store the users

#B User "john" has the role of "ADMIN"

#C User "jane" has the role of "MANAGER"

#D Don't forget you also need to add a `PasswordEncoder`

Let's start with the first scenario: for the requests done with HTTP GET method for the `/a` path, the user needs to authenticate. For the same path, requests using HTTP POST method don't require authentication. The application denies all the other requests.

Listing 8.8 shows the configurations that you need to write to achieve this setup.

#### **Listing 8.8 Authorization configuration for the first scenario**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
    }
}
```

```

    http.authorizeRequests()
        .mvcMatchers(HttpMethod.GET, "/a")
            .authenticated()      #A
        .mvcMatchers(HttpMethod.POST, "/a")
            .permitAll()         #B
        .anyRequest()
            .denyAll();          #C

    http.csrf().disable();    #D
}

```

#A For path /a requests done with HTTP GET method the user needs to authenticate

#B For path /a requests done with HTTP POST method are permitted to anyone

#C Any other request to any other path is denied

#D We have disabled CSRF to be able to call the /a path using the HTTP POST method

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 the path /a using method POST without authenticating:

```
curl -XPOST http://localhost:8080/a
```

The response body is:

Works!

When calling the path /a using HTTP GET and without authenticating:

```
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 the path /a/b, so authenticating with their credentials for this call will generate 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 have seen how to differentiate the 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 the authorization rules, but this is not comfortable for reading if we have too many. As well, we might know from the beginning that a group of paths having the same prefix will always have the same authorization rules. So we want to make sure that if a developer adds a new path to the same group, it doesn't also have to change the authorization configuration. To manage these cases, we will use path expressions. Let's prove them in an example. For the current project, we want to make sure the same rules apply for all the requests for paths starting with /a/b. These paths are in our case: /a/b and /a/b/c. To achieve this, we'll use the \*\* operator. Spring MVC borrows the path matching syntaxes from ANT. You find this example in project ssia-ch8-ex3.

#### Listing 8.9 Changes in the configuration class

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .mvcMatchers( "/a/b/**" )      #A
            .authenticated()
            .anyRequest()
            .permitAll();

        http.csrf().disable();
    }
}

```

#A The /a/b/\*\* expression refers to all the paths prefixed with /a/b

With the configuration given in listing 8.9, you can call the path /a without being authenticated, but for all the paths prefixed with /a/b the user needs to authenticate. The next code snippets present the results of calling the /a, /a/b, and the /a/b/c endpoints.

Calling the /a path without authenticating:

```
curl http://localhost:8080/a
```

The response body is:

Works!

Calling the /a/b path without authenticating:

```
curl http://localhost:8080/a/b
```

The response is:

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

Calling the /a/b/c path without authenticating:

```
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 example at the end so that we match requests with paths having a known prefix. You can use it in the middle of the path as well to just refer to any number of pathnames, or refer to paths ending in a specific pattern like: `/a/**/c`. For `/a/**/c` would match `/a/b/c` but also `/a/b/d/c` or `a/b/c/d/e/c` and so on.

If you only want to match one pathname, then you can use a sole `*`. For example, `a/*/c` would match to `a/b/c` or `a/d/c` but not to `a/b/d/c`.

And because you generally use path variables, you will find it very useful to apply authorization rules for such requests having path variables as well. You can apply even rules referring to the path variable value. Do you remember the discussion from section 8.1 about the `denyAll()` and restricting all the requests? Let's turn it now into a more suitable example with what you have learned in this section. We have an endpoint with a path variable, and we want to deny all the requests that use a value for the path variable that has anything else than only digits. You find this example in project `ssia-ch8-ex4`.

Listing 8.10 presents the controller.

#### **Listing 8.10 The definition of the controller class**

```
@RestController
public class ProductController {

    @GetMapping("/product/{code}")
    public String productCode(@PathVariable String code) {
        return code;
    }
}
```

Listing 8.11 shows you how to configure the authorization such that only the calls which have a value containing only digits are always permitted while all the other calls are denied.

**Listing 8.11 Configuring the authorization**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .mvcMatchers[CA]
            ("/{code:[0-9]*$}")      #A
            .permitAll()
            .anyRequest()
            .denyAll();
    }
}
```

#A The regex refers to strings of any length, containing any digit.

**NOTE** When using parameter expressions with regex, make sure to have no space between the name of the parameter, the colon (:), and the regex.

Running the example, you will see the result as also presented in the next code snippets. The application only accepts the call when the path variable value has only digits.

Calling the endpoint using the value 1234a:

```
curl http://localhost:8080/product/1234a
```

The response is:

```
{
    "status":401,
    "error":"Unauthorized",
    "message":"Unauthorized",
    "path":"/product/1234a"
}
```

Calling the endpoint using the value 12345:

```
curl http://localhost:8080/product/12345
```

The response is:

```
12345
```

We've discussed a lot with plenty of examples of how to refer to requests using MVC matchers. Table 8.1 is a refresher for the MVC expressions you have used in this section, so 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 would match /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, or /a/b/c would be matched by 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.

### 8.3 Selecting requests for authorization using ANT matchers

In this section, we discuss ANT matchers for selecting the requests for which the application applies the authorization rules. Because the MVC expressions used by Spring to match paths to endpoints are borrowed from ANT, the syntaxes that you can use with ANT matchers are the same that you have seen in section 8.2. But there's a trick I'll show you in this section within an example - a significative difference you should be aware of. As you'll see, for this reason, I recommend you to use MVC matchers rather than ANT matchers. However, in the past years, I've seen a lot of times ANT matchers used in applications. For this reason, I want to make you aware of them as well and, of course, about how they act differently. You can still find them in production applications today, and this makes them important as well.

The three methods of using ANT matchers are:

- `antMatchers(HttpServletRequest method, String... patterns)`, which allows you to specify both the HTTP method to which the restrictions will apply and the ANT patterns that refer to the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same group of paths.
- `antMatchers(String... patterns)`, which is simpler and easier to use if you only need to apply the authorization restrictions based on paths. The restrictions will automatically apply for any HTTP method.
- `antMatchers(HttpServletRequest method)`, which is the equivalent of `antMatchers(HttpServletRequest, "/**")`. By using it, you refer to a specific HTTP method, disregarding the paths.

Observe that the way of applying them is also similar to the MVC matchers. Also, the syntaxes we'll use for referring to paths are the same. So what is different then? The MVC matchers refer exactly to how your Spring application understands the matching of the requests to controller actions. And, sometimes, multiple paths could be interpreted by Spring to match the same action.

My favorite example, which is very simple but makes a significant impact in terms of security, is the following: any path, let's take, for example, `/hello`, could also be interpreted by Spring if you append another `/` after it to the same action. In this case, `/hello` and `/hello/` would call the same method. If you use an MVC matcher and configure security for `/hello` path, it will automatically secure the `/hello/` path with the same rules. This is huge! A developer not knowing this, using ANT matchers could leave unprotected a path without

noticing. And this, as you imagine, creates a major security breach for the application. Let's test this behavior with an example. You find this example in project `ssia-ch8-ex5`.

Listing 8.12 shows you how to define the controller.

#### **Listing 8.12 Definition of the controller class**

```
@RestController
public class HelloController {

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

Listing 8.13 describes the configuration class. In this case, an MVC matcher is used to define the authorization configuration for the `/hello` path. Any request to this endpoint requires authentication. I omitted the definition of the `UserDetailsService` and `PasswordEncoder` from the example as they are the same as in listing 8.7.

#### **Listing 8.13 The configuration class using an MVC matcher**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .mvcMatchers( "/hello")
            .authenticated();
    }
}
```

If you start and test the application, you'll observe that authentication is required for both `/hello` and `/hello/` paths. This is also probably what you would expect to happen. The next code snippets show the requests made with `curl` for these paths.

Calling the `/hello` endpoint unauthenticated:

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

The response is:

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

Calling the `/hello` endpoint using the `/hello/` path (with one more / at the end), unauthenticated:

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

The response is:

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

Calling the /hello endpoint authenticating as "jane":

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

The response body is:

```
Hello!
```

Calling the /hello endpoint using the /hello/ path (with one more / at the end), authenticating as "jane":

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

The response body is:

```
Hello!
```

All of these responses are what you've probably expected. But let's see what happens if we change the implementation to use ANT matchers.

If you just change the configuration class to use an ANT matcher now, for the same expression, the result will change. The app doesn't apply the authorization configurations anymore for the /hello/ path. The ANT matchers apply exactly the given ANT expressions for patterns but know nothing about subtle Spring MVC functionality. In this case /hello doesn't also apply as an ANT expression to the /hello/ path. If you also want to secure the /hello/ path, you have to individually add it or write an ANT expression that matches it also. Listing 8.14 shows the change made in the configuration class to use an ANT matcher instead of the MVC matcher.

#### **Listing 8.14 The configuration class using an ANT matcher**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()
            .antMatchers( "/hello").authenticated();
    }
}
```

In the next code snippets, you find the results for calling the endpoint with the `/hello` and `/hello/` paths.

Calling the `/hello` endpoint unauthenticated:

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

The response is:

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

Calling the `/hello` endpoint unauthenticated, but using the path `/hello/` (with one more / at the end):

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

The response is:

```
Hello!
```

To say it again: I recommend and prefer the MVC matchers. Using MVC matchers, you avoid some of the risks involved with the way Spring maps paths to actions. And this is because you know the way paths are interpreted for the authorization rules are the same as Spring itself interprets them for mapping the paths to endpoints. When you use ANT matchers, exercise caution for this aspect, and make sure your expressions indeed match everything you need to apply the authorization rules.

---

### **Effects of communication and knowledge sharing**

I always encourage sharing knowledge in all possible ways: books, articles, conferences, videos, and so on. Sometimes even a short discussion can raise questions that drive dramatic improvements and changes. I'll illustrate what I mean through a story from a course about Spring I delivered a couple of years ago.

The training was designed for a group of intermediate developers who were working for a specific project. It wasn't directly related to Spring Security, but at some point, we started using matchers methods for one of the examples we were working on as part of the training.

I started configuring the endpoints authorization rules with MVC matchers without teaching in first place MVC matchers to the participants. I thought that they would have already used them in their project, I didn't think it mandatory to explain them first. While I was working on the configuration and teaching what I was doing, one of the attendees asked a question. I still remember the shy voice of the lady asking, "Could you introduce these MVC methods you're using? We're configuring our endpoints security with some ANT-something methods."

I realized then that the attendees might not be aware of what they were using. And I was right. They were indeed working with ANT matchers, but didn't understand these configurations and were, I believe, most probably using them mechanically. Copy-paste programming is a risky approach, unfortunately used too often, especially by junior developers. You should never use something without understanding what it does.

While we were discussing the new subject, the same lady found in their implementation precisely a situation in which the ANT matchers were wrongly applied. The training ended with their team scheduling a full sprint to verify and correct such mistakes that could have lead to very dangerous vulnerabilities in their app.

## 8.4 Selecting requests for authorization using regex matchers

In this section, we'll discuss the regular expression (regex) matchers. For this section, you should be aware of what regular expressions are, but you don't need to be an expert in the subject. Any of the books recommended on the page <https://www.regular-expressions.info/books.html> are excellent resources from which you can learn the subject more in-depth. For writing regex, I also often use online generators like [regexr.com](https://regexr.com/).

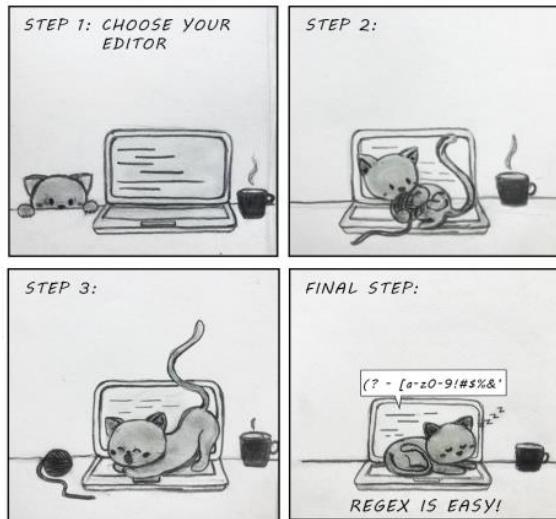


Figure 8.2 Letting your cat play over the keyboard is not the best solution to generate regular expressions. To learn how to generate regex expressions, you could use an online generator like <https://regexr.com/>

You learned in sections 8.2 and 8.3 that in most of the cases you could use MVC and ANT syntaxes to refer to the requests to which you apply the authorization configurations. In some cases, however, you might have requirements that are more particular, and you cannot solve with ANT and MVC expressions. An example of such a requirement could be: "Deny all requests when paths contain specific symbols or characters.". For these scenarios of requirements, you need to use a more powerful expression like a regular expression. You can use regular expressions 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 for simple

scenarios. For this reason, you'll prefer to use MVC or ANT matchers and fallback to regex only when you'll have no other option.

The two methods that you can use to use regex matchers are:

- `regexMatchers(HttpMethod method, String... regex)`, which you can use to specify both the HTTP method to which the restrictions will apply and the regex expressions that refer to the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same group of paths.
- `regexMatchers(String... regex)`, which is simpler and easier to use if you only need to apply the authorization restrictions based on paths. The restrictions will automatically apply for any HTTP method.

To prove how regular expression matchers work, let's put them in action in an example. We build an application that provides video content to its users. The application presenting the video to the users gets the content by calling the endpoint `/video/{country}/{language}`. The application receives the country and language from where the user makes the request, for the sake of the example, in two path variables. We consider that any authenticated user can see the video content if the request comes from the US, Canada, or the UK or use English. You find this example implemented in project `ssia-ch8-ex6`.

The endpoint we have to secure has two path variables, as shown in listing 8.15, which makes the requirement complicated to implement with ANT or MVC matchers.

#### **Listing 8.15 The definition of 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 could have written a regex directly in the ANT or MVC expression. We have referred to such an example also in section 8.3, but I didn't go in-depth at that time since we weren't discussing regex. Let's assume you have an endpoint `/email/{email}`. You want to apply a rule using a matcher only to the requests which send as a value of the `email` parameter an address ending in `.com`. Then you would write an MVC matcher as presented by the next code snippet. You can find the complete example of this example in the projects provided with the book: `ssia-ch8-ex7`.

```
http.authorizeRequests()
    .mvcMatchers("/email/{email}.*(\.+\@\.\com\b)")
        .permitAll()
    .anyRequest()
        .denyAll();
```

If you would test such a restriction, you'll observe that the application only accepts emails ending in `.com`.

Calling the endpoint for email jane@example.com:

```
curl http://localhost:8080/email/jane@example.com
```

The response body is:

```
Allowed for email jane@example.com
```

Calling the endpoint for email jane@example.net:

```
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 clear why we encounter regex matchers less frequently. But as I said earlier also, requirements are complex sometimes.

You'll find it handier to use regex matchers when you find something like:

- “Apply specific configurations for all paths containing phone numbers or email addresses”, or
- “Apply 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 more path patterns a multiple path variable values, you could easier write a regex matcher. In listing 8.16, you find the definition for the configuration class, which 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 8.16 The configuration class using a regex matcher**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @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();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();

        http.authorizeRequests()      #A
            .regexMatchers(".*/[us|uk|ca]+/[en|fr].*")
            .authenticated()
            .anyRequest()
            .hasAuthority("premium");   #B

    }
}

```

#A We use a regex to match the paths for which the user only needs to be authenticated

#B We configure for the other paths that the user needs to have premium access.

Running and testing the endpoints confirm that the application applied the authorization configurations correctly. The user “john” can call the endpoint with country US and language en, but can’t call the endpoint for country FR and language fr due to the restrictions we configured.

Calling the /video endpoint and authenticating with user “john” for region US and language English:

```
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 with user “john” for region FR and language French:

```
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, the user “jane” makes both calls with success.

```
curl -u jane:12345 http://localhost:8080/video/us/en
```

The response body is:

```
Video allowed for us en
```

```
curl -u jane:12345 http://localhost:8080/video/fr/fr
```

The response body is:

```
Video allowed for fr fr
```

Regular expressions are a powerful tool. You can use them to refer to paths for any given requirement. But because regular expressions are hard to read and can become quite long, they should remain your last choice. Use them only if MVC and ANT expressions don't offer you a solution to your problem. In this section, I have 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 regular expression is easy to read. For example, the regex used to match an email address looks like the one in the next code snippet. Can you easily read and understand it?

```
(?:[a-zA-Z0-9!#$%&'*+=?^`{|}~-]+(?:\.[a-zA-Z0-9!#$%&'*+=?^`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x23-\x5b\x5d-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(:[a-zA-Z0-9](?:[a-zA-Z0-9]*[a-zA-Z0-9])?\.)+[a-zA-Z0-9](?:[a-zA-Z0-9]*[a-zA-Z0-9])?\|[(?:(:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?)\.\{3}(?:25[0-5]|2[0-4][0-9]|01)?[0-9][0-9]?|[a-zA-Z0-9]*[a-zA-Z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f]))+)\")
```

## 8.5 Summary

- In real-world scenarios you often apply different authorization rules for different requests.
- You specify the requests for which the authorization rules are configured based on path and HTTP method. To do this, you use matcher methods that come in three flavors: MVC, ANT, and regex.
- The MVC and ANT matchers are very similar, and generally, you'll choose one of these options to refer to the requests for which you apply the authorization restrictions.
- When the requirements are too complex to be solved with ANT or MVC expressions, you can implement them with the more powerful regex expressions.

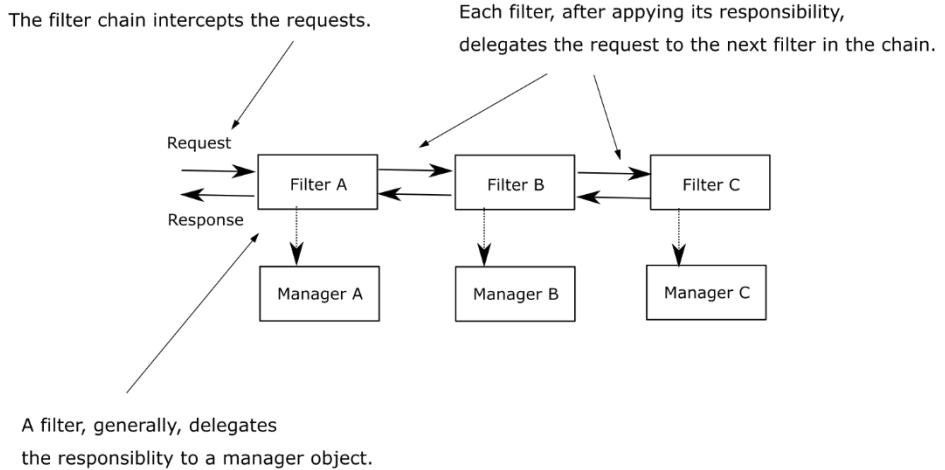
# 9

## *Implementing filters*

### This chapter covers

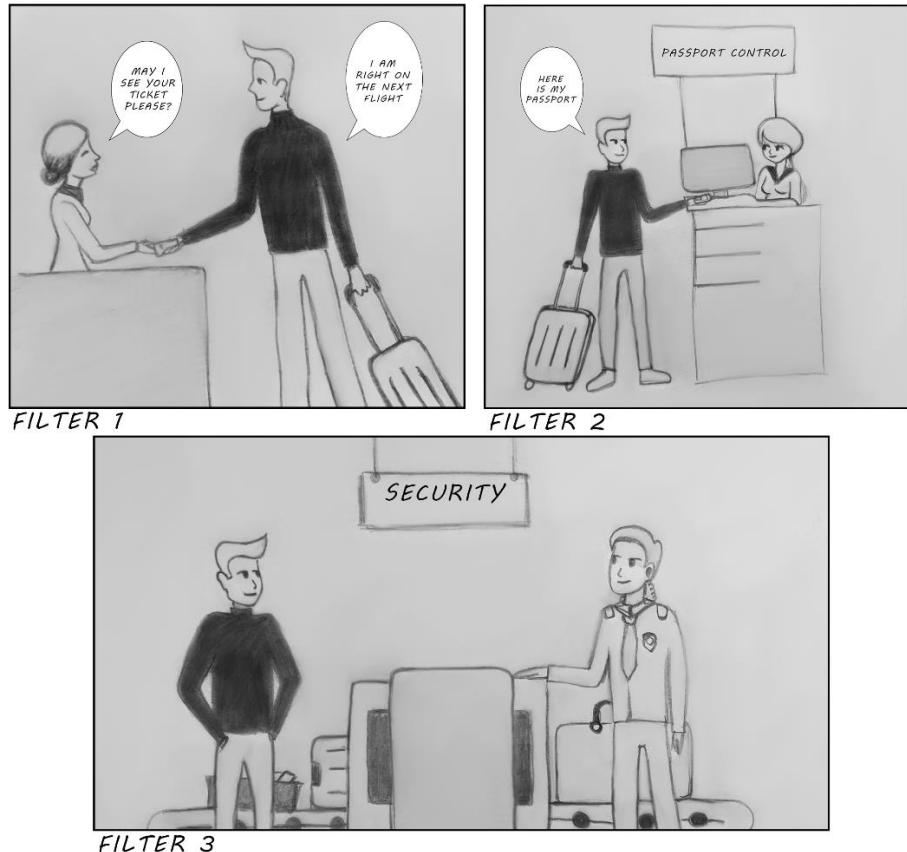
- Working with the filter chain.
- Defining custom filters.
- Using classes provided by Spring Security that implement the `Filter` interface.

In Spring Security, the HTTP filters delegate the different responsibilities that apply to an HTTP request. In chapters 3 through 5, where we discussed the HTTP Basic authentication and authorization architecture, I've often referred to filters. You learned that there is a component we named authentication filter, which delegates the authentication responsibility to the authentication manager. You learned as well that a certain filter takes care of the authorization configuration after successful authentication. In general, in Spring Security, the HTTP filters manage each responsibility that must be applied to the request. The filters form a chain of responsibilities. A filter receives the request, executes its logic, and eventually delegates the request to the next filter in the chain (figure 9.1).



**Figure 9.1** The filters chain receives the request. Each filter uses a manager to apply specific logic to the request and, eventually, delegates the request further in the chain to the next filter.

The idea is very simple. When you go to the airport, from entering the terminal to boarding the aircraft, you go through multiple filters (figure 9.2). You first present your ticket, then your passport is verified, and afterward, you go through security. At the airport decision, 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 will act on the HTTP requests. Spring Security provides filter implementations that you add to the filter chain through customization, but you can also define custom filters.



**Figure 9.2** 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.

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

Knowing to customize the HTTP filters chain of responsibilities is a valuable skill. In practice, applications come with various requirements, where using the default configurations doesn't work anymore. You'll find the need for adding or replacing existing components of this 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'll need to implement a different strategy for authentication, notify an external system about an authorization event, or perhaps simply log a successful or failed authentication later used in tracing and auditing (figure 9.3). Whatever your scenario is, Spring Security offers you this flexibility of modeling the filter chain precisely as you need it.

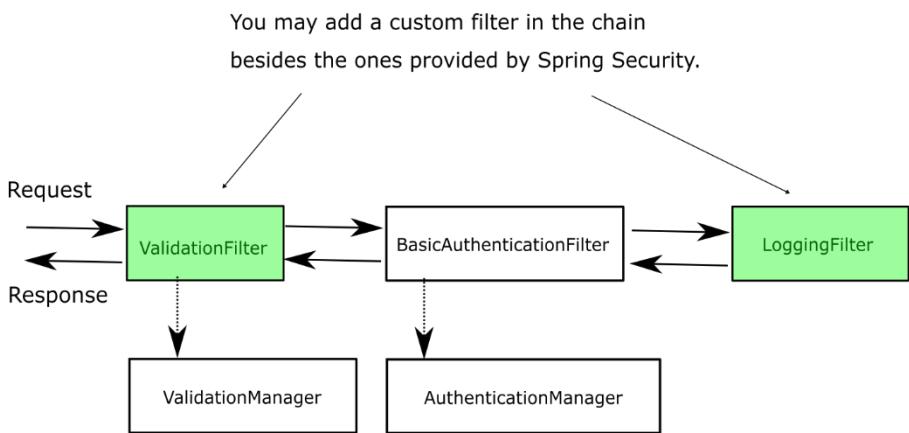


Figure 9.3 You can customize the filter chain by adding new filters before, after, or at the position of existing ones. This way you can customize the authentication as well as the entire process applied to the request and response

## 9.1 Implementing filters in the Spring Security architecture

In this section, we'll discuss the way the filters and the filter chain work in Spring Security architecture. You need this general overview first to understand the implementation examples we'll work on in the next sections of this chapter.

You learned in the previous chapters that the authentication filter intercepts the request and delegates the authentication responsibility further to the authorization manager. If we want to execute certain logic before the authentication, we quickly 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 `javax.servlet` package. Like for any other HTTP filter, you have to override the `doFilter()` method to implement its logic. The `doFilter()`

method receives as parameters the `ServletRequest`, `ServletResponse`, and the `FilterChain`.

- The `ServletRequest` parameter represents the HTTP request. We can use it to retrieve the details about the request.
- The `ServletResponse` is the HTTP response, which we can use to alter the response before sending it back to the client or further on the filter chain.
- The `FilterChain` represents the chain of filters. We use the `FilterChain` object to forward the request to the next filter in the chain.

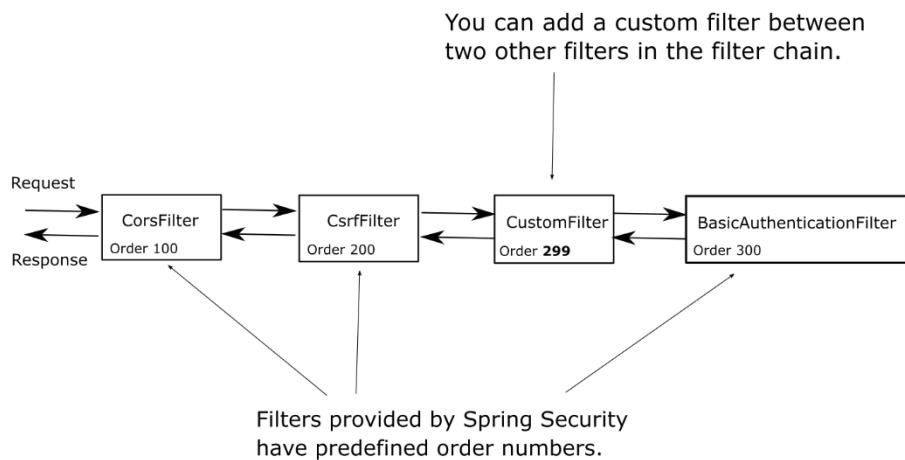
When we refer to the chain of filters, this represents a collection of filters with a defined order in which the filters will act. Spring Security provides some filter implementations and their order. Among the provided filters you find:

- `BasicAuthenticationFilter`, which takes care of the HTTP Basic authentication if present.
- `CsrfFilter`, that takes care of the CSRF protection, which we'll discuss in chapter 10.
- `CorsFilter`, that takes care of Cross-Origin Resource Sharing 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 them directly from your code, but you need to understand how the filter chain works and be aware of a few implementations. In this book, I'll explain those filters that are essential to 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 actually is that if you call the `httpBasic()` method an instance of the `BasicAuthenticationFilter` will be 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. You can either add a filter 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 "order".



**Figure 9.4** Each filter has an order number. This determines the order in which they are applied to the request. You can add custom filters between the filters provided by Spring Security.

That makes it possible to add two or more filters in the same position.

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

We'll encounter a common case in which this might occur and which usually creates confusion among developers in section 9.4.

You can define the same order value for two or more filters in the filter chain. In this case, the order in which those filters having the same order value are called is not defined.

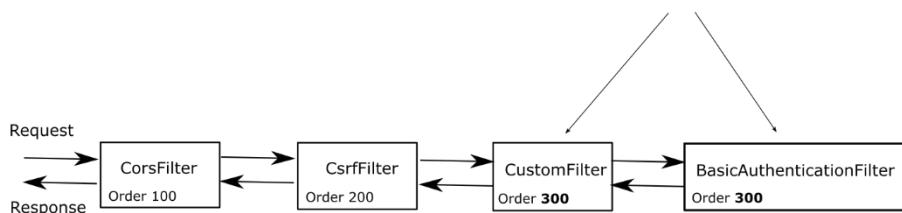


Figure 9.5 You might have more filters with the same order value in the chain. In this case, Spring Security doesn't guarantee the order in which they are called.

## 9.2 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'll find scenarios in which it's useful to apply a filter before an existing one in the filter chain. To approach it as practical as possible, 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. You can then adapt this example to any similar requirement you'll find in a production application.

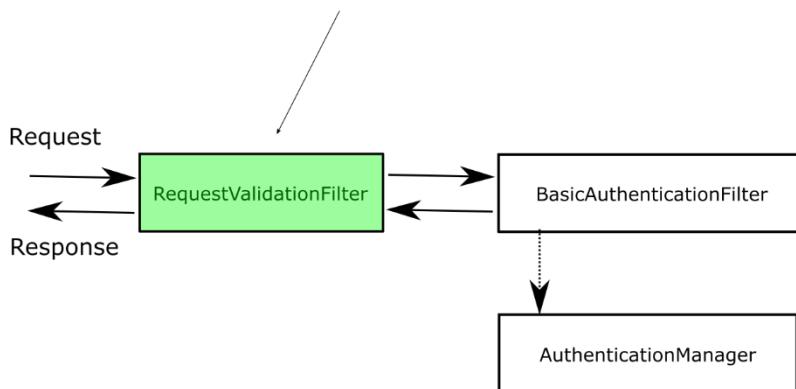
For our first custom filter implementation, let's consider a trivial scenario. We want to make sure, before starting the authentication process, that any request has a header called `Request-Id` (project `ssia-ch9-ex1`). We assume our application uses this header for tracking the requests, and this header is mandatory. At the same time, we want to make such validations before the app does the authentication. The authentication process might involve querying the database or other resource-consuming actions that we don't want the application to execute as long as the format of the request isn't valid anyway.

So, how we do this? To solve the current requirement only takes two steps:

1. Implement the filter - we'll create a class named `RequestValidationFilter`, which checks that the needed header exists in the request.
2. Add the filter to the filter chain - which we'll do in the configuration class, overriding the `configure()` method.

In the end, the filter chain will look as presented in figure 9.6.

We add a filter called RequestValidationFilter before the BasicAuthenticationFilter in the filter chain.



**Figure 9.6** For our example, we add a RequestValidationFilter, which will act before the authentication filter. The RequestValidationFilter makes sure that the authentication won't happen if the validation of the request fails. In our case, the request must have a mandatory header named Request-Id.

Step 1 - We define a custom filter, as presented in listing 9.1.

#### Listing 9.1 Implementing a custom filter

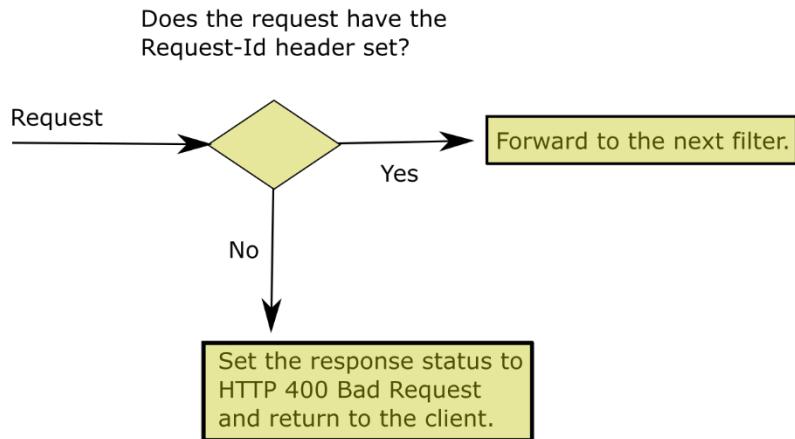
```

public class RequestValidationFilter
    implements Filter {    #A

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

#A To define a filter, the class has to implement the Filter interface and override the doFilter() method.

Inside the filter method, we can write the logic of the filter. In our case, 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 it 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 9.7).



**Figure 9.7** 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 Bad-Request and returns to the client.

Listing 9.2 presents this logic.

#### Listing 9.2 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(HttpStatus.SC_BAD_REQUEST);
        return;      #A
    }

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

#A If the header is missing, the HTTP status is changed to 400 Bad Request, and the request is not forwarded to the next filter in the chain.

#B If the header exists, the request is forwarded to the next filter in the chain.

Step 2 - Once we've written the filter, we apply it within the configuration class. Because we want the application to execute this custom filter before the authentication, we use the `addFilterBefore()` method of the `HttpSecurity` object. This method receives two parameters:

- An instance of the custom filter we want to add to the chain: in our case, an instance of the `RequestValidationFilter` class presented by listing 9.1.
- 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'll have to add it 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 `AuthenticationFilter`. You'll find out in the next chapters that Spring Security also configures other filters. In chapter 10, we'll discuss Cross-Site Request Forgery (CSRF) protection and Cross-Origin Resource Sharing (CORS), which also rely on filters.

Listing 9.3 shows the way to add the custom filter before the authentication filter in the configuration class. To make the example simpler, I've used the `permitAll()` method to allow all the requests unauthenticated.

### Listing 9.3 Configuring the custom filter before authentication

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterBefore(
            #A
            new RequestValidationFilter(),
            BasicAuthenticationFilter.class)
            .authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

#A An instance of the custom filter is added before the authentication filter in the filter chain.

We also need a controller class and an endpoint to test the functionality. In listing 9.4, you find the definition of the controller class.

### Listing 9.4 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 will generate 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!

Calling the endpoint without the `Request-Id` header:

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

This call will generate the following (truncated) response:

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

Calling the endpoint and providing the `Request-Id` header:

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

This call will generate the following (truncated) response:

```
Hello!
```

### 9.3 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'll use this approach whenever 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 9.8). As in section 9.1, we'll implement an example to show you how to add a filter after an existing one. You can quickly adapt it to your needs for a real-world scenario.

For the application we implement now as an example, we have log all the successful authentication events. We do this by adding a filter after the authentication filter (figure 9.8). We consider that what bypasses the authentication filter represents a successfully authenticated event, and we want to log it. Continuing the example from section 9.1, we'll also log the request ID received through the HTTP header.

We add a filter named AuthenticationLoggingFilter after the BasicAuthenticationFilter in the filter chain.

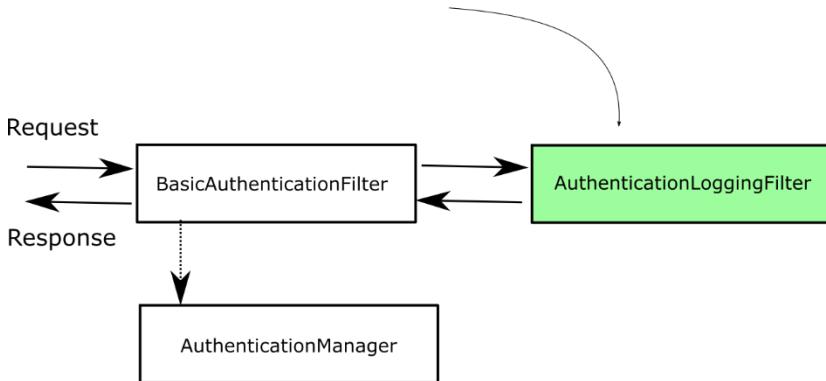


Figure 9.8 We add the AuthenticationLoggingFilter after the BasicAuthenticationFilter to log the requests that the application has authenticated.

Listing 9.5 presents the definition of a filter that logs the requests that passed the authentication filter.

#### **Listing 9.5 Defining a filter to log the 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
    }
}

```

#A We get the request ID from the request headers.

#B We log the event with the value of the request ID.

#C We forward the request to the next filter in the chain.

To add the custom filter in the chain after the authentication filter, you can call the `addFilterAfter()` method of `HttpSecurity`, as presented in listing 9.6.

**Listing 9.6 Adding a custom filter after an existing one in the filter chain**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterBefore(
            new RequestValidationFilter(),
            BasicAuthenticationFilter.class)
        .addFilterAfter( #A
            new AuthenticationLoggingFilter(),
            BasicAuthenticationFilter.class)
        .authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

#A An instance of AuthenticationLoggingFilter is added to the filter chain after the authentication filter.

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

For the following call:

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

The response body is:

```
Hello!
```

In the console, you'll see a line similar to the one in the next code snippet:

```
INFO 5876 --- [nio-8080-exec-2] c.l.s.f.AuthenticationLoggingFilter: Successfully
authenticated request with id 12345
```

## 9.4 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. That is, instead of using a username and a password as input credentials based on which the application authenticates the user, you have to apply another approach. Some examples of scenarios that you could find in applications are

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

Identification based on a static key for authentication (figure 9.9): In this case, the client has to send a string, which is always the same. The application stores these values somewhere (most probably in a database or a secret vault) and identifies the client by making the request based on this value, which the application sends back through an HTTP header. This approach offers weak security related to authentication, but architects and developers often choose it in calls among different backend applications. The reason for choosing this way is mostly its

simplicity. The implementations also execute fast as they don't need to do complex calculations like 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 neither leave the endpoints wholly unprotected.

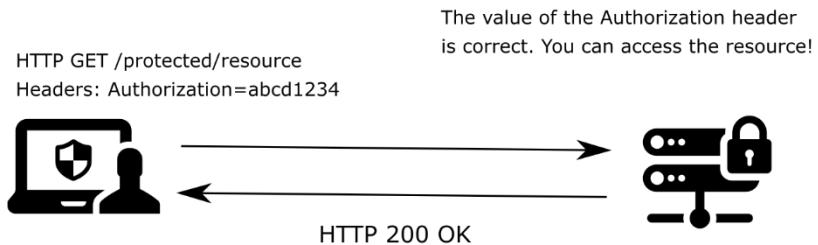


Figure 9.9 The request contains a header with the value of the static key. If this value matches the one known by the application, the request is accepted.

Using symmetric keys to sign and validate the requests (figure 9.10): In this case, both the client and the server know the value of a key (the client and the 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. The server can store individual keys for each client in a database or a vault for secrets. You could use a pair of asymmetric keys very similarly.

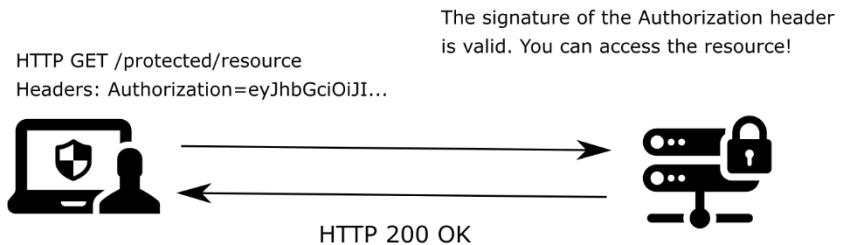
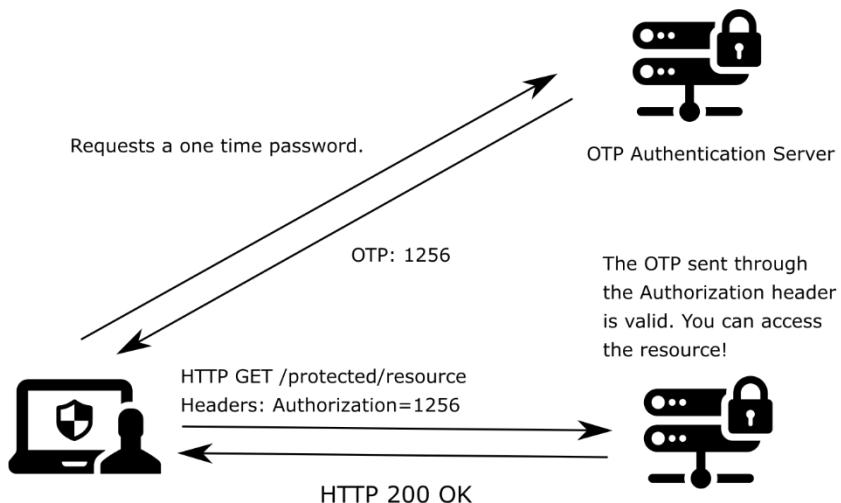


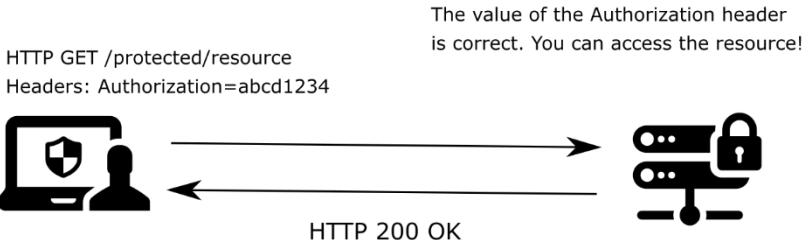
Figure 9.10 The authorization header contains a value signed with a key known by both the client and the server (or a private key for which the server has the public pair). The application checks the signature, and, if correct, allows the request.

Use a one time password (OTP) received by the user via a message or by using an authentication provider app like Google Authenticator (figure 9.11).



**Figure 9.11** 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, the applications use this approach during login when multi-factor authentication is required.

Let's implement an example to demonstrate how to apply a custom filter to solve such a requirement. To keep the case relevant but straightforward, we'll focus on the configuration part, and we'll consider a simple logic for authentication. In our scenario, we'll have a value of a static key, the same for all the requests. To be authenticated, the user has to add the correct value of the static key in the header named `Authorization`, as presented in figure 9.12. The code for this example is located in project `ssia-ch9-ex2`.



**Figure 9.12** The client adds a static key in the Authorization header of the HTTP request. The server checks if it knows the key to authorize the requests.

In chapter 11, which is the next hands-on exercise, we'll examine and implement as well a solution in which we apply cryptographic signatures for authentication.

We'll start with implementing the filter class, which I'll name `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 9.7 presents the definition of the `StaticKeyAuthenticationFilter` class.

#### **Listing 9.7** 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 {
    
```

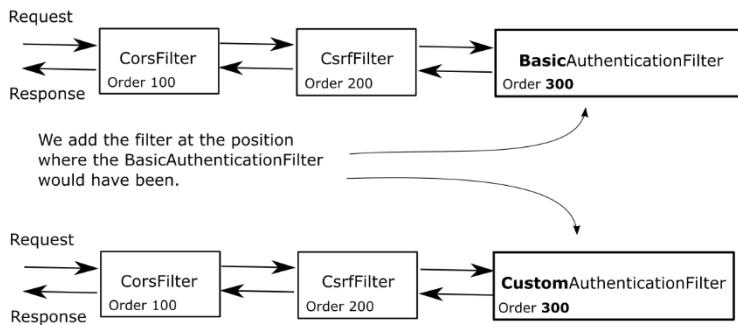
```

        httpResponse.setStatus(
            HttpServletResponse.SC_UNAUTHORIZED);
    }
}

```

#A To allow us to inject values from the property file, the class is added as a component in the Spring context.  
#B The class implements the Filter interface and overrides the doFilter() method to define the authentication logic.  
#C The value of the static key is taken from the properties file using the @Value annotation.  
#D The value of the authorization header is taken from the request to be compared with the static key.

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



**Figure 9.13** We add our custom authentication filter at the location where the 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 9.1: Adding a filter at a specific position 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 they'll act. I tell you this again now because I've seen people confused by how this works. Some developers tend to understand that when you apply a filter at a position of a known one if one already exists there, that one will be replaced. This is not the case! We make sure not to add to the chain filters that we don't need.

**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're used is not defined. It makes sense to have a definite order in which these filters are called. Having a known order makes your application easier to understand and maintain.

In listing 9.8, you find the definition of the configuration class, which 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 9.8 Adding the filter in the configuration class**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired      #A
    private StaticKeyAuthenticationFilter filter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterAt(filter,          #B
                         BasicAuthenticationFilter.class)
            .authorizeRequests()
            .anyRequest().permitAll();
    }

}
```

#A We inject the instance of the filter from the Spring context

#B We add the filter at the position of the basic authentication filter in the filter chain.

To test the application, we'll also need an endpoint. To have a test endpoint, we'll define a controller as presented by listing 9.4. You should add a value for the static key on the server in the `application.properties` file, as shown in the next code snippet.

```
authorization.key=SD9cICj11e
```

**NOTE** Storing passwords, keys, or any other data which is not meant to be seen by everybody in the properties files 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 secret vaults to store such kinds of details.

We can now start and test the application. We expect that the app allows the requests having the correct value for the `Authorization` header and rejects the 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've set on the server-side for the `Authorization` header, the call is successful, and you'll see the response body: Hello!

```
curl -H "Authorization:SD9cICj11e" http://localhost:8080/hello
```

The response body is:

```
Hello!
```

If the `Authorization` header is missing or is incorrect, the response status will be 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 will automatically configure one as you've learned in chapter 2. But in our scenario, you don't need a `UserDetailsService` at all, as the concept of the user doesn't exist. We only validate that the one requesting to call an endpoint on the server knows a given value. Applications scenarios are not usually this simple and often require a `UserDetailsService`, but, if you are in such a case where this component is not needed at all, you can disable this auto-configuration. To disable the configuration of the default `UserDetailsService`, you can use the `exclude` attribute of the `@SpringBootApplication` annotation on the main class to disable this auto-configuration:

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

## 9.5 Filter implementations provided by Spring Security

In this section, we'll discuss classes provided by Spring Security, which implement the `Filter` interface. In the examples, in this chapter, we defined the filter by implementing the `Filter` interface directly. Spring Security offers a few abstract classes which implement the `Filter` interface and which you could extend to define your filters. These classes also add specific 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 you would define in a `web.xml` descriptor file (where applicable).

A more useful class which extends the `GenericFilterBean` is the `OncePerRequestFilter`. When adding a filter to the chain, the framework doesn't guarantee it will be called only once per request. The `OncePerRequestFilter`, as the name suggests, implements logic to make sure that the filter's `doFilter()` method is executed only one time per request.

So if you need such functionality in your application, go on and 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, which don't require the custom logic added by the `GenericFilterBean` class. When asking them why they've done so, it seemed they didn't know either: probably copied the implementation as they found it in examples on the web.

To make it crystal clear on how to use such a class, let's write an example. The logging functionality we implemented in section 9.3 makes a great candidate for using a `OncePerRequestFilter`. We want to avoid logging the same requests multiple times. Spring Security doesn't guarantee the filter won't be called more than one time, so we have to take care of this ourselves. The easiest way is to implement the filter using the `OncePerRequestFilter` class. I'll write this in a separate project called `ssia-ch9-ex3`.

In listing 9.9, you find the change I've done for the `AuthenticationLoggingFilter` class. Instead of implementing the `Filter` interface directly, as was the case of the example of section 9.3, now it extends the `OncePerRequestFilter` class. The method we override is `doFilterInternal()`.

### **Listing 9.9 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 Instead of implementing the `Filter` interface, we extend the `OncePerRequestFilter` class.

#B We override the `doFilterInternal()` method which replaces the purpose of the `doFilter()` method of the `Filter` interface.

#C The `OncePerRequestFilter` only supports HTTP filters. This is why the parameters are directly given as `HttpServletRequest` and `HttpServletResponse`.

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

- It supports only HTTP requests, but that's actually what we're always using. 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 ourselves the request and the response.
- You can implement logic to decide if the filter is applied or not. So, even if you have added the filter to the chain, you might decide it doesn't apply for certain requests. You can do this by overriding the `shouldNotFilter(HttpServletRequest)` method. By default, the filter applies to all the requests.
- By default, a `OncePerRequestFilter` doesn't apply to asynchronous requests or error dispatch requests. You can change this behavior by overriding the `shouldNotFilterAsyncDispatch()` and `shouldNotFilterErrorDispatch()` methods.

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

## 9.6 Summary

- The first layer of the web application architecture, which intercepts the HTTP requests, is a filter chain. As for other components in Spring Security architecture, you can customize it 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 the authentication and authorization to match precisely the requirements of your applications.

# 10

## *Applying CSRF protection and CORS*

### This chapter covers

- Implementing Cross-Site Request Forgery (CSRF) protection with Spring Security.
- Customizing CSRF protection.
- Applying Cross-Origin Resource Sharing (CORS) configurations.

Up to now, you have learned what the filter chain is and its purpose in the Spring Security architecture. We worked on several examples in which we customized the filter chain in chapter 9. But Spring Security also adds its own filters to the chain. In this chapter, we discuss the filter which applies CSRF protection and the one related to the CORS configurations. You'll learn to customize these filters so that the way they work is a perfect fit for your scenarios.

### 10.1 Applying CSRF protection in applications

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 the Cross-Site Request Forgery (CSRF) protection. The reason why you can't directly call an endpoint with HTTP POST is the CSRF protection, which is enabled by default in Spring Security.

In this section, we discuss the CSRF protection, and when to use it within your applications. Cross-Site Request Forgery is a widely spread type of attack, and applications vulnerable to CSRF could force the user to execute unwanted actions on a web application after the web apps have authenticated the users. You don't want the applications you develop to be CSRF vulnerable and allow attackers to trick your users into making unwanted actions, so 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 make use of it to call an endpoint with the HTTP

POST method. We'll prove this with a very small application using REST endpoints.

- Once you've learned how Spring Security implements its CSRF token mechanism for CSRF protection, we'll discuss how this mechanism is used in a real-world application scenario.
- Finally, you'll learn possible customizations of the CSRF token mechanism in Spring Security.

### 10.1.1 How CSRF protection works in Spring Security

In this section, we discuss how Cross-Site Request Forgery protection is implemented in Spring Security. It is important to first understand the underlying mechanism of CSRF protection before using it in your applications. I encountered many situations in which misunderstanding the way CSRF protection works leads the 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.

Explained with an example, you can consider the following scenario (figure 10.1):

- You are at work, and you use a web tool to store and manage your files. With this tool, from 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. You open the page, but the page is blank, or it redirects you to a known website.
- You go back to your work, but observe all your files are gone!

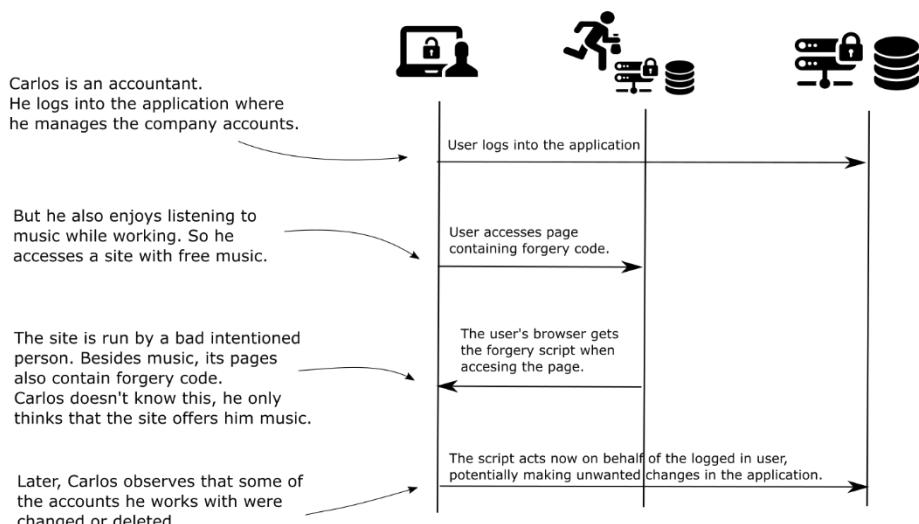


Figure 10.1 After the user logs into their account, they access a page containing forgery code. This code impersonates the user and might execute actions on behalf of the user.

What happened? You were logged into the application so you could manage your files. When you add, change or delete a file, the web page which you interact with calls some endpoints from the server to execute these operations. Instead, when you opened the foreign page by clicking the unknown link in the email, that page called the server and executed actions on your behalf (e.g. deleted your files). It could do that because you logged in previously in the application, so the server trusted the actions are coming from you. You maybe think that someone couldn't trick you so easy to click a link from a foreign mail or message, but trust me, this often happens to a lot of people. Most of the web app users out there aren't aware of security risks. So it's wiser you, who knows all these things protect them by building secure apps rather than rely on your apps' users to protect themselves.

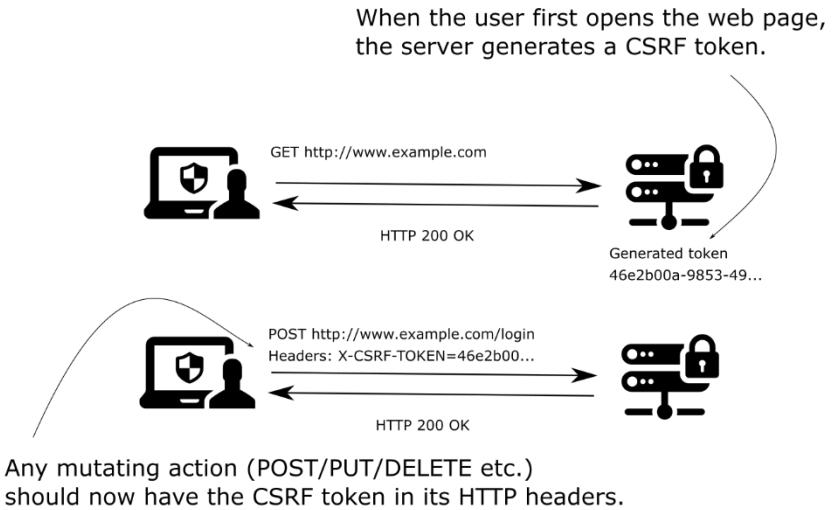
CSRF attacks assume that a user is logged into a web application. They're tricked by the attacker to open a page that contains scripts that execute actions in the same application the user was working on. Because the user has already logged in (as we've assumed from the beginning), the forgery code can now impersonate the user and do actions on their behalf.

How do we protect our users from such scenarios? What CSRF protection wants to ensure is that, if you have a web application, only the frontend of that application can perform mutating operations (by convention, HTTP methods other than GET, HEAD, TRACE or OPTIONS). Then, such a foreign page, like the one in our example, couldn't act on behalf of the user anymore.

How could we achieve this? What you know for sure is that, before being able to do any action that could change data, at least once, a user sends a request using HTTP GET to see the web page. When this happens, the application generates a unique token.

The application now accepts only requests for mutating operations (POST, PUT, DELETE, etc.) 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 etc, should receive through the response the CSRF token, and the page must use this token when making the mutating calls.

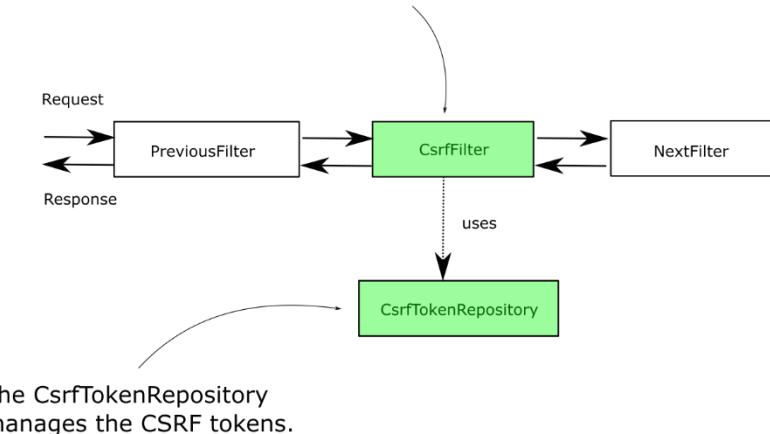
The starting point of the CSRF protection is a filter in the filter chain. This filter, which implements the CSRF protection logic, is called `CsrfFilter`. The `CsrfFilter` intercepts the requests and allows all those that use these HTTP methods: GET, HEAD, TRACE, and OPTIONS. For all the other requests, the filter expects to receive a header containing a token. If this header does not exist, or it contains an incorrect value of the token, the application rejects the request and sets the status of the response to HTTP 403 Forbidden. But 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 whenever you use any other method 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 10.2.



**Figure 10.2** 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 the requests that could be done from the loaded page. This way, only the page loaded can make mutable requests.

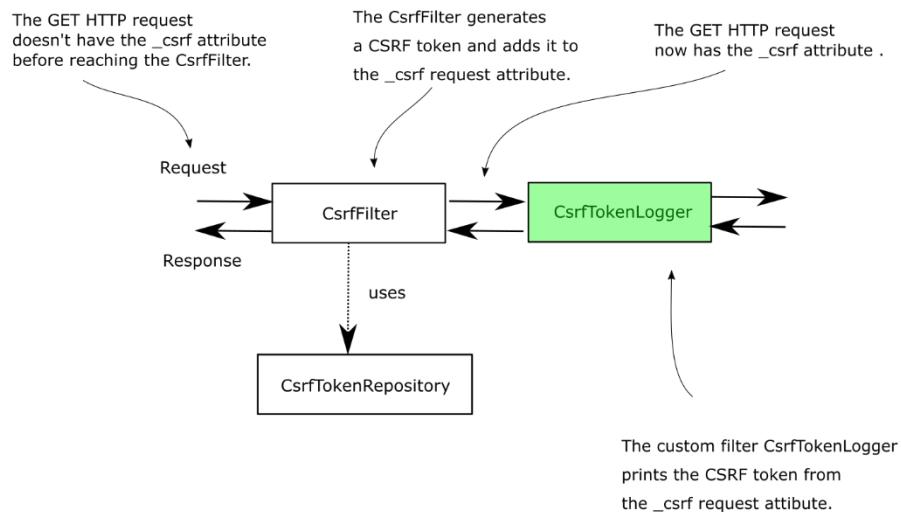
The `CsrfFilter` uses a component named `CsrfTokenRepository` to manage the CSRF token values: generate new tokens, store tokens, and eventually invalidate them. By default, the `CsrfTokenRepository` stores the token on the HTTP session, and the tokens are generated as random UUIDs. In most cases, this is enough, but, as you'll learn in section 10.1.3, you can use your own implementation of `CsrfTokenRepository` if the default one doesn't apply to the requirements you need to implement.

The CsrfFilter intercepts the request and applies the logic for CSRF protection.



**Figure 10.3** 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 the CSRF tokens, it uses a CsrfTokenRepository.

I have explained in this section how CSRF protection works in Spring Security with plenty of paragraphs and figures. But I want to enforce it with a small code example as well. You find this code as part of the project named `ssia-ch10-ex1`. So let's create an application that exposes two endpoints. One of them can be called 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 the CSRF protection. In this example, you'll learn how to call the POST endpoint without disabling the CSRF protection. To achieve this, you need to obtain somehow the CSRF token so that you can use it in the header of the call, which you do with HTTP POST. As you'll learn with this example, the `CsrfFilter` adds the generated CSRF token to the attribute of the HTTP request named `_csrf` (figure 10.4). If we know this, we know that after the `CsrfFilter`, we find this attribute, and we can take the value of the token out of it. For this small application, we'll choose to add a custom filter after the `CsrfFilter`, as you've learned in chapter 9. 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 (figure 10.4). We'll then be able to copy the value of the token from the console and use it to make the mutating call with HTTP POST.



**Figure 10.4** We add the **CsrfTokenLogger** (differently 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** stored it. The **CsrfTokenLogger** prints the CSRF token in the application console, where we'll take it and use it to call an endpoint with the **HTTP POST** method.

In listing 10.1, you find the definition of the controller class with the two endpoints we use for a test.

#### Listing 10.1 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 10.2 presents the definition of the custom filter we'll use to print in the console the value of the CSRF token. I'll name 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 `_csrf` request attribute is the name of the request attribute 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 10.2 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 {
        Object o = request.getAttribute("_csrf");      #A
        CsrfToken token = (CsrfToken) o;

        logger.info("CSRF token " + token.getToken());

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

#A The value of the token is taken from the `_csrf` request attribute and printed in the console.

In the configuration class, we add the custom filter. Listing 10.3 presents the configuration class. Observe that I didn't disable the CSRF protection.

#### **Listing 10.3 Adding the custom filter in the configuration class**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.addFilterAfter(
            new CsrfTokenLogger(), CsrfFilter.class)
            .authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

You can start now to 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'll also need to remember the session id. For this reason, I've added the `-v` flag so I can see more details from the response, including the session id.

Calling the endpoint:

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

The (truncated) response is:

```
...
< 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 : CSRF token c5f0b3fa-
2cae-4ca8-b1e6-6d09894603df
```

**NOTE** You might ask yourself, how do clients get the CSRF token in the end? For sure, they can't either guess it or read the server logs. I designed this example such that it's easier for you to understand how the CSRF protection implementation works. As you'll find further in this chapter, in section 10.1.2, 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 will be 403 Forbidden.

Calling the endpoint with HTTP POST without the CSRF token:

```
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 will be successful. You also need to specify the session ID, because the default implementation of the `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: 1127bfda-57b1-43f0-bce5-bacd7d94694e'
```

The response body is:

```
Post Hello!
```

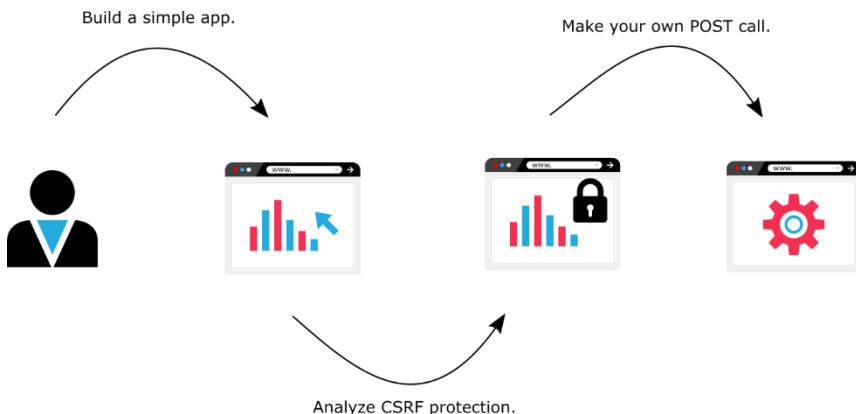
### 10.1.2 Using CSRF protection in practical scenarios

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'll use CSRF protection for the web apps running in a browser, where you expect that mutating operations can be done only by the browser that loaded the display content of the web app. The most

basic example I can provide here is a simple web application developed on the standard Spring MVC flow. We've already made such an application when discussing the Form Login in chapter 5. And that web app from chapter 5 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 it was because we didn't develop any mutating operation within it ourselves. For the default 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 to how CSRF protection works. In this section, we'll

1. build an example of a web application with the login form
2. look at how the default implementation of login uses CSRF tokens
3. implement an HTTP POST call from the main page.



**Figure 10.5 The plan:** In this chapter, we'll start by building and analyzing a simple app to understand how CSRF protection is applied. Then, we'll write our own POST call.

You'll spot 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 a project named `ssia-ch10-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. In listing 10.4, you find the definition of the configuration class, which defines the `UserDetailsService` and adds a user and, as well, configures the form login method.

#### **Listing 10.4 The definition of the configuration class**

```
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @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();
    }

    @Override      #C
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated();

        http.formLogin()
            .defaultSuccessUrl("/main", true);
    }
}
```

#A We add a `UserDetailService` bean with one user to test the application.

#B We add a `PasswordEncoder`.

#C We override the `configure()` method to set the Form Login authentication method and specify that only authenticated users can access any of the endpoints.

We add a controller class for the main page in a package named `controllers` and a `main.html` file in the `resources/templates` folder of the maven project. The `main.html` file can remain empty for the moment as on the first execution of the application, we'll only focus on how the login page uses the CSRF tokens. Listing 10.5 presents the `MainController` class, which serves the main page.

#### **Listing 10.5 The definition of the MainController class**

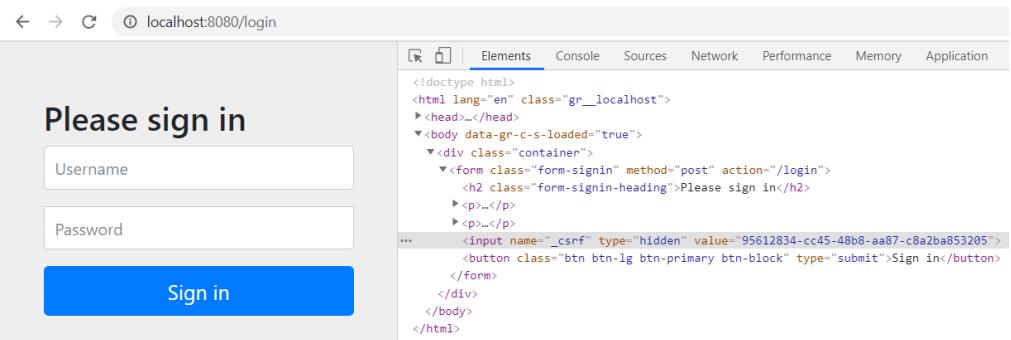
```
@Controller
```

```
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'll 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 10.6 shows how the login form sends the CSRF token through a hidden input.



**Figure 10.6** The default form login uses a hidden input to send the CSRF token in the request. This is why the login request which uses HTTP POST method works with the CSRF protection enabled.

But what about developing our own endpoints which 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'll call this endpoint from the main page. We'll create a second controller for this, which I'll call `ProductController`. Within this controller, we'll define an endpoint `/product/add` which uses HTTP POST. We'll use a form on the main page to call this endpoint. Listing 10.6 presents the definition of the `ProductController` class.

#### **Listing 10.6 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. Listing 10.7 shows the definition of the form defined in the `main.html` file.

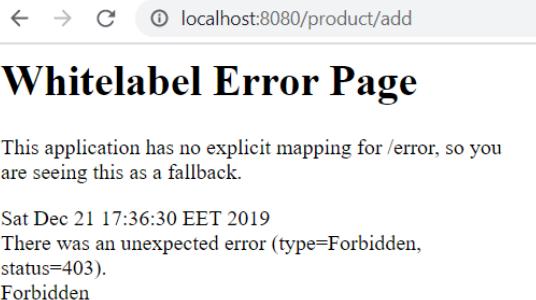
#### **Listing 10.7 the definition of the form in the `main.html` page**

```

<form action="/product/add" method="post">
    <span>Name:</span>
    <span><input type="text" name="name" /></span>
    <span><button type="submit">Add</button></span>
</form>

```

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 10.7). The reason for the HTTP 403 Forbidden status is the absence of the CSRF token.



**Figure 10.7** 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 have to add the CSRF token in the request done through the form. An easy way is to use a hidden input component as you've seen the default form login also does. You can implement this as presented in listing 10.8.

#### **Listing 10.8 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>
```

#A We use a hidden input to add to the request the CSRF token.  
#B The "th" prefix we use is interpreted by Thymeleaf to print the token value.

**NOTE** In the above example, we used Thymeleaf just because it provides a straightforward way to obtain the request attribute value in the view. In our case, we needed to print the CSRF token. We learned in this section that the `CsrfFilter` added 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 logline in the console, proving that the execution succeeded. Also, if you inspect the form, you'll find the hidden input with the value of the CSRF token (figure 10.8).

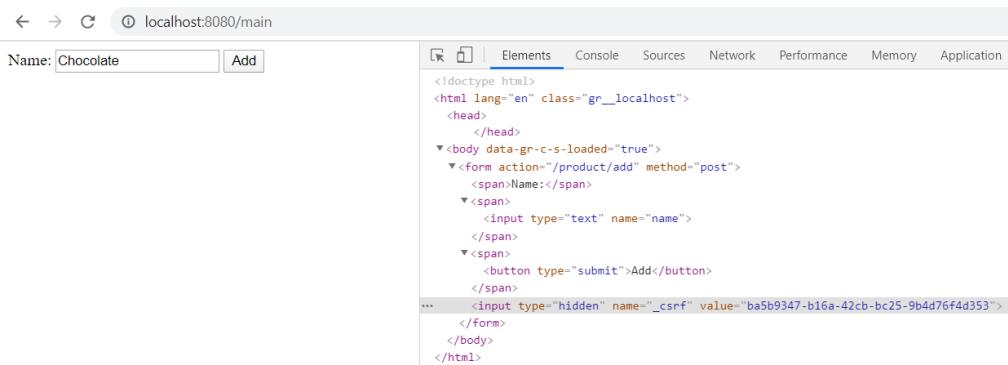


Figure 10.8 The form defined on the main page now sends the value for the CSRF token in the request. This way, the server allows the request and executes the controller action. In the source of the page, you can now find the hidden input used by the form to send the CSRF token in the request.

After submitting the form, you should find in the application console a line similar to the one in the next code snippet.

```
INFO 20892 --- [nio-8080-exec-7] c.l.s.controllers.ProductController : Adding product
Chocolate
```

Of course, for any action or asynchronous Javascript request, your page uses to call a mutable 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 instead. A third party request would try to impersonate the user to execute actions on their behalf.

The CSRF tokens work well in an architecture where the same server is responsible for both frontend and the backend mainly for its simplicity. The CSRF tokens don't work well anymore when the client is independent of the backend solution it consumes. This scenario happens whether 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 today 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 more in chapters 11 to 15.

In chapter 11, we'll work on a hands-on application where we'll solve the requirement of implementing a web application where separate web servers independently support the frontend and the backend solutions. For that example, we'll analyze the applicability of CSRF protection with tokens again.

In chapters 12 to 15, you'll learn to implement the OAuth2 specification, which has excellent advantages in decoupling the component making the authentication from the resources for which the application authorizes the client.

**NOTE** It might look a trivial mistake, but in my experience, I've seen it too many times in applications: never use HTTP GET with mutating operations. Do not implement behavior that changes data and allow it to be called under an HTTP GET endpoint. Remember that the calls to HTTP GET endpoints don't require a CSRF token.

### 10.1.3 Customizing CSRF protection

In this section, you'll 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 makes no exception. In this section, you'll apply with examples the most often encountered needs in customization of the CSRF protection mechanism:

- the configuration of the paths for which the CSRF applies
- the management of the 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 discuss in section 10.2, or a mobile application. In the case of mobile applications, you can use the OAuth2 flow, which we'll discuss in chapters 12 to 15.

By default, the CSRF protection applies to any path for endpoints called with other HTTP methods than GET, HEAD, TRACE, or OPTIONS. You already know (from chapter 7) how to completely disable CSRF protection. But what if you want to disable it only for a part of the paths of your application? You can do this configuration very quickly with a `Customizer` object, similar to the way we've customized the HTTP Basic for Form Login methods in chapter 3. Let's try it with an example. We'll create a new project and add only the web and security dependencies, as presented in the next code snippet. You find this example as project `ssia-ch10-ex3`.

```
<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'll add two endpoints called with HTTP POST, but we want to exclude one of them from using the CSRF protection.

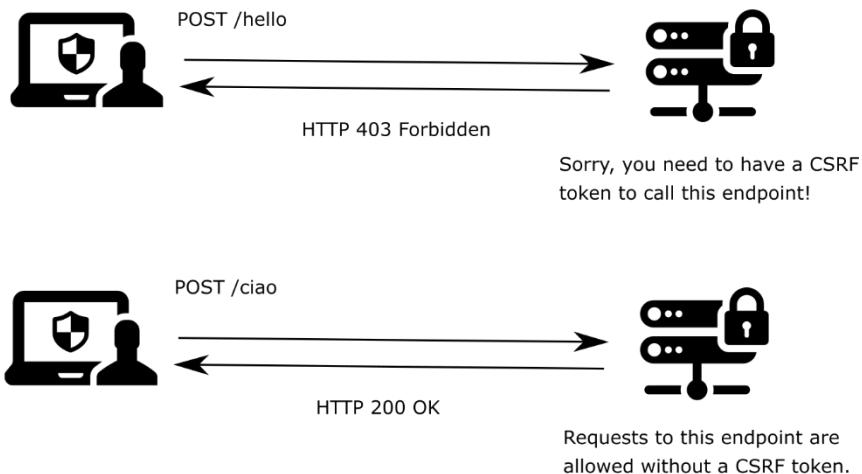


Figure 10.9 The application requires a CSRF token for the /hello endpoint called with HTTP POST but allows the HTTP POST requests to the /ciao endpoint without the need of a CSRF token.

In listing 10.9, you find the definition of the controller class, which I'll name `HelloController`.

#### Listing 10.9 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";
    }
}
```

#A The /hello path remains under CSRF protection. You can't call the endpoint without a valid CSRF token.

**#B** The `/ciao` will be called without needing a CSRF token.

To make customizations on the CSRF protection, you can use the `csrf()` method of the `HttpSecurity` object in the `configuration()` method with a `Customizer` object. Listing 10.10 presents this approach.

#### Listing 10.10 Using a Customizer for the configuration of the CSRF protection

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.csrf(c -> {      #A
            c.ignoringAntMatchers("/ciao");
        });

        http.authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

**#A** The parameter of the lambda expression is a `CsrfConfigurer`. By calling its methods, you can do various configurations for the CSRF protection.

Calling the `ignoringAntMatchers(String... paths)` method, you can specify the ANT expressions representing the paths that you want to exclude from the CSRF protection mechanism. A more general approach is to use a `RequestMatcher`. Using a `RequestMatcher` allows you to apply the exclusion rules with regular MVC expressions as well as using regex. By 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 a `MvcRequestMatcher` instead of `ignoringAntMatchers()`.

```
HandlerMappingIntrospector i = new HandlerMappingIntrospector();
MvcRequestMatcher r = new MvcRequestMatcher(i, "/ciao");
c.ignoringRequestMatchers(r);
```

You can similarly use a regex matcher as presented in the next code snippet.

```
String pattern = ".*[0-9].*";
String httpMethod = HttpMethod.POST.name();
RegexRequestMatcher r = new RegexRequestMatcher(pattern, httpMethod);
c.ignoringRequestMatchers(r);
```

Another need often found in the requirements of the applications is the customization of the management of the CSRF tokens. As you've learned, by default, the application stores the 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 to be horizontally scaled. The HTTP session is stateful and reduces the scalability of the application. Let's suppose you want to change the way the application manages the tokens

and store them somewhere in a database rather than the HTTP session. Spring Security offers two contracts that you need to implement to customize the management of the CSRF tokens:

- The `CsrfToken` interface, which describes the CSRF token itself.
- The `CsrfTokenRepository`, which describes the object that creates, stores and loads the CSRF tokens.

The `CsrfToken` object has three main characteristics which you have to specify when implementing the 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 which stores the value of the token (default named `_csrf`)
- the value of the token

Listing 10.11 presents the definition of the `CsrfToken` contract.

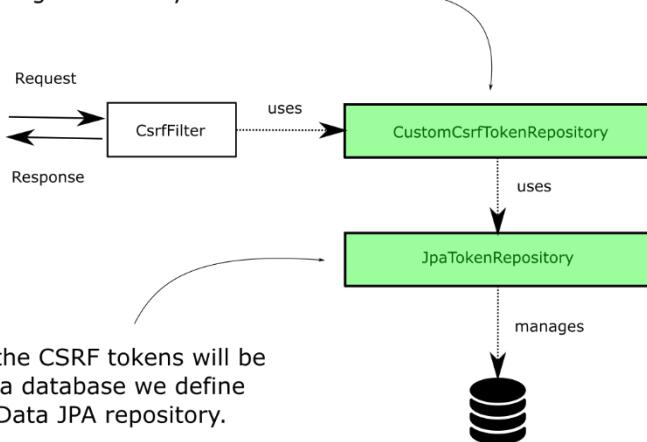
#### **Listing 10.11 The definition of the `CsrfToken` interface**

```
public interface CsrfToken extends Serializable {
    String getHeaderName();
    String getParameterName();
    String getToken();
}
```

Generally, the only need is for the instance of `CsrfToken` type to store these three details in attributes of the instance. For this functionality, Spring Security offers an implementation called `DefaultCsrfToken`, which we'll also use in our example. The `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 `CsrfTokenRepository` is responsible for the management of the CSRF tokens in Spring Security. The interface `CsrfTokenRepository` is the contract that represents the component that manages the CSRF tokens. To change the way the application manages the tokens, you have to implement the `CsrfTokenRepository` interface, which allows you to plug in your custom implementation in the framework. Let's change the current application we've been working on in this section to add a new implementation for `CsrfTokenRepository`, which stores the tokens in a database. Figure 10.10 presents the components we'll implement for this example and the link between them.

We define a custom `CsrfTokenRepository` to manage differently the CSRF tokens.



Because the CSRF tokens will be stored in a database we define a Spring Data JPA repository.

**Figure 10.10** The `CsrfToken` uses a custom implementation of `CsrfTokenRepository`. This custom implementation uses a `JpaReposiroty` to manage the CSRF tokens into a database.

In our example, we'll use a table in a database to store the CSRF tokens. We assume the client has an identifier to identify themselves uniquely. The application needs this identifier to obtain the CSRF token and validate it. Generally, this unique identifier would be obtained during the login and should be different each time the user logs in. This strategy of managing the tokens is very similar to storing them in memory. In the case of storing them in memory, you had a session ID. So this new identifier we chose 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, the tokens would expire after a time you define. You would store in the database the tokens without linking them to a specific identifier of the user. You would only need to check if a token provided via the HTTP request exists and is not expired to decide whether you allow the HTTP request.

**Exercise.** Once you've finished 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'll only focus on the implementation of the `CsrfTokenRepository`, and we consider that the client already has a generated identifier to identify them uniquely.

To work with the database, we'll have to add a couple more dependencies to the `pom.xml` file, as presented in the next code snippet.

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>

```

```
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

In the `application.properties` file of the application, we need to add the properties need for the database connection.

```
spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-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 the next 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'll 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 listing 10.12.

#### **Listing 10.12 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
}
```

#A The identifier of the client.

#B The CSRF token generated by the application for the client.

The `JpaTokenRepository`, which is our `JpaRepository` contract, can be defined as presented by listing 10.13. The only method you'll need is the `findTokenByIdentifier()`, which gets the CSRF token from the database for a specific client.

#### **Listing 10.13 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 start now to write the `CsrfTokenRepository` implementation, which I'll call `CustomCsrfTokenRepository`. Listing 10.14 presents the definition of the `CustomCsrfTokenRepository` class, which overrides the three methods of `CsrfTokenRepository`.

#### **Listing 10.14 The implementation of the `CsrfTokenRepository` contract**

```

public class CustomCsrfTokenRepository implements CsrfTokenRepository {

    @Autowired
    private JpaTokenRepository jpaTokenRepository;

    @Override
    public CsrfToken generateToken(
        HttpServletRequest httpServletRequest) {
        // ...
    }

    @Override
    public void saveToken(
        CsrfToken csrfToken,
        HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse) {
        // ...
    }

    @Override
    public CsrfToken loadToken(
        HttpServletRequest httpServletRequest) {
        // ...
    }
}

```

The `CustomCsrfTokenRepository` injects an instance of `JpaTokenRepository` from the Spring context to gain access to the database. The `CustomCsrfTokenRepository` will use this instance to retrieve or save the CSRF tokens in the database. The CSRF protection mechanism calls `generateToken()` method when the application needs to generate a new token. In listing 10.15, you find the implementation of this method for our current exercise. We use the `UUID` class to generate a new random UUID value, and we keep the same names for the header and attribute of the request as in the default implementation offered by Spring Security: `X-CSRF-TOKEN` and `_csrf`.

#### **Listing 10.15 The implementation of the `generateToken()` method**

```

@Override
public CsrfToken generateToken(HttpServletRequest httpServletRequest) {
    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 began assuming that the client has a unique identifier. We consider the client sends the value of its unique identifier 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 with the new value of the token. If not, we create a new record for this identifier with the new value of the CSRF token. Listing 10.16 presents the implementation of the `saveToken()` method.

#### **Listing 10.16 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);
    }
}
```

#A We try to obtain the token from the database by client identifier.

#B If the identifier exists, we update the value of the token with the newly generated value

#C If the identifier doesn't exist we create a new record for the identifier with a generated value for the CSRF token.

The `loadToken()` method implementation, presented in listing 10.17, loads the token details if they exist or returns null otherwise.

#### **Listing 10.18 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",
            token.getToken(),
            token.getIdentifier(),
            token.isExpired());
    }
}
```

```

        "_csrf",
        token.getToken());
    }

    return null;
}

```

We use the custom implementation of the `CsrfTokenRepository` to declare a bean in the configuration class. We then plug in the bean in the CSRF protection mechanism with the `csrfTokenRepository()` method of the `CsrfConfigurer`. Listing 10.19 presents the definition of the configuration class.

### **Listing 10.19 The configuration class**

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Bean      #A
    public CsrfTokenRepository customTokenRepository() {
        return new CustomCsrfTokenRepository();
    }

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {

        http.csrf(c -> {      #B
            c.csrfTokenRepository(customTokenRepository());
            c.ignoringAntMatchers("/ciao");
        });

        http.authorizeRequests()
            .anyRequest().permitAll();
    }
}

```

#A We define the `CsrfTokenRepository` as a bean in the context.

#B We use the `Customizer<CsrfConfigurer<HttpSecurity>>` object to plug the new `CsrfRepository` implementation in the CSRF protection mechanism

In the definition of the controller class presented in listing 10.20, we also add an endpoint that uses the HTTP GET method. We need this method to obtain the CSRF token when testing our implementation firstly.

```

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

```

You can now start the application and test the new implementation for token management. We call the endpoint using HTTP GET to obtain a value for the CSRF token. When making the call, we have to use an identifier of the client within the `X-IDENTIFIER` header as assumed from the requirement. A new value of CSRF token is generated and stored in the database.

```

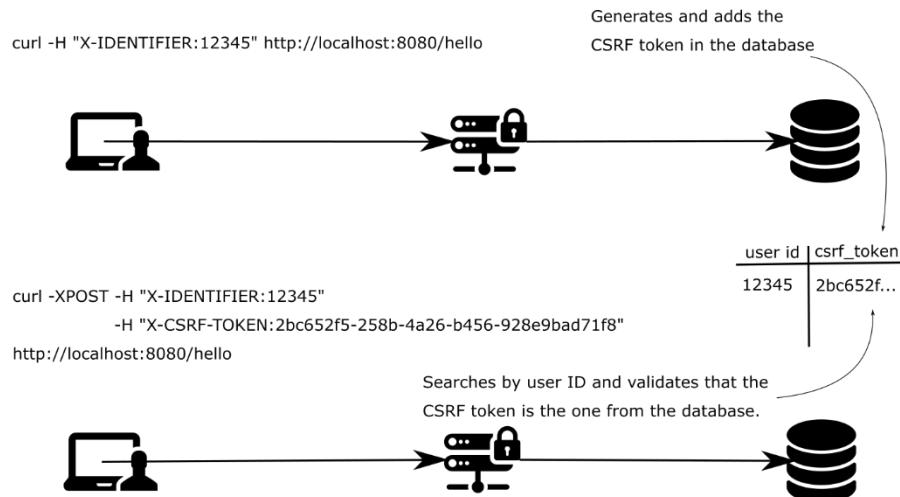
curl -H "X-IDENTIFIER:12345" http://localhost:8080/hello
Get Hello!

```

We search in the token table in the database and 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 presented in the next code snippet. Of course, we have to provide also the identifier for the client, used by the application to retrieve the token from the database to compare with the one we provide in the request.

```
curl -XPOST -H "X-IDENTIFIER:12345" -H "X-CSRF-TOKEN:2bc652f5-258b-4a26-b456-928e9bad71f8"
      http://localhost:8080/hello
Post Hello!
```

In figure 10.11, you find the flow described.



**Figure 10.11** First, the GET request generates the CSRF token and stores its value in the database. Any following POST request must send this value. 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 will get a response back with the HTTP status 403 Forbidden.

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

The response body is :

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
```

}

## 10.2 Using Cross-Origin Resource Sharing (CORS)

In this section, we discuss Cross-Origin Resource Sharing (CORS) and how to apply it with Spring Security. Firstly, what is CORS, and why should you care? The necessity for CORS came from web applications ran in browsers. By default, the browsers don't allow requests made for another domain than the one from which the site was loaded. For example, if you access the site from `example.com`, the browser won't let the site make requests to `api.example.com`, as presented in figure 10.12.

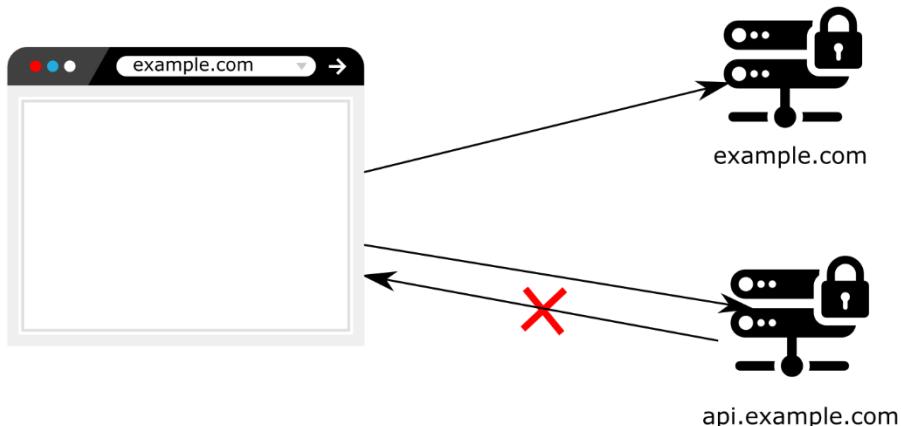
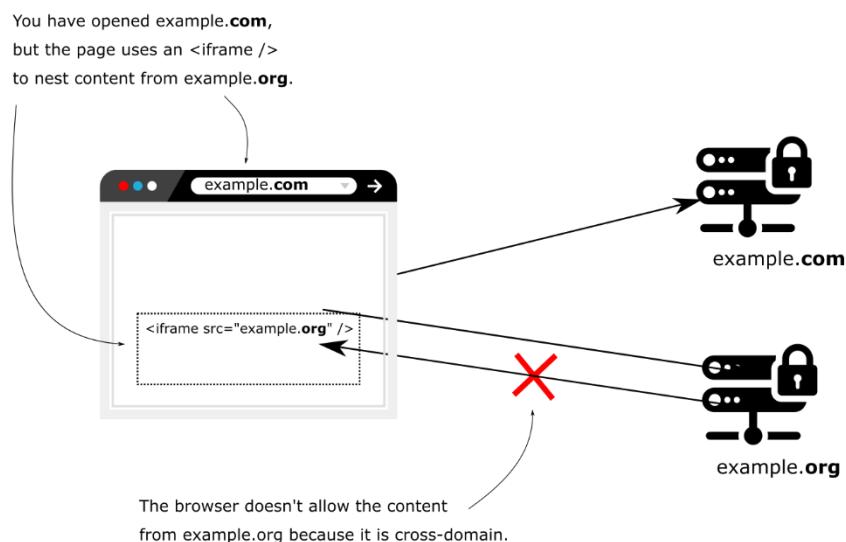


Figure 10.12 Cross-Origin Resource Sharing works like this: When accessed from `example.com`, the website cannot make requests to `api.example.com` because it's cross-domain.

We can briefly say that CORS is a mechanism used by a browser to relax this strict policy and allow requests made between different origins in some conditions. You need to learn this because there's a high chance that you'll have to apply it with applications, especially nowadays, where usually the frontend and the backend are separate applications. It is very common today that a frontend application is developed using a framework like Angular, ReactJS, or Vue to be hosted at a domain like `example.com`, but calls endpoints on the backend hosted at another domain like `api.example.com`. We'll develop some examples from which you'll learn how to apply CORS policies for your web applications, as well as some details that you need to know such that you avoid leaving security breaches in your applications.

### 10.2.1 How does CORS work?

In this section, we discuss how Cross-Origin Resource Sharing (CORS) applies to web applications. If you are the owner of `example.com`, and for some reason, the developers from `example.org` decide to call your REST endpoints from their website, they won't be able to do that. The same could happen if some domain loads your application using an `iframe`, for example (figure 10.13).



**Figure 10.13** 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 fulfill. Even if the application makes a request, the browser won't accept the response.

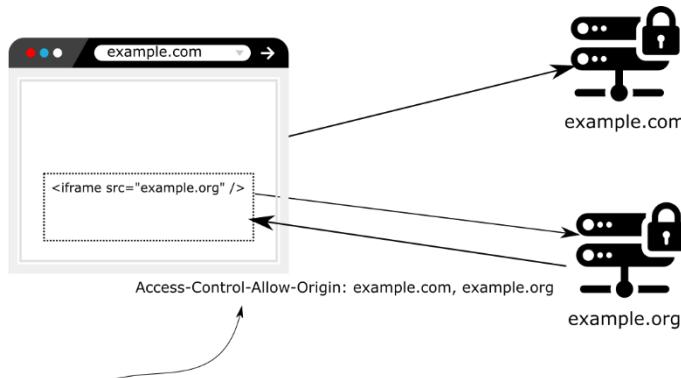
**NOTE** An `iframe` is an HTML element you use to embed content generated by a web page into another web page. For example, integrate the content from `example.org` inside the page of `example.com`.

Any situation in which an application makes calls between two different domains is prohibited. But of course, you'll find cases in which you need to do such calls between two different domains. In these situations, CORS allows you to specify from which domain your application allows the requests and what details can be shared. The CORS mechanism works based on HTTP headers out of which the most important are:

- `Access-Control-Allow-Origin`, which we'll use to specify the foreign domains (origins) that can access resources on our domain.
- `Access-Control-Allow-Methods`, which allow us to refer only to some HTTP methods in situations in which you want to allow access to a different domain, but only to specific HTTP methods. You'll 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`, which you use to add limitations to which headers can be used in a specific request.



`example.org` specifies in the response header what origins accepts.

The browser accepts and displays the content.

**Figure 10.14 To enable the cross origin requests:** The `example.org` server adds the `Access-Control-Allow-Origin` 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.

By default, in Spring Security, none of these headers are added to the response. So let's start with the beginning: what happens when you make a cross-origin call if you configure nothing about CORS in your application. When the application makes the request, it expects that the response has a header `Access-Control-Allow-Origin` containing the origins accepted by the server. If this doesn't happen, as in the case of the default Spring Security behavior, the browser won't accept the response. Let's demonstrate this with a small web application. We'll create a new project using the dependencies presented by the next code snippet. You find this example in project `ssia-ch10-ex4`.

```
<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'll 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, for the REST endpoint, we also

have to add the `@ResponseBody` annotation explicitly. Listing 10.20 presents the definition of the controller.

#### **Listing 10.20 The definition of the controller class**

```
@Controller
public class MainController {

    private Logger logger = #A
        Logger.getLogger(MainController.class.getName());

    @GetMapping("/")
    public String main() {
        return "main.html";
    }

    @PostMapping("/test")
    @ResponseBody
    public String test() { #C
        logger.info("Test method called");
        return "HELLO";
    }
}
```

#A We'll use a logger to observe when the `test()` method is called.

#B We define a `main.html` page which will make the request to the `/test` endpoint

#C The `test()` method defines an endpoint, which we'll call from a different origin to prove how CORS works.

Further, we need to define the configuration class where we'll disable the CSRF protection, to make the example simpler and allow you to only focus on the CORS mechanism. Also, we'll allow unauthenticated access to all the endpoints. Listing 10.21 presents the definition of the configuration class.

#### **Listing 10.21 The definition of the configuration class**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();

        http.authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

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 the browser using the domain `localhost`, but 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 will see them as different strings, and will consider them different domains. Listing 10.22 presents the definition of the `main.html` page.

**Listing 10.22 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>
```

#A We make the call to the endpoint using 127.0.0.1 as host to simulate the cross-origin call.

#B The response body is set to the output div in the page body.

Starting the application and opening the page in the browser with `localhost:8080`, we observe that the page doesn't display anything. We've expected to see a `HELLO` on the page, as 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 in the next code snippet.

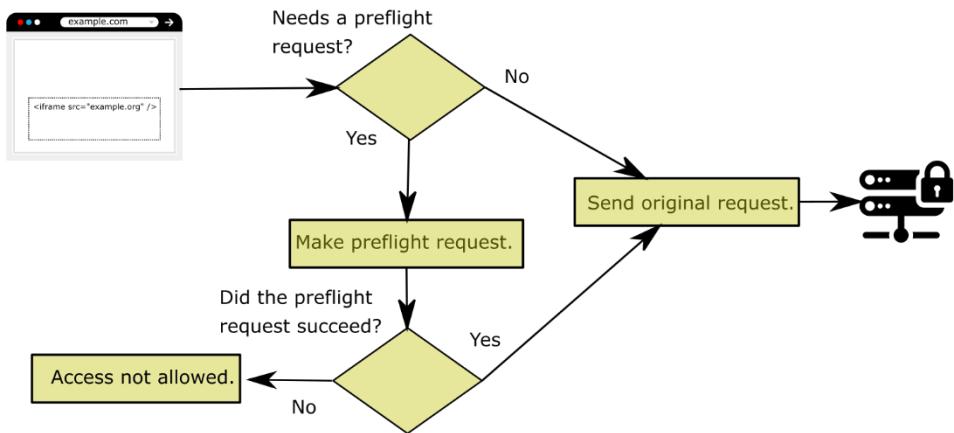
```
Access to XMLHttpRequest at 'http://127.0.0.1:8080/test' from origin
  'http://localhost:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

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 behavior of the browser of not accepting to display 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'll find in the application console.

```
INFO 25020 --- [nio-8080-exec-2] c.l.s.controllers.MainController : Test method
called
```

This aspect is important! I've met many developers who understand CORS as a restriction similar to authorization or CSRF protection. Instead of being a restriction similar to CSRF, CORS helps to relax a rigid restriction for cross-domain calls. And even with the restrictions applied, in some situations, the endpoint could happen to be called. This behavior doesn't always happen. Sometimes, the browser first makes a call using the HTTP OPTIONS method to test whether the actual request would be allowed. We call this test request a preflight request. If the preflight request fails, the browser won't try at all to make 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 10.15 presents this request flow.



**Figure 10.15** 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.

The browser omits to make the preflight request if the HTTP method is GET, POST, or OPTIONS, and it only has some basic headers as described in the official documentation: <https://www.w3.org/TR/cors/#simple-cross-origin-request-0>. In this case, the browser will make the request but won't accept the response if the origin is not specified in the response, as presented in figures 10.9 and 10.10.

But the CORS mechanism is, in the end, related to the browsers and not a way to securing the endpoints from not being called by anybody. The only thing it guarantees you is that from a browser, only origin domains that you allow can make requests from their pages in the browser.

### 10.2.2 Applying CORS policies with the @CrossOrigin annotation

In this section, we'll discuss how to configure CORS to allow requests from different domains, which we allow using the `@CrossOrigin` annotation. You can place the `@CrossOrigin` annotation directly above the method which defines the endpoint and configure using it the allowed origins and methods. As you'll learn in this section, the advantage of using the `@CrossOrigin` annotation is that it makes it very easy to configure CORS per each endpoint.

We'll use the application we've created in section 10.2.1 to demonstrate how `@CrossOrigin` annotation works.

To make the cross-origin call work in the application, the only thing you need to do is add the `@CrossOrigin` annotation over the `test()` method in the controller class. Listing 10.23 shows how to use the annotation to make `localhost` an allowed origin.

#### **Listing 10.23 Making localhost an allowed origin**

```
@PostMapping("/test")
@ResponseBody
@CrossOrigin("http://localhost:8080")      #A
public String test() {
    logger.info("Test method called");
    return "HELLO";
}
```

#A We make the localhost origin allowed for the cross-origin requests.

You can rerun and test the application, which should now display on the page the string returned by the `/test` endpoint: HELLO.

You can specify multiple allowed origins. The value parameter of the `@CrossOrigin` annotation 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 eventually 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, as we discussed in chapter 1. I do personally avoid to allow all the origins even on test environments. I know that applications sometimes happen to be run on wrongly defined infrastructures that use the same datacenters for both test and production. It is wiser to treat all the layers on which security applies independently, as we discussed in chapter 1, and avoid assuming that the application doesn't have particular vulnerabilities because the infrastructure doesn't allow it.

The advantage of using the `@CrossOrigin` annotations 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 10.2.3, we'll discuss applying the CORS configuration centralized within the configuration class.

#### **10.2.3 Applying CORS using a CorsConfigurer**

In this section, we discuss applying the CORS configuration centralized within the configuration class. While using the `@CrossOrigin` annotation is easy as you've learned in section 10.2.2 in a lot of cases, you'll find more comfortable to define the CORS configuration in one place. In this section, we'll change the example we've worked on in sections 10.2.1 and 10.2.2 to apply

the CORS configuration in the configuration class using a `Customizer`. In listing 10.24, you can find the changes we have to make in the configuration class to define the origins we allow.

#### **Listing 10.24 Defining CORS configurations centralized in the configuration class**

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) 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"));
                return config;
            };
            c.configurationSource(source);
        });

        http.csrf().disable();

        http.authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

#A We call the `cors()` method to define the CORS configuration. Within it we create a `CorsConfiguration` object where we set the allowed origins and methods.

The `cors()` method, which we call from the `HttpSecurity` object, receives as parameter a `Customizer<CorsConfigurer>` object. For this object, we set a `CorsConfigurationSource`, which has the purpose of returning a `CorsConfiguration` for an HTTP request. The `CorsConfiguration` is the object that states, which are the allowed origins, methods, and headers. Mind that if using 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 as default.

In this example, to make the explanation straightforward, I provided the implementation for the `CorsConfigurationSource` as a lambda expression in the `configure()` method directly. I strongly recommend in your applications to separate this code in a different class. In real-world applications, you could have a much longer code, so it'll become difficult to read if not separated by the configuration class.

## **10.3 Summary**

- 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 may 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.
- Cross-Over Resource Sharing (CORS) refers to the situation in which a web application hosted on a specific domain (example.com) tries to access content from another domain (example.org). Default, the browser doesn't allow this 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 per endpoint using the `@CrossOrigin` annotation or centralized in the configuration class using the `cors()` method of the `HttpSecurity` object.

# 11

## *Hands-on: A separation of responsibilities*

### This chapter covers

- Implementing and using tokens.
- Working with JSON Web Tokens (JWTs).
- Separating authentication and authorization responsibilities in multiple apps.
- Implementing a multi-factor authentication (MFA) scenario.
- Using multiple custom filters and multiple `AuthenticationProvider` objects.
- Choosing from various possible implementations for a scenario.

We've come a long way, and you're now in front of the second hands-on chapter of the book. It's time again to put in action all you've learned in an exercise which will show you the big picture. Fasten your seat belts, open your IDEs, and get ready for a juicy activity! We'll design a system of three actors: the client, the authentication server, and the business logic server. Out of these three actors, we'll implement the backend part represented by the authentication server and the business logic server. As you observe, advancing with the book, our examples become more complex – this is the sign we get closer and closer to the real-world scenarios.

This exercise is a great chance to recap, apply, and understand better what you've already learned, and is also an excellent chance to touch new subjects, like JSON Web Tokens (JWTs). You'll also see a first demonstration of separating the authentication and authorization responsibilities in a system. We'll extend this discussion in chapters 12 to 15, with the OAuth 2 framework. This example resembles the ones we'll discuss in-depth in chapters 12 to 15 about OAuth2. Getting closer to what we'll discuss in the next chapters is one of the reasons for the design I've chosen for this exercise.

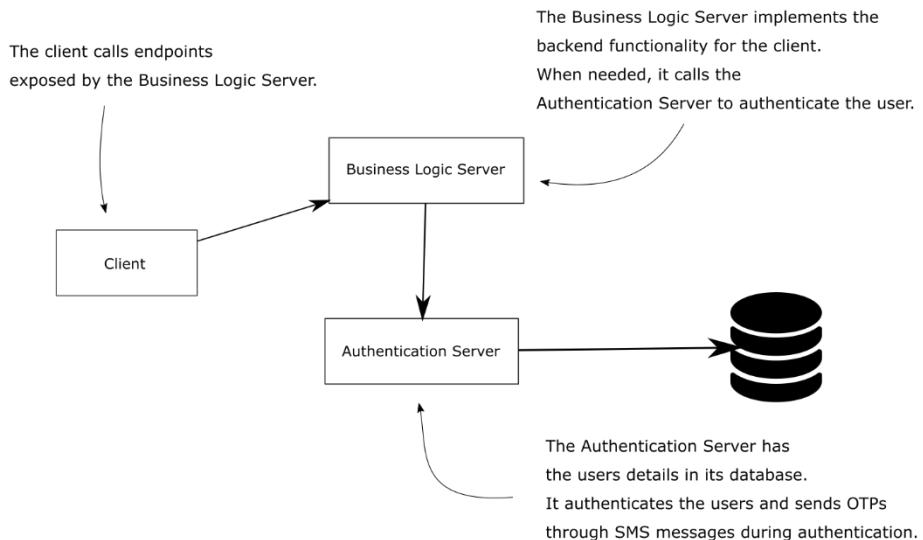
## 11.1 The scenario and requirements of the example

In this section, we discuss the requirements for the applications we'll develop together throughout the chapter. Once you understand what has to be done, in section 11.2, we discuss how to implement the system and which are our best options. Then, in sections 11.3 and 11.4, we get our hands dirty with Spring Security and implement the scenario from head to foot.

The architecture of the system has three components:

- *The Client* – the application consuming our backend – could be a mobile app or a frontend app of a web application developed using a framework like Angular, ReactJS, or Vue.js. We don't implement the client part of the system, but keep in mind that it exists in a real-world application. Instead of using the client to call the endpoints, we will use `curl`.
- *The Authentication Server* – This is an application having a database with user credentials. The purpose of this application is to authenticate users based on their credentials (username and password) and send them a one-time password through SMS. In this chapter, we implement this whole application without the part of sending the SMS. Later, you can also extend it to send messages using a service of your choice, like AWS SNS (<https://aws.amazon.com/sns/>), Twilio (<https://www.twilio.com/sms>), or others. We'll resume reading the generated one-time password from our database for the tests.
- *The Business Logic Server* – This is the application exposing endpoints that our client consumes. We want to secure access to these endpoints. Before being able to call any endpoint, the user must authenticate, first by using their username and password, and then by sending the one-time password (OTP). The users receive the OTP through an SMS message. This application is our target application – the one we secure with Spring Security.

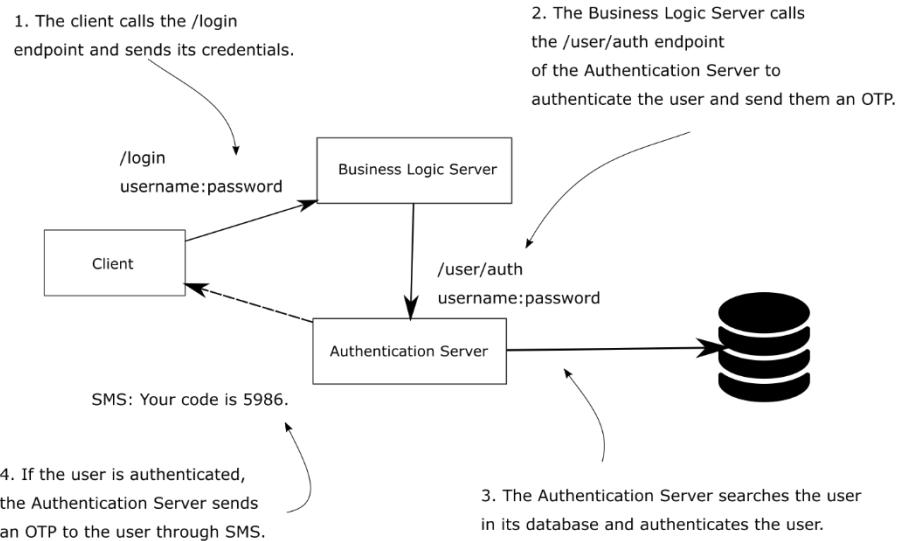
Figure 11.1 presents the three components of the system.



**Figure 11.1** The client calls the endpoints exposed by the Business Logic Server. To authenticate the user, the Business Logic Server uses the responsibility implemented by the Authentication Server. The Authentication Server keeps the users' credentials in its database.

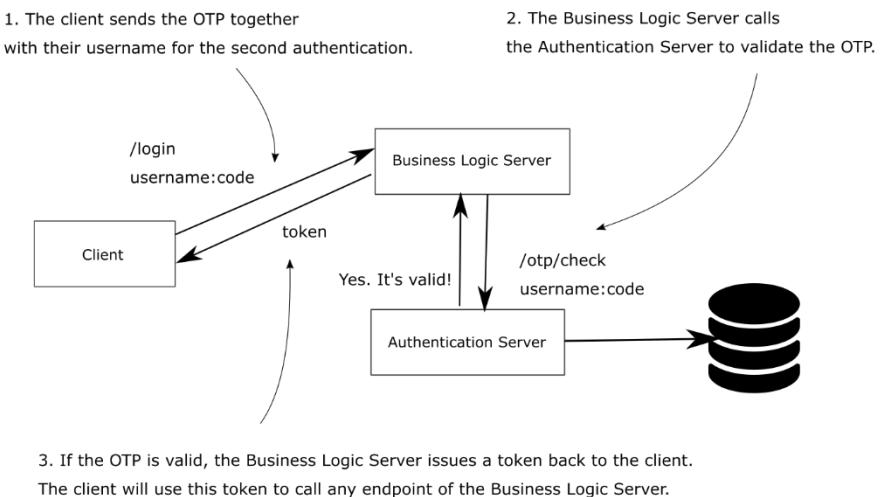
To call any endpoint on the Business Logic Server, the client has to follow three steps:

1. Authenticate with the username and the password by calling the `/login` endpoint on the Business Logic Server. The Business Logic Server sends a request for an OTP to the Authentication Server. After successful authentication, the authentication server sends a randomly generated OTP to the client via SMS (figure 11.2). This way of identifying the user is called multi-factor authentication (MFA), and it's pretty common nowadays. We generally need the users to prove who they are both by using their credentials and other means, like the fact that they own a specific mobile device.



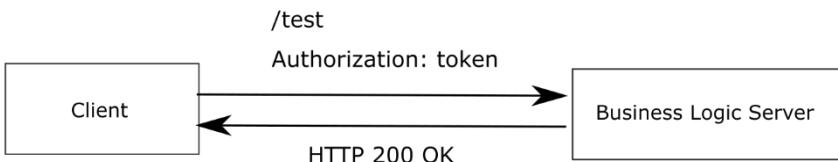
**Figure 11.2** The first authentication step consists of identifying the user using their username and password. The user sends their credentials, and the Authentication Server sends them back an OTP they can use for the second authentication step.

2. Once they have the code from the received SMS, the users can call the `/login` endpoint again with the username and the code. The Business Logic Server validates the code with the Authentication Server. If the code is valid, the client receives back a token that they can use to call any endpoint on the Business Logic Server (figure 11.3). In section 11.2, we talk in detail about what this token is, how we implement it, and why we use it.



**11.3** In the second authentication step, the client has to send the code they've received through SMS message together with their username. The Business Logic Server calls the Authentication Server to validate the OTP. If the OTP is valid, the Business Logic Server issues a token back to the client. The client will use this token to call any other endpoint on the Business Logic Server.

3. The client can now call any endpoint by adding the token they've received in step 2 to the Authorization header of the HTTP request (figure 11.4).



**Figure 11.4** To call any endpoint exposed by the Business Logic Server, the client has to add a valid token in the Authorization HTTP request header.

**NOTE** This example allows us to work on a bigger application, which includes more of the concepts we discussed. To allow you to focus on the Spring Security concepts I wanted to include in the application, I have simplified the architecture of the system. Someone could argue that this architecture uses vicious approaches, as the client should only share passwords with the Authentication Server and never with the Business Logic Server. This is correct! In our case, it's just a simplification. In general, in real-world scenarios, we strive to keep

credentials and secrets known by as few components in the system as possible. Also, someone could argue that the MFA scenario itself could be more easily implemented by using a third party (like Okta, or something similar). It's part of the purpose of the example to teach you how to define your custom filters. For this reason, I've chosen here the hard way: to implement ourselves this part in the authentication architecture.

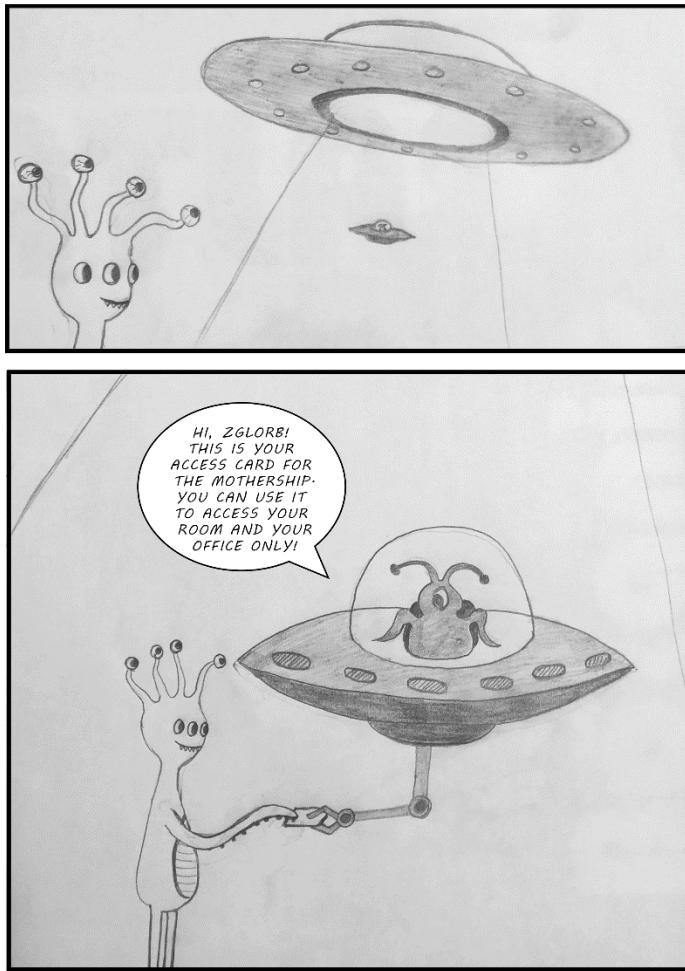
## 11.2 Implementing and using tokens

A token is similar to an access card. An application obtains a token as a result of the authentication process and uses this token to access resources. Endpoints represent the resources in a web application. For a web application, a token is a string, usually sent through an HTTP header by the clients that wish to access a particular endpoint. This string can be a plain one, like a pure Universally Unique Identifier (UUID), or might have a more complex shape, like a JSON Web Token (JWT). Tokens are very often used in authentication and authorization architectures today, and that's why you need to understand them. As you'll find out in chapter 12, they're one of the most important elements in the OAuth 2 architecture, which is also very often used today. As you'll learn, in this chapter but also chapters 12 to 15, tokens offer us advantages like separation of responsibilities in the authentication and authorization architecture, helps us making our architecture stateless, and provides possibilities to validate requests.

### 11.2.1 What is a token?

In this section, we discuss tokens. Using tokens is a method which an application uses to prove they've authenticated a user, which allows them to access their resources. We'll have the chance to implement tokens in this chapter, but you'll also find tokens in examples from chapters 12 to 15, as they're pretty often found in OAuth2 implementations. In section 11.2.2, you'll discover one of the most common token implementations used today: the JSON Web Token (JWT).

So what are tokens in the end? A token is just like an access card. When you visit an office building, you first go to the reception desk. There, you identify yourself (authentication), and you receive an access card (token). You can use the access card to open some of the doors, but not necessarily all of them. This way, the token authorizes your access and decides whether you're allowed to do something.

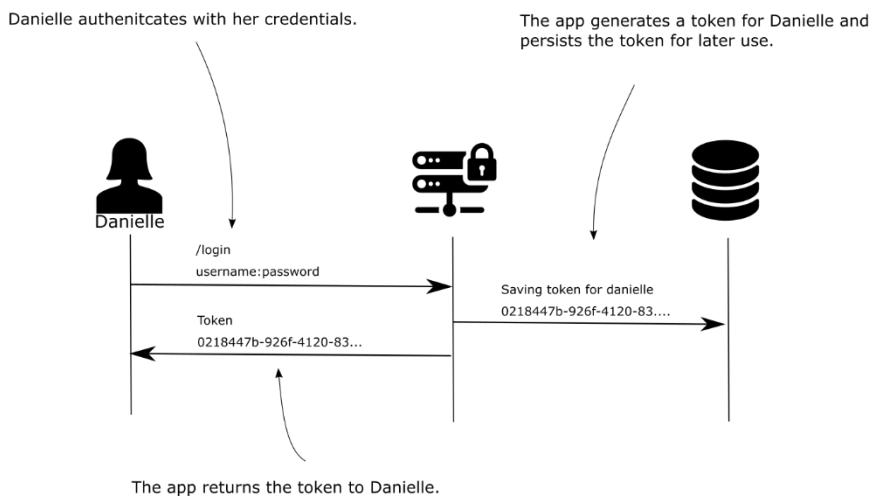


**Figure 11.5** To access the mothership (Business Logic Server), Zglorb needs an access card (token). After being identified, Zglorb gets an access card. This access card (token) only allows him to access his room and his office (resources).

At the implementation level, tokens can even be usual strings. What's most important is to be able to recognize them after you issue them. For example, you can generate UUIDs and store them in memory (or in a database). Let's assume the following scenario:

1. The client proves its identity to the server with its credentials (figure 11.6).

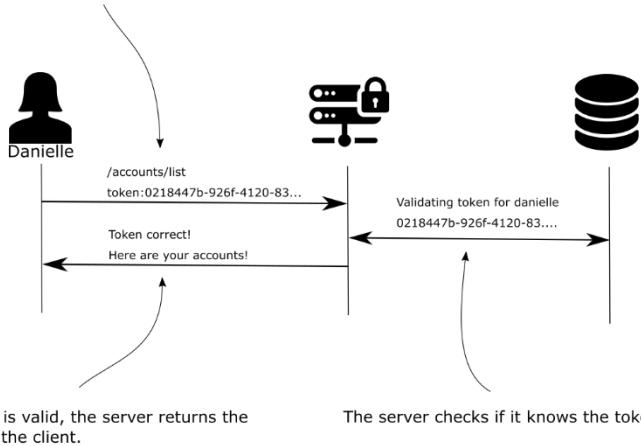
The server issues them a token in the format of a UUID. This token, now associated with the client, is stored in memory by the server.



**Figure 11.6** When the client authenticates, the server generates a token and returns it to the client. This token is then used by the client to access resources on the server.

2. When the client calls an endpoint, the client provides the token and gets authorized (figure 11.7).

When Danielle wants to access her accounts the client needs to send the access token in the request.



**Figure 11.7** When the client needs to access a user resource, the client has to provide a valid token in the request. A valid token is a token that was previously issued by the server when the user authenticated.

This is the general flow associated with using tokens in the authentication and authorization process. Which are its main advantages? Why would you use such a flow? Doesn't it add more complexity than a simple login? You could rely only on the user and the password anyway.

Using tokens bring us more advantages, so let's enumerate these advantages and then discuss them one by one:

- Tokens help you avoid sharing the credentials in all the requests.
- You can define tokens with a short lifetime.
- You can invalidate tokens without invalidating the credentials.
- Tokens may also store details which the client needs to send in the request, like the user authorities.
- Tokens help you delegate the authentication responsibility to another component in the system.

*Tokens help you avoid sharing the credentials in all the requests.* In chapters 2 to 10, we worked with HTTP Basic as the authentication method for all the requests. And this method, as you learned, assumes you send the credentials for each and any of the requests. Sending the credentials with each request isn't ok because this means that you expose them very often. The more often you expose them, the bigger the chances are that someone intercepts them. With tokens, we'll change the strategy. We'll send the credentials only in the first request to authenticate. Once authenticated, we get a token, and we'll use the token to get authorized for calling resources. This way, we'll only have to send credentials once for obtaining the token.

You can define tokens with a short lifetime. If a bad-intentioned individual steals the token, they won't be able to use it forever. Most probably, the token might expire before they find out how to use it to break into your system.

You can invalidate tokens. If you find out a specific token has been exposed, you can refute it. This way, it can't be used anymore by one who would find it.

*Tokens may also store details needed in the request.* We can use tokens to store details like the authorities and the roles of the user. This way, we can replace a server-side session with a client-side session, which offers us better flexibility for horizontal scaling. More about this approach in chapters 12 to 15 when we discuss the OAuth2 flow.

*Tokens help you separate the authentication responsibility to another component in the system.* We might find ourselves in the position to implement a system that doesn't manage its own users. Instead, this system allows the users to authenticate using credentials from accounts they have on other platforms: GitHub, Twitter, and so on. Even if we choose to implement also the component that does the authentication, it's still great that we can make this separate: it helps us to enhance scalability, and it makes the system architecture more natural to understand and develop. Chapters 5 and 6 of API Security in Action by Neil Madden (Manning, 2020) are also a great read related to this topic.

<https://livebook.manning.com/book/api-security-in-action/chapter-5/v-6/>

<https://livebook.manning.com/book/api-security-in-action/chapter-6/v-6/>

### 11.2.2 What is a JSON Web Token (JWT)?

In this section, we discuss a more specific implementation of tokens: the JSON Web Token (JWT). This token implementation has benefits that make it quite common to be used in today's application. This is why we discuss it in this section, and this is also why I've chosen to apply it within the hands-on example of this chapter. You'll also find it in chapters 12 to 15, where we discuss OAuth2.

You've already learned in section 11.2.1 that a token is anything the server can identify later. A UUID, an access card, and even the sticker you receive when you buy a ticket in a museum is a token implementation. Just as those are, JWT is a token implementation. Let's find out what a JWT looks like, and why a JWT is special.

It's easy to understand a lot about JWTs from the name of the implementation itself:

- JSON – it uses JSON to format the data it contains.
- Web – it's designed to be used for web requests.
- Token – it's a token implementation.

A JWT has three parts, each part separated from the others by a dot (a period). You find an example in the next code snippet.

```
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbmlbGx1In0.wg6LFPProg7s_KvFxvnYGizF-Mj4rr-  
0nJA1tVGZNn8U
```

The first two parts are the header and the body. The header (from the beginning of the token to the first dot) and the body (between the first and the second dot) are formatted using JSON and the Base64 encoded to make the value of the token shorter. We use the header and the body to store details in the token. The next code snippet shows what the header and the body look like before they are Base64 encoded.

```
{
  "alg": "HS256"
}

{
  "username": "danielle"
}
```

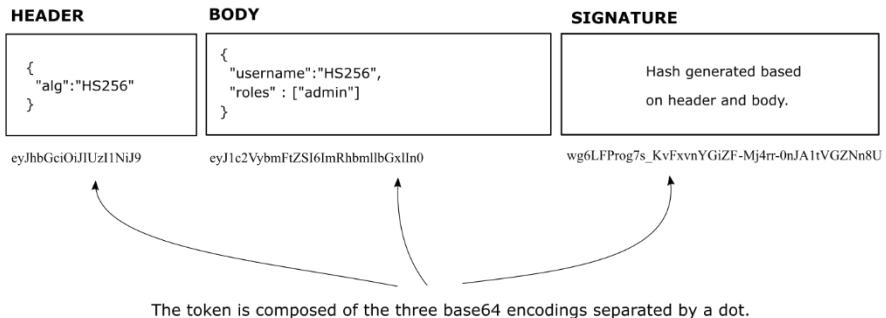
In the header, you store metadata related to the token. In this case, because I chose to sign the token, as you'll soon learn in the example that the header contains the name of the algorithm used to generate the signature.

In the body, you might have details needed later for authorization. In this case, we only have the username. I recommend that you keep the token as short as possible and that you don't add a lot of data in the body. Even if technically there's no limitation, you'll find out that

- If the token is longer, it slows down the request.
- When you sign the token, the longer the token is, the more time the cryptographic algorithm needs for signing it.

The last part of the token (from the second dot to the end) is the digital signature. This part can be missing, but you'll usually prefer to sign the header and the body. When you sign the content of the token, you can later use the signature to check that the content hasn't changed. Without a signature, you can't be sure that one didn't intercept the token while transferred on the network, and changed its content.

So to sum it up (figure 11.8), the JWT is a token implementation. It adds the benefit of easily transferring data during authentication, as well as signing the data to validate its integrity. You'll find a great discussion on JWT in chapter 7 and appendix H of *Microservices Security in Action* by Prabath Siriwardena and Nuwan Dias (Manning, 2020): <https://livebook.manning.com/book/microservices-security-in-action/chapter-7/v-7/> and <https://livebook.manning.com/book/microservices-security-in-action/h-json-web-token-jwt-v-7/>.



**Figure 11.8 A JWT is composed of three parts: the header, the body, and the signature. The header and the body are JSON representations of the data stored in the token. To make them easy to send in a request header, they're base 64 encoded. The last part of the token is the signature. The parts are concatenated with dots.**

In this chapter, we'll use Java JSON Web Token (JJWT) as the library to create and parse JWTs. This is one of the most frequently used libraries to generate and parse JWT tokens in Java applications. Besides all the needed details related to how to use this library, on JJWT's GitHub repository, I've also found a great explanation of JWTs. You might find it useful to read it as well:

<https://github.com/jwtk/jjwt#overview>

### 11.3 Implementing the Authentication Server

In this section, we start the implementation of our hands-on example. The first dependency we have is the authentication server. Even if it's not the application on which we focus on using Spring Security, we need it for our final result. To let you focus on what's essential in this hands-on, I've also taken out some parts of the implementation. I'll mention them throughout the example, and I'll leave them for you to implement as an exercise.

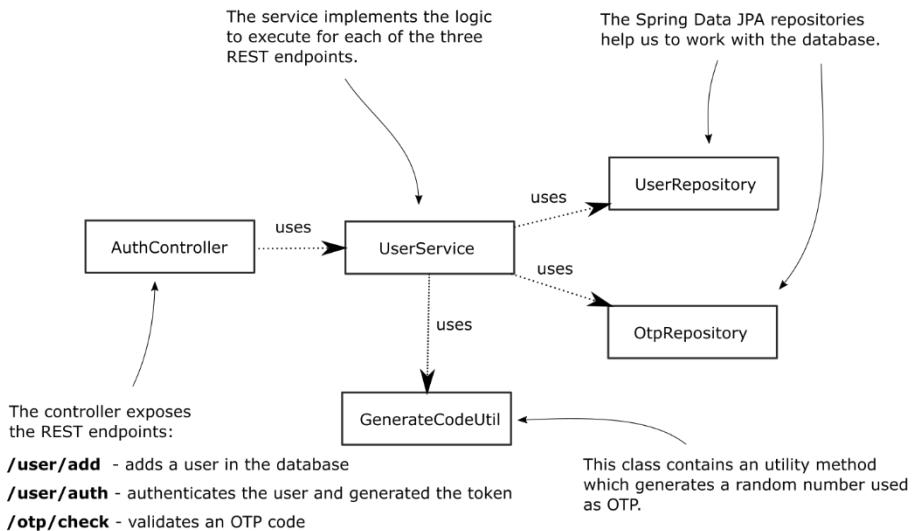
In our scenario, the authentication server connects to a database where it stores the user credentials and the OTPs generated during request authentication events. We'll need this application to expose three endpoints:

- `/user/add` – to add a user we'll use later for testing our implementation.
- `/user/auth` – which authenticates a user by its credentials and sends an SMS with an OTP. Remember, we took out the part with sending the SMS, but you can complete the

example further to do this as well as an exercise.

- `/otp/check` – which verifies that an OTP value is the one that the Authentication Server generated earlier for a specific user.

For a refresher on how to create REST endpoints, I recommend you to read chapter 6 from Spring in Action, Fifth Edition by Craig Walls: <https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-6/>



**Figure 11.9** The class design for an Authentication Server. The controller exposes REST endpoints, which call logic defined in a service class. The two repositories are the access layer to the database. We'll also write a utility class to separate the code which generates the one-time password to be sent through SMS.

We create a new project and add the needed dependencies, as presented by the next code snippet. You find this app implemented in project `ssia-ch11-ex1-s1`.

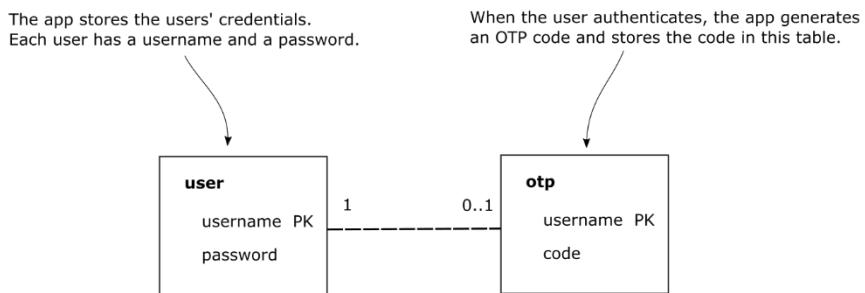
```

<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>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>

```

```
<scope>runtime</scope>
</dependency>
```

We also need to make sure we create the database for the application. Because we store user credentials (username and password), we need a table for this. We'll also need a second table to store the OTP values associated with the authenticated users (figure 11.10).



**Figure 11.10** The app database has two tables. In one of the tables, the app stores the user credentials while in the second one, the app stores the generated one-time password codes.

I'll use a database named `spring` and add the scripts to create the two tables required in a `schema.sql` file. Remember to place the `schema.sql` file in the resources folder of your project, as this is where Spring Boot will pick it up from and execute the scripts. In the next code snippet, you find the content of my `schema.sql` file. If you don't like the approach with the `schema.sql` file, you know you can anytime create the database structure manually or use any other method you prefer.

```

CREATE TABLE IF NOT EXISTS `spring`.`user` (
  `username` VARCHAR(45) NULL,
  `password` TEXT NULL,
  PRIMARY KEY (`username`));

CREATE TABLE IF NOT EXISTS `spring`.`otp` (
  `username` VARCHAR(45) NOT NULL,
  `code` VARCHAR(45) NULL,
  PRIMARY KEY (`username`));
  
```

In the `application.properties` file, we make sure to provide the parameters needed by Spring Boot to create the data source. The next code snippet shows the content of the `application.properties` file.

```

spring.datasource.url=jdbc:mysql://
localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
  
```

You observed that I added Spring Security to dependencies as well for this application. The only reason I've done this for the Authentication Server is to have the BCryptPasswordEncoder that I'd like to use to hash the users' passwords when stored in the database. To keep the example short and relevant to our purpose, I won't implement authentication between the Business Logic Server and the Authentication Server. But I'd like to leave this to you as an exercise, later, after finishing with the hands-on example.

**EXERCISE** Change the applications from this hands-on chapter to validate the requests between the Business Logic Server and the Authentication Server:

1. By using a symmetric key
2. By using an asymmetric key pair

To solve the exercise, you might find it useful to review the example we worked on in section 9.2.

For the implementation, we work on in this chapter, the configuration class of the project looks like as presented in listing 11.1.

### Listing 11.1 The configuration class

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Bean      #A
    public PasswordEncoder passwordEncoder() {      #A
        return new BCryptPasswordEncoder();          #A
    }      #A

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();          #B
        http.authorizeRequests()       #C
            .anyRequest().permitAll();    #C
    }
}
```

#A We define a password encoder we'll use to hash the passwords stored in the database.

#B We disable CSRF to be able to call all the endpoints of the application directly.

#C We allow all the calls without authentication.

With the configuration class in place, we can continue with defining the connection to the database. Because we use Spring Data JPA, we'll have to write the JPA entities and then the repositories. We have two tables, so we'll define two JPA entities and two repository interfaces. Listing 11.2 shows the definition of the `User` entity. It represents the `user` table where we store the users' credentials.

### Listing 11.2 The User entity

```
@Entity
public class User {

    @Id
    private String username;
    private String password;
```

```
// Omitted getters and setters
}
```

Listing 11.3 presents the second entity: `Otp`. This entity represents the `otp` table, where the application stores the generated OTPs for authenticated users.

#### **Listing 11.3 The `Otp` entity**

```
@Entity
public class Otp {

    @Id
    private String username;
    private String code;

    // Omitted getters and setters
}
```

Listing 11.4 presents the Spring Data JPA repository for the `User` entity. We define in this interface a method to retrieve the user by its username. We'll need it for the first step of the authentication, where we validate the username and the password.

#### **Listing 11.4 The `UserRepository` interface**

```
public interface UserRepository extends JpaRepository<User, String> {

    Optional<User> findUserByUsername(String username);
}
```

Listing 11.5 presents the Spring Data JPA repository for the `Otp` entity. In this interface, we'll define a method to retrieve the OTP by username. We'll need this method for the second authentication step, where we validate the OTP for a user.

#### **Listing 11.5 The `OtpRepository` interface**

```
public interface OtpRepository extends JpaRepository<Otp, String> {

    Optional<Otp> findOtpByUsername(String username);
}
```

With the repositories and the entities in place, we can work on the logic of the application. For this, I'll create a service class that I'll call `UserService`. As shown in listing 11.6, the service has dependencies on the repositories and the password encoder. We'll use these objects to implement the application logic, so we need to autowire them.

#### **Listing 11.6 Autowiring the dependencies in the `UserService` class**

```
@Service
@Transactional
public class UserService {

    @Autowired
```

```

private PasswordEncoder passwordEncoder;

@Autowired
private UserRepository userRepository;

@Autowired
private OtpRepository otpRepository;

}

```

We need to define a method to add a user. You find the definition of this method in listing 11.7.

#### **Listing 11.7 Defining the `addUser()` method**

```

@Service
@Transactional
public class UserService {

    // Omitted code

    public void addUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userRepository.save(user);
    }
}

```

Now, what does the Business Logic Server need? The Business Logic Server will need a way to send a username and password to be authenticated. After the user is authenticated, the Authentication Server generates an OTP for the user and sends it via SMS. Listing 11.8 shows the definition of the `auth()` method, which implements this logic.

#### **Listing 11.8 Implementing the first authentication step**

```

@Service
@Transactional
public class UserService {

    // Omitted code

    public void auth(User user) {
        Optional<User> o =      #A
            userRepository.findUserByUsername(user.getUsername());

        if(o.isPresent()) {      #B
            User u = o.get();
            if (passwordEncoder.matches(
                user.getPassword(),
                u.getPassword())) {
                renewOtp(u);      #C
            } else {
                throw new BadCredentialsException("Bad credentials.");      #D
            }
        } else {
            throw new BadCredentialsException("Bad credentials.");      #D
        }
    }
}

```

```

    }

private void renewOtp(User u) {
    String code = GenerateCodeUtil
        .generateCode();      #E

    Optional<Otp> userOtp =      #F otpRepository.findOtpByUsername(u.getUsername());

    if (userOtp.isPresent()) {    #G
        Otp otp = userOtp.get();    #G
        otp.setCode(code);        #G
    } else {      #H
        Otp otp = new Otp();      #H
        otp.setUsername(u.getUsername());    #H
        otp.setCode(code);        #H
        otpRepository.save(otp);    #H
    }
}

// Omitted code
}

```

#A We search for the user in the database.

#B If the user exists, we continue with verifying its password.

#C If the password is correct, we generate a new OTP.

#D If the password is not correct or username doesn't exist, we throw an exception.

#E We generate a random value for the OTP.

#F We search the OTP by username.

#G If an OTP already exists for this username, we update its value.

#H If an OTP doesn't exist for this user, we create a new record with the generated value.

Listing 11.9 presents the GenerateCodeUtil class, which we use in listing 11.8 to generate the new OTP value.

### **Listing 11.9 Generating the OTP**

```

public final class GenerateCodeUtil {

    private GenerateCodeUtil() {}

    public static String generateCode() {
        String code;

        try {
            SecureRandom random =
                SecureRandom.getInstanceStrong();    #A

            int c = random.nextInt(9000) + 1000;    #B

            code = String.valueOf(c);    #C
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(
                "Problem when generating the random code.");
        }

        return code;
    }
}

```

#A We create an instance of `SecureRandom`, which helps us generating a random int value.  
#B We generate a value between 0 and 8999 and we add 1000 to each generated value. This way, we get values between 1000 and 9999 (4 digit random codes).  
#C We convert the int to a String and return it.

The last method we need to have in the `UserService` is one to validate the OTP for a user. You find this method in listing 11.10.

#### **Listing 11.10 Validating an OTP**

```
@Service
@Transactional
public class UserService {
    // Omitted code

    public boolean check(Otp otpToValidate) {
        Optional<Otp> userOtp =      #A
            otpRepository.findOtpByUsername(
                otpToValidate.getUsername());

        if (userOtp.isPresent()) {    #B
            Otp otp = userOtp.get();  #B
            if (otpToValidate.getCode().equals(otp.getCode())) {
                return true;    #B
            }
        }

        return false;    #C
    }

    // Omitted code
}
```

#A We search the OTP by username.

#B If the OTP exists in the database, and it is the same as the one received from the Business Logic Server, we return true.

#C Else we return false.

Finally, in this application, we expose the logic presented above with a controller. Listing 11.11 shows the definition of this controller.

#### **Listing 11.11 The definition of the `AuthController` class**

```
@RestController
public class AuthController {

    @Autowired
    private UserService userService;

    @PostMapping("/user/add")
    public void addUser(@RequestBody User user) {
        userService.addUser(user);
    }

    @PostMapping("/user/auth")
    public void auth(@RequestBody User user) {
        userService.auth(user);
    }
}
```

```

@PostMapping("/otp/check")
public void check(@RequestBody Otp otp, HttpServletResponse response) {
    if (userService.check(otp)) {      #A
        response.setStatus(HttpServletResponse.SC_OK);
    } else {
        response.setStatus(HttpServletResponse.SC_FORBIDDEN);
    }
}

```

#A If the OTP is valid the HTTP response will have the status 200 OK, otherwise the value of the status will be 403 Forbidden.

With this setup, we now have the Authentication Server. Let's start it and make sure that the endpoints work the way we expect. To test the functionality of the Authentication Server, we have to:

1. Add a new user to the database by calling the `/user/add` endpoint.
2. Validate that the user was correctly added by checking the `users` table in the database.
3. Call the `/user/auth` endpoint for the user added in step 1
4. Validate that the application generates and stores an OTP in the `otp` table.
5. Use the OTP generated at step 3 to validate that the `/otp/check` endpoint works as desired.

We begin by adding a user in the database of the Authentication Server. We need at least one user which we use for authentication. We can add it by calling the `/user/add` endpoint we created in the Authentication Server. Because we didn't configure any port in the Authentication Server application, we expect to run on the default one, which is 8080.

```
curl -XPOST
-H "content-type: application/json"
-d "{\"username\":\"danielle\",\"password\":\"12345\"}" http://localhost:8080/user/add
```

After using the `curl` command presented by the previous code snippet to add a user, we check in the database to validate that the record was added correctly. In my case, I can see the following details:

```
Username: danielle
Password: $2a$10$.bI9ix.Y0m70iZitP.RdSuwzSggqPJKnKpRUBQPGhoRvHA.1INYmy
```

The application hashed the password before storing it in the database, which is the expected behavior. Remember, we especially used the `BCryptPasswordEncoder` for this purpose in the Authentication Server.

**NOTE** Remember our discussion from chapter 4, `BCryptPasswordEncoder` uses `bcrypt` as the hashing algorithm. With `bcrypt` the output is generated based on a salt value, which means that you'll obtain different outputs for the same input. So in your case, the hash of the same password will be a different one. You find more details and a great discussion on hash functions in chapter 2 of the Real-World Cryptography by David Wong (Manning, 2020) <https://livebook.manning.com/book/real-world-cryptography/chapter-2/v-4/>

We have a user, so let's generate an OTP for it by calling the `/user/auth` endpoint. The next code snippet provides the `curl` command you can use to call this endpoint.

```
curl -XPOST
-H "content-type: application/json"
-d "{\"username\":\"danielle\",\"password\":\"12345\""} http://localhost:8080/user/auth
```

In the `otp` table in our database, the application generated and stored a random 4-digits code. In my case, its value is 8173. The last step for testing our Authentication Server is to call the `/otp/check` endpoint and validate it returns a 200 OK status code of the HTTP response when the OTP is correct and 403 Forbidden if the OTP is wrong.

The following code snippets show you the tests for the correct OTP value as well as the test for a wrong OTP value.

If the OTP value is correct:

```
curl -v -XPOST -H "content-type: application/json" -d
      "{\"username\":\"danielle\",\"code\":\"8173\""} http://localhost:8080/otp/check
```

The response status is:

```
...
< HTTP/1.1 200
...
```

If the OTP value is wrong:

```
curl -v -XPOST -H "content-type: application/json" -d
      "{\"username\":\"danielle\",\"code\":\"9999\""} http://localhost:8080/otp/check
```

The response status is:

```
...
< HTTP/1.1 403
...
```

We've just proved that the Authentication Server components work! We now dive into the next element for which we'll write most of the Spring Security configurations of our current hands-on example.

## 11.4 Implementing the Business Logic Server

In this section, we implement the Business Logic Server. With this application, you'll recognize a lot of the things we've discussed until now in this book. I'll refer here and there to sections where you learned specific aspects, in case you want to go back and recap them. With this part of the system, you'll learn to implement and use JWTs for authentication and authorization. As well, we'll implement the communication between the Business Logic Server and the Authentication Server to establish the multi-factor authentication in your application.

To accomplish our task, at high-level, we have to:

1. Create an endpoint which will represent our resource we want to secure.

2. Implement the first authentication step in which the client sends to the Business Logic Server the user's credentials (username and password) to log in.
3. Implement the second authentication step in which the client sends to the Business Logic Server the OTP, which the user has received from the Authentication Server after step 2. Once authenticated by OTP, the client gets back a JWT, which it can use to access the user's resources.
4. Implement the authorization based on the JWT. The Business Logic Server has to validate a JWT received from a client and, if valid, allow the client to access the resource.

Technically, to achieve these four high-level points, we need to:

1. Create the Business Logic Server project. I'll name it `ssia-ch11-ex1-s2`.
2. Implement the `Authentication` objects which have the role of representing the two authentication steps.
3. Implement a proxy to establish communication between the Authentication Server and the Business Logic Server.
4. Define the `AuthenticationProvider` objects which implement the authentication logic for the two authentication steps using the `Authentication` objects defined in step 2.
5. Define the custom filter objects which intercept the HTTP request and apply the authentication logic implemented by the `AuthenticationProvider` objects.
6. Write the authorization configurations.

We start with the dependencies. Listing 11.12 shows you which dependencies you should add to the `pom.xml` file. You find this application in project `ssia-ch11-ex1-s2`.

#### **Listing 11.12 The dependencies needed for the Business Logic Server**

```
<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>    #A
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.1</version>
</dependency>
<dependency>    #A
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.1</version>
    <scope>runtime</scope>
</dependency>
<dependency>    #A
    <groupId>io.jsonwebtoken</groupId>
```

```

<artifactId>jjwt-jackson</artifactId>
<version>0.11.1</version>
<scope>runtime</scope>
</dependency>
<dependency> #B
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
    <version>2.3.2</version>
</dependency>
<dependency> #B
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>2.3.2</version>
</dependency>

```

#A We add the jjwt dependency for generating and parsing JWTs.

#B We additionally need this if you use Java 10 or above.

In this application, we'll only define a `/test` endpoint. Everything else we write in this project is to secure this endpoint. The `/test` endpoint is exposed by the `TestController` class, which is presented in listing 11.13.

### Listing 11.13 The TestController class

```

@RestController
public class TestController {

    @GetMapping("/test")
    public String test() {
        return "Test";
    }
}

```

To secure the app now, we have to define the three authentication levels:

- Authentication with username and password to receive an OTP (figure 11.11).

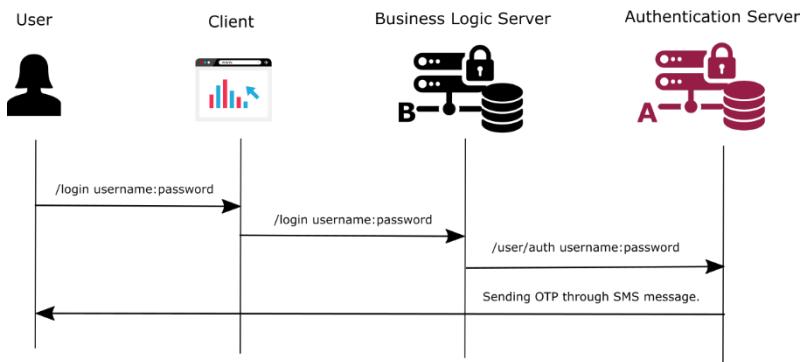
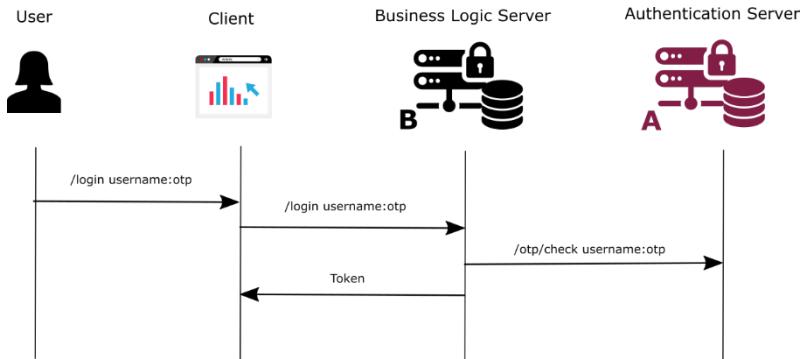


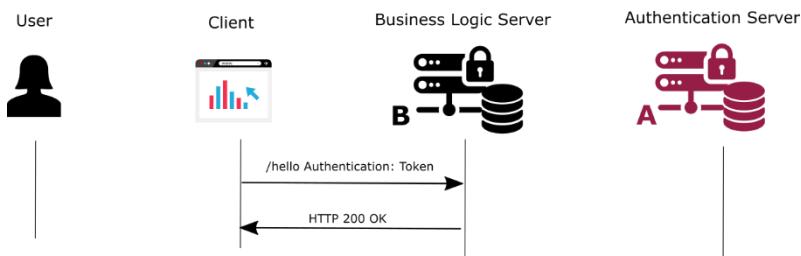
Figure 11.11 The first authentication step. The user sends its credentials for authentication. The authentication server authenticates the user and sends an SMS message containing an OTP code.

- Authentication with OTP to receive a token (figure 11.12).



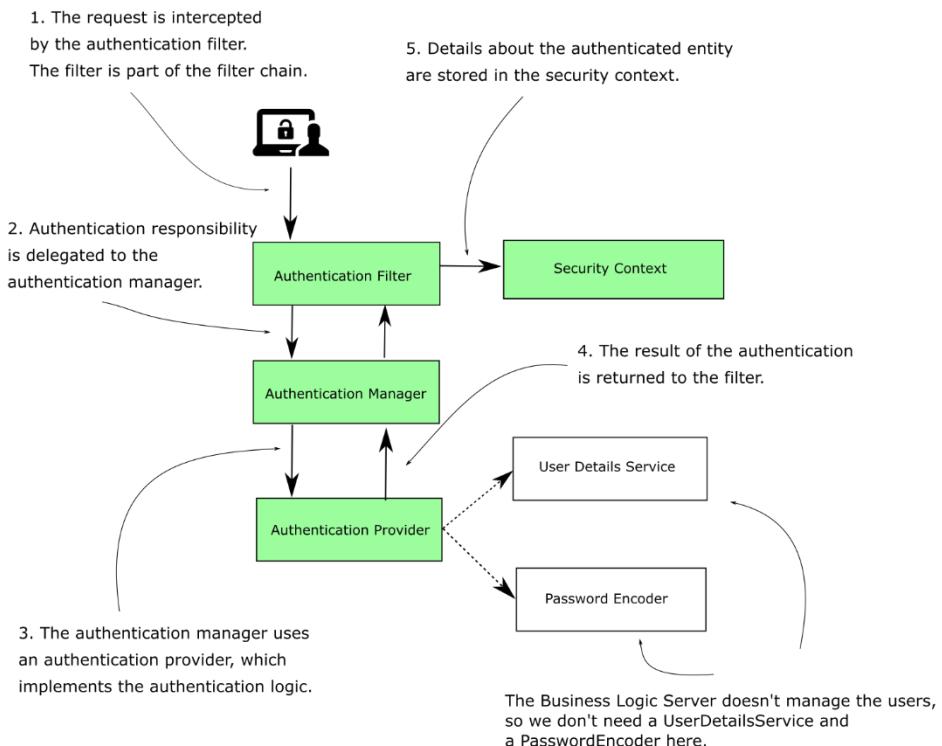
**Figure 11.12** The second authentication step. The user sends the OTP code they received as a result of the first authentication step. The authentication server validates the OTP code and sends back a token to the client. The client uses the token to access the user resources.

- Authentication with the token to access the endpoint (figure 11.13).



**Figure 11.13** The client uses the token obtained in step 2 to access resources exposed by the business logic server.

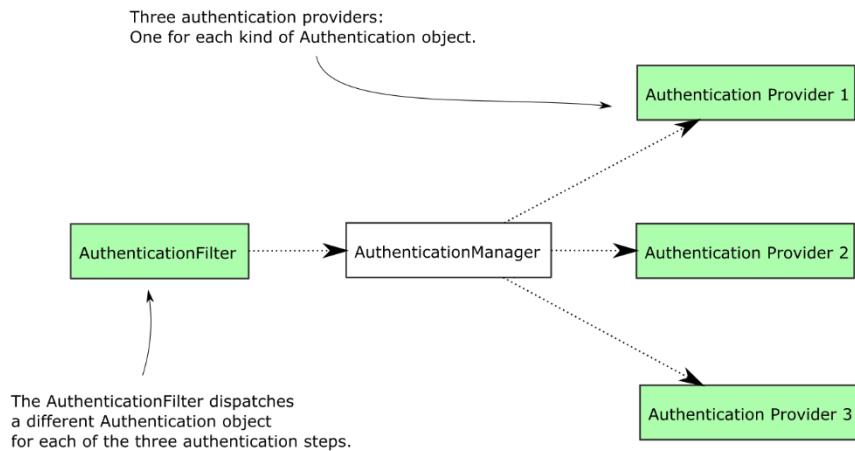
With the given requirement, which is more complex and assumes multiple authentication steps, the HTTP Basic authentication can't help us anymore. We need to implement custom filters and authentication providers to customize the authentication logic for our scenario. Fortunately, you've learned how to define custom filters in chapter 9. Let's remember the authentication architecture in Spring Security (figure 11.14).



**Figure 11.14** The authentication architecture in Spring Security. The `AuthenticationFilter` – which is part of the filter chain – intercepts the request and delegates the authentication responsibility to the `AuthenticationManager`. The `AuthenticationManager` uses an `AuthenticationProvider` to authenticate the request.

Often, when developing an application, we have more than one good solution. When designing an architecture, you should always think about all the possible implementations and choose the best fit for your scenario. If more than one option is applicable and you can't decide which is the best to implement, a proof-of-concept for each to help you decide on which solution to choose. For our scenario, I'll present you two options, and we'll continue the implementation with one of them. I'll leave the other choice as an exercise for you to implement.

The first option for us is to define three custom `Authentication` objects, three custom `AuthenticationProvider` objects, and a custom filter to delegate to them by making use of the `AuthenticationManager` (figure 11.15). You learned how to implement the `Authentication` and `AuthenticationProvider` interfaces in chapter 5.



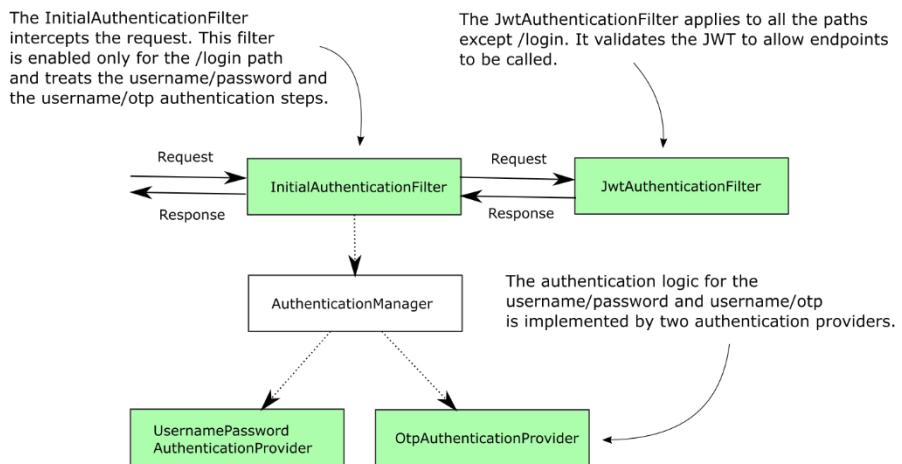
**Figure 11.15** One option for implementing our application: The `AuthenticationFilter` intercepts the request. Depending on the authentication step, it creates a specific `Authentication` object and dispatches it to the `AuthenticationManager`. An `Authentication` object represents each Authentication step. For each Authentication step, an `Authentication provider` implements the logic. I shaded the components that we need to implement.

The second option – which I'll choose to implement in this example – is to have two custom `Authentication` objects and two custom `AuthenticationProvider` objects (figure 11.16). These objects will help me apply the logic related to the `/login` endpoint.

They will:

- Authenticate the user with a username and password.
- Authenticate the user with an OTP.

We'll implement the validation of the token with a second filter.



**Figure 11.16** A second approach to implement the authentication in our scenario is to separate the responsibilities in two filters. The first treats the requests on the /login path and takes care of the two initial authentication steps. The other takes care of the rest of the endpoints for which the JWT token needs to be validated.

Both approaches are equally good. I have described both of them only to enforce that you'll find cases in which you'll have more ways to develop the same scenario, especially with Spring Security that offers a quite flexible architecture. I'll choose the second one because it offers me the possibility to recap more things, like having multiple custom filters and using the `shouldNotFilter()` method of the `OncePerRequestFilter` class. We discussed this class in section 9.5, but I didn't have the chance to apply the `shouldNotFilter()` method with an example. We can take this opportunity now.

**EXERCISE** Implement the Business Logic Server with the first approach I described in this section.

#### 11.4.1 Implementing the Authentication objects

In this section, we implement the two `Authentication` objects we need for our solution to develop the Business Logic Server. At the beginning of section 11.4, we created the project and added the needed dependencies. We also created an endpoint we want to secure and decided on how to implement the class design for our example. We need two types of `Authentication` objects, one to represent the authentication by username and password and a second to represent the authentication by OTP. As you've learned in chapter 5, the `Authentication` contract represents the authentication process for a request. It can be a process in progress or after its completion. We need to implement the `Authentication`

interface for both the case in which the application authenticates the user with their username and the password as well as for the OTP.

In listing 11.14, you find the `UsernamePasswordAuthentication` class, which implements the authentication with username and password. To make the classes shorter, I'll extend the `UsernamePasswordAuthenticationToken` class and not directly the `Authentication` interface. You've seen the `UsernamePasswordAuthenticationToken` class in chapter 5, where we discussed applying custom authentication logic.

#### **Listing 11.14 The UsernamePasswordAuthentication class**

```
public class UsernamePasswordAuthentication
    extends UsernamePasswordAuthenticationToken {

    public UsernamePasswordAuthentication(
        Object principal,
        Object credentials,
        Collection<? extends GrantedAuthority> authorities) {
        super(principal, credentials, authorities);
    }

    public UsernamePasswordAuthentication(
        Object principal,
        Object credentials) {
        super(principal, credentials);
    }
}
```

Note that I'll define both constructors in the class. There's a big difference between them: when you call the one with two parameters, the authentication remains "unauthenticated", while the one with three parameters sets the `Authentication` object as authenticated.

As you learned in chapter 5, when the `Authentication` instance is "authenticated" it means that the authentication process ended.

If the `Authentication` object is not set as authenticated yet, and no exception was thrown during the process, the `AuthenticationManager` still tries to find a proper `AuthenticationProvider` object to authenticate the request.

We'll use the constructor with two parameters when we initially build the `Authentication` object, and it's not yet authenticated. When an `AuthenticationProvider` object authenticates the request, it creates an `Authentication` instance using the constructor with three parameters, which creates an "authenticated" object. The third parameter is the collection of granted authorities, which is mandatory for an authentication process that has ended.

Similarly to the `UsernamePasswordAuthentication`, we'll implement the second `Authentication` object for the second authentication step with OTP. I'll name this class `OtpAuthentication`. As presented by listing 11.15, it extends the `UsernamePasswordAuthenticationToken`. We can use the same class because we'll treat the OTP as a password. So being that it's very similar, we'll use the same approach to save some lines of code.

**Listing 11.15 The OtpAuthentication class**

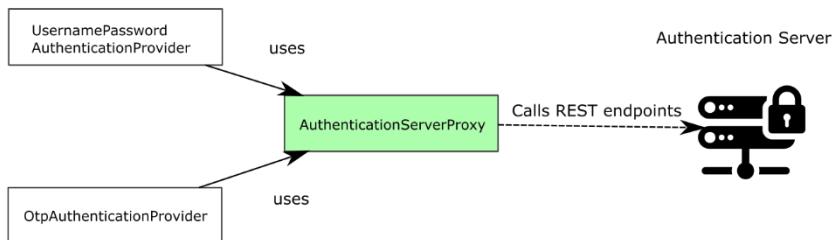
```
public class OtpAuthentication
    extends UsernamePasswordAuthenticationToken {

    public OtpAuthentication(Object principal, Object credentials) {
        super(principal, credentials);
    }

    public OtpAuthentication(
        Object principal,
        Object credentials,
        Collection<? extends GrantedAuthority> authorities) {
        super(principal, credentials, authorities);
    }
}
```

**11.4.2 Implementing the proxy to the Authentication Server**

In this section, we'll build a way to call the REST endpoint exposed by the Authentication Server. Usually, immediately after defining the `Authentication` objects, we would implement the `AuthenticationProvider` objects (figure 11.17). We know, however, that to complete the authentication, we'll need a way to call the Authentication Server. So I suggest continuing now with implementing a proxy to the Authentication Server first, before implementing the `AuthenticationProvider` objects.



**Figure 11.17** The `AuthenticationServerProxy` is used from the authentication logic implemented by the authentication providers to call the Authentication Server.

We need to

1. Define a model class `User`, which we'll use to call the REST services exposed by the Authentication Server.
2. Declare a bean of type `RestTemplate`, which we'll use to call the REST endpoints exposed by the Authentication Server.
3. Implement the proxy class, which defines two methods: one for the username/password authentication and the other for the username/otp authentication.

Listing 11.16 presents the `User` model class.

#### **Listing 11.16 The User model class**

```
public class User {

    private String username;
    private String password;
    private String code;

    // Omitted getters and setters
}
```

Listing 11.17 presents the application configuration class. I'll name this class `ProjectConfig` and define a `RestTemplate` bean to be used by the proxy class we'll develop next.

#### **Listing 11.17 The ProjectConfig class**

```
@Configuration
public class ProjectConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

We can now write the `AuthenticationServerProxy` class, which we'll use to call the two REST endpoints exposed by the Authentication Server application. Listing 11.18 presents the `AuthenticationServerProxy` class.

#### **Listing 11.18 The AuthenticationServerProxy class**

```
@Component
public class AuthenticationServerProxy {

    @Autowired
    private RestTemplate rest;

    @Value("${auth.server.base.url}")    #A
    private String baseUrl;
```

```

public void sendAuth(String username,
                     String password) {

    String url = baseUrl + "/user/auth";

    var body = new User();
    body.setUsername(username);      #B
    body.setPassword(password);     #B

    var request = new HttpEntity<>(body);

    rest.postForEntity(url, request, Void.class);
}

public boolean sendOTP(String username,
                       String code) {

    String url = baseUrl + "/otp/check";

    var body = new User();          #C
    body.setUsername(username);     #C
    body.setCode(code);            #C

    var request = new HttpEntity<>(body);

    var response = rest.postForEntity(url, request, Void.class);

    return response      #D
        .getStatusCode()    #D
        .equals(HttpStatus.OK);   #D
}
}

```

#A We take the base URL from the application.properties file.

#B The HTTP request body needs the username and the password for this call.

#C The HTTP request body needs the username and the code for this call.

#D We return true if the HTTP response status is 200 OK and false otherwise.

They're just regular calls on REST endpoints with a `RestTemplate`. If you need a refresh on how this works, a great choice is chapter 7 of *Spring in Action*, fifth edition, by Craig Walls (Manning, 2018).

<https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-7/>.

Remember to add the base URL for the Authentication Server to your `application.properties` file. I'll also change the port for the current application here because I expect to run them on the same system for my tests. I'll keep the Authentication Server on the default port, which is 8080, and I'll change the port for the current app (the Business Logic Server) to 9090. The next code snippet shows the content for the `application.properties` file.

```

server.port=9090
auth.server.base.url=http://localhost:8080

```

### 11.4.3 Implementing the AuthenticationProvider objects

In this section, we'll implement the `AuthenticationProvider` classes. Only now we have everything we need to start working on the authentication providers. We need them because this is where we write the custom authentication logic.

We'll create a class `UsernamePasswordAuthenticationProvider` to serve the `UsernamePasswordAuthentication` type of `Authentication`, as described by listing 11.19. Because we designed our flow to have two authentication steps, and we have one filter that takes care of both steps, we know that the authentication doesn't finish within this provider. So we use the constructor with two parameters to build the `Authentication` object: new `UsernamePasswordAuthenticationToken(username, password)`. Remember, we discussed in section 11.4.1 that the constructor with two parameters doesn't mark the object as being "authenticated".

#### Listing 11.19 The UsernamePasswordAuthentication class

```
@Component
public class UsernamePasswordAuthenticationProvider
    implements AuthenticationProvider {

    @Autowired
    private AuthenticationServerProxy proxy;

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

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

        proxy.sendAuth(username, password);      #A

        return new UsernamePasswordAuthenticationToken(username, password);
    }

    @Override
    public boolean supports(Class<?> aClass) {    #B
        return UsernamePasswordAuthentication.class.isAssignableFrom(aClass);
    }
}
```

#A We use the proxy to call the Authentication Server. It sends the OTP to the client through SMS.

#B This AuthenticationProvider is designed for the `UsernamePasswordAuthentication` type of `Authentication`.

Listing 11.20 presents the authentication provider designed for the `OtpAuthentication` type of `Authentication`. The logic implemented by this `AuthenticationProvider` is simple. It calls the Authentication Server to find out if the OTP is valid. If the OTP is correct, it returns an instance of `Authentication`. If the OTP isn't correct, the authentication provider throws an exception. Based on this, the filter sends back the token in the HTTP response.

#### Listing 11.20 The OtpAuthenticationProvider class

```
@Component
```

```

public class OtpAuthenticationProvider
    implements AuthenticationProvider {

    @Autowired
    private AuthenticationServerProxy proxy;

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

        String username = authentication.getName();
        String code = String.valueOf(authentication.getCredentials());

        boolean result = proxy.sendOTP(username, code);

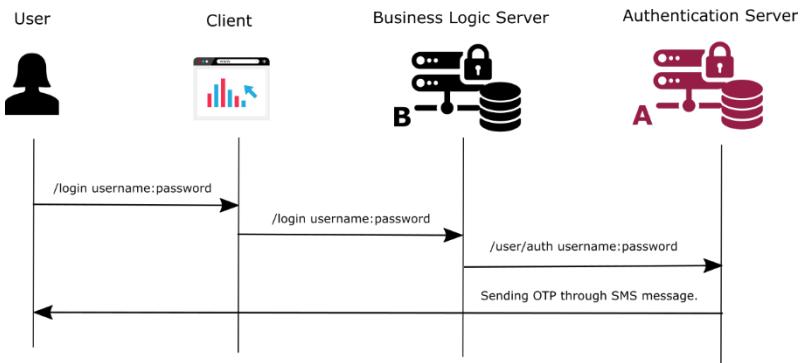
        if (result) {
            return new OtpAuthentication(username, code);
        } else {
            throw new BadCredentialsException("Bad credentials.");
        }
    }

    @Override
    public boolean supports(Class<?> aClass) {
        return OtpAuthentication.class.isAssignableFrom(aClass);
    }
}

```

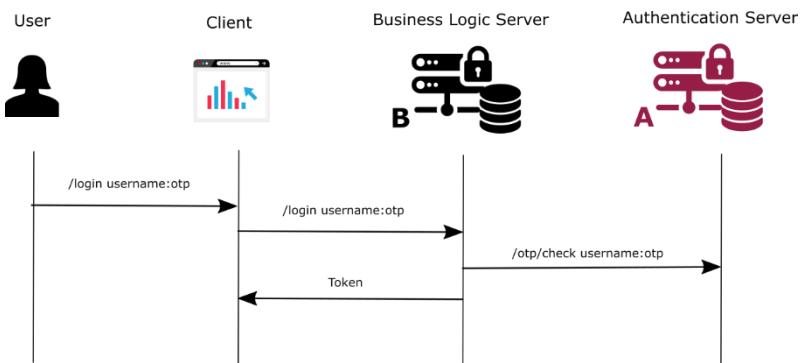
#### 11.4.4 Implementing the filters

In this section, we'll implement the custom filters which we'll add to the filter chain. Their purpose is to intercept the requests and apply the authentication logic. We chose to implement one filter to deal with the authentication done by the Authentication Server and another one for the authentication based on the JWT. We'll implement a class named `InitialAuthenticationFilter`, which deals with the first authentication steps which are done using the Authentication Server. In the first step, the user authenticates with their username and password to receive an OTP (figure 11.18). You've seen these graphics also in figures 11.11 and 11.12, but I'll add them again so that you don't need to flip back pages and search for them.



**Figure 11.18** First, the client needs to authenticate the user using their credentials. After successfully authenticated, the Authentication Server sends an SMS message to the user with a code.

In the second step, the user sends the OTP to prove they really are who they claim, and after successfully authenticated, the app provides them a token to call any endpoint exposed by the Business Logic Server (figure 11.19).



**Figure 11.19** The second authentication step. The user sends the OTP code they received as a result of the first authentication step. The authentication server validates the OTP code and sends back a token to the client. The client uses the token to access the user resources.

Listing 11.21 presents the definition of the `InitialAuthenticationFilter` class. We start by injecting the `AuthenticationManager` to which we'll delegate the authentication responsibility, override the `doFilterInternal()` method, which is called when the request reaches this filter in the filter chain and override the `shouldNotFilter()` method. As we discussed in chapter 9, the `shouldNotFilter()` method is one of the reasons why we would choose to extend the `OncePerRequestFilter` class instead of implementing the `Filter` interface directly. When we override this method, we define a specific condition on when the

filters execute. In our case, we want to execute for any request on the `/login` path and be skipped for all the others.

#### **Listing 11.21 The InitialAuthenticationFilter class**

```
@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {

    @Autowired    #A
    private AuthenticationManager manager;

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

    @Override
    protected boolean shouldNotFilter(
        HttpServletRequest request) {

        return !request.getServletPath().equals("/login");    #C
    }
}
```

#A We autowire the `AuthenticationManager` which will apply the correct authentication logic.

#B We override the `doFilterInternal()` method to require the correct authentication based on the request.

#C We apply this filter only to the `/login` path.

We continue writing the `InitialAuthenticationFilter` class with the first authentication step: the one in which the client sends the username and the password to obtain the OTP. We assume that if the user didn't send a code (OTP) we have to do the authentication based on username and the password. So we take all the values from the HTTP request header where we expect them to be, and if a code wasn't sent, we call the first authentication step by creating an instance of `UsernamePasswordAuthentication` and forwarding the responsibility to the `AuthenticationManager`. We know since chapter 2 that next, the `AuthenticationManager` will try to find a proper `AuthenticationProvider`. In our case, this is the `UsernamePasswordAuthenticationProvider` we wrote in listing 11.19. It will be the one triggered because its `supports()` method states it accepts the `UsernamePasswordAuthentication` authentication type.

#### **Listing 11.22 Implementing the logic for UsernamePasswordAuthentication**

```
@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {

    // Omitted code

    @Override
    protected void doFilterInternal(
```

```

    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain)
    throws ServletException, IOException {

    String username = request.getHeader("username");
    String password = request.getHeader("password");
    String code = request.getHeader("code");

    if (code == null) {      #A
        Authentication a =
            new UsernamePasswordAuthentication(username, password);
        manager.authenticate(a);    #B
    }
}

// Omitted code
}

```

#A If the HTTP request doesn't contain an OTP, we assume we have to authenticate based on username and password.

#B We call the AuthenticationManager with an instance of UsernamePasswordAuthentication.

If, however, a code is sent in the request, we assume it's the second authentication step. In this case, we create an `OtpAuthentication` object to call the `AuthenticationManager`. We know from our implementation of the `OtpAuthenticationProvider` class in listing 11.20 that if the authentication fails, an exception will be thrown. This means that the JWT token will be generated and attached to the HTTP response headers only if the OTP is valid.

#### **Listing 11.23 Implementing the logic for `OtpAuthentication`**

```

@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {

    @Autowired
    private AuthenticationManager manager;

    @Value("${jwt.signing.key}")      #A
    private String signingKey;      #A

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

        String username = request.getHeader("username");
        String password = request.getHeader("password");
        String code = request.getHeader("code");

        if (code == null) {
            Authentication a =
                new UsernamePasswordAuthentication(username, password);
            manager.authenticate(a);
        } else {      #B
            Authentication a =

```

```

        new OtpAuthentication(username, code);      #C

        a = manager.authenticate(a);      #C

        SecretKey key = Keys.hmacShaKeyFor(
            signingKey.getBytes(
                StandardCharsets.UTF_8));

        String jwt = Jwts.builder()      #D
            .setClaims(Map.of("username", username))
            .signWith(key)
            .compact();

        response.setHeader("Authorization", jwt);      #E
    }
}

// Omitted code
}

```

#A We take the value of the key we'll use to sign the JWT token from the properties file.

#B We add the branch for the case in which the OTP code is not null. We consider, in this case, the client sent an OTP for the second authentication step.

#C For the second authentication step, we create an instance of type OtpAuthentication and send it to the AuthenticationManager, which will find a proper provider for it.

#D We build a JWT and store the username of the authenticated user as one of its claims. We also use the key to sign the token.

#E We add the token to the Authorization header of the HTTP response.

**NOTE** I wrote a minimal implementation of our example, and I skipped some details like treating the exceptions and logging the event. These aspects aren't essential for our example now, where I only ask you to focus only on the Spring Security components and architecture. In a real-world application, you should also implement all these details.

The following code snippet builds the JWT. I use the `setClaims()` method to add a value in the JWT body and the `signWith()` method to attach a signature to the token. For our example, I use a symmetric key to generate the signature.

```

SecretKey key = Keys.hmacShaKeyFor(
    signingKey.getBytes(StandardCharsets.UTF_8));

String jwt = Jwts.builder()
    .setClaims(Map.of("username", username))
    .signWith(key)
    .compact();

```

This key is known only by the Business Logic Server. The Business Logic Server signs the token and can use the same key to validate the token when the client calls an endpoint. For the simplicity of the example, I used here one key for all the users. In a real-world scenario, however, I would have a key for each user. As an exercise, change this application to use different keys for each user. The advantage of using individual keys for users is that if you need to invalidate all the tokens for a user, you need just to change its key.

Because we inject the value of the key used to sign the JWT from the properties, we need to change the `application.properties` file to define this value. My `application.properties` file now looks like in the next code snippet.

```
server.port=9090
auth.server.base.url=http://localhost:8080
jwt.signing.key=ymlTU8rq83...
```

If you need to see the full content of the class, remember you find the implementation in project `ssia-ch11-ex1-s2`.

We also need to add the filter that deals with the requests on all the other paths than `/login`. I'll name this filter `JwtAuthenticationFilter`. This filter expects a JWT exists in the Authorization HTTP header of the request. This filter validates the JWT by checking the signature, creates an "authenticated" `Authentication` object and adds it to the `SecurityContext`. Listing 11.24 presents the implementation of the `JwtAuthenticationFilter`.

#### **Listing 11.24 the `JwtAuthenticationFilter` class**

```
@Component
public class JwtAuthenticationFilter
    extends OncePerRequestFilter {

    @Value("${jwt.signing.key}")
    private String signingKey;

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

        String jwt = request.getHeader("Authorization");

        SecretKey key = Keys.hmacShaKeyFor(
            signingKey.getBytes(StandardCharsets.UTF_8));

        Claims claims = Jwts.parserBuilder()      #A
            .setSigningKey(key)      #A
            .build()                #A
            .parseClaimsJws(jwt)    #A
            .getBody();             #A

        String username = String.valueOf(claims.get("username"));

        GrantedAuthority a = new SimpleGrantedAuthority("user");
        var auth = new UsernamePasswordAuthentication(      #B
            username,           #B
            null,              #B
            List.of(a));        #B

        SecurityContextHolder.getContext().setAuthentication(auth);    #C

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

```

}

@Override
protected boolean shouldNotFilter(
    HttpServletRequest request) {

    return request.getServletPath().equals("/login");      #E
}

#A We parse the token to obtain the claims. This is also where the signature is verified, and an exception is thrown if
the signature isn't valid.
#B We create the Authentication instance, which we'll add to the SecurityContext.
#C We add the Authentication object in the SecurityContext.
#D We call the next filters in the filter chain.
#E We configure this filter not to be triggered on the requests for /login path.

```

**NOTE** A signed JWT is also called JWS (JSON Web Token Signed). This is why the name of the method we use  
is `parseClaimsJws()`.

#### 11.4.5 Writing the security configurations

In this section, we finalize writing the application by defining the security configurations. We have to do a few configurations so that our entire puzzle is coherent:

1. Adding the filters to the filter chain, as you learned in chapter 9.
2. Disabling the CSRF protection, because, as you learned in chapter 10, this doesn't apply when using different origins. Here, using the JWT replaces the validation that would be done with the CSRF token.
3. Adding the `AuthenticationProvider` objects to be known by the `AuthenticationManager`.
4. Using matcher methods to configure that all the requests need to be authenticated, as you learned in chapter 8.
5. Adding the `AuthenticationManager` bean in the Spring context so that we can inject it from the `InitialAuthenticationFilter` class as you saw we did in listing 11.23.

##### **Listing 11.25 The SecurityConfig class**

```

@Configuration
public class SecurityConfig
    extends WebSecurityConfigurerAdapter {      #A

    @Autowired      #B
    private InitialAuthenticationFilter initialAuthenticationFilter;

    @Autowired      #B
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Autowired      #B
    private OtpAuthenticationProvider otpAuthenticationProvider;

```

```

@Autowired #B
private UsernamePasswordAuthenticationProvider[CA]
usernamePasswordAuthenticationProvider;

@Override
protected void configure(
    AuthenticationManagerBuilder auth) {

    auth.authenticationProvider( #C
        otpAuthenticationProvider) #C
        .authenticationProvider( #C
            usernamePasswordAuthenticationProvider); #C
}

@Override
protected void configure(HttpSecurity http)
    throws Exception {

    http.csrf().disable(); #D

    http.addFilterAt( #E
        initialAuthenticationFilter, #E
        BasicAuthenticationFilter.class) #E
        .addFilterAfter( #E
            jwtAuthenticationFilter, #E
            BasicAuthenticationFilter.class #E
        ); #E

    http.authorizeRequests() #F
        .anyRequest() #F
        .authenticated(); #F
}

@Override
@Bean #G
protected AuthenticationManager authenticationManager()
    throws Exception {
    return super.authenticationManager();
}
}

```

#A We need to extend the WebSecurityConfigurerAdapter to override the configure() methods for the security configurations.

#B We autowire the filters and the authentication providers which we'll set up in the configuration.

#C We add both authentication providers to the authentication manager.

#D We disable CSRF protection.

#E We add both custom filters into the filter chain.

#F We configure that all the requests need to be authenticated.

#G We add the AuthenticationManager to the Spring context so that we can autowire it from the filter class.

#### 11.4.6 Testing the whole system

In this section, we test the implementation of the Business Logic Server. Now that everything's in place, it's time to run the two components of our system, the Authentication Server and the Business Logic Server, and examine if our custom authentication and authorization work as desired.

We have added a user and checked that the Authentication Server works properly in section 11.3. We can try the first step to authenticate to access the endpoints exposed by the Business Logic Server with the user we added in section 11.3. The Authentication Server opens port 8080, and the Business Logic Server we call now uses port 9090, which we configured earlier in the `application.properties` file of the Business Logic Server.

```
curl -H "username:danielle" -H "password:12345" http://localhost:9090/login
```

Once we've called the `/login` endpoint, providing the correct username and password, we check in the database for the generated OTP value. This should be a record in the `otp` table where the value of the `username` field is "danielle". In my case, I have the following record:

```
Username: danielle
Code: 6271
```

We assume this OTP was sent in an SMS message, and the user has it. We use it for the second authentication step. The curl command in the next code snippet shows you how to call the `/login` endpoint for the second authentication step. I'll also add the `-v` option to see the response headers where I expect to find the JWT.

```
curl -v -H "username:danielle" -H "code:6271" http://localhost:9090/login
```

The (truncated) response is:

```
...
< HTTP/1.1 200
< Authorization:
    eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbm1lbGx1In0.wg6LFProg7s_KvFxvnYGiZF-Mj4rr-
    0nJA1tVGZNn8U
...

```

The JWT is right there where we expected it to be: in the `Authorization` response header. We use the token we obtained to call the `/test` endpoint.

```
curl -H
    "Authorization:eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbm1lbGx1In0.wg6LFProg7s_KvFx
    vnYGiZF-Mj4rr-0nJA1tVGZNn8U" http://localhost:9090/test
```

The response body is:

```
Test
```

Awesome! You've finished the second hands-on chapter! You've managed to write a whole backend system and secure its resources by writing custom authentication and authorization. And you've even used JWTs for this, which brings you a significant step forward and prepares you for what's coming in the next chapters: the OAuth2 flow.

## 11.5 Summary

- When implementing custom authentication and authorization, always rely on the contracts offered by Spring Security: `AuthenticationProvider`, `AuthenticationManager`, `UserDetailsService`, etc. This approach helps you

implement an easier to understand architecture and makes your application less error-prone.

- A token is an identifier of the user. It can have any implementation as long as, after generated, the server recognizes it. Examples of tokens from real-world are an access card, a ticket, or the sticker you receive at the entrance of a museum.
- While an application can use a simple UUID as a token implementation, you'll more often find tokens implemented as JSON Web Tokens (JWT). JWTs have multiple benefits: they can store data exchanged on the request, and you can sign them to make sure they weren't changed while transferred.
- A JWT token can be signed or might be completely encrypted. A signed JWT token is called JSON Web Token Signed (JWS), and one that has its details encrypted is called JSON Web Token Encrypted (JWE).
- Avoid storing too many details within your JWT token. When signed or encrypted, the longer the token is, the more time is needed to sign or encrypt it. Also, remember that we send the token in the header of the HTTP request. The longer the token is, the more data you add to each request, which might affect the performance of your application.
- We prefer to decouple responsibilities in a system to make it easier to maintain and scale. For this reason, in the current example, we've separated the authentication in a different app, which we called the Authentication Server. The backend application serving the client, which we've called the Business Logic Server, uses the separate Authentication Server when it needs to authenticate a client.
- Multi-Factor Authentication (MFA) is an authentication strategy in which the user is asked to authenticate multiple times and in different ways to access a resource. For example, in our case, it had to use its username and password then prove they have access to a specific phone number by validating an OTP received through an SMS message. This way, the user's resources are better protected against credentials theft.
- In many cases, you'll find more than one good solution to solving a problem. Always consider all the possible solutions and if time allows, implement proof-of-concepts for all so understand better, which fits your scenario.

# 12

## How does OAuth 2 work?

### This chapter covers

- What is OAuth 2
- An introduction to implementing the OAuth 2 framework with Spring Security
- Developing an application which uses Single Sign-On (SSO) with OAuth 2

If you're already working with OAuth 2, I know what you're thinking: OAuth 2 framework is a vast subject that could take an entire book to cover. And I can't argue with this, but with four chapters, you'll learn everything you need to know about applying OAuth 2 with Spring Security. We start in this chapter with an overview where you'll discover that the main actors in the OAuth 2 framework are the user, the Client, the resource server, and the authorization server. After the general introduction, you'll learn how to use Spring Security to implement the Client. Then, in chapters 13 to 15, we discuss implementing the last two components: the resource server and the authorization server. I'll give you examples and apps you can adapt to any of your real-world scenarios.

To reach this goal, in this chapter, we discuss what OAuth 2 is, and then we apply it within an application focused on authentication with single sign-on (SSO). The reason why I like starting teaching this subject with the example of a Single Sign-On (SSO): *It's very simple but also very useful* – it allows you to have an overview of OAuth 2, and it gives you the satisfaction of implementing a fully working application without writing too much code.

In chapters 13 to 15, we'll apply what we discuss in this chapter with code examples, as you are already used to from the previous chapters of this book. Once we finish these four chapters, you'll have an excellent overview of the things you need for implementing OAuth 2 with Spring Security in your applications.

As OAuth 2 is such a big subject, I'll refer, where appropriate, to different resources I consider essential for you to read. However, I won't scare you. Spring Security makes the development of applications with OAuth 2 easy. The only prerequisites you need to get started are chapters 2 to 11 of this book, in which you learned the general architecture of

authentication and authorization in Spring Security. What we'll discuss about OAuth 2 is based on the same foundation of the standard authorization and authentication architecture of Spring Security.

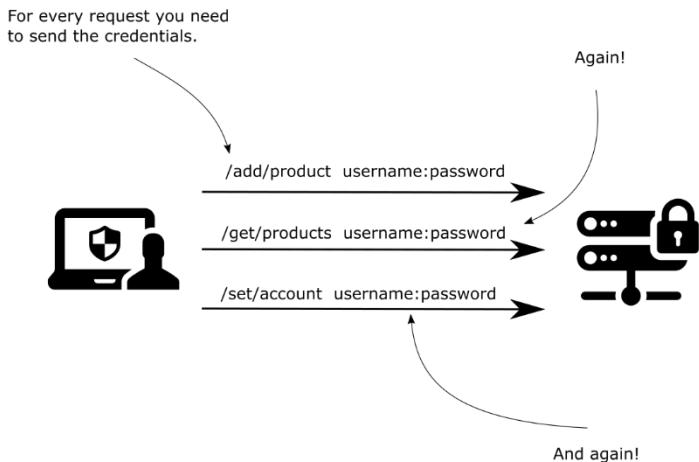
## 12.1 The OAuth 2 framework

In this section, we discuss the OAuth 2 framework. Today, OAuth 2 is pretty common in securing web applications, so, probably, you've already heard about it. The chances are big that you'll need to apply OAuth 2 in your applications. And this is why we need to discuss applying OAuth 2 in Spring applications with Spring Security. We'll start with a little bit of theory and then move to apply it with an application using single sign-on.

In most cases, OAuth 2 is referred to as an authorization framework (specification framework) whose primary purpose is to define a way to allow an entity to access a resource. The access to the resource is done on behalf of somebody else without needing to impersonate them. Sometimes you'll also see people referring to OAuth 2 as being a delegation protocol. Whatever you choose to name it, it's important to remember that OAuth 2 is not a specific implementation or a library. You could as well apply the OAuth 2 flow definitions with other platforms, tools, or languages. In this book, you'll find out how to implement OAuth 2 with Spring Boot and Spring Security.

I think that a great way to understand what OAuth 2 is and its usefulness is to start the discussion with examples we've already analyzed in this book. The most trivial way to authenticate, of which you saw in plenty of examples up to now, is the HTTP Basic authentication method. Wouldn't this be enough in our systems such that we don't have to add more complexity? No. With HTTP Basic authentication, we have two issues we need to take into consideration:

- Sending credentials for each and every request (figure 12.1).
- Having the credentials of the users managed by a separate system.



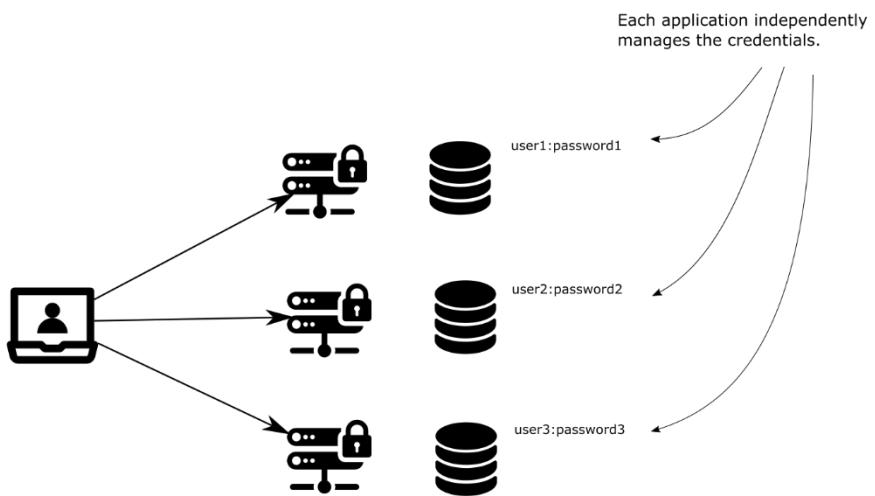
**Figure 12.1** When using HTTP Basic authentication method, you need to send the credentials and repeat the authentication logic with all the requests. This approach implies sharing them often over the network.

*Sending credentials for each and every request* might work with isolated cases, but isn't generally desired because it assumes:

1. Sharing the credentials often over network.
2. Having the Client (browser, in case of a web application) store those credentials somehow so that the Client can send those credentials to the server with the request to get authenticated and authorized.

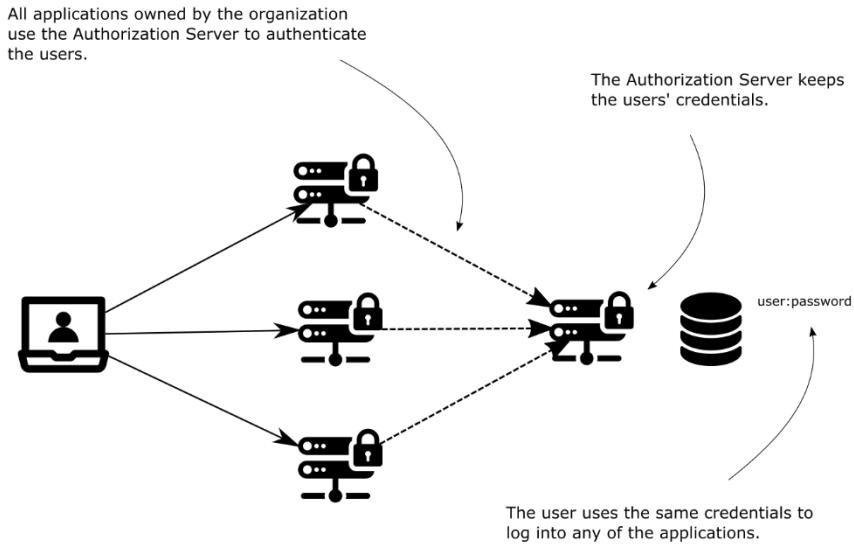
We want to get rid of these two points from our applications' architecture, because they weaken the security by making the credentials vulnerable.

Most often, we'll want to have a *separate system manage the users' credentials*. Imagine just for a second that you'd have to configure and use separate credentials for all the applications you work within your organization (figure 12.2).



**Figure 12.2** In an organization, you work with multiple applications. Most of them need you to authenticate for using them. It would be challenging both for you to know multiple passwords, and for the organization to manage multiple sets of credentials.

It would be better if we isolated the responsibility for the credentials management in one component of our system – let's call it, for now, the Authorization Server (figure 12.3).

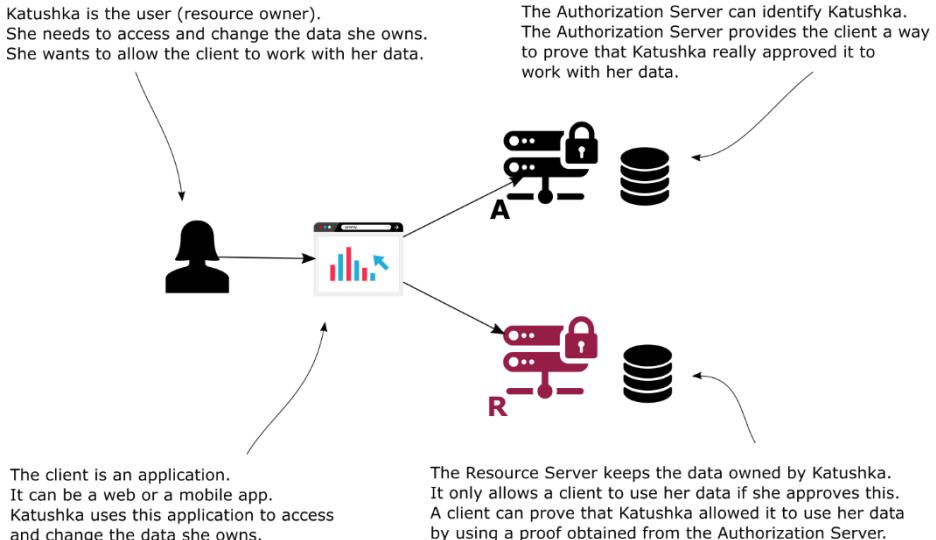


**Figure 12.3** An easier-to-maintain architecture would keep the credentials separately and allow all the applications to use the same set of credentials for users.

This approach would eliminate the duplication of credentials representing the same individual. In this way, the architecture becomes simpler and easier to maintain.

## 12.2 The components of the OAuth 2 authentication architecture

In this section, we discuss the components which act in an OAuth 2 authentication implementations. You need to know these components and the role they play, as we'll refer to them from now on in the next sections. I'll also refer to them through the rest of the book wherever we'll write an implementation related to OAuth 2. In this section, we only discuss what these components are, and their purpose (figure 12.4). As you'll learn in section 12.3, there are more ways in which they "talk" one to the other. And in section 12.3, you'll learn about different flows that cause different interactions between these components.



**Figure 12.4** The main components of the OAuth 2 architecture are the resource owner, the Client, the authorization server, and the resource server. Each of them has its own responsibility, essential in the authentication and authorization process.

- *The resource server* – The application hosting resources owned by users. Resources can be users' data or actions they are authorized to do.
- *The user (also known as the resource owner)* – This is, of course, the individual who owns resources exposed by the resource server. A user generally has a username and a password which they use to identify themselves.
- *The Client* – an application that accesses the resources owned by the user, on behalf of the user. The Client uses a client ID and a client secret to identify itself. Be careful, these credentials are not the same as the user's credentials. The Client needs its own credentials to identify itself when making a request.
- *The authorization server* – the application which authorizes the Client to access the users' resources exposed by the resource server. When the authorization server decides that a client is authorized to access a resource on behalf of the user, it issues a token. The Client uses this token to prove to the resource server that it was authorized by the authorization server. The resource server allows the Client to access the resource they requested if the Client has a valid token.

### 12.3 Implementation choices with OAuth 2

In this section, we discuss how to apply OAuth 2, depending on the architecture of your application. As you'll learn, OAuth 2 implies multiple possible authentication flows, and you need to know which one applies to your case. I'll take the most common cases and evaluate them. It's important to do this before starting with the first implementation so that you know what you're implementing.

So how does OAuth 2 work then? What does it mean to implement OAuth 2 authentication and authorization?

Mainly, OAuth 2 refers to using tokens for authorization. Remember from section 11.2 that tokens are like access cards. Once you obtain a token, you can access specific resources. But OAuth 2 offers multiple possibilities for obtaining the token. We call these ways to obtain the token *grants*.

Here are the most common OAuth 2 grants you can choose from:

- Authorization Code
- Password
- Refresh Token
- Client Credentials

So, when starting an implementation, we need to choose our grant. Do we select it randomly? Of course not, we need to know how tokens are created in each of the grant types. Then, depending on our application requirements, we choose one of them. Let's analyze each one of them shortly, and where it applies. You'll also find an excellent discussion about grant types in section 6.1 of *OAuth 2 In Action* by Justin Richer and Antonio Sanso (Manning, 2017).

<https://livebook.manning.com/book/oauth-2-in-action/chapter-6/6>

### 12.3.1 Implementing the authorization code grant type

In this section, we discuss the Authorization Code grant type (figure 12.5). We'll also use this grant type in the application we implement in section 12.5. This grant type is one of the most used OAuth 2 flows, so it's quite important to understand how it works and how to apply it. There's a high probability that you'll use it in the applications you develop.

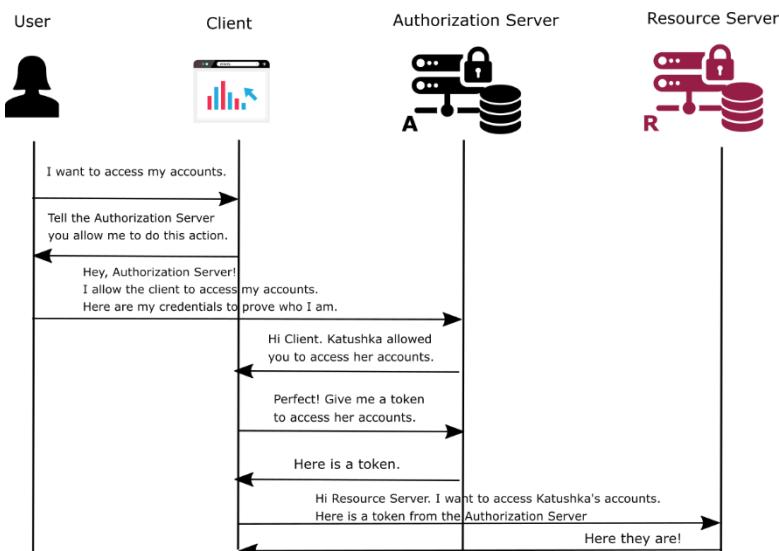


Figure 12.5 The authorization code grant type. The Client asks the user to interact directly with the

Authorization Server to grant them permission for the user's request. Once authorized, the Authorization Server issues a token that the Client uses to access the user resources.

**NOTE** The arrows in the diagram don't necessarily represent HTTP requests and responses. They represent messages exchange among the actors of OAuth 2. For example, when the Client tells the User "Tell the Authorization Server you allow me to do this action" (second arrow on the diagram), the Client redirects there the user to the Authorization Server login page. Same, when the Authorization Server gives the Client an access token, the Authorization Server actually calls the Client on what we call a redirect URI. You'll learn all these throughout chapters 12 to 15, so no worries about it. With this note, I just wanted to make you aware from the beginning that these sequence diagrams are not just representing HTTP requests and responses. They're a simplified description of the communication among the OAuth 2 actors.

Here's how the authorization code grant type works step by step:

1. Making the authentication request.
2. Obtaining the access token
3. Calling the protected resource

#### **STEP 1: MAKING THE AUTHENTICATION REQUEST WITH THE AUTHORIZATION CODE GRANT TYPE**

The Client redirects the user to an endpoint of the authorization server where they need to authenticate. You can imagine you are using app X, and you need to access a protected resource. To access that resource for you, app X needs you to authenticate. It opens a page for you with a login form on the authorization server where you must fill in your credentials.

**NOTE** What's really important to observe here is that the user interacts directly with the authorization server. The user doesn't send the credentials to the Client app.

What technically happens here is that, when the Client redirects the user to the authorization server, the Client calls the authorization endpoint with the following details in the request query:

- **response\_type** with the value "code" tells the authorization server that the Client expects a code. The Client needs the code to obtain an access token, as you'll see in the second step.
- **client\_id** with the value of the client ID, which identifies the application itself.
- **redirect\_uri** – tells the authorization server where to redirect back the user after successful authentication. Sometimes the authorization server already knows for each Client a default redirect URI. For this reason, you'll see the Client doesn't send the redirect URI anymore.
- **scope** – you can think of scopes as the granted authorities we discussed in chapter 5.
- **state** – a Cross-Site Request Forgery (CSRF) token, used for CSRF protection. We discussed CSRF in chapter 10.

After successful authentication, the authorization server calls back the Client on the redirect URI and provides a code and the state value. The client checks that the state value is the same as the one it sent in the request to confirm that was not someone else to call the redirect URI.

The Client will use the code to obtain an access token, as presented in step 2.

#### STEP 2: OBTAINING THE ACCESS TOKEN WITH THE AUTHORIZATION CODE GRANT TYPE

The code resulted from step 1 is the Client's proof that the user authenticated to allow them to access resources. You guessed correctly, this is why this grant type is called "authorization code." Now the Client calls the authorization server with the code they have to get the token.

**NOTE** Mind again. In the first step, the interaction was between the user and the authorization server. In this step, the interaction is between the Client and the authorization server (figure 12.6).

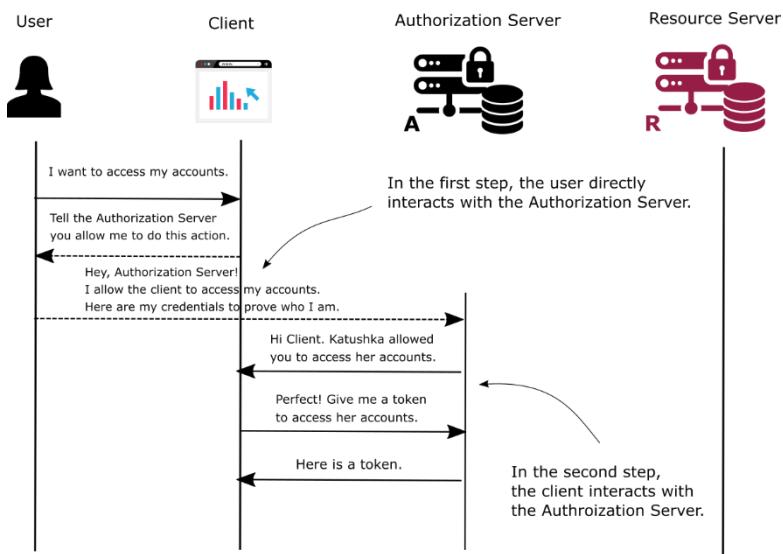


Figure 12.6 The first step implies direct interaction between the user and the Authorization Server. In the second step, the Client requests an access token from the Authorization Server providing the authorization code obtained previously in step 1.

In many cases, these first two steps create confusion. People are generally confused about why the flow needs two calls to the authorization server and two different "tokens": the authorization code and the access token. Take a moment to understand this.

- The first code is generated as proof that the user directly interacted with the authorization server and successfully authenticated to allow grants to the Client.
- The second token is the key that the Client can use to access resources on the resource server.

So why didn't the authorization server directly return the second token (access token) then? Well, OAuth 2 defines a flow called the "implicit grant type" where the authorization server directly returns the access token. The implicit grant type is not enumerated in section 12.3 because it's not recommended to be used, and most of the authorization servers today don't allow using it. The simple fact that the authorization server would call the redirect URI

directly with the access token without making sure that it was indeed the right Client receiving that token makes the flow less secure. By sending an authorization code first, the Client has to prove again who they are by using their credentials to obtain the access token. So the Client will make a final call to get the access token sending

- the authorization code (which proves the user authorized them)
- their credentials (to prove they really are that Client and not someone else who intercepted the authorization codes)

To return to step 2 – technically – the Client makes now a request to the authorization server containing:

- **code** – authorization code received in step 1, which proves the user authenticated.
- **client\_id** and **client\_secret** – which are the credentials of the Client.
- **redirect\_uri** – should be the same as the one used in step 1 for validation.
- **grant\_type** – needs to have the value “`authorization_code`” and identifies, of course, the kind of flow used. A server might support multiple flows, so it’s essential always to specify which is the current executed authentication flow.

As a response, the server sends back the `access_token`. This `access_token` is a value that the Client can use to call resources exposed by the resource server.

### **STEP 3: CALLING THE PROTECTED RESOURCE WITH THE AUTHORIZATION CODE GRANT TYPE**

The Client uses the access token in the `Authorization` request header when calling an endpoint of the resource server.

#### **AN ANALOGY FOR THE AUTHORIZATION CODE GRANT TYPE**

I'll end this section with an analogy to this flow. I sometimes buy books from a small shop I've known for ages. I have to order books in advance and then pick them up a couple of days later. But the shop isn't on my daily route, so I can't go myself to collect the books. I usually ask a friend who lives near me to go there and collect them for me. When my friend asks for my order, the lady from the shop calls me to confirm I've sent someone to collect my books. After I confirm, my friend collects the package and brings it to me later in the evening.

In this analogy, the books are the resources. I own them, so I'm the user (resource owner). My friend picking them up for me is the Client. The lady selling the books is the authorization server (we can also consider her or the book store as being the Resource Server). Observe that, to grant permission to my friend (Client) to collect the books (resources), the lady (authorization server) selling the books called me (user) directly. This analogy describes the processes of the authorization code and the implicit grant types. Of course, because we have no token in the story, the analogy is partial and describes both cases.

**NOTE** The Authorization Code grant type has the great advantage of enabling the user to allow a client to execute specific actions without needing to share their credentials with the Client. But the Authorization Code grant type has a weakness: what happens if someone intercepts the authorization code? Of course, as we discussed, the Client needs to authenticate with its credentials. But if the Client credentials are also stolen somehow? Even if this scenario isn't easy to achieve, we can consider it a vulnerability of the grant type. To mitigate this vulnerability of the flow, you need to rely on a more complex scenario, as presented by the Proof

Key for Code Exchange (PKCE) Authorization Code grant type. You find an excellent description of the PKCE Authorization Code grant type directly in the RFC 7636: <https://tools.ietf.org/html/rfc7636>. For an excellent discussion on this subject, I also recommend you read section 7.3.2 of API Security in Action by Neil Madden (Manning, 2020). <https://livebook.manning.com/book/api-security-in-action/chapter-7/v-8/123>

### 12.3.2 Implementing the password grant type

In this section, we discuss the password grant type (figure 12.7). This grant type is also known as the resource owner credentials grant type. Applications using this flow assume that the Client collects the user's credentials and uses them to authenticate and obtain an access token from the authorization server. Remember our hands-on example in chapter 11? That architecture we implemented is quite close to what happens in the password grant type. We'll also implement a "real" OAuth 2 password grant type architecture with Spring Security in chapters 13 to 15.

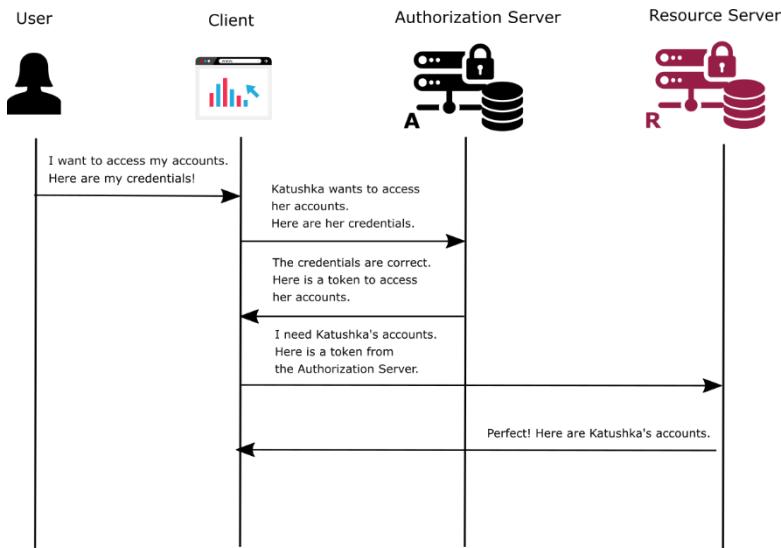


Figure 12.7 The password grant type assumes that the user shares their credentials with the Client. The Client uses them to obtain the token from the Authorization Server. It then accesses the resources from the resource server on behalf of the user.

**NOTE** You might already wonder at this point about how the Resource Server knows whether a token is valid. In chapters 13 and 14, we discuss the approaches a Resource Server uses to validate the token. For the moment, I'd like to allow you to focus on the grant types discussion, and we'll only refer to how the Authorization Server issues access tokens.

You'll use this flow only if the Client and authorization server are built and maintained by the same organization. Why? Let's assume you build a microservices system, and you decide

to separate the authentication responsibility as a different microservice. You would do this separation to enhance scalability and to keep responsibilities separated for each service. (This separation is used widely in many systems. Let's assume further that your system's users use either a client web application developed with a frontend framework like Angular, ReactJS, or Vue.js. Or, they use a mobile app. In this case, the users might consider it strange to be redirected *from* your system *to* the same system for authentication and then back. This is what would happen with a flow like the authorization code grant type. You would instead expect to have the application present a login form to the user, and let the Client take care of sending the credentials to the server to authenticate. The user doesn't need to know how you designed the authentication responsibility in your application.

Let's see what happens when using the password grant type. The two tasks are:

1. Requesting the access token
2. Using the access token to call resources

#### **STEP 1: REQUESTING THE ACCESS TOKEN WHEN USING THE PASSWORD GRANT TYPE**

The flow is much simpler with the password grant type. The Client collects the user's credentials and calls the authorization server to obtain the access token. When requesting the access token, the Client also sends in the request the following details:

- **grant\_type** with the value "password"
- **client\_id** and **client\_secret** – the credentials used by the Client to authenticate itself
- **scope** – which you can understand as the granted authorities
- **username** and **password** – which are the credentials of the user. They are sent in plain text as values of the request headers.
- The Client receives back the access token in the response.

#### **STEP 2: USING THE ACCESS TOKEN TO CALL RESOURCES WHEN USING THE PASSWORD GRANT TYPE**

Exactly as for the authorization code grant type, once the Client has an access token, it uses the token to call the endpoints on the resource server. The Client adds the access token to the requests in the `Authorization` request header.

#### **THE PASSWORD GRANT TYPE IN THE ANALOGY**

To refer back to the analogy I made in section 12.3.1, imagine the lady selling the books won't call me to confirm I want my friend to collect the books. I would instead give my ID to my friend to prove that I delegated my friend to get the books. See the difference? In this flow, I need to share my ID (credentials) with the Client. For this reason, we say that this grant type applies only if the resource owner "trusts" the Client.

**NOTE** The password grant type is less secure than the authorization code grant type, mainly because it assumes sharing the user credentials with the client app. While it's true that it's more straightforward than the authorization code grant type, and this is the main reason you'll also find it used plenty in theoretical examples, try to avoid it in real-world scenarios. Even if the Authorization Server and the Client are both built by the same organization, you should first think about using the authorization code grant type. Take the password grant type as your second option.

### 12.3.3 Implementing the client credentials grant type

In this section, we discuss the client credentials grant type (figure 12.8). This is the simplest of the grant types described by OAuth 2. You use it when no user is involved – that is when implementing authentication between two applications. I like to think about the client credentials grant type being a combination between the password grant type and an API key authentication flow.

We assume you have a system that implements authentication with OAuth 2. Now you need to allow an external server to authenticate and call a specific resource that your server exposes. In chapter 9, we discussed filter implementations. You learned how to create a custom filter to augment your implementation with a case of authentication using an API key. You could still apply this approach using an OAuth 2 approach. And if your implementation uses OAuth 2, it's undoubtedly cleaner to use the OAuth 2 framework in all the cases rather than augmenting it with a custom filter in your implementation, which lies outside of the OAuth 2 framework.

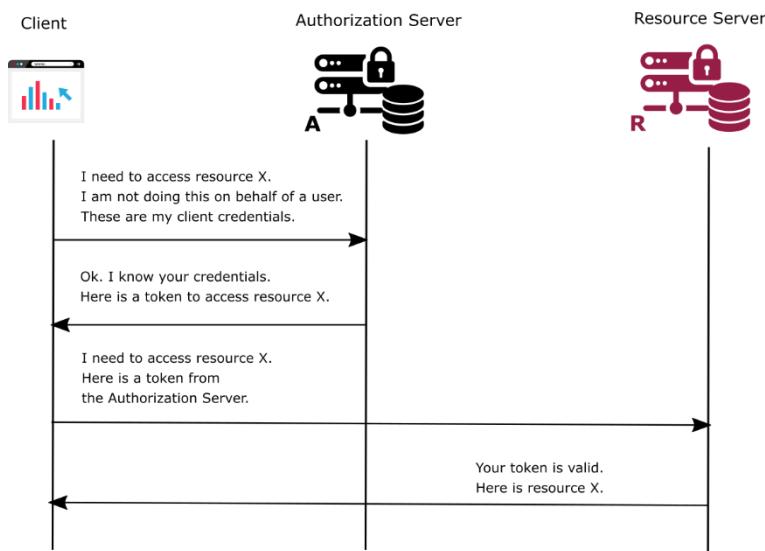


Figure 12.8 The client credentials grant type. We use this flow if a client needs to access a resource, but not on behalf of a resource owner. This resource can be an endpoint that isn't owned by a user.

The steps for the client credentials grant type are very similar as for the password grant type, just that the request for the access token doesn't need any user credentials:

1. Requesting the access token
2. Using the access token to call resources

#### **STEP 1: USING THE ACCESS TOKEN WITH THE CLIENT CREDENTIAL GRANT TYPE**

To obtain an access token, the Client requests the authorization server with the following details:

- **grant\_type** with the value “client\_credentials”
- **client\_id** and **client\_secret** representing the client credentials
- **scope** which represents the granted authorities

In response, the Client receives an access token. The Client can now use the access token to call endpoints of the resource server.

#### **STEP 2: USING THE ACCESS TOKEN TO CALL RESOURCES WITH THE CLIENT CREDENTIAL GRANT TYPE**

Exactly as for the authorization code grant type and password grant type, once the Client has an access token, it uses that token to call the endpoints on the resource server. The Client adds the access token value to the requests in the `Authorization` request header.

#### **12.3.4 Using refresh tokens to obtain new access tokens**

In this section, we discuss refresh tokens. Up to now, you learned that the result of an OAuth 2 flow, which we also name grant, is an access token. But we didn’t discuss too much about this token. In the end, OAuth 2 doesn’t assume a specific implementation for tokens. But what you’ll learn now is that a token, no matter how implemented, might expire. It’s not mandatory—you can create tokens with an infinite lifespan—but in general, you’d prefer making them short-lived. The refresh tokens, of which we discuss in this section, represent an alternative to using the credentials for obtaining a new access token again. I’ll show you now how the refresh tokens work in OAuth 2, and you’ll also see them implemented within an application in chapter 13.

Let’s assume in your app, you implemented tokens that never expire. That means that the Client can use the same token, again and again, to call resources on the resource server. What if the token is stolen? In the end, don’t forget that the token is attached as a simple HTTP header on each and every request. If the token doesn’t expire, someone who gets their hands on the token can use it to access resources. A token that doesn’t expire is too powerful. It becomes almost as powerful as the user credentials. We prefer to avoid this and make it short-lived. This way, at some point, an expired token can’t be used anymore. The Client has to obtain another access token.

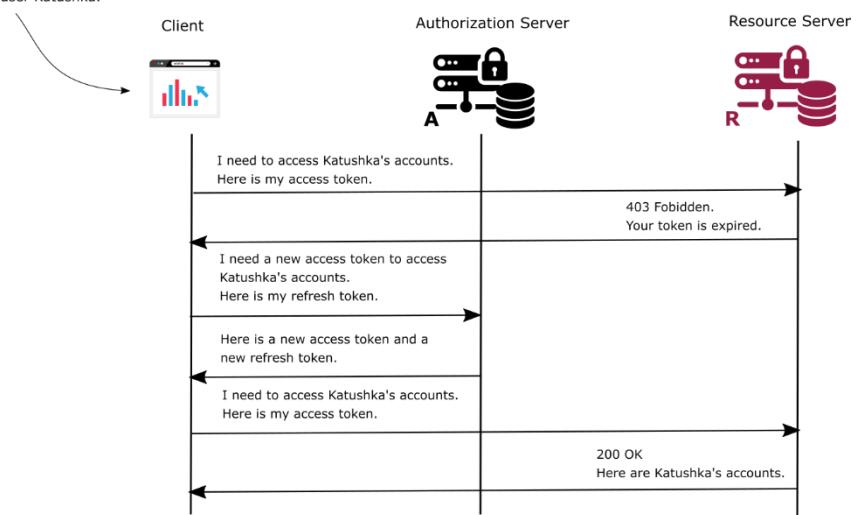
To obtain a new access token, the Client can rerun the flow depending on the grant type used. For example, if the grant type is authentication code, the Client would redirect the user to the authorization server login endpoint, and the user must again fill in their username and password. Not really user-friendly, is it? Imagine that the token has a 20 minutes lifespan and you work a couple of hours with the online app, the app would redirect you back to log in for about six times. Oh no! That app logged me out again!

To avoid the need to re-authenticate, the authorization server can issue a refresh token. This refresh token has a different value and purpose than the access token. The app uses the refresh token to obtain a new access token instead of having to re-authenticate.

Refresh tokens also have advantages over reauthentication in the password grant type. Even if with the password grant type, if we don’t use refresh tokens, we either would have to

ask the user again to authenticate or store their credentials. Storing the users' credentials when using the password grant is one of the biggest mistakes you could do! And I've seen this approach used in real applications! Don't do it! If you store the username and password – and assuming you save them as plaintext or something reversible because you have to be able to reuse them – you expose those credentials. Refresh tokens help you solve this problem easier and safer. Instead of unsafely storing the credentials and without needing to redirect the user every time, you can store a refresh token and use it to obtain a new access token when needed. Storing the refresh token is safer because you can revoke it if you find that it was exposed. Moreover, don't forget that people tend to have the same credentials for multiple apps. So losing the credentials is way worse than losing a token that one could use with a specific application.

The client has a token previously issued for user Katushka.



**Figure 12.9 The refresh token grant: The Client has an access token, which expired. To avoid forcing the user to log in again, the Client uses the refresh token to issue a new access token.**

Finally, let's find out how to use a refresh token. Where do you get a refresh token from? The authorization server returns the refresh token together with the access token when using a flow like authorization code grant or password grant. With the client credentials grant, there's no refresh token because this flow doesn't need user credentials. Once the Client has a refresh token, when the access token expires, the Client should issue a request with the following details:

- **grant\_type** with value "refresh\_token"
- **refresh\_token** with the value of the refresh token
- **client\_id** and **client\_secret** with the client credentials
- **scope** – can define the same granted authorities or less. If more granted authorities

need to be authorized, a re-authentication is needed.

In response to this request, the authorization server issues a new access token and a new refresh token.

## 12.4 The sins of OAuth 2

In this section, we discuss possible vulnerabilities of the applications using OAuth 2 authentication and authorization. It's important to understand what can go wrong when using OAuth 2 so that you can avoid these scenarios when developing your applications. Of course, like anything else in software development, OAuth 2 isn't bulletproof. It has its vulnerabilities of which we must be aware and which we must consider when building our applications. I enumerate here some of the most common:

- *Using Cross-Site Request Forgery (CSRF) on the Client* – With a user logged in, CSRF is possible if the application doesn't apply any CSRF protection mechanism. We had a great discussion on the CSRF protection implemented by Spring Security in chapter 10.
- *Stealing client credentials* – Storing or transferring the credentials unprotected can create breaches that allow attackers to steal and use them.
- *Replaying tokens* – As you'll understand in chapters 13 and 14, tokens are the "keys" we use within an OAuth 2 authentication and authorization architecture to access the resources. You send them over network, and sometimes they might be intercepted. If intercepted, they are stolen and can be reused. Imagine you lose the key from your home's front door. What could happen? Somebody else could use it to open the door as many times as they like (replay). We'll learn in chapter 14 more about tokens and how to avoid token replaying.
- *Token hijacking* – which implies you interfere in the authentication process and steal tokens that you can use to access resources. This is also a potential vulnerability of using refresh tokens, as they, as well, can be intercepted and used to obtain new access tokens. I recommend this helpful article <http://blog.intothesymmetry.com/2015/06/on-oauth-token-hijacks-for-fun-and.html>

Remember, OAuth 2 is a framework. The vulnerabilities are the result of wrongly implementing functionality over it. Using Spring Security already helps us mitigate most of those vulnerabilities in our applications. When implementing an application with Spring Security, as you'll see in this chapter, we need to set the configurations, but we rely on the flow as implemented by Spring Security.

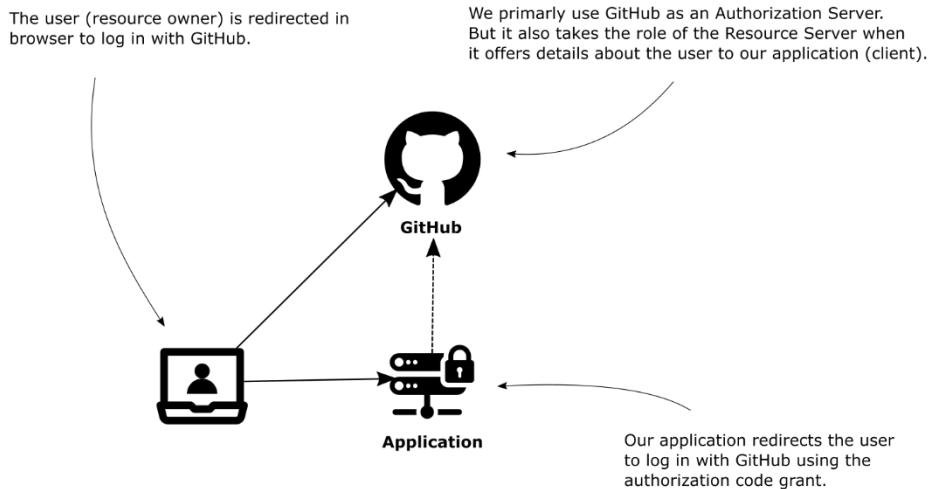
For more details on vulnerabilities related to the OAuth 2 framework and how could a bad intentioned individual exploit them, you find a great discussion in part 3 of the OAuth 2 In Action by Justin Richer and Antonio Sanso (Manning, 2017).

<https://livebook.manning.com/book/oauth-2-in-action/part-3>

## 12.5 Implementing a simple Single Sign-On application

In this section, we implement the first application of our book that uses the OAuth 2 framework with Spring Boot and Spring Security. This example shows you a general overview of how to apply OAuth 2 with Spring Security and teaches you some of the first contracts you need to

know. A single sign-on application is, as the name suggests, one in which you authenticate through an authorization server, and then the app keeps you logged in, using a refresh token. In our case, it represents only the Client from the OAuth 2 architecture (figure 12.10). In this application, we'll use GitHub as the Authorization Server and Resource Server, and we'll focus on the communication between the components with the authorization code grant. In chapters 13 and 14, we'll implement both an Authorization Server and a Resource Server in an OAuth 2 architecture.



**Figure 12.10** Our application takes the role of the Client in the OAuth 2 architecture. We use GitHub as the authorization server, but also GitHub takes the role of the Resource Server when it allows us to retrieve details of the user.

### 12.5.1 Managing the authorization server

In this section, we'll configure the authorization server. In this chapter, we won't implement our own authorization server, but instead, we'll use an existing one. I chose to use GitHub in this example as the authorization server. In chapter 13, you'll learn how to implement your own authorization server.

So what should we do to use a third-party like GitHub as the authorization server? This means in the end that our application won't manage its users, but a person would be able to log in to our application using their GitHub account. Like any other authorization server, GitHub needs to know the client application to which it issues tokens. Remember, in section 12.3, where we discussed the OAuth 2 grants, the requests used a client ID and a client secret. A client uses these credentials to authenticate itself at the authorization server. The OAuth application must be registered with the GitHub authorization server. To do this, complete a short form using the following link:

<https://github.com/settings/applications/new>

Here, you add a new OAuth application by filling the form presented in figure 12.11. You need to specify a name for the application, the homepage, and the link to which GitHub will make the call back to your application. The OAuth 2 grant type on which this works is the authorization code grant type. This grant type assumes that the Client redirects the user to the authorization server (GitHub in our case) for login, and then the authorization server calls back the Client at a defined URL, as we discussed in section 12.3.1. This is why you need to fill in this callback URL here. In both cases, because I'll run the example on my system, I'll use the localhost. And because I won't change the port (which is 8080 by default, as you already know). This makes my homepage URL be <http://localhost:8080>, and I'll use the same URL for the callback.

**NOTE** It will be the client-side of GitHub (which is from your browser) calling the localhost. This is the reason why you can test your application locally.

The screenshot shows a web browser window with the URL `github.com/settings/applications/new`. The page title is "New OAuth Application". The main content is a form titled "Register a new OAuth application". The form fields are as follows:

- Application name \***: `spring_security_in_action`
- Homepage URL \***: `http://localhost:8080`
- Application description**: A placeholder text area containing "Application description is optional". Below it, a note says "This is displayed to all users of your application."
- Authorization callback URL \***: `http://localhost:8080`

At the bottom of the form are two buttons: a green **Register application** button and a blue **Cancel** button.

**Figure 12.11** To use your application as an OAuth 2 client with GitHub as the authorization server, you must register it first. You do this by filling the form to add a new OAuth application on GitHub.

Once you fill out the form and press **Register application**, GitHub provides you a client ID and a client secret you can use (figure 12.12).

spring\_security\_in\_action

**ispil** owns this application.

Transfer ownership

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. [List this application in the Marketplace](#)

**1 USER**

Client ID  
a7553955a0c534ec5e6b

Client Secret  
1795b30b425ebb79e424afa51913f1c724da0dbb

[Revoke all user tokens](#) [Reset client secret](#)

Application logo

Drag & drop [Upload new logo](#)  
You can also drag and drop a picture from your computer.

Application name \*

spring\_security\_in\_action

Something users will recognize and trust.

**Figure 12.12** When you register an OAuth application with GitHub, you receive the credentials for your Client. You'll use them in your application configuration.

**NOTE** I deleted the application you see in the image. As these credentials offer access to confidential information, I cannot let them stay alive. For this reason, you can't reuse them, and you'll need to generate your own, as presented in this section. Also, be very careful when writing an application using such credentials, especially if you use a public Git repo to store it.

This configuration is everything we needed to do for the authorization server. Now that we have the client credentials, we can start working on our application.

### 12.5.2 Starting the implementation

In this section, we begin implementing a single sign-on application. You find this example as project `ssia-ch12-ex1`. We create a new Spring Boot application and add the following dependencies to the `pom.xml` file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</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>
```

We need first to have something to secure: a web page. So we create a controller class and a simple HTML page which represents our application. Listing 12.1 presents the `MainController` class, which defines the single endpoint of our app.

### **Listing 12.1 The controller class**

```

@Controller
public class MainController {

    @GetMapping("/")
    public String main() {
        return "main.html";
    }
}
```

I'll also define the `main.html` page in the `resources/static` folder of my Spring Boot project. It contains only a heading text so that I observe the following when I've accessed the page:

```
<h1>Hello there!</h1>
```

And now the real job! Let's make the security configurations to allow our application to use the login with GitHub. We start by writing a configuration class as we are used to. We extend the `WebSecurityConfigurerAdapter` and override the `configure(HttpSecurity http)` method. And now a difference: instead of using `httpBasic()` or `formLogin()` as you learned in chapter 4, we call a different method named `oauth2Login()`. This code is presented in listing 12.2.

### **Listing 12.2 The configuration class**

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

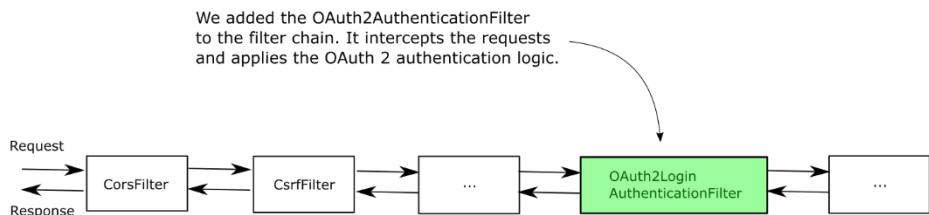
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();      #A

        http.authorizeRequests()   #B
            .anyRequest()         #B
                .authenticated();  #B
    }
}
```

#A We set the authentication method.

#B We specify that a user needs to be authenticated to make any request.

We called a new method on the `HttpSecurity` object: the `oauth2Login()`. But you know for sure what's going on. Of course, like in the case of the `httpBasic()` or `formLogin()` it simply adds a new authentication filter to the filter chain. We discussed filters in chapter 9, where you learned that Spring Security has some filter implementations, and you can also add custom ones to the filter chain. In this case, the filter that the framework will add to the filter chain when you call the `oauth2Login()` method is the `OAuth2LoginAuthenticationFilter` (figure 12.13). This filter intercepts the requests and applies the needed logic for OAuth 2 authentication.



**Figure 12.13** By calling the `oauth2Login()` method on the `HttpSecurity` object, we added the `OAuth2LoginAuthenticationFilter` to the filter chain. It intercepts the request and applies the OAuth 2 authentication logic.

### 12.5.3 Implementing the ClientRegistration

In this section, we discuss implementing the link between the OAuth 2 client and the authorization server. This is vital if you want your application to really do something. If you start it as is right now, you won't be able to access the main page. The reason why you can't access the pages is that you have specified that for any request, the user needs to authenticate, but you didn't provide any way to authenticate. We need to set up somehow that GitHub is, in our case, the authorization server. For this purpose, Spring Security defines the `ClientRegistration` contract.

The `ClientRegistration` interface represents the Client in the OAuth 2 architecture. For the Client, you have to define all its needed details among which we have:

- The client ID and secret
- The grant type used for authentication
- The redirect URI
- The scopes

You remember from section 12.3 that the application needs all these details for the authentication process. Spring Security also offers an easy way to create an instance of this type: a builder, very similar to the one you've already used to build `UserDetails` instances starting with chapter 2. Listing 12.3 shows you how to build such an instance representing our needed client implementation with the builder Spring Security provides. First, in listing 12.3, I show you how to provide all the details, but for some known providers you'll learn later in this section it's even easier than this.

### **Listing 12.3 Creating a `ClientRegistration` instance**

```
ClientRegistration cr =
    ClientRegistration.withRegistrationId("github")
        .clientId("a7553955a0c534ec5e6b")
        .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
        .scope(new String[]{"read:user"})
        .authorizationUri(
            "https://github.com/login/oauth/authorize")
        .tokenUri("https://github.com/login/oauth/access_token")
        .userInfoUri("https://api.github.com/user")
        .userNameAttributeName("id")
        .clientName("GitHub")
        .build();
```

Oh! Where did all those details come from? I know listing 12.3 might look scary at first glance, but it's nothing more than setting up the client ID and secret. Also, in listing 12.3, I've defined the scopes (granted authorities), a client name, and a registration ID at my choice.

Besides these details I had to provide the URLs of the authorization server:

- **Authorization URI** – the URI to which the Client redirects the user for authentication.
- **Token URI** – the URI which the Client calls to obtain the access token and refresh token as we discussed in section 12.3.
- **User info URI** – A URI that the Client can call after obtaining an access token to get more details about the user.

Where did I get all those URIs? Well, if the authorization server is not developed by you – like in our case – you need to get them from the documentation. For GitHub, for example, you could find them here:

<https://developer.github.com/apps/building-oauth-apps/authorizing-oauth-apps/>

Wait! Spring Security is even smarter than this. The framework defines a class named `CommonOAuth2Provider`. This class partially defines the instances of `ClientRegistration` for the most common providers you would use for authentication:

- Google
- GitHub
- Facebook
- Okta

If you use one of these providers, you can define your `ClientRegistration` as presented in listing 12.4.

### **Listing 12.4 Using the `CommonOAuth2Provider` class**

```

ClientRegistration cr =
    CommonOAuth2Provider.GITHUB      #A
        .getBuilder("github")       #B
            .clientId("a7553955a0c534ec5e6b")   #C
            .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")   #C
        .build();      #D

```

#A We select the GitHub provider to have the corresponding URIs already set.

#B We give an ID for the client registration.

#C We set the client credentials.

#D We build the ClientRegistration instance.

As you can see, much cleaner, and you don't have to find and set the URLs for the authorization server manually. Of course, this applies only to common providers. If your authorization server is not among the common providers, then you have no other option than defining the `ClientRegistration` entirely, as presented by listing 12.3.

**NOTE** Using the values from the `CommonOAuth2Provider` class also means that you rely on the fact that the provider you used won't change the URLs and the other values relevant to using it. While this is less probable to happen, if you want to avoid this situation the option is to implement the `ClientRegistration` as presented in listing 12.3. This enables you to configure the URLs and related provider values in a configuration file.

We end this section by adding a private method to our configuration class, which returns the `ClientRegistration` as presented in listing 12.5. In section 12.5.4, you'll learn how to register this client registration object for Spring Security to use it for authentication.

#### Listing 12.5 Building the `ClientRegistration` object in the configuration class

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    private ClientRegistration clientRegistration() {      #A
        return CommonOAuth2Provider.GITHUB      #B
            .getBuilder("github")
            .clientId(
                "a7553955a0c534ec5e6b")   #C
            .clientSecret(
                "1795b30b425ebb79e424afa51913f1c724da0dbb")   #C
            .build();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();

        http.authorizeRequests()
            .anyRequest()
            .authenticated();
    }
}

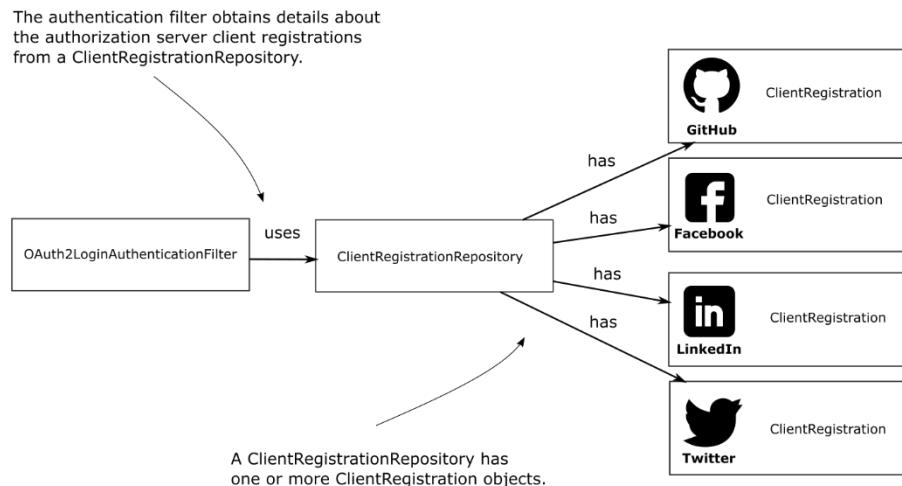
```

#A We'll call this method later to obtain the returned ClientRegistration  
 #B We start from the configuration Spring Security provides for GitHub common provider.  
 #C We provide our client credentials.

**NOTE** The client ID and the client secret are credentials, which makes them sensitive data. In a real-world application, they should be obtained from a secrets vault, and you should never directly write credentials in the source code.

#### 12.5.4 Implementing a ClientRegistrationRepository

In this section, you'll learn how to register the `ClientRegistration` instances for Spring Security to use them for authentication. In section 12.5.3, you learned how to represent the OAuth 2 client for Spring Security by implementing the `ClientRepository` contract. But you also need to set it up to be used for authentication. Spring Security uses, for this purpose, an object of type `ClientRegistrationRepository` (figure 12.14).



**Figure 12.14** The responsibility of the `ClientRegistrationRepository` is to retrieve the `ClientRegistration` details (client ID and client secret, URLs, scopes, and so on). The authentication filter needs these details for the authentication flow.

The `ClientRegistrationRepository` interface is very similar to the `UserDetailsService` interface, which you learned in chapter 2. Same as a `UserDetailsService` finds a `UserDetails` by its `username`, a `ClientRegistrationRepository` finds a `ClientRegistration` by its `registration id`.

You can implement the `ClientRegistrationRepository` interface to tell the framework where to find the `ClientRegistration` instances. Spring Security offers us an implementation for the `ClientRegistrationRepository`, which stores in-memory the instances of `ClientRegistration`: the `InMemoryClientRegistrationRepository`. As you guessed, this works very similarly to how the `InMemoryUserDetailsManager` works for the `UserDetails` instances. We discussed the `InMemoryUserDetailsManager` in chapter 3.

To end our application implementation, I'll define a `ClientRegistrationRepository` using the `InMemoryClientRegistrationRepository` implementation and register it as a bean in the Spring context. I'll add the `ClientRegistration` instance we built in section 12.5.3 to the `InMemoryClientRegistrationRepository` by providing it as a parameter to its constructor. You find this code in listing 12.6.

#### **Listing 12.6 Registering the ClientRegistration object**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Bean    #A
    public ClientRegistrationRepository clientRepository() {
        var c = clientRegistration();
        return new InMemoryClientRegistrationRepository(c);
    }

    private ClientRegistration clientRegistration() {
        return CommonOAuth2Provider.GITHUB.getBuilder("github")
            .clientId("a7553955a0c534ec5e6b")
            .clientSecret("1795b30b425eb79e424afa51913f1c724da0ddb")
            .build();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login()

        http.authorizeRequests()
            .anyRequest().authenticated();
    }
}
```

#A We add a bean of type `ClientRegistrationRepository` in Spring context. The bean contains the reference to a `ClientRegistration`.

As you can see, adding the `ClientRegistrationRepository` as a bean in the Spring context is enough for Spring Security to find it and work with it. An alternative to this way of registering the `ClientRegistrationRepository`, you can use a `Customizer` object as a parameter of the `oauth2Login()` method of the `HttpSecurity` object. You've learned to do something similar with the `httpBasic()` and `formLogin()` methods in chapters 7 and 8 and then with the `cors()` and `csrf()` methods in chapter 10. The same principle applies here. You find this configuration in listing 12.7. I have also separated it into a project named `ssia-ch12-ex2`.

**Listing 12.7 Configuring the ClientRegistrationRepository with a Customizer**

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login(c -> { #A
            c.clientRegistrationRepository(clientRepository());
        });

        http.authorizeRequests()
            .anyRequest()
            .authenticated();
    }

    private ClientRegistrationRepository clientRepository() {
        var c = clientRegistration();
        return new InMemoryClientRegistrationRepository(c);
    }

    private ClientRegistration clientRegistration() {
        return CommonOAuth2Provider.GITHUB.getBuilder("github")
            .clientId("a7553955a0c534ec5e6b")
            .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
            .build();
    }
}

```

#A We use a Customizer to set the ClientRegistrationRepository instance.

**NOTE** One configuration option is as good as the other. But remember what we've discussed in chapter 2. To keep your code easy to understand, try not to mix the configuration approaches. Either use an approach where you set everything with beans in the context or use the code inline configuration style.

### 12.5.5 The pure magic of the Spring Boot configuration

In this section, I'll show you a third approach to configuring the application we build in this chapter. Spring Boot is designed to use its magic and build the `ClientRegistration` and `ClientRegistrationRepository` objects directly from the properties files. This approach isn't unusual in a Spring Boot project. We see this happening with other objects as well. Very often, we see data sources configured based on the properties file, for example. In the following code snippet, you find how to set the client registration for our example in the `application.properties` file.

```

spring.security.oauth2.client[CA]
    .registration.github.client-id=a7553955a0c534ec5e6b

spring.security.oauth2.client[CA]
    .registration.github.client-secret=[CA]
    1795b30b425ebb79e424afa51913f1c724da0dbb

```

In this snippet, I only needed to specify the client ID and client secret. Because the name for the provider is “github,” Spring Boot knows to take all the details regarding the URIs from the `CommonOAuth2Provider` class. Now my configuration class looks like the one presented in listing 12.8. You also find this example in a separate project named `ssia-ch12-ex3`.

#### **Listing 12.8 The configuration class**

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();

        http.authorizeRequests()
            .anyRequest()
            .authenticated();
    }

}
```

We don’t need to specify any details about the `ClientRegistration` and the `ClientRegistrationRepository` because they’ll be created automatically by Spring Boot based on the properties file. If we were using a provider not among the common ones known by Spring Security, we would need to also specify the details for the authorization server using the property group starting with `spring.security.oauth2.client.provider`. The next code snippet provides you with an example.

```
spring.security.oauth2.client.provider
.myprovider.authorization-uri=<some uri>

spring.security.oauth2.client.provider
.myprovider.token-uri=<some uri>
```

When everything I need is to have one or more authentication providers in memory, as we do in the current example, I prefer to configure it as I presented in this section. It’s cleaner and more manageable. But if we need something different, like storing the client registration details in a database or obtaining them from a web service, then we would need to create our custom implementation of `ClientRegistrationRepository`. This way, we would need to set it up, as you learned in section 12.5.5.

**Exercise** Change the current application to store the authorization server details in a database.

#### **12.5.6 Obtaining details about the authenticated user**

In this section, we discuss getting and using the details of the authenticated user. You’re already aware that in the Spring Security architecture, the `SecurityContext` stores the details of the authenticated user. Once the authentication process ends, the responsible filter stores the `Authentication` object in the `SecurityContext`. The application can take the user details from there and use them whenever needed. The same happens with an OAuth 2 authentication.

The implementation of the `Authentication` object, used by the framework, in this case, is named `OAuth2AuthenticationToken`. You can take it directly from the `SecurityContext` or let Spring Boot inject it for you in a parameter of the endpoint, as you learned in chapter 6. Listing 12.9 shows how I changed the controller to receive and print in the console details about the user.

#### **Listing 12.9 Using the details of the logged-in user**

```
@Controller
public class MainController {

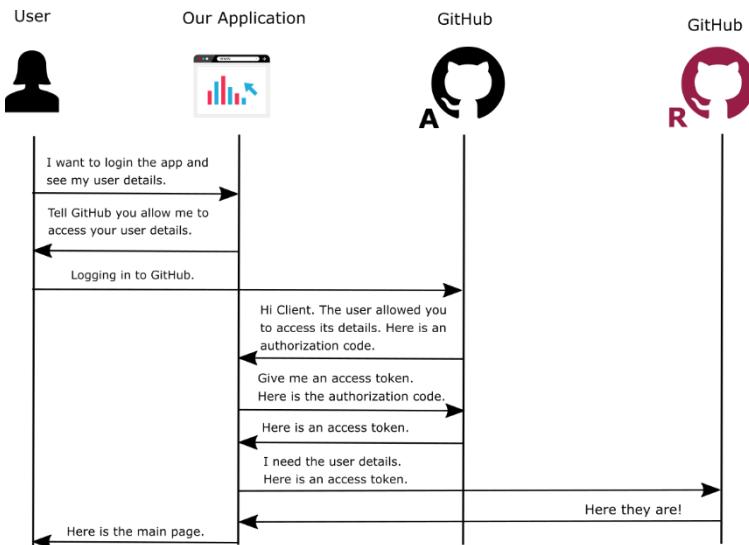
    private Logger logger =
        Logger.getLogger(MainController.class.getName());

    @GetMapping("/")
    public String main(OAuth2AuthenticationToken token) {      #A
        logger.info(String.valueOf(token.getPrincipal()));
        return "main.html";
    }
}
```

#A Spring Boot automatically injects the `Authentication` object representing the user in the parameter of the method.

#### **12.5.7 Testing the application**

In this section, we test the app we worked on in this chapter. Together with checking the functionality, we'll follow the steps of the OAuth 2 authorization code grant (figure 12.15) to make sure you understood it correctly, and you observe how Spring Security applies it with the configuration we made. You can use any of the three projects we wrote during this chapter. They define the same functionality with different ways of writing the configuration, but the result is the same for any of the three.



**Figure 12.15** The application uses GitHub as an authorization server and also as a resource server. When the user wants to log in, the Client redirects them to the GitHub login page. When the user logs in successfully, GitHub calls back our application with an authorization code. Our application uses the authorization code to request an access token. The application can then access the user details from the Resource Server (GitHub) by providing the access token. The response from the resource server provides the user details along with the URL for the main page.

I first make sure I'm not logged into GitHub. I'll also make sure I open the Chrome console to check the history of request navigation. This history gives me an overview of the steps that happen in the OAuth 2 flow — the steps we discussed in section 12.3.1. If I were, then the application would log me directly. Then I start the app and access it in the browser: <http://localhost:8080>.

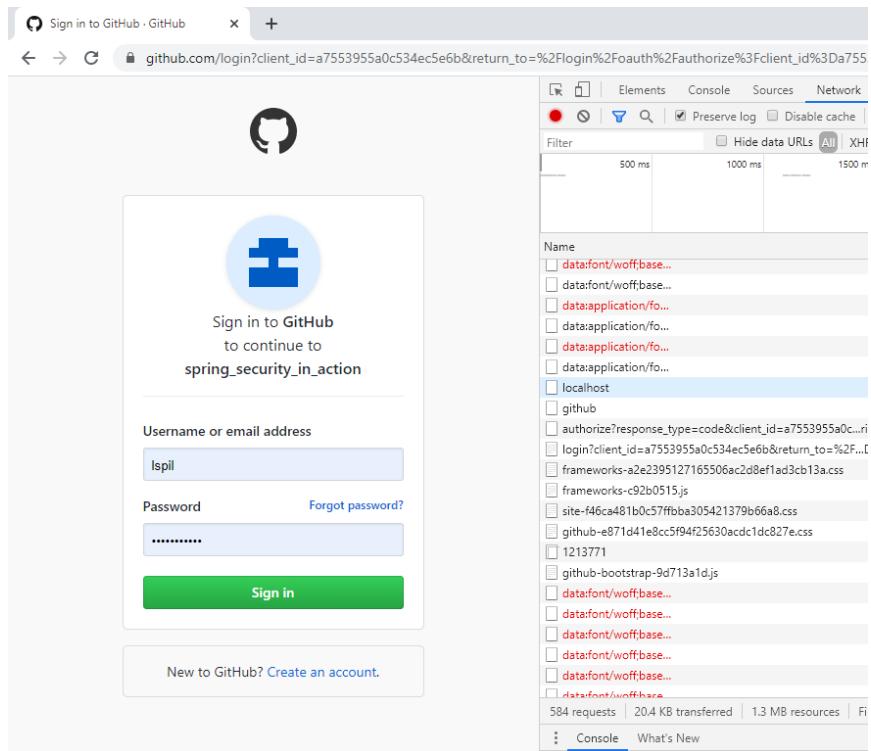
We access the main page of our application.

<http://localhost:8080/>

The application redirects us to the URL in the following code snippet (figure 12.16). This URL is configured in the `CommonOAuth2Provider` class for GitHub as the authorization URL. Our application attached the needed query parameters to the URL as we discussed in section 12.3.1:

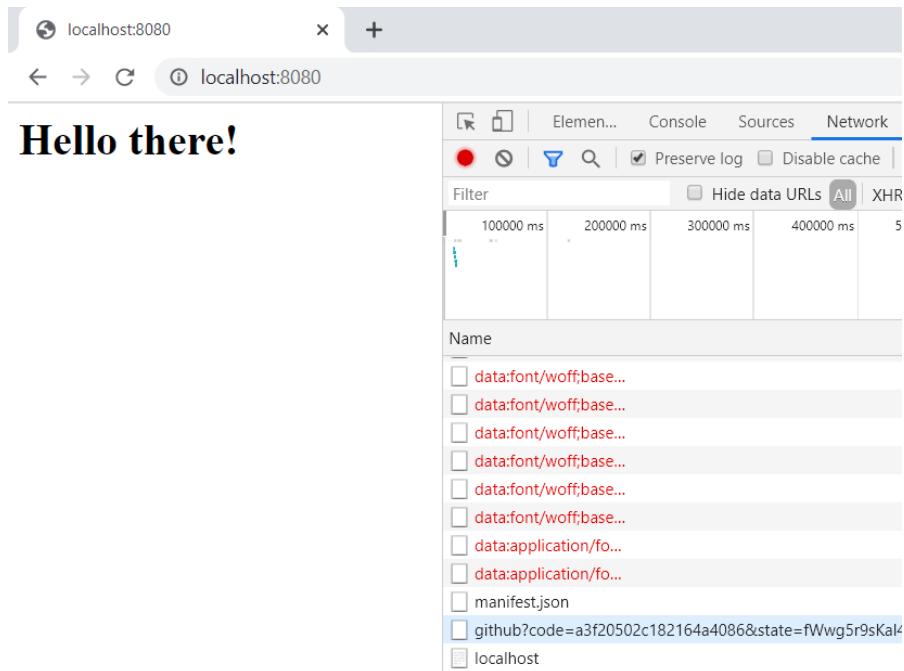
- **response\_type** with the value “code”
- **client\_id**
- **scope** – the value `read:user` is also defined in the `CommonOAuth2Provider` class.
- **state** – with the CSRF token.

[https://github.com/login/oauth/authorize?response\\_type=code&client\\_id=a7553955a0c534ec5e6b&scope=read:user&state=fWwg5r9sKaI4BMubg1oXBRnN5y7VDW1A\\_rQ4UITbjk%3D&redirect\\_uri=http://localhost:8080/login/oauth2/code/github](https://github.com/login/oauth/authorize?response_type=code&client_id=a7553955a0c534ec5e6b&scope=read:user&state=fWwg5r9sKaI4BMubg1oXBRnN5y7VDW1A_rQ4UITbjk%3D&redirect_uri=http://localhost:8080/login/oauth2/code/github)



**Figure 12.16** After accessing the main page, the browser redirects us to the GitHub login. In the Chrome console tool, we can see the calls to the localhost and then to the authorization endpoint of GitHub.

I use my GitHub credentials and log into our application with GitHub. We are authenticated and redirected back, as you can see in figure 12.17.



**Figure 12.17** After filling our credentials, GitHub redirects back to our application. We can see the main page, and the application can access the user details from GitHub by making use of the access token.

The following code snippet has the URL on which GitHub called us back. You observe GitHub provided the authorization code, which our application uses to request an access token.

```
http://localhost:8080/login/oauth2/code/github?code=a3f20502c182164a4086&state=fWwg5r9sKa14BMubg1oXBRRnN5y7VDW1A\_r04UITbJK%3D
```

We won't see the calls to the token endpoint from the browser, as this happens directly from our application. But we can trust that the application managed to get a token as we can see the user details printed in the console. This means the app managed to call the endpoint to retrieve the user details. The next code snippet partially shows you this output.

```
Name: [43921235],
Granted Authorities: [[ROLE_USER, SCOPE_read:user]], User Attributes: [{login=lspil,
id=43921235, node_id=MDQ6VXNlcjQzOTIxMjM1,
avatar_url=https://avatars3.githubusercontent.com/u/43921235?v=4, gravatar_id=,
url=https://api.github.com/users/lspil, html_url=https://github.com/lspil,
followers_url=https://api.github.com/users/lspil/followers,
following_url=https://api.github.com/users/lspil/following{/other_user}, ...}
```

## 12.6 Summary

- The OAuth 2 framework describes ways to allow an entity to access resources on behalf of somebody else. We use it in applications to implement the authentication and

- authorization logic.
- The different flows an application can use to obtain an access token are named grants. Depending on the system architecture, you need to choose a suitable grant type:
    - The authentication code grant type works by allowing the user directly authenticate at the Authorization Server to enable the Client to obtain an access token. We choose this grant type when the user can't trust the Client to share credentials with it.
    - The password grant type implies that the user shares its credentials with the Client. You can apply this only if the Client can be trusted.
    - The client credentials grant type, which implies that the Client obtains a token by authenticating only with its credentials. We choose this grant type when the Client needs to call an endpoint of the Resource Server, which isn't a resource of the user.
  - Spring Security implements the OAuth 2 framework, which allows you to configure it into your application with very few lines of code.
  - In Spring Security, you represent a registration of a client at an authorization server using an instance of `ClientRegistration`.
  - The component of the Spring Security OAuth 2 implementation responsible for finding a specific client registration is called `ClientRegistrationRepository`. You need to define a `ClientRegistrationRepository` with at least one `ClientRegistration` available when implementing an OAuth 2 client with Spring Security.

# 13

## *OAuth 2 – Implementing the authorization server*

### This chapter covers

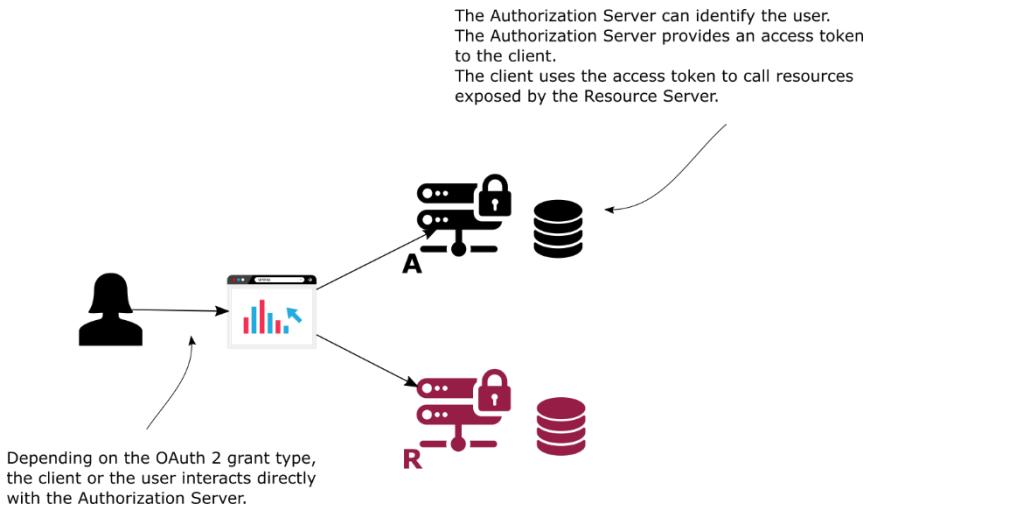
- **Implementing an OAuth 2 authorization server.**
- **Managing clients for the authorization server.**
- **Using the OAuth 2 grant types.**

In this chapter, we discuss implementing an authorization server with Spring Security. As you learned in chapter 12, the Authorization Server is one of the components acting in the OAuth 2 architecture (figure 13.1). The role of the Authorization Server is to authenticate the user and provide a token to the client. The client uses this token to access resources exposed by the resource server on behalf of the user.

Also, in chapter 12, you learned that the OAuth 2 framework defines multiple flows for obtaining the token. We call these flows “grants.” You’ll choose one of the different flows according to your scenario. The behavior of the Authorization Server is different depending on the chosen grant. In this chapter, you’ll learn how to configure an authorization server with Spring Security for the most common OAuth 2 grant types:

- Authorization code grant type
- Password grant type
- Client credentials grant type

Besides these three grant types, you’ll also learn to configure the authorization server to issue refresh tokens. A client uses the refresh tokens to obtain new access tokens. If an access token expires, the client has to get a new one. To do so, the client has two choices: reauthenticate using the user credentials, or use a refresh token. We discussed the advantages of using the refresh tokens over the user reauthentication in section 12.3.4.



**Figure 13.1** The Authorization Server is one of the OAuth 2 actors. It identifies the resource owner and provides an access token to the client. The client needs the access token to access resources on behalf of the user.

For months rumors said the authorization server development with Spring Security would no longer be supported (<https://spring.io/projects/spring-security-oauth#overview>). Finally, the Spring Security OAuth2 dependency was deprecated. With this action, we had alternatives (the ones you learn in this book) for implementing the client and the resource server, but not for an authorization server. Luckily, the Spring Security team announced a new Authorization Server is starting to be developed: <https://spring.io/blog/2020/04/15/announcing-the-spring-authorization-server>. I'd also recommend you stay aware of the implemented features in different Spring Security project using this link: <https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Features-Matrix>

It'll take time for the new Spring Security Authorization Server to become mature. Until then, the only choice we have for developing a custom Authorization Server with Spring Security is the current way. We discuss this way of implementing the Authorization Server in this chapter. Implementing a custom Authorization Server, as presented in this chapter, will help you understand better how this component work. Of course, it's also the only way for the moment to implement an Authorization Server. I've seen this applied by developers in projects. If you'd have to deal with such a project that implemented the Authorization Server this way, it's still important you understand it before we'll be able to use the new implementation. And, say you want today to start a new Authorization Server implementation: It's still the only way to go using Spring Security because you simply don't have another choice. Instead of implementing a custom Authorization Server, you could go with a third-party tool like Keycloak or Okta. In chapter 18, we'll use Keycloak in our hands-on example. But in my experience, sometimes stakeholders don't accept using such a solution, and you need to go with

implementing custom code. Let's learn how to do this and understand better the Authorization Server in the following sections of this chapter.

### 13.1 Writing your own authorization server implementation

There's no OAuth 2 flow without an Authorization Server. I've said it earlier: OAuth 2 is mainly about obtaining an access token. And the Authorization Server is the component of the OAuth 2 architecture that issues access tokens. So you need first to know how to implement it. Then, in chapters 14 and 15, you'll learn how the Resource Server authorizes requests based on the access token a client obtains from the Authorization Server. Let's start building an Authorization Server. To begin with, you'll have to create a new Spring Boot project and add the dependencies as presented in the following code snippet. I'll name this project `ssia-ch13-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.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

Inside the `projects` tag, you'll also need to add the `dependenciesManagement` tag for the `spring-cloud-dependencies` artifact id as presented in the next code snippet.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

We can now define a configuration class, which I'll call `AuthServerConfig`. Besides the classic `@Configuration` annotation, we also have to annotate this class with `@EnableAuthorizationServer`. This way, we instruct Spring Boot to enable the configuration specific to the OAuth 2 authorization server. We'll be able to customize this configuration by extending the `AuthorizationServerConfigurerAdapter` class and overriding specific methods that we'll discuss in this chapter. Listing 13.1 presents the `AuthServerConfig` class.

#### **Listing 13.1 The `AuthServerConfig` class**

```
@Configuration
```

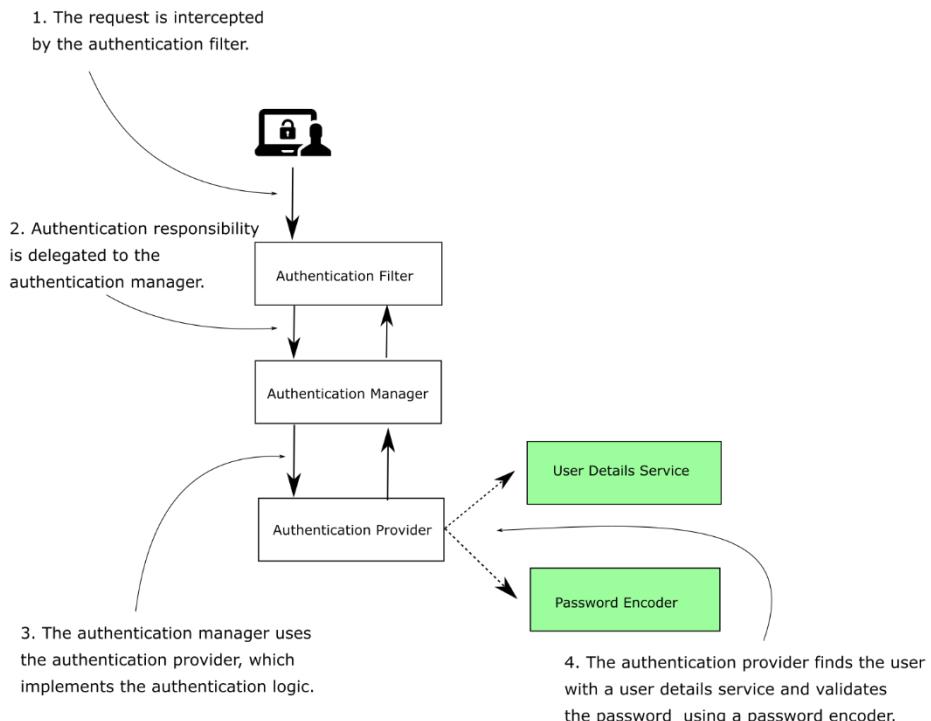
```
@EnableAuthorizationServer  
public class AuthServerConfig  
    extends AuthorizationServerConfigurerAdapter {  
}
```

We already have the minimal configuration for our Authorization Server. This is awesome! However, to make it usable, we still have to implement the user management, register at least a client, and decide which grants to support.

## 13.2 Defining user management

In this section, we discuss user management. The Authorization Server is the component that deals with authenticating the user in the OAuth 2 framework. So, naturally, it needs to manage the users. Fortunately, the user management implementation doesn't change from what you learned in chapters 3 and 4. We'll continue to use the `UserDetails`, `UserDetailsService`, and `UserDetailsManager` contracts to manage credentials. And to manage passwords, we'll continue to use the `PasswordEncoder` contract. Here, they have the same roles and work the same as you learned in chapters 3 and 4. Behind the scenes is the standard authentication architecture, which we've discussed throughout the previous chapters.

Figure 13.2 reminds you of the main components acting in the authentication process in Spring Security. What you observe differently from the way we used to describe the authentication architecture until now is the fact that we don't have a `SecurityContext` anymore in this diagram. This change happens because the result of the authentication is not stored anymore in a `SecurityContext`. The authentication is instead managed with a token from a `TokenStore`. You'll learn more about the `TokenStore` in chapter 14, where we discuss the Resource Server.



**Figure 13.2** The authentication process. A filter intercepts the user request and delegates the authentication responsibility to an authentication manager. Further, the authentication manager uses an authentication provider that implements the authentication logic. To find the user, the authentication provider uses a `UserDetailsService`, and to verify the password, the authentication provider uses a `PasswordEncoder`.

Let's find out how to implement user management in our authorization server. I always prefer to separate the responsibilities of the configuration classes. For this reason, I choose to define a second configuration class in our application, where I only write the configurations needed for user management. I named this class `WebSecurityConfig`; you can see its implementation in listing 13.2.

#### **Listing 13.2** The `WebSecurityConfig` class contains configurations for the user management

```
@Configuration
public class WebSecurityConfig {

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

        var u = User.withUsername("john")
                    .password("12345")
                    .roles("USER")
                    .build();

        uds.createUser(u);
    }
}
```

```

        .authorities("read")
        .build();

    uds.createUser(u);

    return uds;
}

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

}

```

As you observe in listing 13.2, I've declared an `InMemoryUserDetailsManager` as my `UserDetailsService`, and I use the `NoOpPasswordEncoder` as `PasswordEncoder` implementation. You know that you can use any implementation of your choice for these components, as you may recall from chapters 3 and 4. I'll keep them as simple as possible in my implementation, to let you focus on the OAuth 2 aspects of the app.

Now that we have users, we only need to link the user management to the Authorization Server configuration. To do this, I'll expose the `AuthenticationManager` as a bean in the Spring context, and then I'll use it in the `AuthServerConfig` class. Listing 13.3 shows you how to add the `AuthenticationManager` as a bean in the Spring context.

### **Listing 13.3 Adding the `AuthenticationManager` instance in Spring context**

```

@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {      #A

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

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

        uds.createUser(u);

        return uds;
    }

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

    @Bean      #B
    public AuthenticationManager authenticationManagerBean()
        throws Exception {
        return super.authenticationManagerBean();
    }
}

```

```
}
```

#A We extend the WebSecurityConfigurerAdapter to access the AuthenticationManager instance.

#B We add the AuthenticationManager instance as a bean in the Spring context.

We can now change the AuthServerConfig class to register the AuthenticationManager to the Authorization Server. Listing 13.4 shows you the changes you need to do to the AuthServerConfig class.

#### **Listing 13.4 Setting the AuthenticationManager in the Authorization Server configurations**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired    #A
    private AuthenticationManager authenticationManager;

    @Override    #B
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

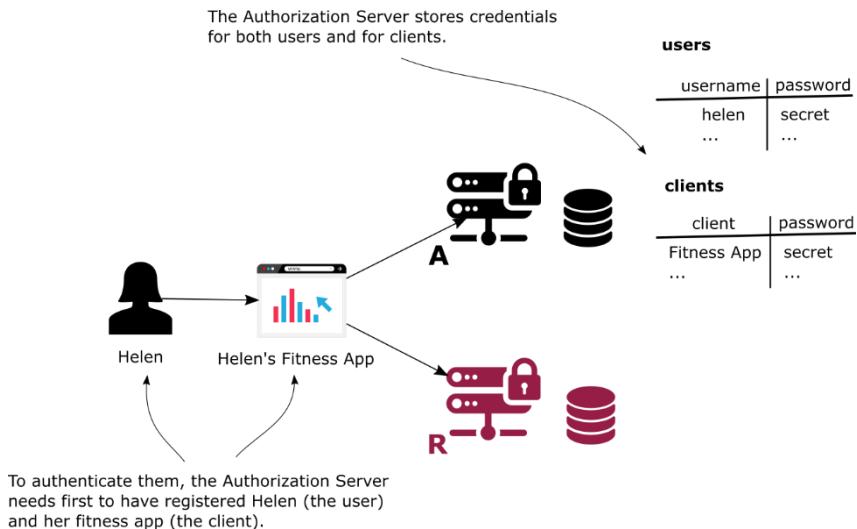
#A We inject the AuthenticationManager instance from the context.

#B We override the configure() method to set the AuthenticationManager.

With these configurations in place, we now have users who can authenticate at our Authentication Server. But the OAuth 2 architecture implies that the users grant the privileges to a client. So it is a client that'll be able to use resources on behalf of a user. In section 13.1.2, you'll learn how to configure the clients for the Authorization Server.

### **13.3 Registering clients with the Authorization Server**

In this section, you'll learn how to make your clients known by the Authorization Server. To call the Authorization Server, an app acting as a client in the OAuth 2 architecture needs its own credentials. The Authorization Server also manages these credentials and only allows requests from known clients (figure 13.3).

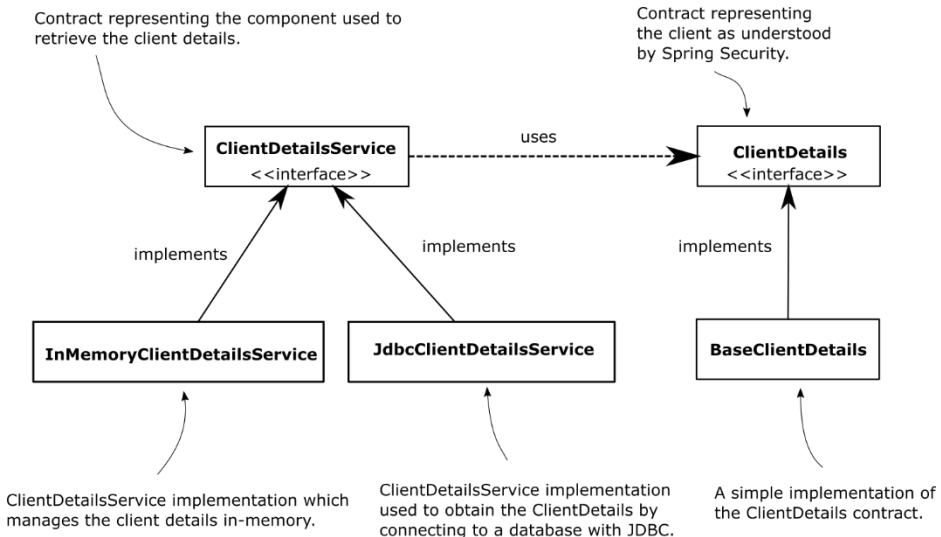


**Figure 13.3 The Authorization Server stores the users' credentials and client credentials. The Authorization Server uses client credentials so that it only allows known applications to be authorized by it.**

Do you remember the client application we developed in chapter 12? We used GitHub as our Authentication Server. GitHub needed to know about the client app, so the first thing we did was to register the application at GitHub. After registering the application, we received a client ID and a client secret – the client credentials. We configured these credentials, and our app used them to authenticate with the Authorization Server (GitHub).

The same applies in this case: our Authorization Server needs to know its clients because it accepts requests from these clients. Here the process becomes familiar. The contract that defines the client for the Authorization Server is `ClientDetails`. The contract defining the object to retrieve `ClientDetails` by their ID is `ClientDetailsService`.

Do these names sound like you know them already? These interfaces work like the `UserDetails` and the `UserDetailsService` interfaces, but they represent the clients. You'll find many of the things we discussed in chapter 3 work similarly for the `ClientDetails` and `ClientDetailsService`. For example, we have an `InMemoryClientDetailsService` that is an implementation of the `ClientDetailsService` interface, which manages the `ClientDetails` in-memory. It works similarly to the `InMemoryUserDetailsService` class for `UserDetails`. Likewise, we have a `JdbcClientDetailsService`, which is similar to the `JdbcUserDetailsService`. Figure 13.4 show these classes and interfaces and the relationships among them.



**Figure 13.4** The dependencies between classes and interfaces we use to define the client management for the authorization server.

We can sum up these similarities in a few points you can easily remember:

- The `ClientDetails` is for the client what the `UserDetails` is for the user.
- The `ClientDetailsService` is for the client what the `UserDetailsService` is for the user.
- The `InMemoryClientDetailsService` is for the client what the `InMemoryUserDetailsService` is for the user.
- The `JdbcClientDetailsService` is for the client what the `JdbcUserDetailsService` is for the user.

Listing 13.5 shows you how to define a client configuration and set it up using an `InMemoryClientDetailsService`. The `BaseClientDetails` class I use in the snippet is an implementation for the `ClientDetails` interface provided by Spring Security. In listing 13.6, you also find a shorter way of writing the same configuration.

#### **Listing 13.5 Using an `InMemoryClientDetailsService` to configure a client**

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Ommited code

    @Override    #A
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
  
```

```

var service = new InMemoryClientDetailsService();      #B

var cd = new BaseClientDetails();      #C
cd.setClientId("client");      #C
cd.setClientSecret("secret");      #C
cd.setScope(List.of("read"));      #C
cd.setAuthorizedGrantTypes(List.of("password"));      #C

service.setClientDetailsStore(Map.of("client", cd));      #D

clients.withClientDetails(service);      #E
}

}

#A We override the configure() method to set up the ClientDetailsService instance.
#B We create an instance using the implementation of ClientDetailsService.
#C We create an instance of ClientDetails and set the needed details about the client.
#D We add the ClientDetails instance to the InMemoryClientDetailsService.
#E We configure the ClientDetailsService to be used by our Authorization Server.

```

Listing 13.6 presents a shorter method for writing the same configuration, which enables us to avoid repetition and to write cleaner code.

#### **Listing 13.6 Configuring the ClientDetails in-memory**

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()      #A
            .withClient("client")      #B
            .secret("secret")      #B
            .authorizedGrantTypes("password")      #B
            .scopes("read");      #B
    }
}

```

#A We use a ClientDetailsService implementation, which manages the in-memory stored ClientDetails.  
#B We build and add an instance of ClientDetails.

To write less code, I prefer using the shorter version over the more detailed one in examples. But if you write an implementation where you store the client details in a database – which is mainly the case for a real-world scenario – then you'll use the contracts from listing 13.5.

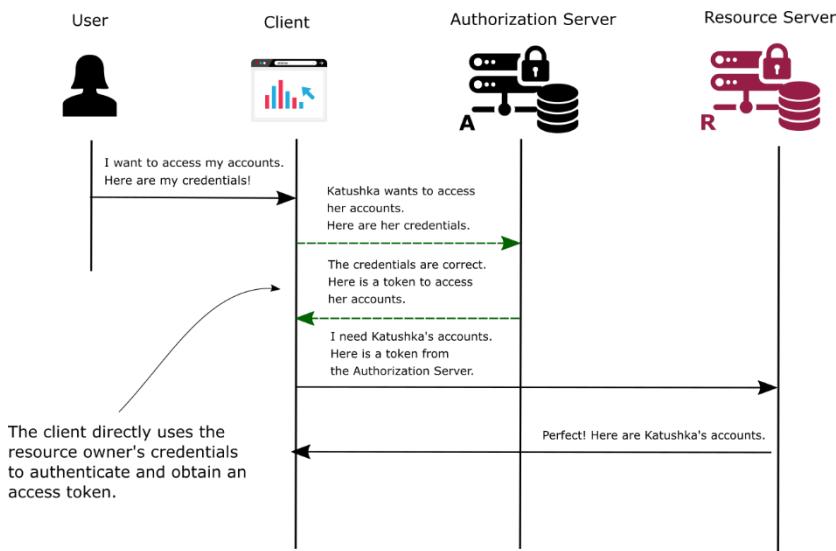
**Exercise.** Write an implementation to manage the client details in a database. You can use a similar implementation to the one we worked on for `UserDetailsService` in section 3.3.

**NOTE** As we did for the `UserDetailsService`, here we used an implementation that manages the details in-memory. This approach only works for examples and study purposes. In a real-world scenario, you'd use an implementation that persists somehow these details – usually a database.

## 13.4 Using the password grant

In this section, we'll use the authorization server with the OAuth 2 password grant. Well, we'll mainly test it's working, because with the implementation we've done until now in sections 13.1.1 and 13.1.2, we already have a working Authorization Server that can use the password grant type. I told you it's easy!

Figure 13.5 reminds you of the password grant type and the place of the Authorization Server within this flow.



**Figure 13.5 The password grant type:** The authorization server receives the user credentials and authenticates the user. If the credentials are correct, the authorization server issues an access token which the client can use to call resources that belong to the authenticated user.

Now, let's start the application and test it. We can request a token at the `/oauth/token` endpoint. Spring Security automatically configured this endpoint for us. We'll use the client credentials with HTTP Basic to access the endpoint and send the needed details as query parameters. As you know from chapter 12, the parameters we need to send in this request are:

- **grant\_type** with the value “password”
- **username** and **password** – which are the user credentials.
- **scope** – which is the granted authority

In the next code snippet, you find the curl command.

```
curl -v -XPOST -u client:secret
http://localhost:8080/oauth/token?grant\_type=password&username=john&password=12345&scope=read
```

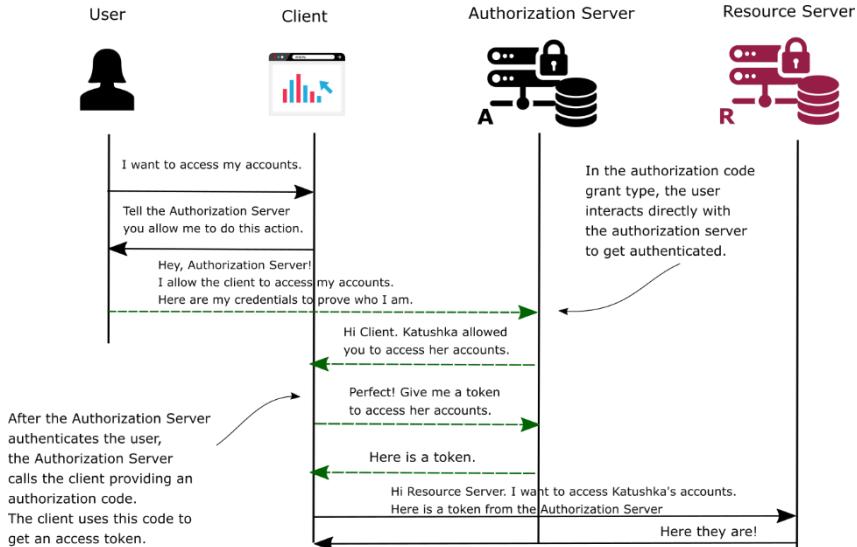
Running this command, you'll get the response as presented in the next code snippet.

```
{
  "access_token": "693e11d3-bd65-431b-95ff-a1c5f73aca8c",
  "token_type": "bearer",
  "expires_in": 42637,
  "scope": "read"
}
```

Observe the access token in the response. With the default configuration in Spring Security, a token is a simple UUID. The client can now use this token to call the resources exposed by the resource server. In section 13.2, you'll learn how to implement the Resource Server and also there you'll learn more about customizing tokens.

## 13.5 Using the authorization code grant

In this section, we discuss configuring the authorization server for the authorization code grant. You have used this grant with the client application we developed in chapter 12, and you know it's one of the most used OAuth 2 grant types. It's essential to understand how to configure your authorization server to work with this grant type as it's highly probable to find this requirement in a real-world system. So in this section, we'll write some code to prove how to make it work with Spring Security. I will create another project I'll name `ssia-ch13-ex2` to be easier for you to find the examples in the projects. From figure 13.6, you can remember how the authorization code grant type works and how the authorization server interacts with the other components in this flow.



**Figure 13.6** In the authorization code grant type, the client redirects the user to the Authorization Server to authenticate. The user directly interacts with the Authorization Server and, once authenticated, the Authorization Server calls back the client on a redirect URI. When it calls back the client, it also provides an authorization code. The client uses the authorization code to obtain an access token.

As you learned in section 13.1.3, it's all about how you register the client. So, everything you have to do to use another grant type is to set it up in the client registration, as presented in listing 13.7. For the authorization code grant type, you'll also need to provide the redirect URI. This is the URI to which the Authorization Server redirects the user once it completes the authentication. When calling the redirect URI, the Authorization Server also provides the access code.

#### **Listing 13.7 Setting the authorization code grant type**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("authorization_code")
            .scopes("read")
```

```

    .redirectUris("http://localhost:9090/home");

}

@Override
public void configure(
    AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints.authenticationManager(authenticationManager);
}
}

```

You can have multiple clients, and each might use different grants. It's also possible to set up more grants for one client. The authorization server will act according to the client's request. Take a look at listing 13.8 to see how you'd configure different grants for different clients.

### **Listing 13.8 Configuring clients with different grant types**

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()

            .withClient("client1")
            .secret("secret1")
            .authorizedGrantTypes(      #A
                "authorization_code")
            .scopes("read")
            .redirectUris("http://localhost:9090/home")
            .and()

            .withClient("client2")
            .secret("secret2")
            .authorizedGrantTypes(      #B
                "authorization_code", "password", "refresh_token")
            .scopes("read")
            .redirectUris("http://localhost:9090/home");

    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}

```

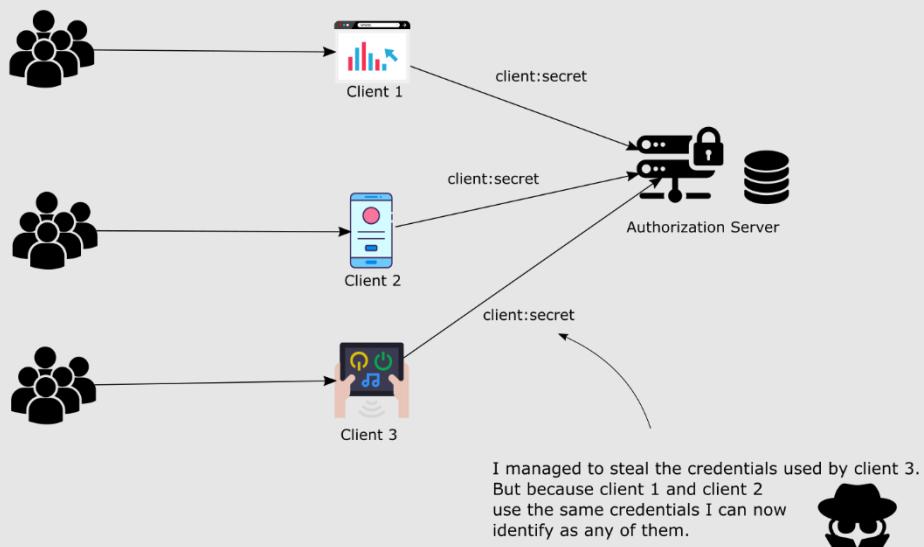
#A Client with ID client1 can only use the authorization\_code grant.

#B Client with ID client2 can use any of authorization\_code, password and refresh tokens.

## Using multiple grant types for a client

As you learned, it's possible to allow multiple grant types to one client. But you have to be careful with this approach as it might reveal that wrong practices are used in your architecture from the security perspective. The grant type is the flow in which a client (an application) obtains an access token so that it can access a specific resource. When you implement the client in such a system (as we've also done in chapter 12), you write logic depending on the grant type you use.

So what's the reason for having multiple grant types assigned to the same client on the authorization server side? What I've seen in a couple of systems, which I consider a very bad and worthy to avoid practice, is sharing client credentials. Sharing client credentials means to have different client applications sharing the same client credentials.



**Figure 13.7 Sharing client credentials: Multiple clients use the same credentials to obtain access tokens from the Authorization Server.**

In the OAuth 2 flow, the client, even if it's an application, acts as an independent component having its own credentials, which it uses to identify itself. Same as you don't share the user credentials, you shouldn't share the client credentials either. Even if all applications which define clients are part of the same system, nothing stops you from registering them as separate clients at the authorization server level.

Registering the clients individually with the authorization server brings the following benefits:

- 1) It gives you the possibility to audit events individually from each application – when you log events, you know which client has generated them.
- 2) It allows you stronger isolation – if one pair of credentials is lost, only one client is affected.
- 3) It allows you separation of scope – you assign different scopes (granted authorities) to a client that obtains the token in a specific way.

Scope separation is fundamental, and managing it incorrectly could lead to strange scenarios. Let's assume you defined a client as presented in the next code snippet.

```
clients.inMemory()
    .withClient("client")
    .secret("secret")
    .authorizedGrantTypes(
```

```

    "authorization_code",
    "client_credentials")
.scopes("read")

```

The client configuration presented by the previous code snippet is configured for the authorization code, and client credentials grant types. Using any of them, the client obtains an access token, which provides it the “read” authority. What is strange here is that the client can get the same token either by authenticating a user or by only using its own credentials. This doesn’t make sense, and one could even argue this is a security breach. And even if it sounds strange to you, I’ve seen that in practice in a system for which I was asked to audit. Why was the code designed that way for that system? Most probably, the developers didn’t understand the purpose of the grant types and used some code they’ve found somewhere around the web. Or that’s the only thing I could imagine when I’ve seen that all the clients in the system were configured with the same list containing all the possible grant types (some of them being strings that don’t even exist as a grant type). Make sure you avoid such mistakes!

Be careful, to specify the grant types, you use strings, not enum values, and this design could lead to mistakes. And yes, you can write a configuration like the one presented in the next code snippet.

```

clients.inMemory()
    .withClient("client")
    .secret("secret")
    .authorizedGrantTypes("password", "hocus_pocus")
    .scopes("read")

```

As long as you don’t try to use the “hocus\_pocus” grant type, the application will actually work.

Let’s start the application using the configuration presented in listing 13.7. When we want to accept the authorization code grant, the server also needs to provide a page where the client redirects the user for login. We implement this using the form login configuration, as you learned in chapter 5. You need to override the `configure()` method, as presented in listing 13.9.

#### **Listing 13.9 Configuring the form login authentication for the authorization server**

```

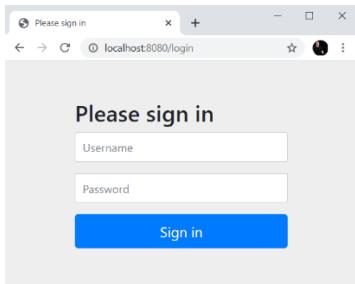
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {
    // Omitted code
    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.formLogin();
    }
}

```

You can now start the application and access in browser the link presented in the next code snippet.

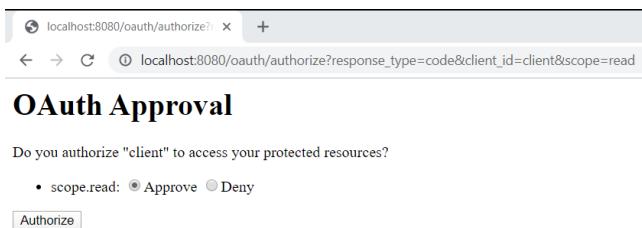
[http://localhost:8080/oauth/authorize?response\\_type=code&client\\_id=client&scope=read](http://localhost:8080/oauth/authorize?response_type=code&client_id=client&scope=read)

You’ll be redirected to the login page, as presented in listing 13.8.



**Figure 13.8** The Authorization Server redirects you to the login page. After it authenticates you, it will redirect you to the provided redirect URL.

After logging in, the Authorization Server explicitly asks you to grant or reject the requested scopes (figure 13.9).



**Figure 13.9** After the authentication, the Authorization Server asks you to confirm the scopes you want to authorize.

Once you grant the scopes, the Authorization Server redirects you to the redirect URI and provides the access token. In the next code snippet, you find the URL to which the Authorization Server redirected me. Observe the access code the client got through the query parameter in the request.

<http://localhost:9090/home?code=qeSLSt>

Your application can use the authorization code now to obtain a token calling the /oauth/token endpoint, as presented in the next code snippet.

```
curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=authorization_code&scope=read&code=qeSLSt"
```

The response body is:

```
{
  "access_token": "0fa3b7d3-e2d7-4c53-8121-bd531a870635",
  "token_type": "bearer",
  "expires_in": 43052,
  "scope": "read"
```

```
}
```

Mind that an authorization code can only be used once. If you try to call the /oauth/token endpoint using the same code again, you'll receive an error like the one presented in the next code snippet. You can only obtain another valid authorization code by asking the user to log in again.

```
{
  "error": "invalid_grant",
  "error_description": "Invalid authorization code: qeSLSt"
}
```

## 13.6 Using the client credentials grant

In this section, we discuss implementing the client credentials grant type. You may remember from chapter 12 that we use this grant for backend-to-backend authentications. It's not mandatory in this case, but sometimes we see this grant type as an alternative to the API key authentication method we discussed in chapter 8. We might use the client credentials grant type also when we secure an endpoint that's unrelated to a specific user and for which the client needs access. Let's say you want to implement an endpoint that returns the status of the server. The client calls this endpoint to check the connectivity and eventually display a connection status to the user or an error message. Because this endpoint only represents a deal between the client and the resource server, not being involved with any user-specific resource, the client should be able to call it without needing the user to authenticate. For such a scenario, we'd use the client credentials grant type.

Figure 13.9 reminds you how client credentials grant type work and how the Authorization Server interacts with the other components in this flow.

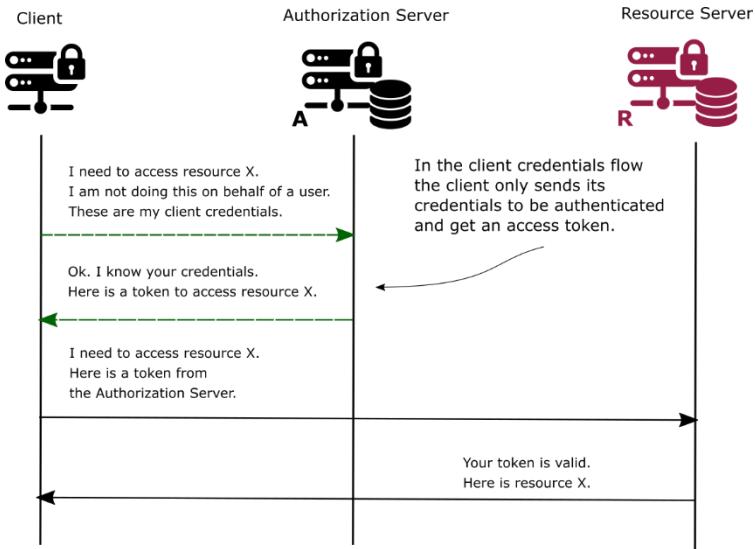


Figure 13.10 The client credentials grant type doesn't involve the user. Generally, we use this grant type as authentication between two backend solutions. The client needs only its credentials to authenticate and obtain an access token.

**NOTE** Don't worry for the moment about how the Resource Server validates the tokens. We'll discuss all the possible scenarios for how the Resource Server validates the access tokens in detail in chapters 14 and 15.

As you expect, to use the client credentials grant type, a client must be registered with this grant. I have defined a separate project `ssia-ch13-ex3` to prove this grant type. In listing 13.10, you find the configuration of the client, which uses the client credentials grant type.

#### Listing 13.10 The client registration for the client credentials grant type

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("client_credentials")
            .scopes("info");
    }
}
```

```
}
```

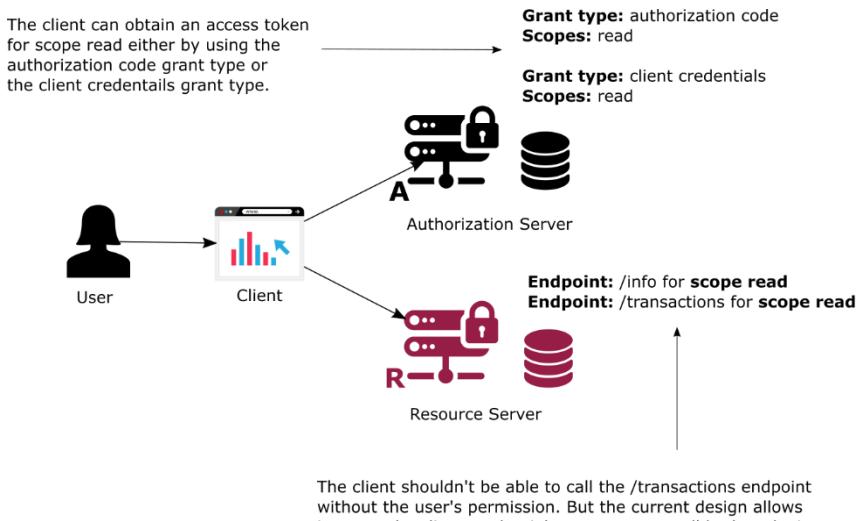
You can start the application now and call the `/oauth/token` endpoint to get an access token. The next code snippet shows you how to obtain the access token.

```
curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=client_credentials&scope=info"
```

The response body is:

```
{
  "access_token": "431eb294-bca4-4164-a82c-e08f56055f3f",
  "token_type": "bearer",
  "expires_in": 4300,
  "scope": "info"
}
```

But be careful with the client credentials grant type. As you observe, this grant type only requires the client to use its credentials. Make sure that you don't offer it access to the same scopes as for the flows that require user credentials. Otherwise, you might allow the client access to users' resources without needing the permission of the user. Figure 13.11 presents such a design in which the developer created a security breach by allowing the client to call a user's resource endpoint without needing the user to authenticate first.



**Figure 13.11** The figure presents a vicious design of the system. The developers wished to offer the client the possibility to call the `/info` endpoint without the need for the user permission. But because they used the same scope, they've now allowed it to call also the `/transactions` endpoint, which is a user's resource.

## 13.7 Using the refresh token grant

In this section, we discuss using refresh tokens with the Authorization Server developed with Spring Security. As you may recall from chapter 12, refresh tokens offer several advantages when using them together with another grant type. You can use refresh tokens with the authorization code grant type and with the password grant type (figure 13.12).

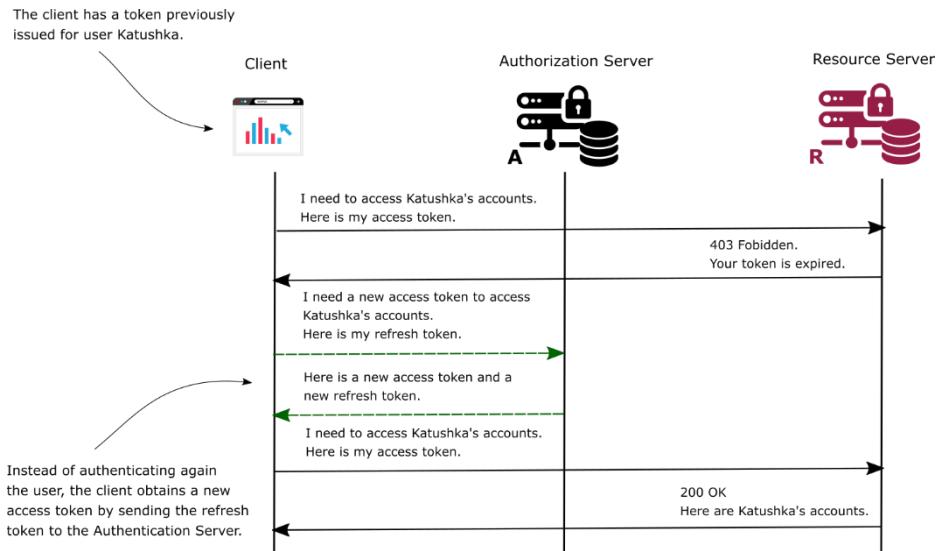


Figure 13.12 When the user authenticates, besides the access token, the client also receives a refresh token. The client uses the refresh token to get a new access token.

If you wish your Authorization Server to support refresh tokens, you need to add the refresh token grant to the grant list of the client. For example, if you wish to change the project we created in section 13.1.3 to prove the password grant, you would change the client as presented in listing 13.11.

### Listing 13.11 Adding the refresh token grant

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("client")
    }
}

```

```

        .secret("secret")
        .authorizedGrantTypes(
            "password",
            "refresh_token")      #A
        .scopes("read");
    }

}

```

#A We add the refresh\_token grant in the authorized grant types list of the client.

Now try the same curl command you used in section 13.4. You'll see the response is similar but now includes a refresh token.

```
curl -v -XPOST -u client:secret
http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&s
cope=read
```

The next code snippet presents the response of the previous command.

```
{
  "access_token": "da2a4837-20a4-447d-917b-a22b4c0e9517",
  "token_type": "bearer",
  "refresh_token": "221f5635-086e-4b11-808c-d88099a76213",
  "expires_in": 43199,
  "scope": "read"
}
```

## 13.8 Summary

- The `ClientRegistration` interface defines the OAuth 2 client registration in Spring Security. The `ClientRegistrationRepository` interface describes the object responsible for managing the client registrations. These two contracts allow you to customize how your authorization server manages the client registrations.
- For the Authorization Server implemented with Spring Security, the client registration dictates the grant type. The same authorization server can offer different grant types to different clients. This means that you don't have to implement something specific in your authorization server to define multiple grant types.
- For the authorization code grant type, the authorization server has to offer the possibility to the user to log in. This requirement is a consequence of the fact that in the authorization code flow, the user (resource owner) directly authenticates itself at the authorization server to grant access to the client.
- A `ClientRegistration` can request multiple grant types. This means that a client can use, for example, both password and authorization code grant type in different circumstances.
- We use the client credentials grant type for backend-to-backend authorization. It's technically possible, but uncommon that a client requests the client credentials grant type together with another grant type.
- We can use the refresh token grant type together with the authorization code grant type and with the password grant type. By adding the refresh token grant type to the client registration, we instruct the authorization server also to issue a refresh token besides the access token. The client uses the refresh token to obtain a new access token

without needing to authenticate the user again.

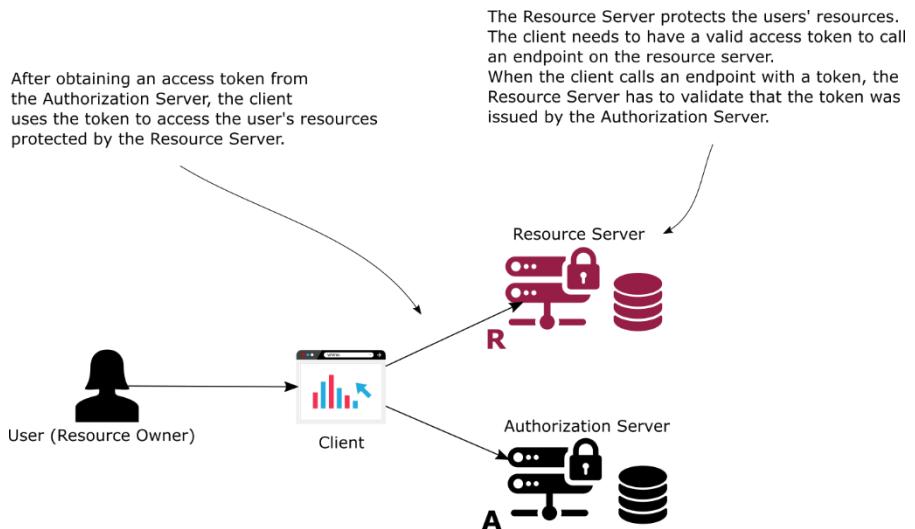
# 14

## *OAuth 2 – Implementing the resource server*

### This chapter covers

- **Implementing an OAuth 2 resource server with Spring Security.**
- **Implementing token validation by direct communication between the resource server and the authorization server.**
- **Using token stores to customize token management.**
- **Implementing token validation through blackboarding.**

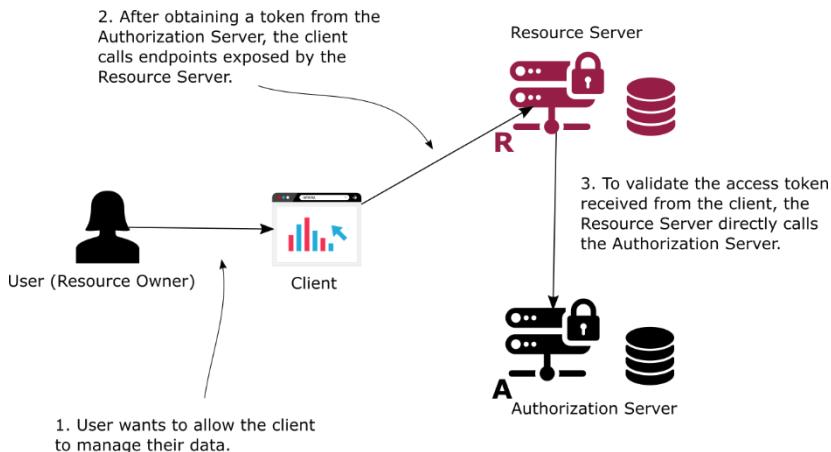
In this chapter, we discuss implementing a resource server with Spring Security. The Resource Server is the component that manages users' resources. The name Resource Server might not be suggestive at the beginning. Still, it actually represents in terms of OAuth 2 the backend you secure (just like any other app we've secured in the previous chapters – remember, for example, the Business Logic Server we implemented in chapter 11?). To allow a client to access the resources, the Resource Server requires a valid access token. A client obtains the access token from the Authorization Server and can use this token to call resources on the Resource Server by adding it in the HTTP request headers. Figure 14.1 is a refresher from chapter 12 on the place of the Resource Server in the OAuth 2 authentication architecture.



**Figure 14.1** The Resource Server is one of the components acting in the OAuth 2 authentication architecture. The Resource Server manages the users' data. To call an endpoint on the Resource Server, a client needs to prove with a valid access token that the user approved them to work with their data.

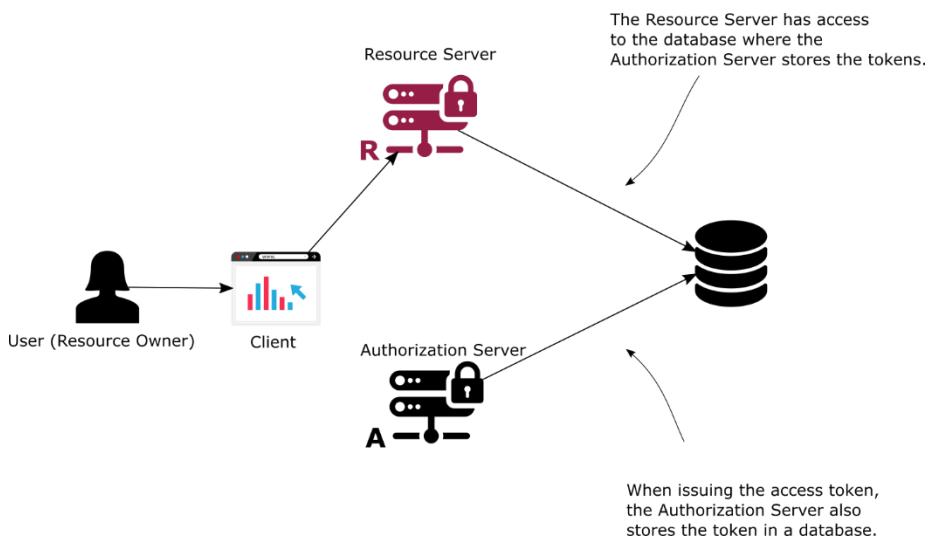
In chapters 12 and 13, we discussed implementing a client and an authorization server. In this chapter, you'll learn how to implement the Resource Server. What's most important when discussing the Resource Server implementation is to choose how the Resource Server validates the tokens. You have multiple options for implementing the validation of the tokens at the Resource Server level. I briefly describe the three options below, and then we'll detail them one by one.

1. Allow the Resource Server to directly call the Authorization Server to verify an issued token.



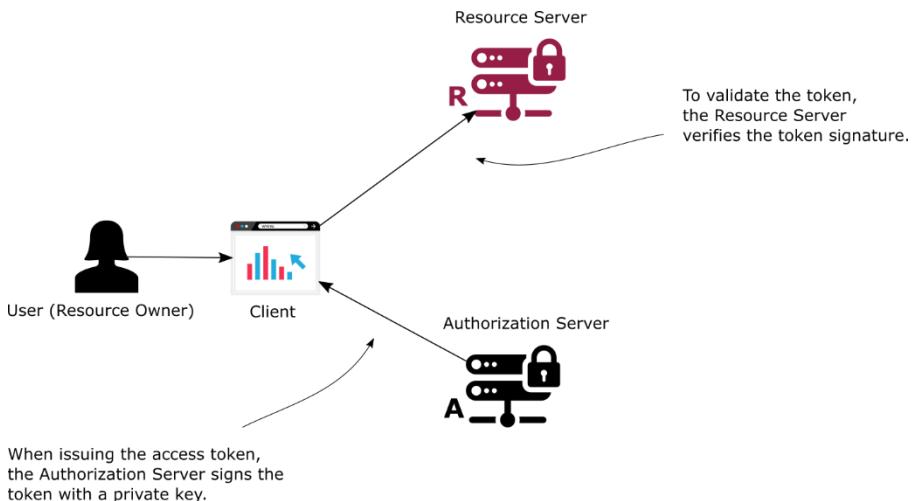
**Figure 14.2** To validate the token, the Resource Server calls the Authorization Server directly. The Authorization Server knows whether it issued a specific token.

2. Use a common database where the Authorization Server stores the tokens, and then the Resource Server can access and validate the tokens. This approach is also called blackboarding.



**Figure 14.3** Blackboarding: Both the Authorization Server and the Resource Server access a shared database. The Authorization Server stores the tokens into this database when it issues them, and the Resource Server can access them to validate the tokens it receives.

3. Using cryptographic signatures: the authorization server signs the token when issuing it, and the resource server validates the signature. Here's where we generally use JWT nowadays. We'll discuss this approach in chapter 15.

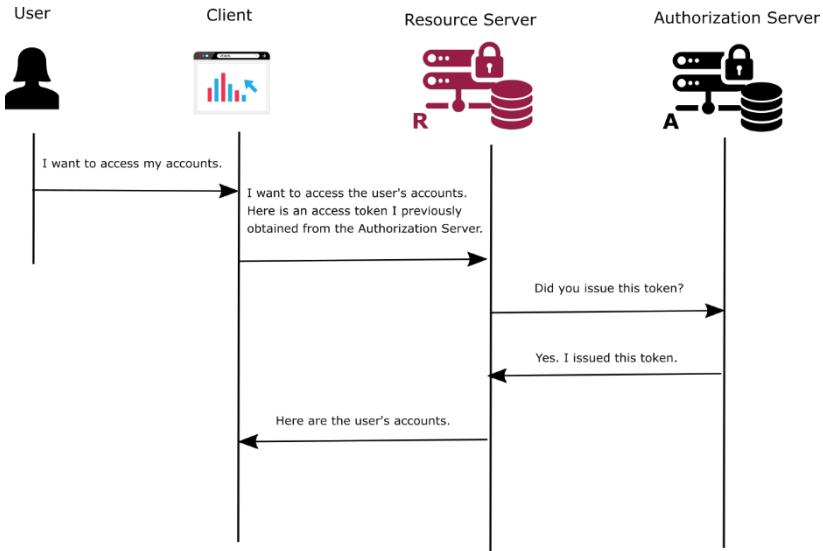


**Figure 14.4** When issuing the access token, the Authorization Server uses a private key to sign the token. To verify a token, the Resource Server only needs to check if the signature is valid.

## 14.1 Implementing a resource server

We start with the implementation of our first Resource Server application. The last piece of the OAuth 2 puzzle is the Resource Server. The whole reason why we have an Authorization Server that issues tokens is to allow clients to access users' resources. The Resource Server manages and protects the users' resources, so for this reason, you need to know how to implement a Resource Server. We'll use the default implementation provided by Spring Boot, which is allowing the Resource Server to directly call the Authorization Server to find out if a token is valid (figure 14.5).

**NOTE** Like in the case of the Authorization Server, the implementation of the Resource Server suffered changes. These changes affect us because now, you'll find in practice different ways in which developers implemented the Resource Server. I'll provide you examples for both ways in which you can configure the Resource Server such that, when you encounter them in real-world scenarios, you will understand and be able to use both.



**Figure 14.5** When the Resource Server needs to validate a token, it directly calls the Authorization Server. If the Authorization Server confirms it issued the token, then the Resource Server considers the token valid.

To implement a Resource Server, we create a new project and add the needed dependencies as presented in the next code snippet. I'll name this project `ssia-ch14-ex1-rs`.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
  
```

Besides the dependencies, you'll add the dependency management tag for the `spring-cloud-dependencies` artifact, as presented in the next code snippet.

```

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
  
```

```
</dependencyManagement>
```

The purpose of the Resource Server is to manage and protect the users' resources. So to prove how it works, we need a resource that we want to access. We'll create an endpoint /hello for our tests by defining a usual controller, as presented in listing 14.12.

#### **Listing 14.12 The controller class**

```
@RestController
public class HelloController {

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

The other thing we need is a configuration class in which we'll use the `@EnableResourceServer` annotation to allow Spring Boot configure for us what's needed for our app to become a resource server. Listing 14.13 presents the configuration class.

#### **Listing 14.13 The configuration class**

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig { }
```

We have a resource server now. But it's not useful if you can't access the endpoint. And that's because we didn't configure any way in which the resource server can check the tokens. You know that the requests made for resources have also to provide a valid access token. Even if they do provide a valid access token, the request still won't work. Our Resource Server cannot verify that they're valid tokens - that the authorization server indeed issued them.

We didn't implement any of the options the Resource Server has to validate the access tokens. Let's take these approaches and discuss them in detail in the next sections.

**NOTE** As I mentioned in an earlier note, the Resource Server implementation changed as well. The `@EnableResourceServer` annotation, which is part of the Spring Security OAuth project, was recently marked as deprecated. In the Spring Security migration guide (<https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide>), the Spring Security team invites us to use configurations methods directly from Spring Security. However, currently, I still mostly encountered the usage of the Spring Security OAuth project in most of the apps I recently saw. For this reason, I consider it really important that you understand both approaches. In our examples, we'll cover both.

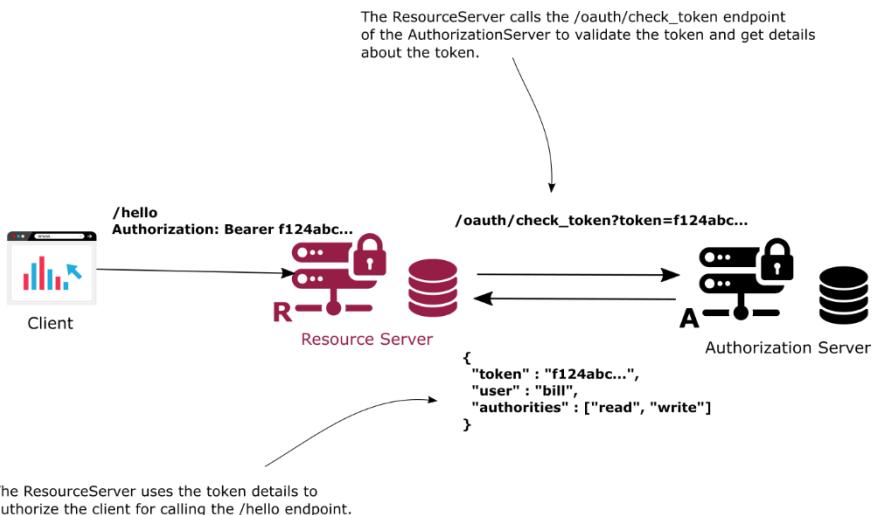
## **14.2 Checking the token remotely**

In this section, we implement the validation of the token by allowing the Resource Server to call the Authorization Server directly. This approach is the simplest you can implement to enable access to the resource server with a valid access token. You choose this approach if the

tokens in your system are plain (for example, simple UUIDs as in the default implementation of the Authorization Server with Spring Security). We start by discussing the approach, and then we'll implement it within an example.

This mechanism for validating the tokens is simple (figure 14.6):

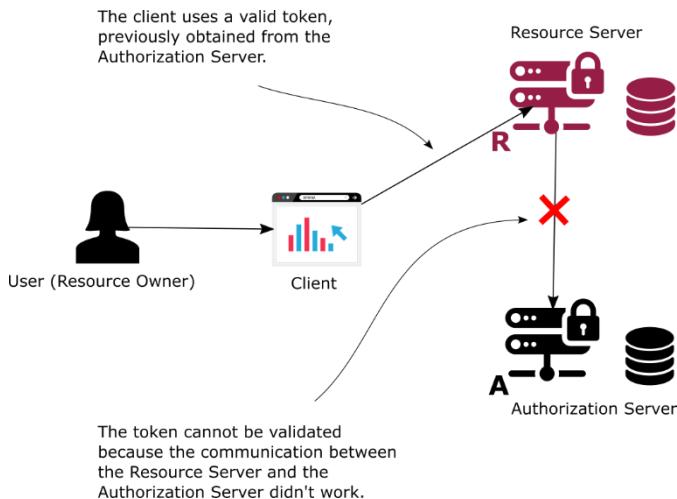
1. The Authorization Server exposes an endpoint which, for a valid token, returns the granted authorities of the user to whom it was issued earlier. Let's call this endpoint the check token endpoint.
2. The Resource Server calls the check token endpoint for each request. This way, it validates the token received from the client and also obtains the client granted authorities.



**Figure 14.6** To validate the token and obtain details about it, the Resource Server calls the `/oauth/check_token` endpoint of the Authorization Server. The Resource Server uses the details retrieved about the token to authorize the call.

The advantage of this approach is its simplicity - you can apply it to any kind of token implementations.

The disadvantage of this approach is that for each request on the resource server having a new, unknown yet, token, the resource server calls the authorization server for validating the token. These calls may apply unnecessary stress to the authorization server. Also, remember the rule of thumb: the network is not 100% reliable. You need to keep this in mind every time you design a new remote call in your architecture. You might need to also apply some alternative solutions for what happens if the call fails because of some network instability (figure 14.7).



**Figure 14.7** The network is not 100% reliable. If the communication between the Resource Server and the Authorization Server is down, the tokens cannot be validated. This implies that the Resource Server will refuse the client to access the user's resources even if it has a valid token.

Let's continue our resource server implementation in project `ssia-ch14-ex1-rs`. What we want is to allow a client to access the `/hello` endpoint if it provides an access token issued by an authorization server. We've already developed authorization servers in chapter 13. We could use, for example, the project `ssia-ch13-ex1` as our authorization server. But to avoid changing the project we already discussed in the previous section, I'll create a separate project for this discussion, and I'll name it `ssia-ch14-ex1-as`. Mind that it now has the same structure as the project `ssia-ch13-ex1` and what I'll present you in this section is only the changes I do in regards to our current discussion.

**NOTE** You can use the configuration we discuss here with any other grant type (grant types, which I described in chapter 12, are the flows implemented by the OAuth 2 framework in which the Authorization Server issues the token). So you can choose to continue our discussion using the Authorization Server we implemented in `ssia-ch13-ex2`, `ssia-ch13-ex3`, or `ssia-ch13-ex4` projects if you'd like.

By default, the authorization server implements the endpoint `/oauth/check_token` that can be used by the resource server to validate a token. However, the authorization server implicitly denies all the requests to that endpoint. Before using the `/oauth/check_token` endpoint you need to make sure the resource server can call this endpoint. To allow authenticated requests to the `/oauth/check_token` endpoint, we override the `configure(AuthorizationServerSecurityConfigurer c)` method in the `AuthServerConfig` class of the authorization server. Overriding the `configure()` method allows us to set the condition in which the `/oauth/check_token` endpoint can be called. Listing 14.14 shows you how to do this configuration.

**Listing 14.14 Enabling authenticated access to the check token endpoint**

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }

    public void configure(
        AuthorizationServerSecurityConfigurer security) {
        security.checkTokenAccess("isAuthenticated()");      #A
    }
}

```

#A We specify the condition in which the check token endpoint can be called.

**NOTE** You can even make this endpoint accessible without authentication using `permitAll()` instead of `isAuthenticated()`. But it's not recommended to leave endpoints unprotected. So preferably, in a real-world scenario, use authentication for this endpoint.

Besides making this endpoint accessible, if we decide to allow only authenticated access, then we need a client registration of the resource server itself. For the authorization server, the resource server is also a client now and requires its own credentials. We add them as for any other client. For the resource server, you don't need any grant type or scope, but only a set of credentials which the resource server will use to call the check token endpoint. In listing 14.15, you find the change in the configuration I've done to add the credentials for the resource server in our example.

**Listing 14.15 Adding a group of credentials for the resource server**

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

```

```
// Omitted code

@Override
public void configure(
    ClientDetailsServiceConfigurer clients)
    throws Exception {

    clients.inMemory()
        .withClient("client")
        .secret("secret")
        .authorizedGrantTypes("password", "refresh_token")
        .scopes("read")
        .and() #A
        .withClient("resourceserver") #A
        .secret("resourceserversecret"); #A
    }

}
```

#A We added a set of credentials that the resource server will use to authenticate when calling the /oauth/check\_token endpoint

You can now start the authorization server and obtain a token, as you learned in chapter 13.

```
curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&
      scope=read"
```

The response body is:

```
{
    "access_token": "4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b",
    "token_type": "bearer",
    "refresh_token": "a4bd4660-9bb3-450e-aa28-2e031877cb36",
    "expires_in": 43199, "scope": "read"
}
```

Now we call the check token endpoint to find the details about the access token we obtained in the previous code snippet.

```
curl -XPOST -u resourceserver:resourceserversecret
      "http://localhost:8080/oauth/check_token?token=4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b"
```

The response body is:

```
{
    "active": true,
    "exp": 1581307166,
    "user_name": "john",
    "authorities": ["read"],
    "client_id": "client",
    "scope": ["read"]
}
```

Observe the response we got from the check token endpoint. It tells us all the details needed about the access token:

- If the token is still active and when it expires

- The user the token was issued for
- The authorities which represent the privileges
- The client the token was issued for

If we could call the endpoint using curl, the resource server should now be also able to use it to validate the tokens. We just need to configure the endpoint of the authorization server and the credentials the resource server has to use to access the endpoint. We can do all these in the `application.properties` file, as presented in the next code snippet.

```
server.port=9090
security.oauth2.resource.token-info-uri=http://localhost:8080/oauth/check_token
security.oauth2.client.client-id=resourceServer
security.oauth2.client.client-secret=resourceServerSecret
```

**NOTE** When we use authentication for the `/oauth/check_token` (token introspection) endpoint, the Resource Server acts as a client for the Authorization Server. For this reason, it needs to have some credentials registered, which it uses to authenticate using HTTP Basic authentication when calling the introspection endpoint.

By the way, if you plan to run both applications on the same system, as I do, don't forget to set a different port using the `server.port` property. I use port 8080 (the default one) for running the authorization server and port 9090 for the resource server.

You can run both applications and test the whole setup by calling the `/hello` endpoint. You need to set the access token in the Authorization header of the request, and you need to prefix its value with the word "bearer". For the word "bearer" the case is insensitive. That means that you can also write "Bearer" or "BEARER".

```
curl -H "Authorization: bearer 4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b"
      "http://localhost:9090/hello"
```

The response body is:

```
Hello!
```

If you had called it without a token or with a wrong one, the result would have been a 401 Unauthorized status on the HTTP response, as presented in the next code snippet.

```
curl -v "http://localhost:9090/hello"
```

The (truncated) response is:

```
...
< HTTP/1.1 401
...
{
  "error": "unauthorized",
  "error_description": "Full authentication is
                      required to access this resource"
}
```

## Using token introspection without Spring Security OAuth

A common question nowadays is, how do we implement the same without Spring Security OAuth. In the end, it's said Spring Security OAuth to be deprecated (mind that my opinion is you should still understand it because there're great chances you'll find these classes out there in existing projects). To clarify this aspect, I'll add a comparison with the way to implement the same thing without the Spring Security OAuth where relevant.

In this sidebar, we discuss the implementation of a Resource Server, using token introspection without using the Spring Security OAuth project, but directly with Spring Security configurations.

Fortunately, it's easier than you might imagine. Remember, we discussed `httpBasic()`, `formLogin()`, and other authentication methods in the previous chapters? I've thought you at that time that when calling such a method, you simply add a new filter to the filter chain, which enables a different authentication mechanism in your app. Guess what? Spring Security also offers you in the latest versions an `oauth2ResourceServer()` method, which enables a Resource Server authentication method in your app. You just have to use it like any other method you've used until now to set up the authentication method, and you need no longer the Spring Security OAuth project in your dependencies.

However, mind that this functionality isn't the same mature yet, and for using it, you'd need to add other dependencies which are not automatically figured out by Spring Boot as you might expect. In the next code snippet, you find the required dependencies for implementing a Resource Server using token introspection.

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>oauth2-oidc-sdk</artifactId>
    <version>8.4</version>
    <scope>runtime</scope>
</dependency>
```

Once you've added the needed dependencies to your `pom.xml` file, you can configure the authentication method as shown in the next code snippet.

```
@Configuration
public class ResourceServerConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2ResourceServer(
                c -> c.opaqueToken(
                    o -> {
                        o.introspectionUri("...");
```

```

        o.introspectionClientCredentials("client", "secret");
    })
);
}
}

```

To make the code snippet easier to be read, I've omitted the parameter value of the `introspectionUri()` method, which is the `check_token` URI, also known as the introspection token URI.

As a parameter of the `oauth2ResourceServer()` method, I've added a `Customizer` instance. Using the `Customizer` instance, you specify the parameters needed for the Resource Server to work depending on the approach you choose. For a direct token introspection, you need to specify the URI the Resource Server calls to validate the token and the credentials the Resource Server needs to authenticate when calling this URI.

You find this example implemented in project `ssia-ch14-ex1-rs-migration`.

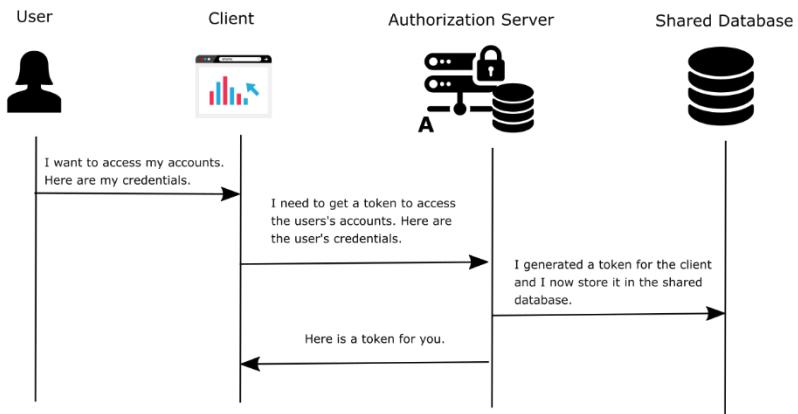
## 14.3 Implementing blackboarding with a `JdbcTokenStore`

In this section, we'll implement an application where the Authorization Server and the Resource Server use a shared database. We call this architectural style "blackboarding", because it's like the Authorization Server and the Resource Server use a blackboard to manage the tokens. This approach for issuing and validating the tokens has the advantage of eliminating the direct communication between the Resource Server and the Authorization Server. However, it implies adding a shared database, which might become a bottleneck. Like any architectural style, you'll find it applicable in various situations. For example, if you already have the services sharing a database, maybe it makes sense to use this approach also for your access tokens. For this reason, I consider it important for you to know how to implement this approach.

As for the previous implementation, we'll work on an application to demonstrate the implementation of such an architecture. You find this application in the projects provided with the book as `ssia-ch14-ex2-as` for the Authorization Server, and `ssia-ch14-ex2-rs` for the Resource Server.

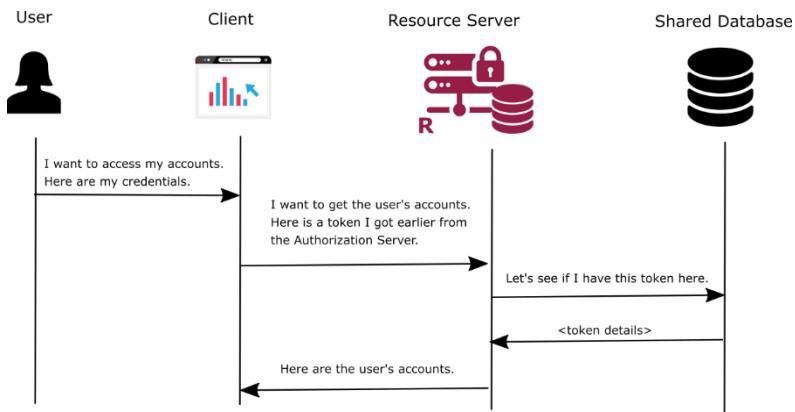
This architecture implies that:

1. When the authorization server issues a token, the Authorization Server stores the token in the database shared with the Resource Server (figure 14.8).



**Figure 14.8** When the Authorization Server issues a token, it also stores the token in the shared database. This way, the Resource Server will be able to get the token and validate it later.

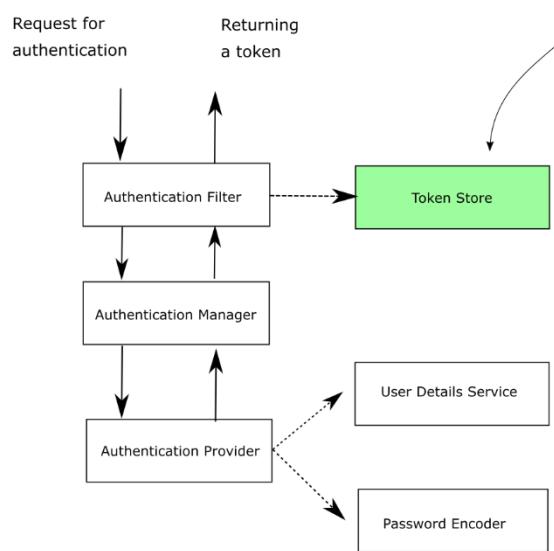
2. The Resource Server access this database when needed to validate the token (figure 14.9).



**Figure 14.9** The Resource Server searches for the token in the shared database. If the token exists, the Resource Server finds in the database also details related to it including the username and its authorities. With these details, the Resource Server can authorize the request.

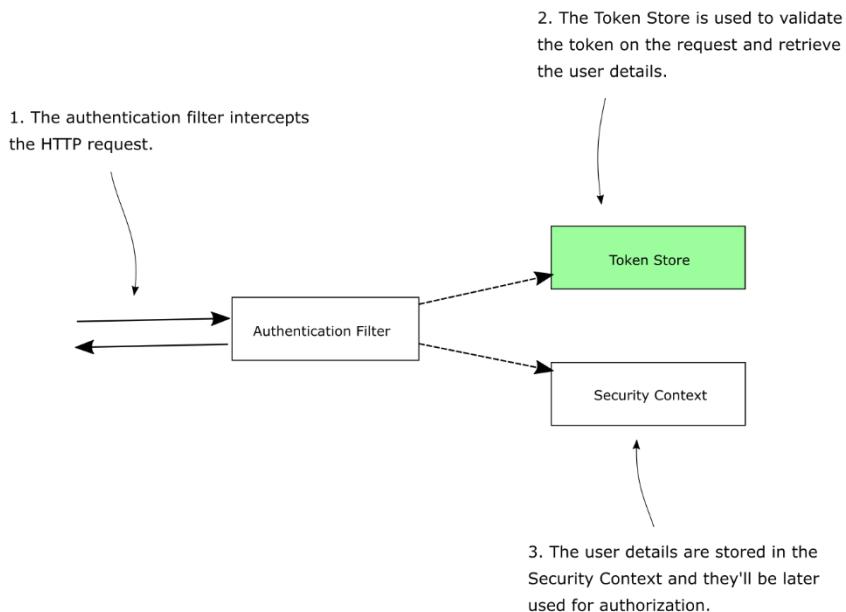
The contract representing the object which manages the tokens in Spring Security, both on the Authorization Server side as well as for the Resource Server is called `TokenStore`. For the Authorization Server, you can visualize the `TokenStore`'s place in the authentication architecture is where we've previously had the `SecurityContext`. Once the authentication is finished, the Authorization Server uses the token store to generate a token (figure 14.10).

Once the request is authenticated, the Token Store is used to generate a token. The token will be returned in the HTTP response.



**Figure 14.10** The Authorization Server uses the Token Store to generate tokens at the end of the authentication process. The client uses these tokens to access resources managed by the Resource Server.

For the Resource Server, the authentication filter uses the `TokenStore` to validate the token and find the user details that it'll use later for authorization. The Resource Server stores the user details in the Security Context (figure 14.11).



**Figure 14.11** The Resource Server uses the TokenStore to validate the token and retrieve details needed for authorization. These details are then stored in the Security Context.

**NOTE** The Authorization Server and the Resource Server are two different responsibilities. But they don't necessarily have to be implemented by two separate applications. In most of the real-world implementations, you'll develop them in different applications, and this is also why we do the same in our examples in this book. But you can choose to implement both as the same application. In this case, being the same application, you don't need to establish any call or have a shared database. If you'd implement the two responsibilities into the same app, then both the Authorization Server and Resource Server can access the same beans. As such, they can use the same TokenStore without needing to do network calls or access a database.

Spring Security offers various implementations for the `TokenStore` contract, and in most cases, you won't need to write your own implementation. For example, for all the Authorization Server implementations we did until now in this book, we did not specify any `TokenStore`. Spring Security provided a default token store of type `InMemoryTokenStore`. As you can imagine, in all our cases until now, the tokens were stored in the application memory. They haven't persisted anyhow, and, if you'd restart the authorization server, the tokens issued before the restart won't be valid anymore.

To implement the token management with blackboarding, Spring Security offers the `JdbcTokenStore` implementation. As the name suggests, this `TokenStore` works with a database directly via JDBC. It works very similarly to the `JdbcUserDetailsManager` we discussed in chapter 3, but instead of managing users, the `JdbcTokenStore` manages tokens.

**NOTE** In this example, we use the `JdbcTokenStore` to implement blackboarding. But you could choose to use this `TokenStore` implementation just to persist the tokens and continue using the `/oauth/check_token` endpoint. You would choose to do so if you don't want to use a shared database, but you need to persist the tokens such that, if the Authorization Server restarts, the previously issued tokens can still be used.

The `JdbcTokenStore` expects you to have two tables in the database. It uses a table to store the access tokens – the name for this table should be `oauth_access_token`, and a table to store refresh tokens. The name for the table in which the token store persists the refresh tokens should be `oauth_refresh_token`.

**NOTE** As in the case of the `JdbcUserDetailsManager` component, which we discussed in chapter 3, you can customize the `JdbcTokenStore` to use other names for tables or columns. The `JdbcTokenStore` offers methods to override any of the SQL queries it uses to retrieve or store the details of the tokens. To keep it short, in our example, we'll use the default namings.

We need to change our `pom.xml` file to declare the necessary dependencies to connect to our database. The next code snippet presents the dependencies I have in my `pom.xml` file.

```
<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.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

In the Authorization Server project, `ssia-ch14-ex2-as`, I'll define the `schema.sql` file with the queries needed to create the structure for these tables. Don't forget that this file needs to be in the `resources` folder to be picked up by Spring Boot when the application starts. The

next code snippet presents the definition of the two tables, as should be written in the schema.sql file.

```
CREATE TABLE IF NOT EXISTS `oauth_access_token` (
    `token_id` varchar(255) NOT NULL,
    `token` blob,
    `authentication_id` varchar(255) DEFAULT NULL,
    `user_name` varchar(255) DEFAULT NULL,
    `client_id` varchar(255) DEFAULT NULL,
    `authentication` blob,
    `refresh_token` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`token_id`));

CREATE TABLE IF NOT EXISTS `oauth_refresh_token` (
    `token_id` varchar(255) NOT NULL,
    `token` blob,
    `authentication` blob,
    PRIMARY KEY (`token_id`));
```

In the application.properties file, you need to add the definition of the data source as presented in the next code snippet.

```
spring.datasource.url=jdbc:mysql://localhost/spring?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

Listing 14.16 presents the AuthServerConfig class, the way we already have it from the first example.

#### **Listing 14.6 The AuthServerConfig class**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

```
}
```

```
}
```

We change this class to inject the data source and then define and configure the `TokenStore` as presented in listing 14.7.

#### **Listing 14.7 Defining and configuring the `JdbcTokenStore`**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private DataSource dataSource;    #A

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore());    #B
    }

    @Bean
    public TokenStore tokenStore() {    #C
        return new JdbcTokenStore(dataSource);    #C
    }    #C
}
```

#A We inject the data source we configured in the `application.properties` file.

#B We configure the token store.

#C We create an instance of `JdbcTokenStore` providing access to the database through the data source earlier configured in the `application.properties` file.

We can already start our Authorization Server and issue tokens. We issue the tokens in the same way we were doing until now in chapters 13 and this chapter. From this perspective, nothing changed. Just that now, we'll be able to see our tokens stored in the database as well. The next code snippet shows the curl command you can use to issue a token.

```
curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&
```

```
scope=read"
```

The response body is:

```
{
  "access_token": "009549ee-fd3e-40b0-a56c-6d28836c4384",
  "token_type": "bearer",
  "refresh_token": "fd44d772-18b3-4668-9981-86373017e12d",
  "expires_in": 43199,
  "scope": "read"
}
```

The same access token got in the response can also be found now as a record in the `oauth_access_token` table. Because I also have configured the refresh token grant type, I also receive a refresh token. For this reason, I also find a record for the refresh token in the `oauth_refresh_token` table.

Because the tokens are now persisted, the Resource Server will be able to validate the already issued tokens even if the Authorization Server is down or after its restart.

It's time now to configure the Resource Server, so it also uses the same database. For this purpose, I'll work in the project `ssia-ch14-ex2-rs`. I'll start with the implementation we've worked on in section 14.1.

As for the Authorization Server, we need to add the necessary dependencies in the `pom.xml` file. Because the Resource Server needs to connect to the database, we have to add the `spring-boot-starter-jdbc` dependency and the JDBC driver. The next code snippet shows the dependencies I have configured in the `pom.xml` file.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

In the `application.properties` file I configure the data source, so the Resource Server connects to the same database as the Authorization Server. The next code snippet presents the content of the `application.properties` file for the Resource Server.

```
server.port=9090
spring.datasource.url=jdbc:mysql://localhost/spring
```

```
spring.datasource.username=root
spring.datasource.password=
```

In the configuration class of the resource server, we inject the data source and configure the `JdbcTokenStore`. Listing 14.8 shows the changes to the Resource Server configuration class.

#### **Listing 14.8 The Resource Server configuration class**

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Autowired
    private DataSource dataSource;      #A

    @Override
    public void configure(
        ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());      #B
    }

    @Bean
    public TokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);      #C
    }
}
```

#A We inject the data source we have configured in the application.properties file.

#B We configure the token store.

#C We create a `JdbcTokenStore` based on the injected data source.

You can now start your Resource Server as well and call the `/hello` endpoint with the access token you have previously issued. The next code snippet shows you how to call the endpoint using `curl`.

```
curl -H "Authorization:Bearer 009549ee-fd3e-40b0-a56c-6d28836c4384"
      "http://localhost:9090/hello"
```

The response body is:

```
Hello!
```

Fantastic! We've implemented in this section a blackboarding approach for the communication between the Resource Server and Authorization Server. We used the implementation of `TokenStore` called `JdbcTokenStore` for this implementation. Now we persist the tokens into the database, and we avoid the direct calls between the Resource Server and the Authorization Server for validating the tokens. But we have the disadvantage of depending on the same database from both the Authorization Server and the Resource Server. For a large number of requests, this dependency might become a bottleneck and slow down the system. Do we have another implementation option to avoid using a shared database? Yes, in chapter

15, we discuss the alternative to the already presented approaches by using signed tokens with JWT.

**NOTE** Writing the configuration of the Resource Server without the Spring Security OAuth makes it not possible for the moment to use the blackboarding approach.

## 14.4 A short comparison of the approaches

In this chapter, you learned to implement two approaches for allowing the Resource Server to validate the tokens it receives from the client:

- *Directly calling the Authorization Server* – when the Resource Server needs to validate a token, it directly calls the Authorization Server that issued that token.
- *Using a shared database (blackboarding)* – Both the Authorization Server and the Resource Server work with the same database. The Authorization Server stores the issued tokens into the database, and the Resource Server reads them for validation.

Let's briefly sum up in table 14.1 the advantages and disadvantages of these two approaches we've discussed in this chapter.

**Table 14.1 Advantages and disadvantages of implementing the presented approaches for the Resource Server to validate the tokens**

Approach	Advantages	Disadvantages
Directly calling the Authorization Server	Easy to implement. It can be applied to any token implementation.	It implies direct dependency between the Authorization Server and the Resource Server. Might cause unnecessary stress on the Authorization Server.
Using a shared database (blackboarding)	Eliminates the need for direct communication between the Authorization Server and the Resource Server. It can be applied to any token implementation. Persisting the tokens allows authorization to work after an Authorization Server restart or if the Authorization Server is down.	More difficult to implement than directly calling the Authorization Server. Requires one more component in the system – the shared database. The shared database might become a bottleneck and affect the system performance.

## 14.5 Summary

- The Resource Server is a Spring component that manages users' resources.
- The Resource Server needs a way to validate the tokens issued to the client by the Authorization Server. One option for verifying the tokens for the Resource Server is to call the Authorization Server directly. This approach can cause too much stress on the

Authorization Server. I would generally avoid using this approach.

- So that the Resource Server can validate the tokens, we can choose to implement a blackboarding architecture. In this implementation, the Authorization Server and the Resource server access the same database where they manage the tokens. This approach has the advantage of eliminating the direct dependencies between the Resource Server and the Authorization Server. But it implies adding a database to persist the tokens, which could become a bottleneck and affect the system performance for a large number of requests.
- To implement token management, we need to use an object of type `TokenStore`. We can write our own implementation of `TokenStore`, but in most cases, we'll use an implementation provided by Spring Security.
- The `JdbcTokenStore` is a `TokenStore` implementation that you can use to persist the access and refresh tokens in a database.

# 15

## *OAuth 2 – Using JWT and cryptographic signatures*

### This chapter covers

- Validating the tokens using cryptographic signatures.
- Using JSON Web Token (JWT) in the OAuth 2 architecture.
- Signing tokens with symmetric and asymmetric keys.
- Adding custom details to a JWT.

In this chapter, we discuss using JWT as token implementation. You learned in chapter 14 that the Resource Server needs to validate the tokens issued by the Authorization Server. And I told you about three ways to do this:

- Using direct calls between the Resource Server and the Authorization Server, which we implemented in section 14.2.
- Using a shared database for storing the tokens, which we implemented in section 14.3.
- Using cryptographic signatures, which we discuss in this chapter.

Using cryptographic signatures to validate the tokens has the advantage of allowing the Resource Server to validate the tokens without needing to call the Authorization Server directly and without the need for a shared database. This approach to implementing tokens validation is the most used in systems implementing authentication and authorization with OAuth 2 today. For this reason, you need to know this way of implementing the token validation as well. We'll write an example for this case as we've done for the other two methods in chapter 14.

## 15.1 Using tokens signed with symmetric keys with JWT

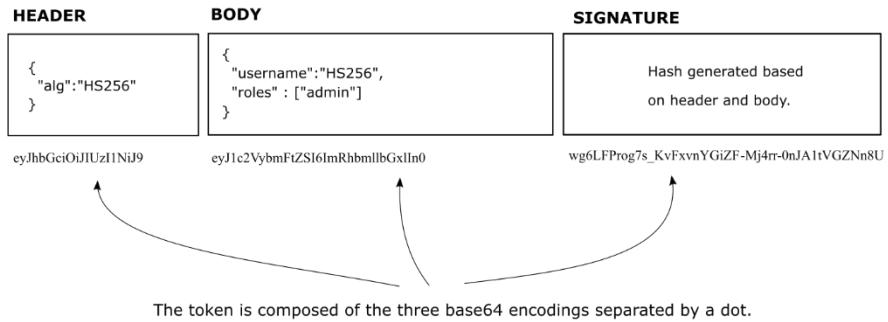
The most straightforward approach to sign the tokens is using symmetric keys. With this approach, using the same key, you can both sign the token and validate its signature. Using symmetric keys for signing the tokens has the advantage of being simpler than other approaches we'll discuss further in this chapter and also faster. As you'll see, however, it has disadvantages too – you can't always share the key used to sign the tokens with all the applications involved in the authentication process. We discuss these advantages and disadvantages when comparing symmetric keys with asymmetric key pairs in section 15.2.

For now, let's start a new project to implement a system that uses JWTs signed with symmetric keys. For this implementation, I'll name the projects `ssia-ch15-ex1-as` for the Authorization Server and `ssia-ch15-ex1-rs` for the Resource Server. We'll start with a brief recap of the JSON Web Tokens (JWTs), which we detailed in chapter 11. Then, we'll implement them in an example.

### 15.1.1 Using JSON Web Tokens

In this section, we briefly recap the JSON Web Tokens (JWTs). We discussed JWTs in chapter 11 in detail, but I think it's better if we start with a refresher on how JWTs work. We'll then continue with implementing the Authorization Server and the Resource Server. Everything we discuss in this chapter relies on JWTs, so this is why I find it essential to start with this refresher first before going further with our first example.

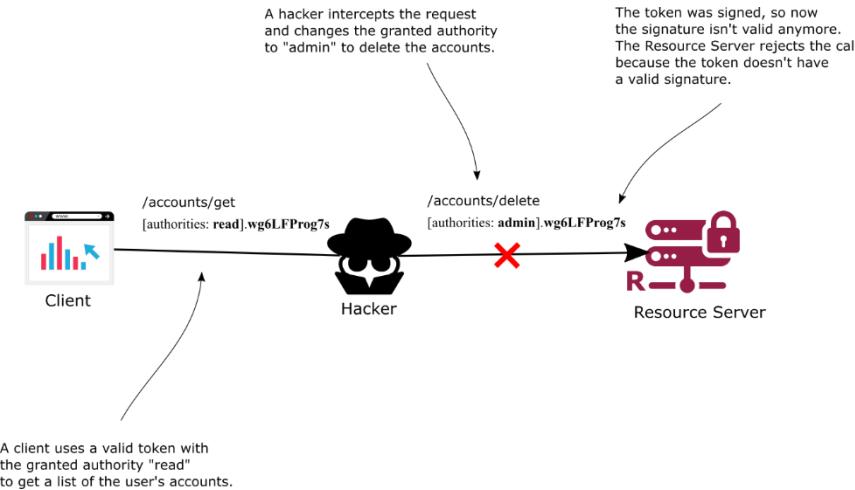
A JSON Web Token (JWT) is a token implementation. A token consists of three parts: the header, the body, and the signature. The details in the header and the body are represented with JSON, and they are Base64 encoded. The third part is the signature generated using a cryptographic algorithm that uses as input the header and the body (figure 15.1). The cryptographic algorithm used to sign the token also implies the need for a key. The key is like a password. Someone having a proper key can sign the token or validate that the signature is authentic. If the signature on the token is authentic, that guarantees that nobody altered the token after it was signed.



**Figure 15.1 A JWT is composed of three parts: the header, the body, and the signature. The header and the body contain details represented with JSON. These parts are Base64 encoded and then signed. The token is a string formed of these three parts separated by dots.**

When a JWT is signed, we also call it a JWS (JSON Web Token Signed). Usually, applying a cryptographic algorithm for signing the token is enough, but sometimes you can choose to encrypt the token. If the token is signed, you can see its contents without having any key or password. But even if they see the contents in the token, they can't change the token's contents because if they do so, the signature becomes invalid (figure 15.2). To be valid, a signature has to

- be signed with the correct key.
- match the content that was signed.



**Figure 15.2 A hacker intercepts the token and changes its content. The Resource Server rejects the call because the signature of the token doesn't match anymore to the content.**

If a token is encrypted, we also call it a JWE (JSON Web Token Encrypted). You can't see the contents of the encrypted token without a valid key.

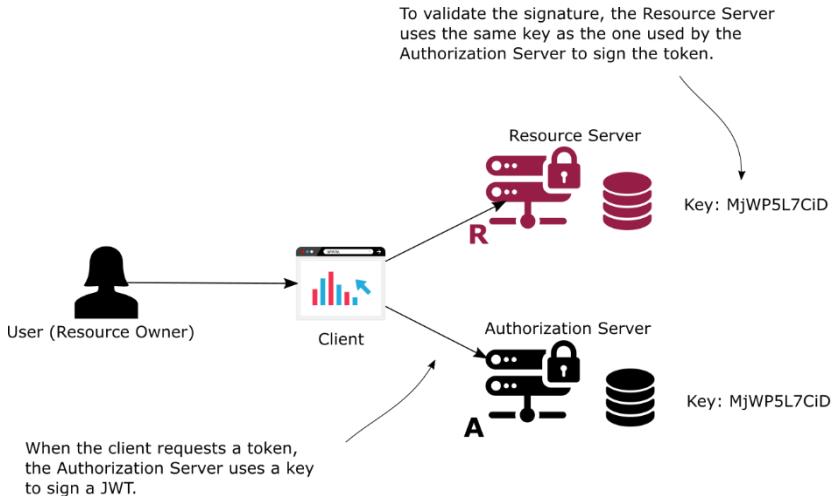
### 15.1.2 Implementing an Authorization Server which issues JWT

In this section, we implement an Authorization Server that issues JWT tokens to the client for authorization. You learned in chapter 14 that the component managing the tokens is the `TokenStore`. So what we'll actually do in this section is to use a different implementation of `TokenStore` provided by Spring Security. The name of the implementation we'll use is `JwtTokenStore`, and it manages JWT tokens. We implement and test the Authorization Server in this section. Later, in section 15.1.3, we'll implement a Resource Server and have a complete system that uses JWTs.

You can implement the token validation with JWT in two ways:

- If you use the same key for signing the token as well as for verifying the signature, we say that the key is symmetric.
- If you use one key to sign the token but a different one to verify the signature, we say that we use asymmetric keys.

In this example, we'll implement signing with a symmetric key. This approach implies that both the Authorization Server and the Resource Server know and use the same key. The Authorization Server signs the token with the key, and the Resource Server validates the signature using the same key (figure 15.3).



**Figure 15.3 Using symmetric keys:** Both the Authorization Server and the Resource Server share the same key. The Authorization Server uses the key to sign the issues tokens, and the Resource Server uses the key to validate the signature.

Let's create the project and add the needed dependencies. In my case, the name of the project is `ssia-ch15-ex1-as`. The next code snippet presents the dependencies we need to add. They are the same we also used in chapters 13 and 14 for the Authorization Server.

```
<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.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

We configure a `JwtTokenStore` in the same way we did in chapter 14 for the `JdbcTokenStore`. Additionally, we'll have to define an object of type `JwtAccessTokenConverter`. With the `JwtAccessTokenConverter`, we configure how the Authorization Server validates the token—in our case, using a symmetric key. Listing 15.1 shows you how to configure the `JwtTokenStore` in the configuration class.

#### **Listing 15.1 Configuring the JwtTokenStore**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {
```

```

@Value("${jwt.key}")
private String jwtKey;      #A

@Autowired
private AuthenticationManager authenticationManager;

@Override
public void configure(
    ClientDetailsServiceConfigurer clients)
throws Exception {
    clients.inMemory()
        .withClient("client")
        .secret("secret")
        .authorizedGrantTypes("password", "refresh_token")
        .scopes("read");
}

@Override
public void configure(
    AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints
        .authenticationManager(authenticationManager)
        .tokenStore(tokenStore())      #B
        .accessTokenConverter(        #B
            jwtAccessTokenConverter()); #B
}

@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(      #C
        jwtAccessTokenConverter()); #C
}

@Bean
public JwtAccessTokenConverter jwtAccessTokenConverter() {
    var converter = new JwtAccessTokenConverter();
    converter.setSigningKey(jwtKey); #D
    return converter;
}
}

```

#A We get the value of the symmetric key from the application.properties file.

#B We configure the token store and the access token converted objects.

#C Create a token store with an access token converter associated to it.

#D We set the value of the symmetric key for the access token converter object.

I stored the value of the symmetric key for this example in the application.properties file, as presented in the next code snippet. However, don't forget that the signing key is sensitive data, and you should store it in a secrets vault in a real-world scenario.

```
jwt.key=MjWP5L7CiD
```

Remember from our previous examples with the Authorization Server from chapters 13 and 14 that for every Authorization Server, we also have defined the UserDetailsService and PasswordEncoder. Listing 15.2 reminds you how to configure these components for the

Authorization Server. To keep the explanations short, I won't repeat the same listing for all the following examples of the chapter.

### **Listing 15.2 Configuring user management for the Authorization Server**

```
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

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

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

        uds.createUser(u);

        return uds;
    }

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

    @Bean
    public AuthenticationManager authenticationManagerBean()
        throws Exception {
        return super.authenticationManagerBean();
    }
}
```

We can now start the Authorization Server and call the `/oauth/token` endpoint to obtain an access token. The next code snippet shows you the curl command you can use to call the `/oauth/token` endpoint and the result of the call.

```
curl -v -XPOST -u client:secret
    "http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&
scope=read"
```

The response body is:

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV...",
    "token_type": "bearer",
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV...",
    "expires_in": 43199,
    "scope": "read",
    "jti": "7774532f-b74b-4e6b-ab16-208c46a19560"
}
```

You observe in the response that both the access and the refresh tokens are now JWTs. In the code snippet, I have shortened the tokens to make the code snippet readable. You'll see

in the response in your console that the tokens are much longer. In the next code snippet, you find the decoded (JSON) form of the token's body.

```
{
  "user_name": "john",
  "scope": [
    "read"
  ],
  "generatedInZone": "Europe/Bucharest",
  "exp": 1583874061,
  "authorities": [
    "read"
  ],
  "jti": "38d03577-b6c8-47f5-8c06-d2e3a713d986",
  "client_id": "client"
}
```

Having the Authorization Server, we can go further now and implement the Resource Server.

### 15.1.3 Implementing a Resource Server which uses JWT

In this section, we implement the Resource Server, which uses the symmetric key to validate the tokens issued by the Authorization Server we implemented in section 15.1.2. With this, at the end of this section, you will know how to write a complete OAuth 2 system that uses JWTs signed using symmetric keys.

We create a new project and add the needed dependencies in `pom.xml`, as presented in the next code snippet. I'll name this project `ssia-ch15-ex1-rs`.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

I didn't add any new dependency to what we've already used in chapters 13 and 14.

Because we need one endpoint to secure, I'll define a controller and a method to expose a simple endpoint, which we'll use to test the Resource Server. Listing 15.3 shows you how to declare the controller.

#### **Listing 15.3 The HelloController class**

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
```

```

        return "Hello!";
    }
}

```

Now that we have an endpoint to secure, we declare the configuration class where we configure the `TokenStore`. We'll configure the `TokenStore` for the Resource Server as we've done for the Authorization Server. The most important aspect is to be sure we use the same value for the key. The Resource Server needs the key to validate the tokens' signatures. Listing 15.4 shows you the definition of the Resource Server configuration class.

#### **Listing 15.4 The Resource Server configuration class**

```

@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${jwt.key}")      #A
    private String jwtKey;

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());      #B
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(      #C
            jwtAccessTokenConverter());
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();      #D
        converter.setSigningKey(jwtKey);      #D
        return converter;      #D
    }
}

```

#A We inject the key value from the `application.properties` file.

#B We configure the `TokenStore`.

#C We declare the `TokenStore` and add it to the Spring context.

#D We create an access token converter and set the symmetric key which will be used to validate the token signatures.

Don't forget to set the value for the key in the `application.properties` file. A key used for symmetric encryption or signing is just a random string of bytes. You generate it using an algorithm for randomness. In your example, you could use any string value say "abcde". In a real-world scenario, use a randomly generated value with a length, preferably, longer than 258 bytes. I would also recommend "Real-World Cryptography" by David Wong (Manning, 2020). In chapter 8 of David Wong's book, you find a details discussion on randomness and secrets: <https://livebook.manning.com/book/real-world-cryptography/chapter-8/v-5/>.

Because I'll run both the Authorization Server and the Resource Server locally on the same machine, I need to configure a different port for this application. The next code snippet presents the content of the `application.properties` file.

```
server.port=9090
jwt.key=MjWP5L7CiD
```

We can now start our Resource Server and call the `/hello` endpoint using a valid JWT that you obtained earlier from the Authorization Server. You have to add the token to the Authorization HTTP header on the request prefixed by the word "Bearer". The next code snippet shows you how to call the endpoint with `curl`.

```
curl -H "Authorization:Bearer eyJhbGciOiJIUzI1NiIs..." http://localhost:9090/hello
```

The response body is:

```
Hello!
```

**NOTE** Remember that I've truncated the JWTs in the examples of the book to save space and make the call easier to read.

You've just finished implementing a system that uses OAuth 2 with JWT as a token implementation. As you've found out, Spring Security makes these implementations very easy. In this section, you learned how to use a symmetric key to sign and validate the tokens. But you'll find requirements in real-world scenarios where having the same key on both Authorization Server and Resource Server is not applicable anymore. In section 15.2, you'll learn how to implement a similar system that uses asymmetric keys for the token validation.

## Using symmetric key without the Spring Security OAuth project

Like using opaque tokens and introspection, as we discussed in chapter 14, you can also configure your resource server to use JWTs with the `oauth2ResourceServer()` method.

As we discussed in chapter 14, this approach is more advisable for future projects, but currently, you have fewer chances to find it used in existing apps. You need to know this approach for future implementations and, of course, if you'd like to migrate an existing project to it.

The next code snippet shows you how to configure JWT authentication using symmetric keys without the classes of the Spring Security OAuth project.

```
@Configuration
public class ResourceServerConfig
    extends WebSecurityConfigurerAdapter {

    @Value("${jwt.key}")
    private String jwtKey;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
```

```

http.authorizeRequests()
    .anyRequest().authenticated()
    .and()
    .oauth2ResourceServer(
        c -> c.jwt(
            j -> j.decoder(jwtDecoder());
        )));
}

// Omitted code
}

As you observe, this time, I use the jwt() method of the Customizer object sent as a parameter to oauth2ResourceServer(). Using the jwt() method, we configure further the details needed by our app to validate the tokens. In this case, we discuss validation using symmetric keys. To provide the value of the symmetric key, I've created a JwtDecoder in the same class, as presented in the next code snippet. I set this decoder using the decoder() method.

@Bean
public JwtDecoder jwtDecoder() {
    byte [] key = jwtKey.getBytes();
    SecretKey originalKey = new SecretKeySpec(key, 0, key.length, "AES");

    NimbusJwtDecoder jwtDecoder =
        NimbusJwtDecoder.withSecretKey(originalKey)
            .build();

    return jwtDecoder;
}

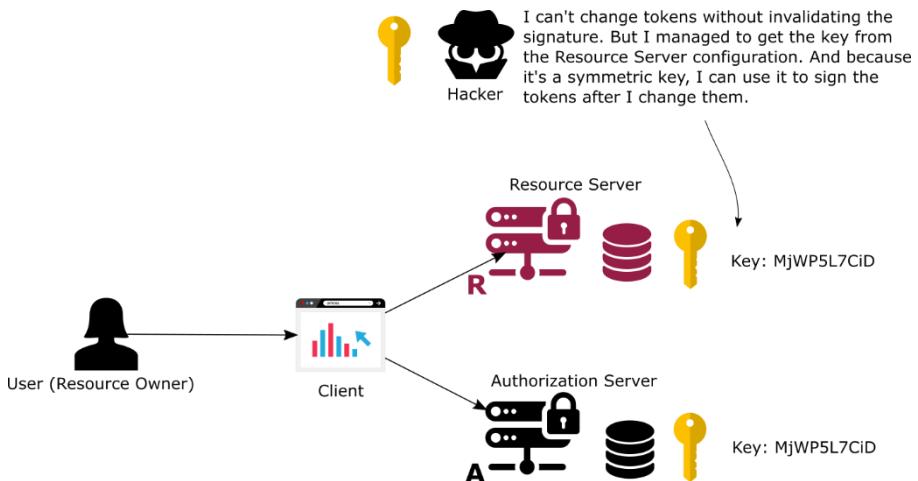
As you observe, the elements we configure are the same. It's only the syntax that differs if you'll choose to use this approach to set up your Resource Server. You find this example implemented in project ssia-ch15-ex1-rs-migration.

```

---

## 15.2 Using tokens signed with asymmetric keys with JWT

In this section, we implement an example of OAuth 2 authentication where the Authorization Server and the Resource Server use an asymmetric key pair to sign and validate the tokens. Sometimes having only a key shared by the Authorization Server and the Resource Server, as we implemented in section 15.1, is not applicable. Often, this scenario happens if the Authorization Server and the Resource Server aren't developed both by the same organization. In this case, we say that the Authorization Server doesn't "trust" the Resource Server. So the Authorization Server cannot share the same key with the Resource Server. With symmetric keys, the Resource Server has too much power: the possibility to sign tokens, not only to validate them (figure 15.4).

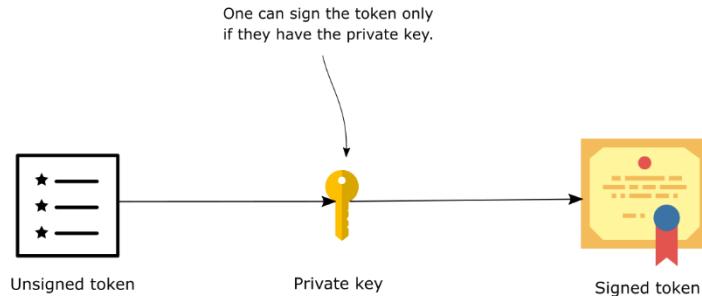


**Figure 15.4** If a hacker manages somehow to get the symmetric key, they can change the tokens and sign them. This way, they get access to the user's resources.

**NOTE** While working as a consultant for different projects, I've seen cases in which symmetric keys were exchanged by mail or other unsecured channels. Never do this! A symmetric key is a private key. One having such a key can use it to access the system. My rule of thumb is: "If you need to share the key outside your system, it shouldn't be symmetric."

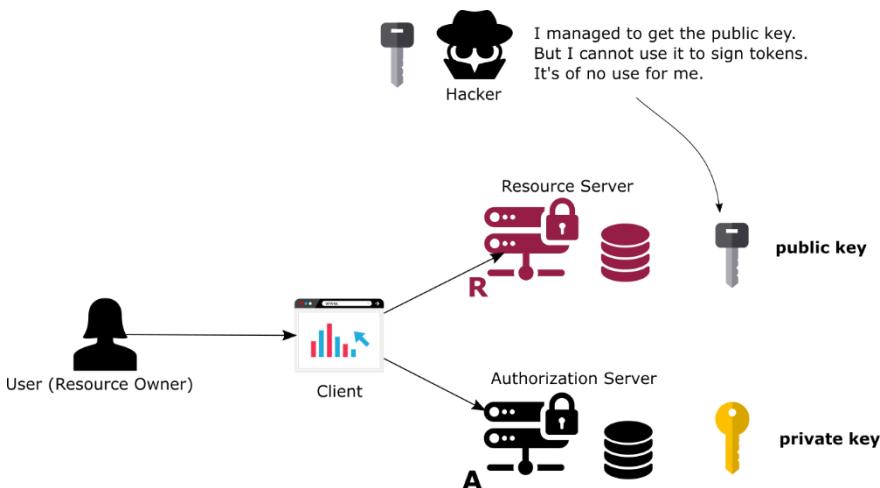
When we want to avoid the need of a trustful relationship between the Authorization Server and the Resource Server, we use asymmetric key pairs. For this reason, you need to know how to implement such a system. In this section, we'll work on an example which shows you all the required aspects of how to achieve this goal.

First, what is an asymmetric key pair, and how does it work? The concept is quite simple: An asymmetric key pair has two keys: one of them is called the private key, and the other is called the public key. The Authorization Server uses the private key to sign the tokens. One can sign the tokens only by using the private key (figure 15.5).



**Figure 15.5** To sign the token, one needs to use the private key. The public key of the key pair would then be used by anyone to verify the identity of the signer.

The public key is linked to the private key, and this is why we call it *a pair*. But the public key can only be used to validate the signature. One cannot sign the token using the public key (figure 15.6).



**Figure 15.6** If a hacker manages to obtain the public key, they won't be able to use it to sign tokens. The public key can only be used to validate the signature.

### 15.2.1 Generating the key pair

In this section, I will teach you how to generate an asymmetric key pair. We need a key pair to configure the Authorization Server and the Resource Server we'll implement in sections 15.2.2 and 15.2.3. That is an asymmetric key pair (which means it has a private part used by

the Authorization Server to sign the token and a public part used by the Resource Server to validate the signature). To generate the key pair, I'll use `keytool` and `openssl`, which are two simple-to-use command-line tools. `keytool` gets installed with your JDK, and probably you already have it on your computer. For `openssl` you need to download it from <https://www.openssl.org/>. You can also use the git bash – git bash comes with `openssl`, so you don't need to install it separately. I always prefer using git bash for these operations because it doesn't require me to install these tools separately.

Once you have the tools, you need to run two commands:

1. Generate the private key.
2. Obtain the public key for the previously generated private key.

#### **GENERATING THE PRIVATE KEY**

To generate a private key, run the command in the next code snippet. It will generate a private key in a file named `ssia.jks`. I have also used the password "ssia123" to protect the private key, and I have used the alias "ssia" to give the key a name. The algorithm used to generate the key – RSA – is also present in the command.

```
keytool -genkeypair -alias ssia -keyalg RSA -keypass ssia123 -keystore ssia.jks -storepass ssia123
```

#### **OBTAINING THE PUBLIC KEY**

To get the public key for the previously generated private key, you can run the command in the next code snippet.

```
keytool -list -rfc --keystore ssia.jks | openssl x509 -inform pem -pubkey
```

You will be prompted to fill the password used when generating the public key – in my case, `ssia123`, and then you should find the public key and a certificate in the output. Only the value of the key is essential for us. This key should look similar to the next code snippet.

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBggkqhkiG9w0BAQEFAOAQ8AMIIIBgKCAQEAIjLqDcBHwtnsBw+WFSzG
VkjtcB06NwK1YjS2PxE114Xwf9H2j0dWmBu7Nk+1v/JqpiOi0GzaLYYf4XtCJxTQ
DD2CeDUKczcd+fpnppripN5jRzhASJpr+ndj8431iAG/rvXrmZt3jLD3v6nwLdxz
pJGmVWzcV/OBXQZkd1LHOK5LEG0YCQ0jAU30N70ZAnFn/DMJyDCky994UtaAYyAJ
7mr7I01uHQxsBg7SiQGpApgDEK3Ty8gaFuafnExsYD+aqua1Ese+pluYnQxuxkk2
Ycsp48qtUv1TWp+TH3kooTM6eKcnP SweaYDvHd/ucNg8UDNpIqynM1eS7KpffKQm
DwIDAQAB
-----END PUBLIC KEY-----
```

This is it! We now have a private key we can use to sign the JWTs and a public key we can use to validate the signature. Now we just have to configure them in our Authorization Server and Resource Server.

### **15.2.2 Implementing an Authorization Server which uses the private key**

In this section, we configure the Authorization Server to use a private key for signing the JWTs. In section 15.2.1, you learned how to generate a private and public key. For this section, I

create a separate project called `ssia-ch15-ex2-as`, but I use the same dependencies in the `pom.xml` file as for the Authorization Server we implemented earlier in section 15.1.

I copy the private key file, `ssia.jks`, in the `resources` folder of my application. I add the key in the `resources` folder because it's easier for me to read it directly from the classpath. However, it's not mandatory to be in the classpath.

In the `application.properties` file I store the name of the file, the alias of the key, and the password I used to protect the private key when I generated the password. We need these details to configure the `JwtTokenStore`. The next code snippet shows you the contents of my `application.properties` file.

```
password=ssia123
privateKey=ssia.jks
alias=ssia
```

Compared with the configurations we did for the Authorization Server to use a symmetric key, the only thing that changes is the definition of the `JwtAccessTokenConverter` object. We still use a `JwtTokenStore`, and you remember we've used the `JwtAccessTokenConverter` to configure the symmetric key in section 15.1. We'll now use the same `JwtAccessTokenConverter` object to set up the private key. Listing 15.5 shows the configuration class of the Authorization Server.

#### **Listing 15.5 The configuration class of the Authorization Server**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Value("${password}")    #A
    private String password;    #A

    @Value("${privateKey}")    #A
    private String privateKey;    #A

    @Value("${alias}")    #A
    private String alias;    #A

    @Autowired
    private AuthenticationManager authenticationManager;

    // Omitted code

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();

        KeyStoreKeyFactory keyStoreKeyFactory =    #B
        new KeyStoreKeyFactory(    #B
            new ClassPathResource(privateKey),    #B
            password.toCharArray()    #B
        );    #B

        converter.setKeyPair(    #C
    }
```

```

        keyStoreKeyFactory.getKeyPair(alias));

    return converter;
}
}

```

#A We inject the name of the private key file, the alias, and the password from the application.properties file.

#B We create a KeyStoreKeyFactory object which reads the private key file from the classpath.

#C We use the KeyStoreKeyFactory object to retrieve the key pair, and we set the key pair to the JwtAccessTokenConverter object.

You can now start the Authorization Server and call the /oauth/token endpoint to generate a new access token. Of course, you'll only see a normal JWT created, but the difference is now that, to validate its signature, you need to use the public key in the pair. By the way, don't forget the token is only signed, not encrypted. The next code snippet shows you how to call the /oauth/token endpoint and the response of the call.

```

curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&
      scope=read"

```

The response body is:

```

{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5...",
  "token_type": "bearer",
  "refresh_token": "eyJhbGciOiJSUzI1NiIsInR5...",
  "expires_in": 43199,
  "scope": "read",
  "jti": "8e74dd92-07e3-438a-881a-da06d6cbbe06"
}

```

### 15.2.3 Implementing a Resource Server which uses the public key

In this section, we implement a Resource Server that uses the public key to verify the tokens' signatures. When we finish this section, you'll have a full system that implements authentication over OAuth 2 and uses a public-private key pair to secure the tokens. The Authorization Server uses the private key to sign the tokens, and the Resource Server uses the public one to validate the signature. Mind, we use the keys only to sign the tokens and not to encrypt them. I'll name the project we work on to implement this Resource Server ssia-ch15-ex2-rs.

We use the same dependencies in pom.xml as for the examples in the previous sections of this chapter.

The Resource Server needs to have the public key of the pair to validate the tokens' signatures, so let's add this key to the application.properties file. In section 15.2.1, you learned how to generate the public key. The next code snippet shows the content of my application.properties file.

```

server.port=9090

publicKey=====BEGIN PUBLIC KEY=====MIIBIjANBghk=====END PUBLIC KEY=====

```

I've abbreviated the public key for better readability. Listing 15.6 shows you how to configure this key in the configuration class of the Resource Server.

#### **Listing 15.6 The configuration class of the Resource Server**

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${publicKey}")      #A
    private String publicKey;

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(          #B
            jwtAccessTokenConverter());    #B
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();
        converter.setVerifierKey(publicKey);  #C
        return converter;
    }
}
```

#A We inject the key from the application.properties file.

#B We create and add a JwtTokenStore in the Spring context.

#C We set the public key which the token store will use to validate the tokens.

Of course, to have an endpoint, we also need to add the controller, as presented in the next code snippet.

```
@RestController
public class HelloController {

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

Let's run and call the endpoint to test the Resource Server.

```
curl -H "Authorization:Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6I..." http://localhost:9090/hello
```

The response body is:

```
Hello!
```

## Using asymmetric keys without the Spring Security OAuth project

In this sidebar, we discuss the changes you'd need to do to migrate your Resource Server using the Spring Security OAuth project to a simple Spring Security one if the app uses asymmetric keys for the token validation.

Actually, using asymmetric keys doesn't differ too much from using a project with symmetric keys. The only change is the `JwtDecoder` you need to use. In this case, instead of configuring the symmetric key for the token validation, you'd need to configure the public part of the key pair. You do this as presented in the next code snippet.

```
public JwtDecoder jwtDecoder() {
    try {
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        var key = Base64.getDecoder().decode(publicKey);

        var x509 = new X509EncodedKeySpec(key);
        var rsaKey = (RSAPublicKey) keyFactory.generatePublic(x509);
        return NimbusJwtDecoder.withPublicKey(rsaKey).build();
    } catch (Exception e) {
        throw new RuntimeException("Wrong public key");
    }
}
```

Once you have a `JwtDecoder` using the public key to validate the token, you just have to set up the decoder using the `oauth2ResourceServer()` method. You do this like in the case of a symmetric key, as also presented in the next code snippet.

```
@Configuration
public class ResourceServerConfig
extends WebSecurityConfigurerAdapter {

    @Value("${publicKey}")
    private String publicKey;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2ResourceServer(
            c -> c.jwt(
                j -> j.decoder(jwtDecoder())
            )
        );

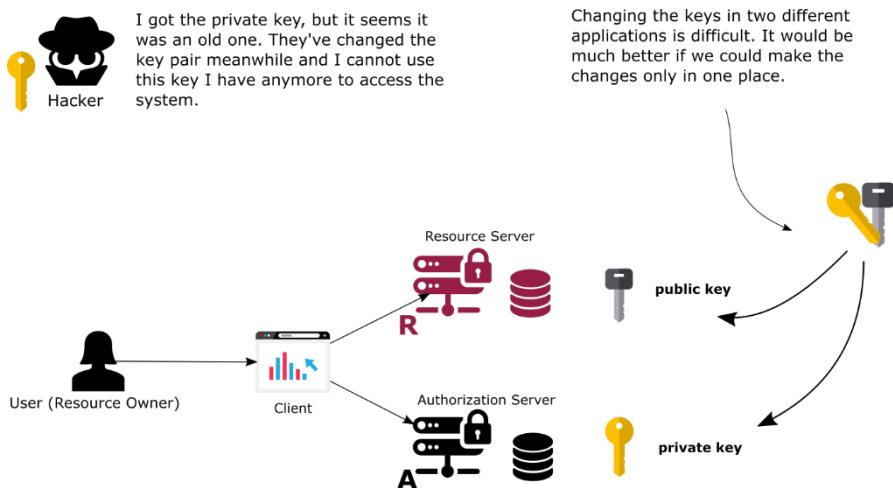
        http.authorizeRequests()
            .anyRequest().authenticated();
    }

    // Omitted code
}
```

You find this example implemented in the project named `ssia-ch15-ex2-rs-migration`.

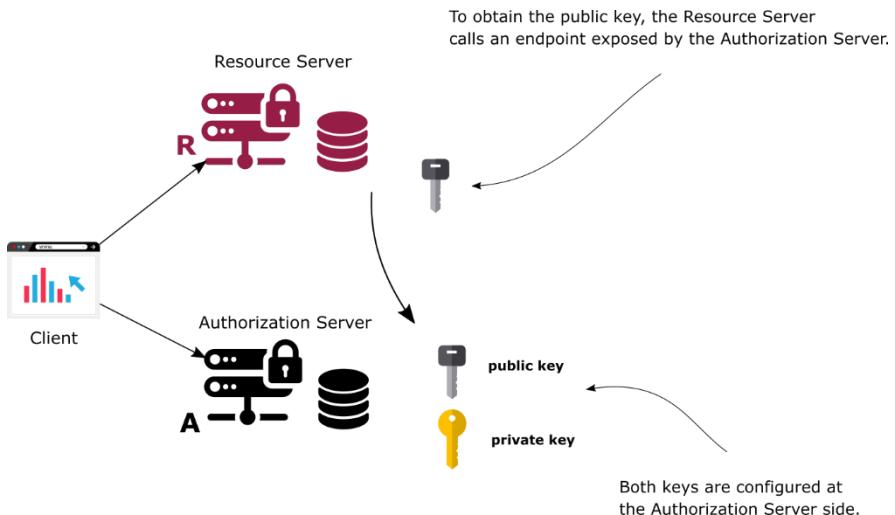
#### 15.2.4 Using an endpoint to expose the public key

In this section, we discuss a way of making the public keys known to the Resource Server, in which the Authorization Server exposes the public keys. In the system we implemented in section 15.2, we use private-public key pairs to sign and validate the tokens. We configured the public key at the Resource Server side. The Resource Server uses the public key to validate the JWTs. But what happens if you want to change the key pair? It is a good practice not to keep the same key pair forever, and this is what you'll learn to implement in this section. At a certain time, you should rotate the keys, this way, making your system less vulnerable to key theft (figure 15.7).



**Figure 15.7** If the keys are changed once in a while, the system is less vulnerable to key theft. But if the keys are configured in both applications, it's more difficult to rotate them.

Up to now, we have configured the private key at the Authorization Server side and the public key at the Resource Server side (figure 15.7). Being set in two places makes the keys more difficult to manage. If we configured them on one side, you could manage them easier. The solution is moving the whole key pair on the Authorization Server side, and allow the Authorization Server to expose the public keys with an endpoint.



**Figure 15.8 Both keys are configured at the Authorization Server. To get the public key, the Resource Server calls an endpoint from the Authorization Server. This approach allows us to rotate the keys easier as we only have to configure them in one place.**

We'll work on a separate application to prove how to make this configuration with Spring Security. You find the Authorization Server of this example in project `ssia-ch15-ex3-as` and the Resource Server of this example in project `ssia-ch15-ex3-rs`.

For the Authorization Server, we keep the same setup as for the project we developed in section 15.2.3. We only need to make sure we make accessible the endpoint, which exposes the public key. Yes, Spring Boot already configures such an endpoint, just that, by default, all the requests for it are denied. We need to override the endpoint's configuration and allow anyone with client credentials to access it. In listing 15.7, you find the changes you need to do to the configuration class of the Authorization Server. These configurations allow anyone with valid client credentials to call the endpoint to obtain the public key.

#### **Listing 15.7 The configuration class of the Authorization Server**

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
    }
}
```

```

        .authorizedGrantTypes("password", "refresh_token")
        .scopes("read")
        .and()      #A
        .withClient("resourceserver")    #A
        .secret("resourceserversecret");   #A
    }

    @Override
    public void configure(
        AuthorizationServerSecurityConfigurer security) {
        security.tokenKeyAccess("isAuthenticated()"); #B
    }
}

```

#A Add client credentials, which will be used by the resource server to call the endpoint, which exposed the public key.  
#B Configure the Authorization Server to expose the endpoint for the public key for any request authenticated with valid client credentials.

You can start the Authorization Server and call the `/oauth/token_key` endpoint to make sure you correctly implemented the configuration. The next code snippet shows you the curl call and its response.

```
curl -u resourceserver:resourceserversecret http://localhost:8080/oauth/token\_key
```

The response body is:

```
{
    "alg": "SHA256withRSA",
    "value": "-----BEGIN PUBLIC KEY----- nMIIBIjANBgkq... -----END PUBLIC KEY-----"
}
```

For the Resource Server to use this endpoint and obtain the public key, you only need to configure the endpoint and the credentials in its properties file. In the next code snippet, you find out how to define the `application.properties` file of the Resource Server.

```
server.port=9090

security.oauth2.resource.jwt.key-uri=http://localhost:8080/oauth/token_key

security.oauth2.client.client-id=resourceserver
security.oauth2.client.client-secret=resourceserversecret
```

Because the Resource Server now takes the public key from the `/oauth/token_key` endpoint of the Authorization Server, you don't need to configure it anymore in the Resource Server configuration class. The configuration class of the Resource Server may remain empty, as presented in the next code snippet.

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {
}
```

You can start the Resource Server as well now, and call the `/hello` endpoint it exposes to see that the entire setup works as expected. The next code snippet shows you how to call the

/hello endpoint using curl. You obtain a token as we did in section 15.2.3 and use it to call the test endpoint of the Resource Server.

```
curl -H "Authorization:Bearer eyJhbGciOiJSUzI1NiIsInR5cCI..." http://localhost:9090/hello
```

The response body is:

```
Hello!
```

### 15.3 Adding custom details to the JWT

In this section, we discuss adding custom details to the JWT token. In most of the cases, you'll need no more than what Spring Security already adds to the token. However, in real-world scenarios, you'll sometimes find requirements for which you need to add custom details in the token. In this section, we implement an example in which you'll learn how to change the Authorization Server to add custom details on the JWT and how to change the Resource Server to read these details.

If you take one of the tokens we generated in previous examples and decode it, you'll find the defaults that Spring Security adds to the token. Listing 15.8 presents you these defaults.

**Listing 15.8 The default details in the body of a JWT issued by the Authorization Server**

```
{
  "exp": 1582581543,      #A
  "user_name": "john",    #B
  "authorities": [        #C
    "read"
  ],
  "jti": "8e208653-79cf-45dd-a702-f6b694b417e7",    #D
  "client_id": "client", #E
  "scope": [            #F
    "read"
  ]
}
```

#A The timestamp when the token expires.

#B The user that authenticated to allow the client to access their resources.

#C Permissions granted to the user.

#D A unique identifier of the token.

#E The client that requested the token.

#F Permissions granted to the client.

As you can see in listing 15.8, generally, by default, a token stores all the details needed for basic authorization. But what if the requirements of your real-world scenarios ask for something more. Some examples are:

1. You use as an Authorization Server an application where readers review books. Some endpoints should only be accessible for users who have given more than a specific number of reviews.
2. You need to allow the calls only if the user authenticated from a specific timezone.
3. Your authorization server is a social network, and some of your endpoints should be accessible only by users having a minimum number of connections.

For my first example, you would need to add the number of reviews to the token. For the second, the timezone from where the client connected is needed. For the third example, you need to add the number of connections of the user. No matter which is your case, you need to know how to customize your JWT.

### 15.3.1 Configuring the Authorization Server to add custom details in the token

In this section, we discuss the changes we need to do to the Authorization Server for adding custom details to the tokens. To make the example simple, I consider the requirement is to add the timezone of the Authorization Server itself. The project I work on for this example is `ssia-ch15-ex4-as`.

To add additional details to your token, you need to create an object of type `TokenEnhancer`. In listing 15.9, you find the definition of the `TokenEnhancer` object I created for the current example.

#### Listing 15.9 A custom token enhancer

```
public class CustomTokenEnhancer
    implements TokenEnhancer {    #A

    @Override
    public OAuth2AccessToken enhance(    #B
        OAuth2AccessToken oAuth2AccessToken,
        OAuth2Authentication oAuth2Authentication) {

        var token =    #C
            new DefaultOAuth2AccessToken(oAuth2AccessToken);

        Map<String, Object> info =    #D
            Map.of("generatedInZone",
                ZoneId.systemDefault().toString());

        token.setAdditionalInformation(info);    #E

        return token;    #F
    }
}
```

#A We implement the `TokenEnhancer` contract.

#B We override the `enhance()` method, which receives the current token and returns the enhanced token.

#C We create a new token object based on the one we received.

#D We define as a Map the details we want to add to the token.

#E We add on the token the additional details.

#F We return the token containing the additional details.

The `enhance()` method of a `TokenEnhancer` object received as a parameter the token we enhance and returns the “enhanced” token. The “enhanced” token is the token containing the additional details.

I use for this example the same application, which we developed in section 15.2 and only change the `configure()` method to apply the token enhancer. Listing 15.10 presents these changes.

#### Listing 15.10 Configuring the `TokenEnhancer` object

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
        public void configure(
            AuthorizationServerEndpointsConfigurer endpoints) {

        TokenEnhancerChain tokenEnhancerChain
            = new TokenEnhancerChain();      #A

        var tokenEnhancers =      #B
            List.of(new CustomTokenEnhancer(),
                    jwtAccessTokenConverter());

        tokenEnhancerChain      #C
            .setTokenEnhancers(tokenEnhancers);

        endpoints
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore())
            .tokenEnhancer(tokenEnhancerChain);    #D

    }
}

```

#A We define a TokenEnhancerChain.

#B We add our two token enhancer objects in a list.

#C We add the token enhancer's list to the chain.

#D We configure the token enhancer objects.

As you observe, configuring our custom token enhancer is a little bit more complicated. We have to create a chain of token enhancers and set the entire chain instead of only one object. The reason why we have to do this is the fact that the access token converter object is also a token enhancer. If we had configured only our custom token enhancer, we would have overridden the behavior of the access token converter. Instead, we add both in a "chain of responsibilities", and we configure the chain containing both objects.

Let's start the Authorization Server, generate a new access token, and inspect it to see how it looks like. The next code snippet shows you how to call the /oauth/token endpoint to obtain the access token.

```
curl -v -XPOST -u client:secret
      "http://localhost:8080/oauth/token?grant_type=password&username=john&password=12345&
      scope=read"
```

The response body is:

```
{
    "access_token": "eyJhbGciOiJSUzI...",
    "token_type": "bearer",
    "refresh_token": "eyJhbGciOiJSUzI1...",
    "expires_in": 43199,
    "scope": "read",
    "generatedInZone": "Europe/Bucharest",
```

```

    "jti": "0c39ace4-4991-40a2-80ad-e9fdeb14f9ec"
}
```

If you decode the token, you'll observe that its body looks like the one presented in listing 15.11. You observe that the framework also adds the custom detail by default in the response as well. But I recommend you always refer to it from the token. Remember that by signing the token, we make sure that if anybody altered the content of the token, the signature doesn't get validated anymore. This way, we know that if the signature is correct, nobody changed the contents of the token. You don't have the same guarantee on the response itself.

#### **Listing 15.11 The body of the enhanced JWT**

```
{
  "user_name": "john",
  "scope": [
    "read"
  ],
  "generatedInZone": "Europe/Bucharest",      #A
  "exp": 1582591525,
  "authorities": [
    "read"
  ],
  "jti": "0c39ace4-4991-40a2-80ad-e9fdeb14f9ec",
  "client_id": "client"
}
```

#A The custom details we added appear in the token's body.

#### **15.3.2 Configuring the Resource Server to read the custom details of a JWT**

In this section, we discuss the changes we need to do to the Resource Server to read the additional details we added to the JWT. Once you've changed your Authorization Server to add custom details to the JWT, you'd like the Resource Server to be able to read these details. The changes you need to do in your Resource Server to access the custom details are straightforward. You find the example we work on in this section in project `ssia-ch15-ex4`-rs.

We discussed in section 15.1 that the `AccessTokenConverter` is the object which converts the token to an `Authentication`. This is the object we need to change so that it also takes into consideration the custom details in the token. Previously you were creating a bean of type `JwtAccessTokenConverter` as presented in the next code snippet.

```
@Bean
public JwtAccessTokenConverter jwtAccessTokenConverter() {
    var converter = new JwtAccessTokenConverter();
    converter.setSigningKey(jwtKey);
    return converter;
}
```

We used this token to set the key used by the Resource Server for the token validation. We create a custom implementation of `JwtAccessTokenConverter`, which also takes into consideration our new details on the token. The simplest way is to extend this class and override the `extractAuthentication()` method. The `extractAuthentication()` method is

responsible for converting the token in an `Authentication`. Listing 15.12 shows you how to implement a custom `AccessTokenConverter`.

#### **Listing 15.12 Creating a custom AccessTokenConverter**

```
public class AdditionalClaimsAccessTokenConverter
    extends JwtAccessTokenConverter {

    @Override
    public OAuth2Authentication
        extractAuthentication(Map<String, ?> map) {

        var authentication =      #A
            super.extractAuthentication(map);

        authentication.setDetails(map);    #B

        return authentication;    #C
    }
}
```

#A We apply the logic implemented by the `JwtAccessTokenConverter` class and get the initial authentication object.

#B We add the custom details to the authentication.

#C We return the authentication object.

In the configuration class of the Resource Server, you now use the custom access token converter. Listing 15.13 shows you how to define the `AccessTokenConverter` bean in the configuration class.

#### **Listing 15.13 Defining the new AccessTokenConverter bean**

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    // Omitted code

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter =
            new AdditionalClaimsAccessTokenConverter();    #A
        converter.setVerifierKey(publicKey);
        return converter;
    }
}
```

#A We create an instance of the new `AccessTokenConverter` object.

An easy way to test the changes is to inject them in the controller class and return them in the HTTP response. Listing 15.14 shows you how to define the controller class.

#### **Listing 15.14 The controller class**

```
@RestController
public class HelloController {
```

```

@GetMapping("/hello")
public String hello(OAuth2Authentication authentication) {
    OAuth2AuthenticationDetails details = #A
        (OAuth2AuthenticationDetails) authentication.getDetails();

    return "Hello! " + details.getDecodedDetails();      #B
}

```

#A We get the extra details added to the Authentication object.

#B We return the details in the HTTP response.

You can now start the Resource Server and test the endpoint with a JWT containing custom details. The next code snippet shows you how to call the `/hello` endpoint and the results of the call. The `getDecodedDetails()` method returns a `Map` containing the details of the token. In this example, to keep it simple, I have directly printed out the entire value returned by `getDecodedDetails()`. If you'd need only to use a specific value, you get the returned `Map` and obtain the desired value using its key.

```
curl -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp... "
http://localhost:9090/hello
```

The response body is:

```
Hello! {user_name=john, scope=[read], generatedInZone=Europe/Bucharest, exp=1582595692,
authorities=[read], jti=982b02be-d185-48de-a4d3-9b27337d1a46, client_id=client}
```

You can spot in the response the new attribute `generatedInZone=Europe/Bucharest`.

## 15.4 Summary

- Using cryptographic signatures is one of the most often used ways in applications today to validate tokens in an OAuth 2 authentication architecture.
- When we use token validation with cryptographic signatures, JSON Web Token (JWT) is the most used token implementation.
- You can use symmetric keys to sign and validate the tokens. While using symmetric keys is a straightforward approach, you cannot use it for the cases in which the Authorization Server doesn't trust the Resource Server.
- If symmetric keys don't apply to your implementation, you can implement the token signing and validation using asymmetric key pairs.
- It's recommended to change the keys regularly to make the system less vulnerable to key theft. We also refer to changing the keys periodically as "keys rotation".
- You can configure public keys directly at the Resource Server side. While this approach is very simple it makes keys rotations more difficult.
- To simplify keys rotations, you can configure the keys at the Authorization Server side and allow the Resource Server to read them at a specific endpoint.
- You can customize JWTs by adding details to their body according to the requirements of your implementations. The Authorization Server adds custom details to the token's body, and the Resource Server uses these details for authorization.

# 16

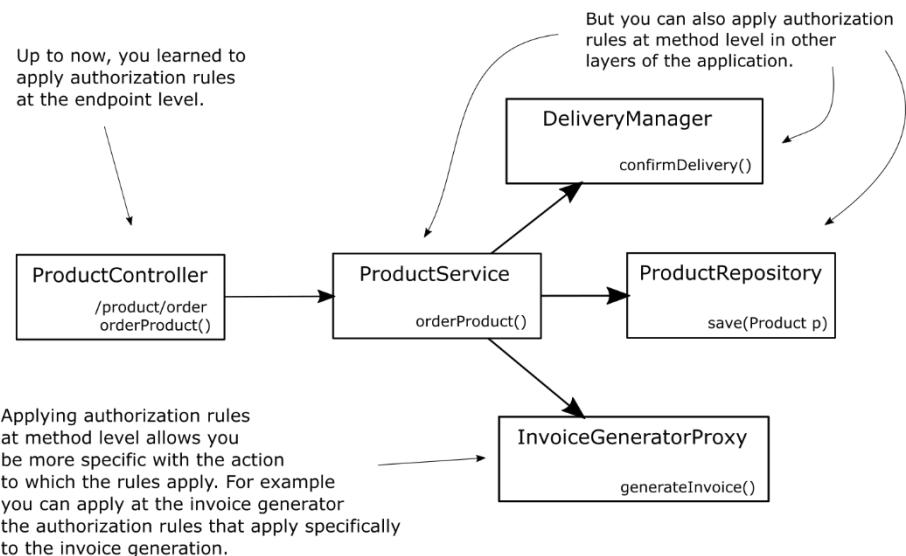
## *Global Method Security – Pre/Post Authorization*

### This chapter covers

- Enabling global method security in Spring applications.
- Using pre-authorization on methods based on authorities, roles, and permissions.
- Using post-authorization on methods based on authorities, roles, and permissions.

Up to now, we discussed various ways of configuring authentication. We started from the most straightforward approach – HTTP Basic in chapter 2, and then I showed you how to set Form Login in chapter 5, and we have, of course, covered OAuth 2 in chapters 12 to 15. But in terms of authorization, we only discussed the configurations at the endpoint level. Say you don't have a web application. Wouldn't you use Spring Security anymore for authentication and authorization? Spring Security is a good fit as well for scenarios in which your app isn't used via HTTP endpoints.

In this chapter, you'll learn how to configure authorization at the method level. We use this approach to configure authorization in both web and non-web applications, and we name it the Global Method Security.



**Figure 16.1 Global Method Security** enables you to apply authorization rules at any layer of your application. This approach allows you to be more granular and apply the authorization rules specifically to the subject to which they apply.

For non-web applications, it offers us 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 Global Method Security.

## 16.1 Enabling global method security

In this section, you'll learn how to enable authorization at the method level and the different options that Spring Security offers to apply various authorization rules. This approach offers you great flexibility in applying authorization. It's an essential skill to have, which allows you to solve situations in which authorization simply cannot be configured just at the endpoint level.

By default, Global Method Security is disabled, so if you wish to use this functionality, you first need to enable it. Also, the Global Method Security offers multiple approaches for applying authorization. We'll discuss these approaches, and then we'll implement them in examples in the following sections of this chapter and chapter 17.

Briefly, you do two main things with Global Method Security:

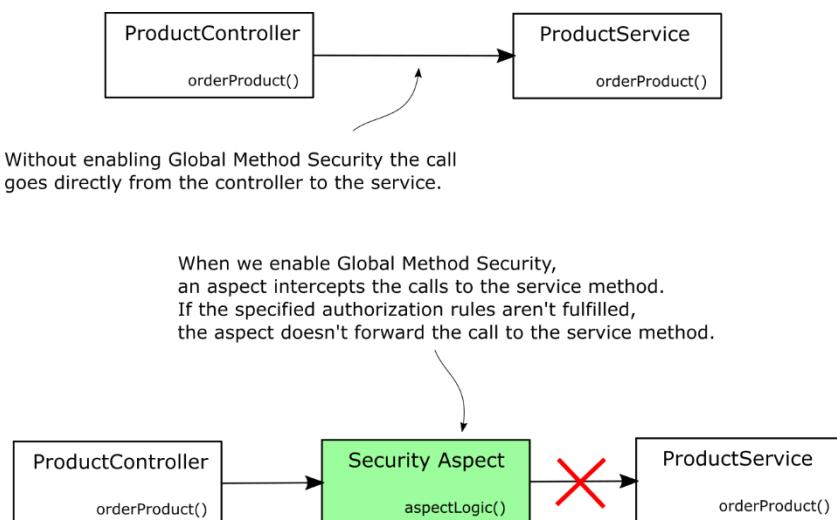
- **Call authorization** – deciding whether one can call a method according to some implemented privilege rules (pre-authorization), or if the caller may access what the method returns after the method was executed (post-authorization).
- **Filtering** – deciding what a method can receive through its parameters (pre-filtering),

and what the caller may receive back from the method after the method's execution (post-filtering). We'll discuss and implement filtering in chapter 17.

### 16.1.1 Understanding call authorization

One of the approaches for configuring authorization rules you use with Global Method Security is the call authorization. The call authorization approach refers to applying authorization rules which decide if a method may be called, or allow the method to be called and decide after if the value returned by the method can be accessed. Often we need to decide if someone may access a piece of logic either depending on the provided parameters or depending on its result. So let's discuss call authorization in this section and then apply it in examples.

How does Global Method Security work? What's the mechanism behind applying the authorization rules? When we enable Global 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 16.2).



**Figure 16.2** When we enable Global Method Security, an aspect intercepts the calls to the protected methods. If the given authorization rules aren't respected, the aspect won't delegate the call to the protected method.

Plenty of implementations in Spring framework rely on aspect-oriented programming (AOP). Global 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 5 of *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools* by Clarence Ho et al. (Apress, 2017).

Shortly, we classify the call authorization as:

- **Pre-authorization** – the framework checks the authorization rules before the call of

the method.

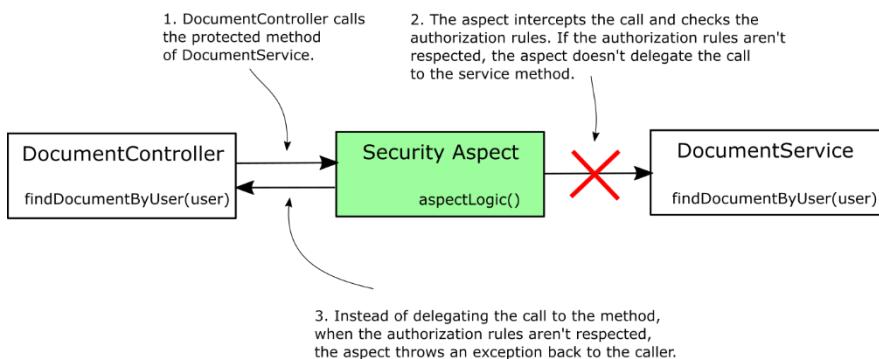
- **Post-authorization** – the framework checks the authorization rules after the method is executed.

Let's take both approaches, detail them, and implement them within examples.

#### USING PRE-AUTHORIZATION TO SECURE ACCESS TO METHODS

Say we have a method `findDocumentsByUser(String username)` that returns the caller the documents for a specific user. The caller provides through the method's parameter the user's name for which the method retrieves the documents. Now assume you need to make sure that the authenticated user may only obtain their own documents. Could we apply a rule to this method such that only the method calls which receive the username of the authenticated user as a parameter are allowed? Yes! This is something we'd do with pre-authorization.

When we apply authorization rules that completely forbid anyone to call the method in specific situations, we name this pre-authorization. This case 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 defined, the framework doesn't delegate the call to the method. Instead, the framework throws an exception. This is by far the most often approach we apply with the Global Method Security.



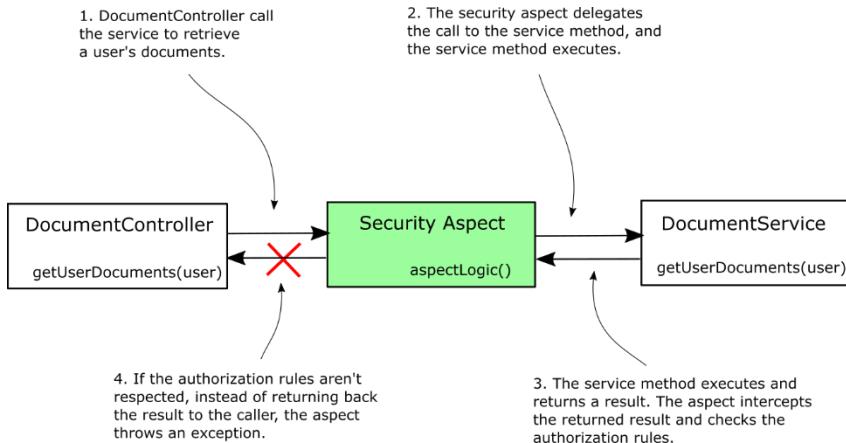
**Figure 16.3 With pre-authorization, the authorization rules are verified before delegating the method call further. The framework won't delegate the call if the authorization rules aren't respected and, instead, will throw an exception to the method caller.**

Usually, we don't want a specific functionality to be executed at all if some conditions aren't met. You may apply these conditions based on the authenticated user, and you may also refer to the values the method received through its parameters.

#### USING POST-AUTHORIZATION TO SECURE A METHOD CALL

When we apply authorization rules that allow one to call a method but not necessarily to obtain the result returned by the method, we say we implement post-authorization. With post-authorization, Spring Security checks the authorization rules after the method executes. You can use this kind of authorization to restrict the caller access to what the method returned in

certain conditions. Because post-authorization happens after the method execution, you can apply the authorization rules on the result returned by the method.



**Figure 16.4** With post-authorization, the aspect delegates the call to the protected method. After the protected method finishes its 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 post-authorization to apply the authorization rules based on what the method returned after execution. But be very careful with the post-authorization! If the method mutates something during its execution, the change happens whether or not the authorization succeeded in the end.

**NOTE** Even with the `@Transactional` annotation, a change isn't rolled back if the post-authorization fails. The exception thrown by the post-authorization functionality happens after the transaction manager commits the transaction.

### 16.1.2 Enabling global method security in your project

In this section, we'll start working on a project to apply the pre-authorization and post-authorization features offered by the Global Method Security. The Global Method Security functionality isn't enabled by default in a Spring Security project. To use it, you need first to enable it. However, enabling this functionality is straightforward. You do this by simply using the `@EnableGlobalMethodSecurity` annotation on the configuration class. I'll create a new project for this example, which I'll call `ssia-ch16-ex1`. For this project, I write a `ProjectConfig` configuration class, as presented in listing 16.1. On the configuration class, we add the `@EnableGlobalMethodSecurity` annotation. Global Method Security offers us three approaches to define the authorization rules which we discuss in this chapter:

- The pre/post-authorization annotations
- The JSR 250 annotation - `@RolesAllowed`

- The `@Secured` annotation

Because in almost all of the cases, the only used approach is the pre/post-authorization annotations, in this chapter, we'll discuss this approach. To enable this approach, we use the `prePostEnabled` attribute of the `@EnableGlobalMethodSecurity` annotation. We'll have a short overview of the other two options at the end of this chapter.

#### **Listing 16.1 Enabling Global Method Security**

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {
```

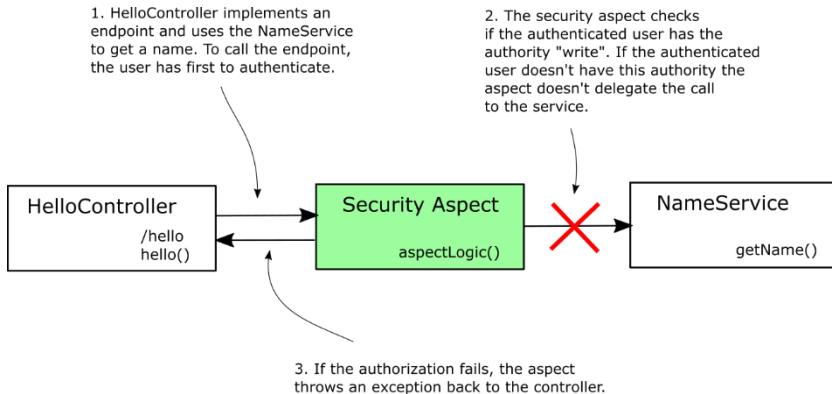
You can use global method security with any authentication approach, from HTTP Basic authentication to OAuth 2. To keep it simple and allow you to focus on the new details, we'll prove Global Method Security with HTTP Basic authentication. For this reason, the `pom.xml` file for the projects of this chapter only needs the web and spring security dependencies, as presented in the next code snippet.

```
<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>
```

## **16.2 Applying pre-authorization for authorities and roles**

In this section, we implement an example of pre-authorization. For our example, we'll continue with the project `ssia-ch16-ex1` started in section 16.1. As we discussed in section 16.1, pre-authorization implies defining authorization rules which Spring Security applies before calling a specific method. If the rules aren't respected, the framework doesn't call the method at all.

The application we'll implement in this section has a very 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. This method applies a pre-authorization rule to verify the user has an authority named "write".



**Figure 16.5** To call the `getName()` method of the `NameService`, the authenticated user needs to have the "write" authority. If the user doesn't have this authority, the framework won't allow the call and will throw an exception.

I'll add a `UserDetailsService` and a `PasswordEncoder` for the beginning to my configuration class to make sure I have some users to authenticate with. To validate our solution, we need two users. One user to have the "write" authority and another one that doesn't have the "write" authority. We will prove 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. Listing 16.2 shows the complete definition of the configuration class, which defines the `UserDetailsService` and the `PasswordEncoder`.

**Listing 16.2 The configuration class defines the `UserDetailsService` and the `PasswordEncoder`**

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)      #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();
    }
}

```

#A Enabling the global method security with pre and post authorization approach.

#B Adding to the Spring context an `UserDetailsService` with two users for test.

#C Adding to the Spring context a `PasswordEncoder`.

To define the authorization rule for the method, we use the `@PreAuthorize` annotation. The `@PreAuthorize` annotation receives as value a Spring Expression Language (SpEL) expression, which describes the authorization rule. In this example, we apply a very simple rule. You can define restrictions for users based on their authorities using the `hasAuthority()` method. You also learned about the `hasAuthority()` method in chapter 7, where we discussed applying authorization at the endpoint level. In listing 16.3, you find the definition of the controller class, which provides the value for the name.

### **Listing 16.3 The service class defines the pre-authorization rule on the method**

```

@Service
public class NameService {

    @PreAuthorize("hasAuthority('write')")      #A
    public String getName() {
        return "Fantastico";
    }
}

```

#A Defining the authorization rule. Only users having the “write” authority can call the method.

We define the controller class, which uses `NameService` as a dependency. Listing 16.4 shows you how to define the controller class.

### **Listing 16.4 The controller class implements the endpoint and uses the service**

```

@RestController
public class HelloController {

    @Autowired      #A
    private NameService nameService;

    @GetMapping("/hello")
    public String hello() {
        return "Hello, " + nameService.getName();      #B
    }
}

```

#A Injecting the service from the context.

#B Calling the method for which we applied the pre-authorization rules.

You can now start the application and test its behavior. We expect only user “emma” to be authorized to call the endpoint because it has the “write” authorization. The next code snippet presents the calls for the endpoint with our two users “emma” and “natalie”.

Calling the `/hello` endpoint and authenticating with user "emma":

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

The response body is:

```
Hello, Fantastico
```

Calling the `/hello` endpoint and authenticating with user "natalie":

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

The response body is:

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

Similarly, you could use any other expression we've discussed in chapter 7 for endpoint authentication. Below a short recap of them:

- `hasAnyAuthority()` – to specify multiple authorities. The user must have at least one of these authorities to call the method.
- `hasRole()` – to specify a role a user must have to be allowed to call the method.
- `hasAnyRole()` – to specify multiple roles. The user must have at least one of these roles to call the method.

Let's extend our example now also to prove how you can use the values of the method parameters to define the authorization rules. You find this example in the project named `ssia-ch16-ex2`.

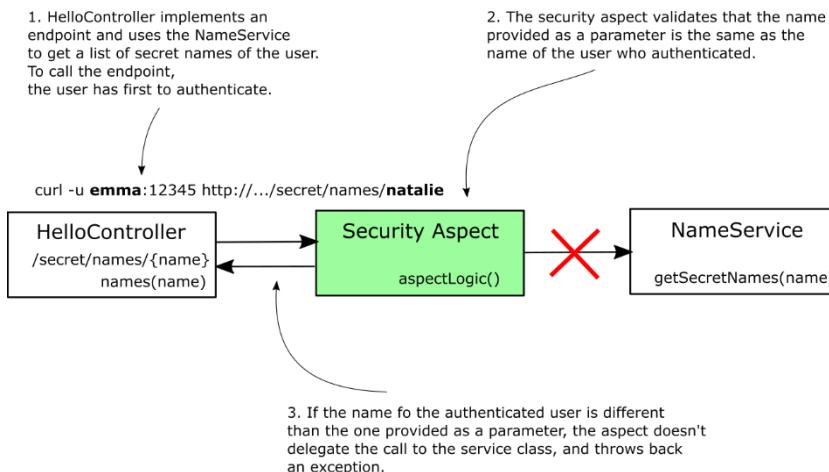


Figure 16.6 When implementing pre-authorization, we can use the values of the method parameters in the

authorization rules. In our example, only the authenticated user can retrieve information about their secret names.

In this project, I defined the same `ProjectConfig` class as our first example to 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 user name. Of course, in this case, the “secret names” are just an invention of mine referring to a characteristic of the user, which is not open for everyone. Precisely, I defined the controller class as presented in listing 16.5.

#### **Listing 16.5 The controller class defines an endpoint for test**

```
@RestController
public class HelloController {

    @Autowired #A
    private NameService nameService;

    @GetMapping("/secret/names/{name}") #B
    public List<String> names(@PathVariable String name) {
        return nameService.getSecretNames(name); #C
    }
}
```

#A Injecting from the context an instance of the service class that defines the protected method.

#B Defining an endpoint that takes a value from a path variable.

#C Calling the protected method to obtain the secret names of the user.

Now let’s take a look at how to implement the `NameService` class. You find it in listing 16.6. The expression we now use for authorization is `#name == authentication.principal.username`. In this expression, we use `#name` to refer to the value of the method parameter called “name”, and we have access directly to the authentication object which we use to refer to the currently authenticated user. The expression we used 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 16.6 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);
    }
}
```

#A We use the #name to represent the value of the method parameters in the authorization expression.

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"]
```

But if authenticating with the user Emma we try to get the secret names of Natalie, the call won'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 their secret names as presented in the next code snippet.

```
curl -u natalie:12345 http://localhost:8080/secret/names/natalie
```

The response body is:

```
["Energico","Perfecto"]
```

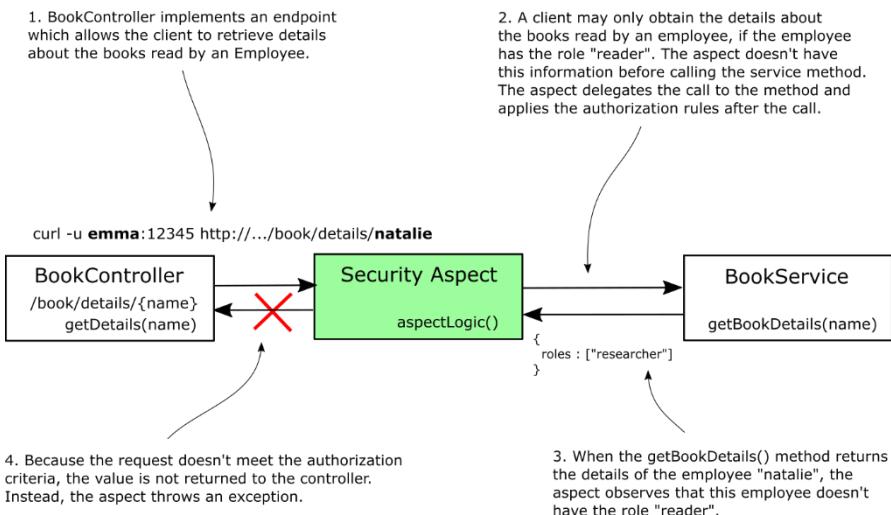
**NOTE** Remember, you can apply global method security to any layer of your application. In the examples presented in this chapter, you find the authorization rules applied for methods of the service classes. But you can apply authorization rules with global method security in any part of your application: repositories, managers, proxies, and so on.

## 16.3 Applying post-authorization

Say you want to allow the call to the method, but in certain circumstances, you want to make sure the caller of the method doesn't receive the returned value. Whenever we want to apply an authorization rule that is verified after the call of the method, we use post-authorization. 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, here's not about the method itself, but imagine this method retrieves some data from a data source, say a web service or a database. You can be confident about what your method does, but you can't bet on the third-party your method calls. So you allow the method to execute, but you validate in the end what it returns. And if that's not right, you don't let the value returned to be accessed by the caller.

To apply the post-authorization rules with Spring Security, we use the `@PostAuthorize` annotation. The `@PostAuthorize` annotation is very similar to `@PreAuthorize`, which we discussed in section 16.2. The annotation receives as value the SpEL defining the authorization rule. We'll continue with an example in which you'll learn how to use the `@PostAuthorize` annotation and define post-authorization rules for a method.

The scenario of our example for which I create a project named `ssia-ch16-ex3` defines an object `Employee`. Our `Employee` has a name, a list of books, and a list of roles. We associate each `Employee` to a user of the application. To keep consistent with the other examples of this chapter, we'll 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 the role "reader". Because we don't know the roles 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'll use the `@PostAuthorize` annotation.



**Figure 16.7** With post-authorization, 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 configuration class is the same we used in the previous examples, but for your convenience, I repeat it in listing 16.7.

#### Listing 16.7 The configuration class enables global method security and defines the users

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
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. Listing 16.8 shows the definition of the `Employee` class.

#### **Listing 16.8 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
}

```

Usually, we'd probably get the employee details from a database. To keep our example shorter, I'll just use a `Map` with a couple of records, which we'll consider our data source. In listing 16.9, 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 used with the `@PostAuthorize` annotation refers to the value returned by the method - `returnObject`. The post-authorization expression can use the value returned by the method, which is available after the method executes.

#### **Listing 16.9 The BookService class defines 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);
}
}

```

#A The @PostAuthorize annotation defines the expression for post-authorization.

Let's also write a controller and implement an endpoint to call the method for which we applied the authorization rule. Listing 16.10 presents this controller class.

#### **Listing 16.10 The controller class implements the endpoint**

```

@RestController
public class BookController {

    @Autowired
    private BookService bookService;

    @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 snippet, 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":

```
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":

```
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”:

```
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”:

```
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 request to have both pre-authorization and post-authorization.

## 16.4 Implementing permissions for methods

Up to now, you learned how to define rules with simple expressions for pre-authorization and post-authorization. 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 a hard to read code, and by this affects the app’s maintainability. When you need to implement complex authorization rules, instead of writing long SpEL expression, take the logic out in a separate class. Spring Security provides the concept of permission, which makes it easy to write the authorization rule in a separate class to make it easier to read and understand.

In this section, we’ll apply authorization rules using permissions within a project. I’ll name this project `ssia-ch16-ex4`. The scenario is the following: 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’ll implement a permission evaluator to solve this requirement. The `Document`, which is only a plain Java object, is presented in listing 16.11.

### Listing 16.11 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'll create a repository class that manages a few document instances in a `Map`. You find this class in listing 16.12.

#### **Listing 16.12 The DocumentRepository class manages 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 Each document is identified by a unique code and has an owner.

#B A document can be obtained using its unique identification code.

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'll 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 `hasPermission()` SpEL expression. This method allows us to refer to an external authorization expression that we'll 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 the access – "ROLE\_admin". You find the definition of this class in listing 16.13.

#### **Listing 16.13 The DocumentService class implements the protected method**

```
@Service
public class DocumentService {

    @Autowired
    private DocumentRepository documentRepository;

    @PostAuthorize  #A
    ("hasPermission(returnObject, 'ROLE_admin')")
    public Document getDocument(String code) {
        return documentRepository.findDocument(code);
    }
}
```

#A Using the `hasPermission()` expression to refer to an authorization expression.

But 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 offers two ways to implement the permission logic:

- By object and permission – this is the one we use in the current example. It assumes the permission evaluator receives an object which is subject to the authorization rule and an object that offers extra details that are needed in implementing the permission logic.
- By object ID, object type, and permission – This permission implementation assumes the permission evaluator receives an object ID, which it may use to retrieve the needed object. Besides, it also receives a type of object – which might be used if the same permission evaluator applies to multiple types of object and an object offering extra details needed for evaluating the permission.

In listing 16.14, you find the `PermissionEvaluator` contract with its two methods.

#### **Listing 16.14 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'll also send the role name "ROLE\_admin", which, as defined by the example's scenario, may access any document. Of course, in our example, we could have directly used the name of the role in the permission evaluator class and avoid sending it as a value of the `hasPermission()` object. Here, we only do this for the sake of the example, but in a real-world scenario, which might be more complex, you'd have multiple methods, and details needed in the authorization process might differ between each of them. For this reason, you have this parameter, so you can send from the method level the needed details to be used in the authorization logic. I'd also like to mention here, for your awareness and to avoid confusion, that you don't have to pass the `Authentication` object. Spring Security will automatically provide this parameter value when calling the `hasPermission()` method. The framework knows the value of the authentication instance as, at this time, it is already in the `SecurityContext`.

In listing 16.15, you find the `DocumentsPermissionEvaluator` class, which, in our example, implements the `PermissionEvaluator` contract to define the custom authorization rule.

**Listing 16.15 The DocumentsPermissionEvaluator class implements 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
    }
}

```

#A Implementing the PermissionEvaluator contract.

#B Casting the target object to Document.

#C The permission object in our case is the role name, so we cast it to a String.

#D We check if the authentication user has the role we got as a parameter.

#E If admin or the authenticated user is the owner of the document, the permission is granted.

#F We don't need to implement the second method as we don't use it.

To make Spring Security aware of our new PermissionEvaluator implementation, we have to define a MethodSecurityExpressionHandler in the configuration class. Listing 16.16 shows you how to define a MethodSecurityExpressionHandler to make the custom PermissionEvaluator known.

**Listing 16.16 Configuring the PermissionEvaluator in the configuration class**

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig
    extends GlobalMethodSecurityConfiguration {

    @Autowired

```

```

private DocumentsPermissionEvaluator evaluator;

@Override    #A
protected MethodSecurityExpressionHandler createExpressionHandler() {
    var expressionHandler =    #B
        new DefaultMethodSecurityExpressionHandler();

    expressionHandler.setPermissionEvaluator(
        evaluator);    #C

    return expressionHandler;    #D
}

// Omitted definition of the UserDetailsService and PasswordEncoder beans
}

```

#A Overriding the `createExpressionHandler()` method.

#B Defining a default security expression handler to set up the custom permission evaluator.

#C Setting up the custom permission evaluator.

#D Returning the custom expression handler.

**NOTE** We use here an implementation for `MethodSecurityExpressionHandler` named `DefaultMethodSecurityExpressionHandler` which is provided by Spring Security. You could as well implement a custom `MethodSecurityExpressionHandler` to define custom SpEL expressions you'd use to apply the authorization rules. You'll very 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, but I'd like to make you aware that this is possible.

I separated the definition of the `UserDetailsService` and `PasswordEncoder` to let you focus only on the new code. In listing 16.17, 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. The user emma is a manager and can only access her documents.

### Listing 16.17 The full definition of the configuration class

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig
    extends GlobalMethodSecurityConfiguration {

    @Autowired
    private DocumentsPermissionEvaluator evaluator;

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        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 as presented in listing 16.18.

#### **Listing 16.18 Defining the controller class and implementing an endpoint**

```

@RestController
public class DocumentController {

    @Autowired
    private DocumentService documentService;

    @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 the document's 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":

```
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":

```
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":

```
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":

```
curl -u emma:12345 http://localhost:8080/documents/abc123
```

The response body is:

```
{
  "status": 403,
  "error": "Forbidden",
  "message": "Forbidden",
  "path": "/documents/abc123"
}
```

In a very similar manner, you could use the second method `PermissionEvaluator` offers to write your authorization expression. The second method refers to using an identifier and type of the subject instead of the object itself. Say we want to change the current example to apply the authorization rules before the method is executed using `@PreAuthorize`. In this case, you don't have the returned object yet. But instead of having the object itself, we have the code of the document, which is its unique identifier. Listing 16.19 shows you how to change the permission evaluator class to implement this scenario. I've separated the examples in a project named `ssia-ch16-ex5`, which you can run individually.

#### **Listing 16.19 Changes in the `DocumentsPermissionEvaluator` class**

```
@Component
public class DocumentsPermissionEvaluator
    implements PermissionEvaluator {

    @Autowired
    private DocumentRepository documentRepository;

    @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());
}
}

```

#A We no longer define the authorization rules through the first method.

#B Instead of having the object, we have its ID, and we get the object using the ID.

#C We check if the user is an admin.

#D If the user is admin or it's the owner of the document, the user can access the document.

Of course, we also need to use the proper call to the permission evaluator with the @PreAuthorize annotation. In listing 16.20, you find the change I made in the DocumentService class to apply the authorization rules with the new method.

#### **Listing 16.20 The DocumentService class**

```

@Service
public class DocumentService {

    @Autowired
    private DocumentRepository documentRepository;

    @PreAuthorize  #A
    ("hasPermission(#code, 'document', 'ROLE_admin')")
    public Document getDocument(String code) {
        return documentRepository.findDocument(code);
    }
}

```

#A Applying the pre-authorization rules by using the second method of the permission evaluator.

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":

```
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":

```
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":

```
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":

```
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 you can enable it using the `@EnableGlobalMethodSecurity` annotation over the configuration class. Moreover, you must specify a certain way to apply the authorization rules using an attribute of the `@EnableGlobalMethodSecurity` annotation. We used the annotations like this:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

The `prePostEnabled` attribute enabled the `@PreAuthorize` and `@PostAuthorize` annotations we used to specify the authorization rules. The `@EnableGlobalMethodSecurity` annotation offers two other similar attributes that you use to enable different annotations. You use the `jsr250Enabled` attribute to enable the

`@RolesAllowed` annotation and the `securedEnabled` attribute to enable the `@Secured` annotation. These two methods are way less powerful than using `@PreAuthorize` and `@PostAuthorize`, and the chances you'll find them in real-world scenarios are really low. I'd like to make you aware of them, but without spending too much time on the details.

You enable the use of these annotations the same way we've done for the pre-authorization and post-authorization: by setting to true the attributes of the `@EnableGlobalMethodSecurity`. You enable the attributes which represent the usage of one kind of annotation, either `@Secure` or `@RolesAllowed`. You find an example on how to do this in the next code snippet.

```
@EnableGlobalMethodSecurity(
    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 should have to call a certain method. The next code snippet shows you how to use the `@RolesAllowed` annotation to specify that only the users having the role admin can call the `getName()` method.

```
@Service
public class NameService {

    @RolesAllowed("ROLE_ADMIN")
    public String getName() {
        return "Fantastico";
    }
}
```

Very similar, you could use the `@Secured` annotation instead of the `@RolesAllowed` annotation, as presented in the next code snippet.

```
@Service
public class NameService {

    @Secured("ROLE_ADMIN")
    public String getName() {
        return "Fantastico";
    }
}
```

You can now test your example as presented by the next code snippet.  
`curl -u emma:12345 http://localhost:8080/hello`

The response body is:

Hello, Fantastico

Calling the endpoint and authenticating with the user "natalie":

`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 project `ssia-ch16-ex6`.

---

## 16.5 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 Global Method Security functionality.
- The Global Method Security functionality is disabled by default. To enable Global Method Security, you use the `@EnableGlobalMethodSecurity` annotation over the configuration class of your application.
- You can apply authorization rules that the application checks before the call of a method. If these authorization rules aren't followed, the framework doesn't allow the method execution. When we test the authorization rules before the call of the method, we say we do pre-authorization.
- To implement pre-authorization, you use the `@PreAuthorize` annotation with a value of a SpEL expression that defines the authorization rule.
- We say we do post-authorization if we only decide after the method call if the caller can use the returned value and if the execution flow may proceed.
- To implement post-authorization, we use the `@PostAuthorize` annotation with the value of a SpEL expression representing the authorization rule.
- When implementing complex authorization logic, you should separate this logic in another class to make your code easier to read. In Spring Security, a common way to do this is implementing a `PermissionEvaluator`.
- Spring Security offers compatibility with older specifications like the `@RolesAllowed` and `@Secured` annotations. You can use them, but they are way less powerful than `@PreAuthorize` and `@PostAuthorize`, and chances you'll find these annotations used with Spring in a real-world scenario are very low.

# 17

## *Global method security: Pre/post filtering*

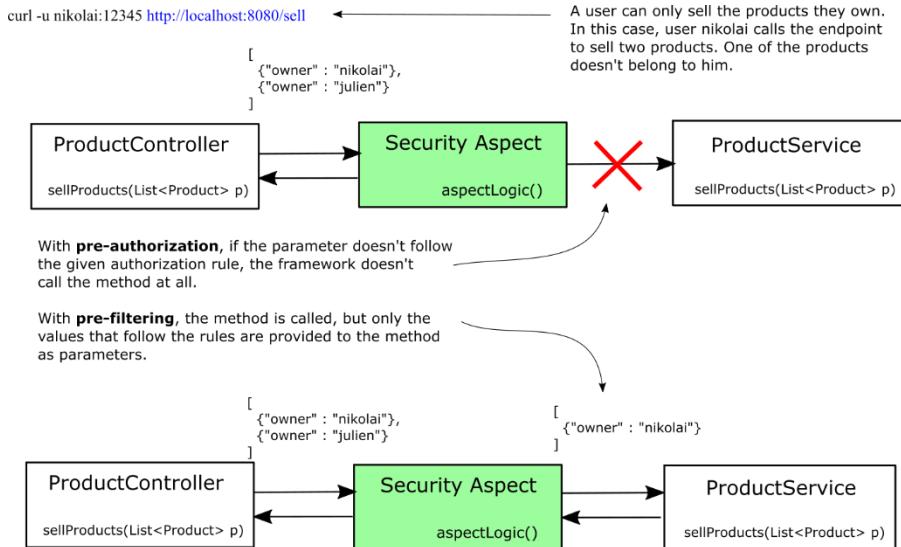
### This chapter covers

- Using pre-filtering to restrict what a method receives as parameter values.
- Using post-filtering to restrict what a method returns.
- Integrating filtering with Spring Data.

In chapter 16, you learned how to apply authorization rules using Global Method Security. We worked on examples using the `@PreAuthorize` and `@PostAuthorize` annotations. But using these annotations, you apply an approach in which the application either allows the method call or it completely rejects the call. Say you don't want to forbid the call to a method, but you want to make sure that the parameters sent to the method follow some rules. Or, in another scenario, you want to make sure that, after someone called 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 into two categories:

- **Pre-filtering** – when the framework filters the values of the parameters before calling the method.
- **Post-filtering** – when the framework filters the returned value after the method call.

Observe how filtering works differently than call authorization (figure 17.1). 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 the elements that don't follow the conditions you specify.

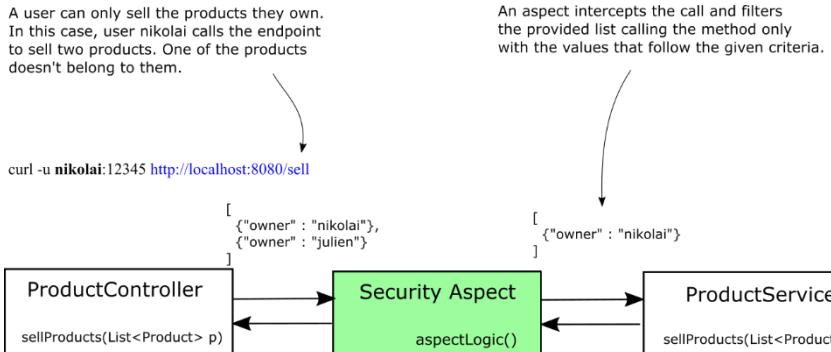


**Figure 17.1** The client calls the endpoint providing a value that doesn't follow the authorization rule. With pre-authorization, the method wouldn't be called at all, and the caller would receive an exception. With pre-filtering, the aspect calls the method, but only provides the values that follow the given rules.

It's important to mention from the beginning, however, that you can only apply filtering to collections and arrays. So you use pre-filtering only if the method receives as a parameter an array or a collection of objects. The framework will filter this collection or array according to rules you define. Same for post-filtering, 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.

## 17.1 Applying pre-filtering for method authorization

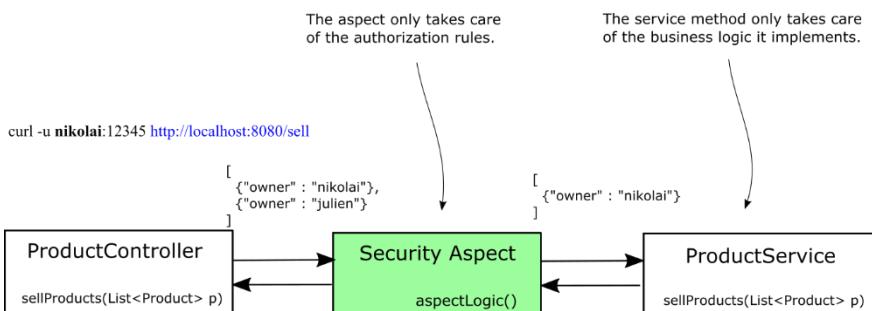
In this section, we discuss the mechanism behind pre-filtering, and then we implement pre-filtering in an example. You can use filtering to instruct the framework to validate the values sent via the method parameters when someone calls the method. The framework filters the values that don't match the given criteria and calls the method only with the values that match the criteria (figure 17.2). We name this functionality pre-filtering.



**Figure 17.2** With pre-filtering, an aspect intercepts the call to the protected method. The aspect filters the values which the caller provided as the parameter and provides the method only the values that follow the rules you define.

You'll find requirements in real-world examples where pre-filtering applies really well as it decouples the authorization rules from the business logic the method implements. Say you implement a use-case where you process only specific details that have to be owned by the authenticated user. This use-case may 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. So 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 the authorization logic from the business logic enhances the maintainability of your code and makes it easier for others to read and understand it.

Like in the case of call authorization we discussed in chapter 16, Spring Security also implements filtering by using aspects. Aspects intercept specific method calls and may augment the method call with other instructions. For pre-filtering, an aspect intercepts the methods annotated with the `@PreFilter` annotation and filters the values in the collection provided as a parameter according to the criteria you define (figure 17.3).



**Figure 17.3** With pre-filtering, we decouple the authorization responsibility from the business implementation. The aspect provided by Spring Security only takes care of the authorization rule, 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 16, you set the authorization rule as the value of the `@PreFilter` annotation. In these rules, which you provide as a SpEL expression, you use `filterObject` to refer to any element inside the collection or array you provide as a parameter to the method.

To see pre-filtering applied, let's start working on a project. I name this project `ssia-ch17-ex1`, and you find it in the projects provided with the book. Say you have an application used to buy and sell products. This application's backend implements an endpoint `/sell`. The application's frontend calls this endpoint when a user sells a product. But the logged-in user may only sell the products they own. Let's implement the very simple scenario of a service method called `to sell` the products received as a parameter. With this example, you'll learn how to apply the `@PreFilter` annotation, as this is the method we'll use to make sure that the method only receives products owned by the currently logged-in user.

Once we created 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 17.1. The configuration class that I called `ProjectConfig` only declares a `UserDetailsService` and a `PasswordEncoder`, and I annotated it with `@GlobalMethodSecurity(prePostEnabled=true)`. For using the filtering annotation, we still need to use the `@GlobalMethodSecurity` annotation and enable the pre/post-authorization annotations. The provided `UserDetailService` defines two users we'll need in our tests: `nikolai` and `julien`.

#### **Listing 17.1 The configuration class defines the users and enables Global Method Security**

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
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'll describe the product using a model class you find in listing 17.2.

### **Listing 17.2 The Product class definition**

```

public class Product {

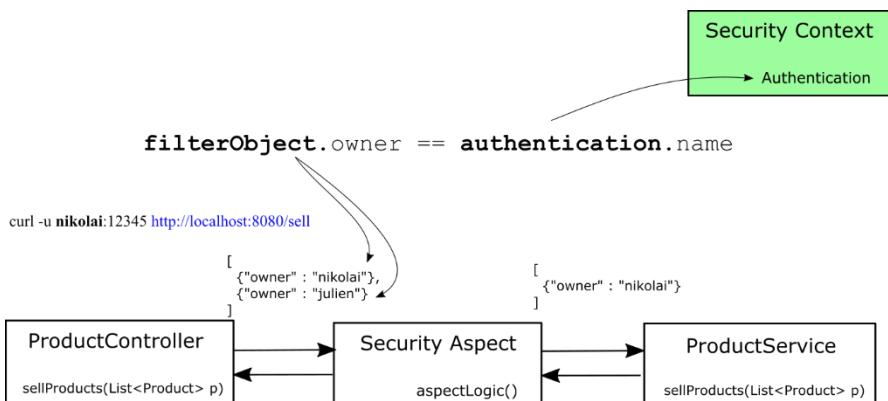
    private String name;
    private String owner;      #A

    // Omitted constructor, getters, and setters
}

```

#A Attribute owner has the value of the username.

The `ProductService` class defines the service method we protect with `@PreFilter`. You find the `ProductService` class presented in listing 17.3. Above the `sellProducts()` method, you observe the use of the `@PreFilter` annotation. The Spring Expression Language (SpEL) used with the annotation is `filterObject.owner == authentication.name` which allows in the list only the values where the `owner` attribute of the `Product` equals the username of the logged-in user. On the left side of the equal operator in the expression, we used `filterObject`. With `filterObject` we refer to objects in the list provided as a parameter. 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 attribute named `owner`. On the right side of the equal operator in the expression, we used the `authentication` object. For the `@PreFilter` and `@PostFilter` annotations, we can directly refer to the `authentication` object, which is available in the `SecurityContext` after the authentication (figure 17.4).



**Figure 17.4** When using pre-filtering, by `filterObject`, we refer to the objects inside the list, which the caller provided as a parameter. The `authentication` object is the one stored after the authentication process in the `SecurityContext`.

The service method returns, in the end, the list exactly as the method received. 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 17.3 Using the @PreFilter annotation in the ProductService class**

```
@Service
public class ProductService {

    @PreFilter[CA]      #A
        ("filterObject.owner == authentication.name")
    public List<Product> sellProducts(List<Product> products) {
        // sell products and return the sold products list
        return products;      #B
    }
}
```

#A In the list given as a parameter, only products owned by the authenticated user are allowed.

#B We return the products for test purposes.

To make our tests easier, I'll define an endpoint to call the protected service method. In listing 17.4, you find the definition of 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've been provided by the client in the request body. You may also observe I use `@GetMapping` for an operation which suggests a mutating operation, which is non-standard. But you know that I do like this to avoid dealing with the CSRF protection in our example and by this allow you to focus on the subject we discuss. You learned about CSRF protection in chapter 10.

### **Listing 17.4 The controller class implements the endpoint we use for tests**

```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @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 assigned two of the products to the user nikolai and another one to the user julien. When we call the endpoint and authenticate with the user nikolai we expect only to see the two products associated with nikolai. When we call the endpoint, and we authenticate with julien, we should

only find in the response the product associated with julien. In the following code snippet, you find the test calls and their results.

Calling the endpoint `/sell` and authenticating with the user "nikolai":

```
curl -u nikolai:12345 http://localhost:8080/sell
```

The response body is:

```
[  
  {"name":"beer","owner":"nikolai"},  
  {"name":"candy","owner":"nikolai"}  
]
```

Calling the endpoint `/sell` and authenticating with the user "julien":

```
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. So 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 will result in an exception at execution because the filtering aspect won't be able to change the collection's contents (figure 17.5).

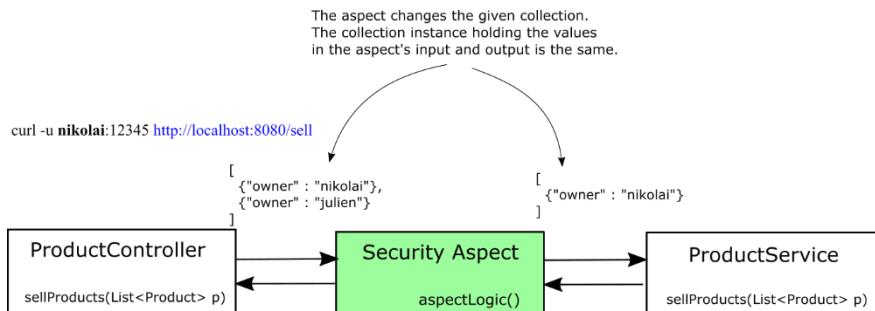


Figure 17.5 The aspect intercepts and changes the collection given as the parameter. You need to provide a mutable instance of a collection so the aspect can change it.

Listing 17.5 presents the same project we worked on in this section, but I've changed the `List` definition with an immutable instance as returned by the `List.of()` method to test what happens in this situation.

**Listing 17.5 Using an immutable collection**

```

@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @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);
    }
}

```

#A List.of() returns an immutable instance of list.

I have separated this example in project `ssia-ch17-ex2` so that you can test it yourself as well. Running the application and calling the `/sell` endpoint will result 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:

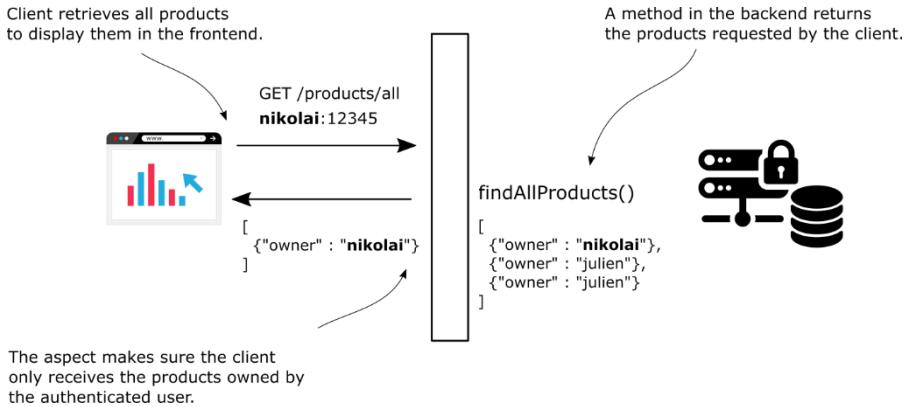
```
{
    "status":500,
    "error":"Internal Server Error",
    "message":"No message available",
    "path":"/sell"
}
```

In the application console, you'll find an exception similar to the one presented in the following code snippet.

```
java.lang.UnsupportedOperationException: null
    at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:73)
    ~[na:na]
...
```

## 17.2 Applying post-filtering for method authorization

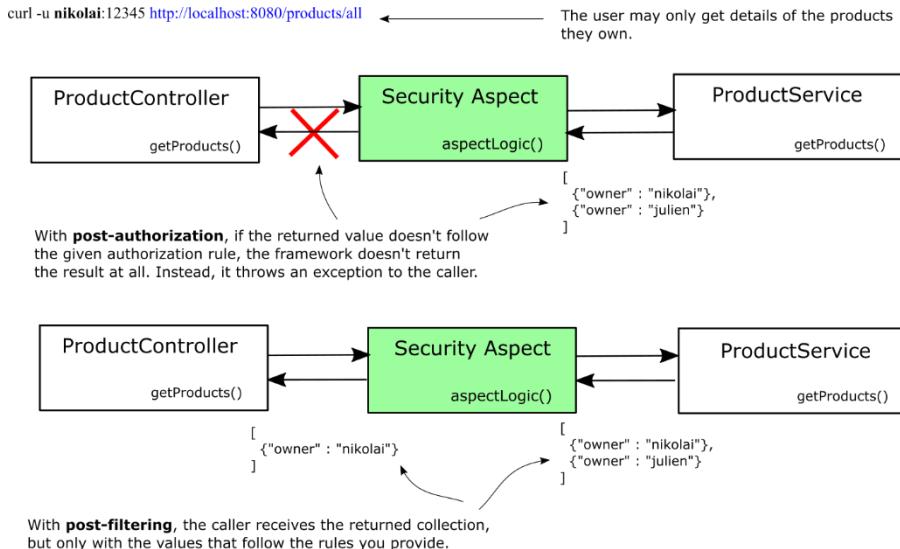
In this section, we implement post-filtering. Say we have the following scenario. Our application, which has a frontend implemented in Angular and a Spring-based backend, manages products. Users own products, and they can obtain details only for their products. To get the details of their products, the frontend calls endpoints exposed by the backend (figure 17.6).



**Figure 17.6 Post-filtering scenario:** A client calls an endpoint to retrieve the data it needs to display in the frontend. A post-filtering implementation makes sure that the client only gets the data owned by the currently authenticated user.

On the backend side, 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 named post-filtering. Further, in this section, we discuss how post-filtering works and demonstrate its implementation in an application.

Similar to pre-filtering, post-filtering also relies on an aspect. This aspect allows the call to the method, but once the method returns, the aspect takes the returned value and makes sure that it follows the rules you define. Like in the case of pre-filtering, post-filtering 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.



**Figure 17.7 Post-filtering:** An aspect intercepts the collection returned by the protected method and filters out the values that don't follow the rules you provide. Unlike post-authorization, post-filtering doesn't throw an exception to the caller when the returned value doesn't follow the authorization rules.

To apply post-filtering, you need to use the `@PostFilter` annotation. The `@PostFilter` annotation works very similar to all the other pre/post annotations we used in chapter 14 and this chapter. You provide the authorization rule as a SpEL for the annotation's value, and that rule is the one that'll be used by the filtering aspect. Also, similar to `@PreFilter`, the post-filtering only works with arrays and collections. Make sure you apply the `@PostFilter` annotation only for methods that have a return type array or a collection.

Let's apply post-filtering in an example. I'll create a project named `ssia-ch17-ex3` for this example. To be consistent, I'll keep the same users as in our previous examples of this chapter so the configuration class won't change. For your convenience, in listing 17.6, I repeat the configuration presented in listing 17.1.

### Listing 17.6 The configuration class

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
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 `Product` class remains unchanged as well, as presented in the next code snippet.

```

public class Product {

    private String name;
    private String owner;

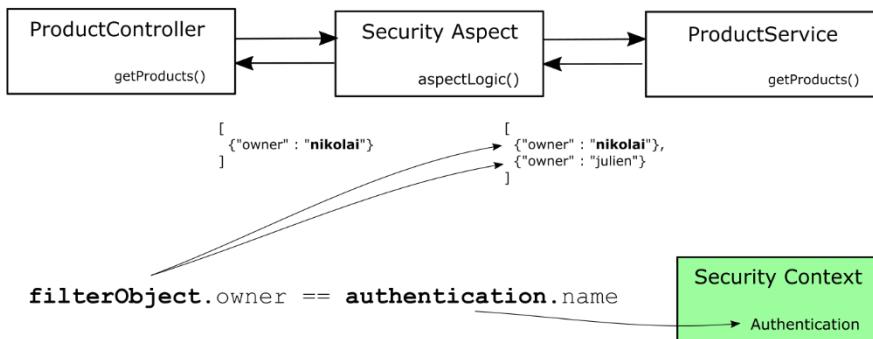
    // Omitted constructor, getters, and setters
}

```

But 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 discussed aspects, we'll use a simple collection, as presented in listing 17.7.

I annotated the `findProducts()` method, which returns the list of products, with the `@PostFilter` annotation. The condition I added as the value of the annotation, `filterObject.owner == authentication.name`, only allows products that have the owner equal to the authenticated user to be returned. On the left side of the equal 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`.

```
curl -u nikolai:12345 http://localhost:8080/products/all
```



**Figure 17.8** 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 17.7 The ProductService class**

```
@Service
public class ProductService {

    @PostFilter[CA]      #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;
    }
}
```

#A Adding the filtering condition of the objects in the collection returned by the method.

We define a controller class to make our method accessible through an endpoint. Listing 17.8 presents the controller class.

#### **Listing 17.8 The ProductController class**

```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @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 the products owned by the authenticated user. The next code snippet shows the result for calling the endpoint with each of our users nikolai and julien.

Calling the endpoint `/find` and authenticating with the user "julien":

```
curl -u julien:12345 http://localhost:8080/find
```

The response body is:

```
[{"name":"chocolate","owner":"julien"}]
```

Calling the endpoint `/find` and authenticating with the user "nikolai":

```
curl -u nikolai:12345 http://localhost:8080/find
```

The response body is:

```
[{"name":"beer","owner":"nikolai"}, {"name":"candy","owner":"nikolai"}]
```

## 17.3 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 very often use databases to persist the data of our applications. It is pretty common nowadays 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 them in examples. The first approach we take is the one you've learned up to now in the chapter: using the `@PreFilter` and `@PostFilter` annotations. The second approach we'll discuss is the direct integration of the authorization rules in queries. As you'll learn in this section, you have to be attentive when choosing the way you apply filtering at the Spring Data repositories.

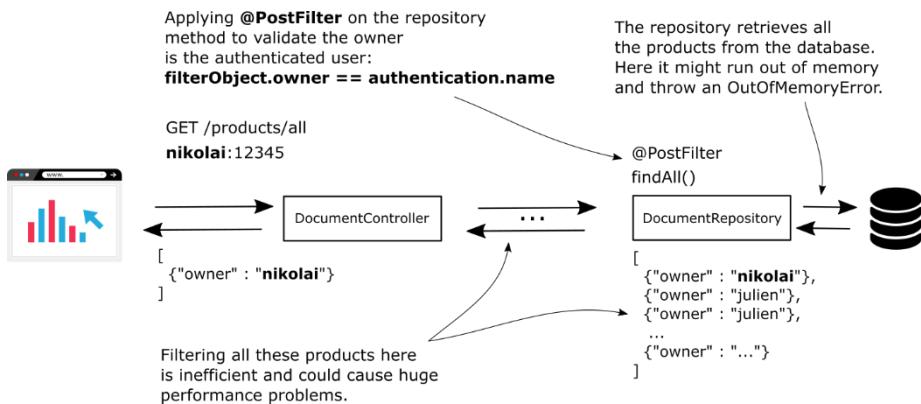
Let's start our discussion on using filtering with Spring Data. As mentioned, we have two options:

- Using `@PreFilter` and `@PostFilter` annotations.
- Directly applying filtering within the queries.

Using the `@PreFilter` annotation in the case of repositories is merely the same thing as applying this annotation at any other layer of your application. But when it comes to post-filtering, the situation changes. Using `@PostFilter` on repository methods technically works fine, but it's rarely a good choice for applying filtering from the 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. This approach is clearly wrong, as it allows the application to retrieve from the database all the records and then filter them 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 that big to fill up the heap, it's still less performant to filter the records in your application rather than retrieving from the beginning only what you need from the database (figure 17.9).



**Figure 17.9** When you know the filtering conditions from the repository level, it's better to retrieve only the data you need. Otherwise, your application could face heavy memory and performance issues.

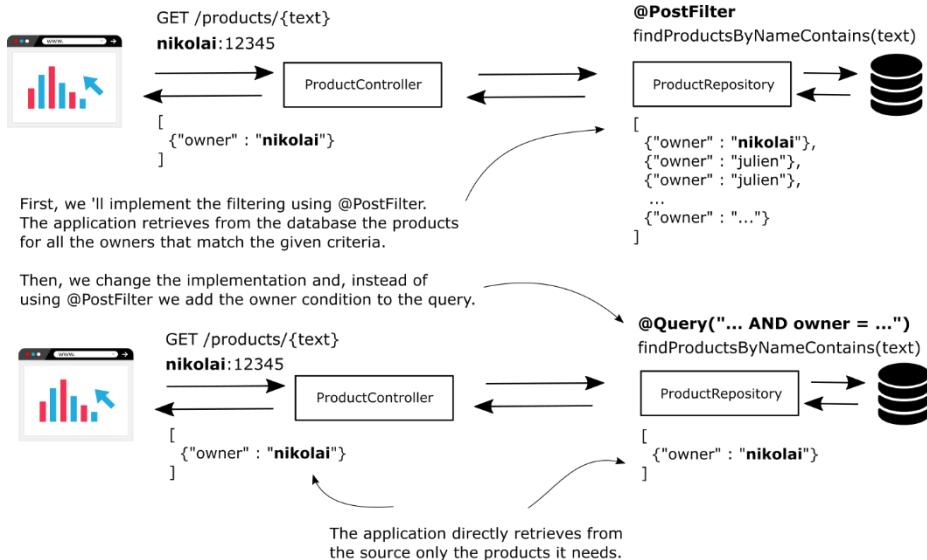
At the service level, you have no other option than to filter them in the app. Still, if you know from the repository level that you need to retrieve only the records owned by the logged-in user, you should implement a query that extracts from the database only the required documents.

**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 the 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 it to the second approach where we write the condition directly in the query. This way, we'll have the opportunity to experiment with both approaches and compare them.

I'll create a new project named `ssia-ch17-ex4`, where I'll use the same configuration class as for our previous three examples of this chapter. Like in the earlier examples of this chapter, we'll write an application managing products, but this time we retrieve the product details from a table in our database. For our example, we'll implement a search functionality for the products. We'll 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 only to return the products associated with the authenticated user.



**Figure 17.10** In our scenario, we start by implementing the application using `@PostFilter` to filter the 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` and a connection driver according to your database management server technology. In the next code snippet, you find 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've 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 products' details our application will retrieve. 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. To be found and executed at the start of the application, you need to place both files (`schema.sql` and `data.sql`) in the `resources` folder of the Spring Boot project.

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'll write three insert statements as presented in the next code snippet. They'll create the test data we'll need later to prove the application's behavior.

```
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('1', 'beer',
  'nikolai');
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('2', 'candy',
  'nikolai');
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('3', 'chocolate',
  'julien');
```

**NOTE** Remember, we have 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 would probably like to drop them before starting with this project. An alternative is also using a different schema.

To map the product table in our application, we need to write an entity class. Listing 17.9 contains the definition of the `Product` entity.

### Listing 17.9 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 as presented in listing 17.10. Observe this time we use the `@PostFilter` annotation directly on the method declared by the repository interface.

#### **Listing 17.10 The ProductRepository interface**

```
public interface ProductRepository
    extends JpaRepository<Product, Integer> {

    @PostFilter("filterObject.owner == authentication.name")
    List<Product> findProductByNameContains(String text);
}
```

#A Using the `@PostFilter` annotation for the method declared by the Spring Data repository.

Listing 17.11 shows you how to define a controller class which implements the endpoint we use for testing the behavior.

#### **Listing 17.11 The ProductController class**

```
@RestController
public class ProductController {

    @Autowired
    private ProductRepository productRepository;

    @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 “c” while authenticating with user `nikolai`, the HTTP response only contains product “candy”. Even if “chocolate” contains “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 snippet.

Calling the endpoint `/products` and authenticating with the user “`nikolai`”:

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is:

```
[{"id":2,"name":"candy","owner":"nikolai"}]
```

Calling the endpoint `/products` and authenticating with the user “`julien`”:

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is:

```
[{"id":3,"name":"chocolate","owner":"julien"}]
```

But 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 at all from the database what we don't need. So how can we change our example to directly select the required data instead of filtering it after selection? We may use SpEL expressions directly in the queries used by the repositories 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 repositories classes with the proper clauses for selection.

In our project, to add the `SecurityEvaluationContextExtension` bean in the context, the configuration class will change as presented in listing 17.12. To keep all the code associated with the examples of the book, I have used here another project I called `ssia-ch17-ex5`.

#### **Listing 17.12 Adding the `SecurityEvaluationContextExtension` to the context.**

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {

    @Bean      #A
    public SecurityEvaluationContextExtension
        securityEvaluationContextExtension() {

        return new SecurityEvaluationContextExtension();
    }

    // Omitted declaration of the UserDetailsService and PasswordEncoder
}
```

#A Adding a `SecurityEvaluationContextExtension` to the Spring context.

In the `ProductRepository` interface, we add the query above the method, and we adjust the `where` clause with the proper condition, using a SpEL expression. Listing 17.13 presents the change.

#### **Listing 17.13 Using SpEL in the query in the repository interface**

```
public interface ProductRepository
    extends JpaRepository<Product, Integer> {

    @Query("SELECT p FROM Product p[CA]
        WHERE p.name LIKE %:text% AND[CA]      #A
            p.owner=?#{authentication.name}")
    List<Product> findProductByNameContains(String text);
}
```

#A Using SpEL in the query to add a condition on the owner of the record.

We can now start the application and test it by calling the `/products/{text}` endpoint. We expect that the behavior remained the same as for the case where we were using `@PostFilter`. But now, the records for the right owner are directly retrieved from the database, which makes the functionality faster and more reliable. The next code snippet presents the calls to the endpoint.

Calling the endpoint `/products` and authenticating with the user "nikolai":

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is:

```
[{"id":2,"name":"candy","owner":"nikolai"}]
```

Calling the endpoint `/products` and authenticating with the user "julien":

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is:

```
[{"id":3,"name":"chocolate","owner":"julien"}]
```

## 17.4 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 the 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 out from the collection given as parameter all the 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 using `@PostFilter` with methods of Spring Data repositories.

# 18

## *Hands-on: An OAuth 2 application*

### This chapter covers

- Configuring Keycloak as an Authorization Server in your OAuth 2 system.
- Using Global Method Security in an OAuth 2 Resource Server.

In chapters 12 to 15, we discussed in detail how an OAuth 2 system works and how you implement it with Spring Security. We then changed the subject, and in chapters 16 and 17, you learned how to apply authorization rules at any layer of your application using Global Method Security. In this chapter, we'll combine these two essential subjects, and we'll apply Global Method Security within an OAuth 2 Resource Server.

Besides defining authorization rules at different layers of our Resource Server implementation, you'll also learn how to use a tool named Keycloak as the Authorization Server for your system. The example we'll work on this chapter will be helpful for you for the following reasons:

- Systems often use third-party tools such as Keycloak in real-world implementations to define an abstraction layer for authentication. There's a good chance you'll need to use Keycloak or a similar third-party tool in your OAuth 2 implementation. You find many possible alternatives to Keycloak like Okta, Auth0, or LoginRadius. This chapter focuses on a scenario in which you'll need to use such a tool in the system you develop.
- In real-world scenarios, we use authorization applied not only for the endpoints but also at other layers of the application. And this also happens for an OAuth 2 system.
- Putting once again into an example what you learned in chapters 12 to 17 helps you better understand the big picture of the technologies and approaches we discussed.

Let's dive into section 18.1 and find out the scenario of the application we'll implement in this hands-on chapter.

## 18.1 The application's scenario

Say we have to build the backend for a fitness application. Besides other great features, the app also stores a history of the users' workouts. In this chapter, we'll focus on the implementation of the part of the application storing the history of workouts. We presume our backend needs to implement three use-cases. For each action defined by the use-cases, we have specific security restrictions as described:

- **Use case:** Add a new workout record for a user – In a database table named “workout”, add a new record storing: the user, the start and the end time of the workout, and the difficulty of the workout using an integer on a scale from 1 to 5.

**Authorization restriction:** The authenticated users may only add workout records for themselves. The client calls an endpoint exposed by the Resource Server to add a new workout record.

- **Use case:** Find all the workouts for a user – The client needs to display a list for the workouts in the user's history. The client calls an endpoint to retrieve the list of workouts.

**Authorization restriction:** A user may only get their own workout records.

- **Use case:** Delete a workout – Any user having the admin role may delete a workout for any other user. The client calls an endpoint to delete a workout record.

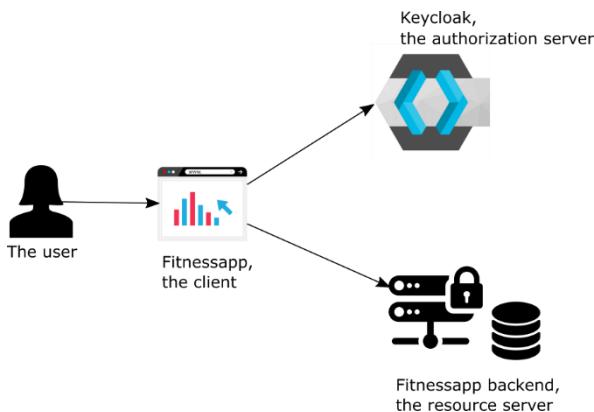
**Authorization restriction:** Only an admin may delete records.

We need to implement three use cases for which we have two roles acting. The two roles are the standard user, which we'll name `fitnessuser`, and the admin, which we'll name `fitnessadmin`. A `fitnessuser` can add a workout for themselves and may see their own workout history. The `fitnessadmin` may only delete workout records for any of the users. Of course, the admin may also be a user, and in this case, they can also add workouts for themselves or see their recorded workouts.



**Figure 18.1** Whether it's the workout history or the bank account, an application needs to implement proper authorization rules to protect the user's data from theft or unwanted changes.

The backend that we'll implement with these three use-cases is an OAuth 2 Resource Server. We'll need an Authorization Server as well. For this example, we'll use a tool named Keycloak to configure the Authorization Server for the system. Keycloak offers us all the possibilities to set our users either locally, or by integrating with other user management services.

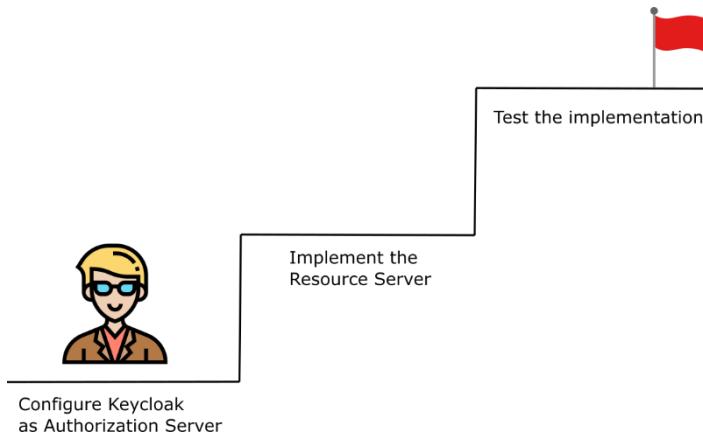


**Figure 18.2** The actors in the system are the user, the client, the Authorization Server, and the Resource Server. We'll use Keycloak to configure the Authorization Server, and we'll implement the Resource Server using Spring Security.

We'll start the implementations by configuring a local Keycloak instance as our Authorization Server. We then implement the Resource Server and set up the authorization rules using Spring Security. Once we have a working application, we test it by calling the endpoint with `curl`.

## 18.2 Configuring Keycloak as an Authorization Server

In this section, we start to configure Keycloak as the Authorization Server for the system. Keycloak is an excellent tool designed for open source identity and access management. You download Keycloak from [keycloak.org](https://keycloak.org). Keycloak offers us the possibility to manage simple users locally but also advanced features as user federation. You could connect it to your LDAP and Active Directory services, or different identity providers. For example, you could use Keycloak as a high-level authentication layer by connecting it to one of the common OAuth 2 providers we discussed in chapter 12.



**Figure 18.3** As part of the hands-on application we implement in this chapter, we'll follow three major steps. In this section, we configure Keycloak as the Authorization Server for the system.

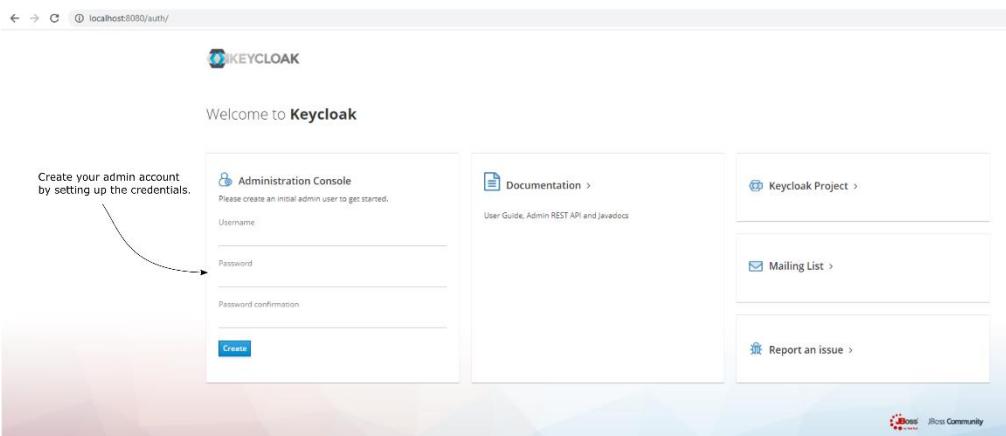
Keycloak's configuration is flexible, though it can become complex, depending on what you'd like to achieve. For this chapter, we'll discuss only the setup we need to do for our example. The setup we'll make only defines a few users with their roles. But Keycloak can do much more than this. If you plan to use Keycloak in real-world scenarios, I recommend you first read the detailed official documentation at their official website: <https://www.keycloak.org/documentation>. In chapter 9 of *Enterprise Java Microservices* by Ken Finnigan (Manning, 2018), you'll also find a good discussion on securing microservices where the author uses Keycloak for user management.

<https://livebook.manning.com/book/enterprise-java-microservices/chapter-9>

If you enjoy a discussion on microservices, I recommend you read Ken Finnigan's entire book. The author makes excellent statements on subjects anyone implementing microservices with Java should know.

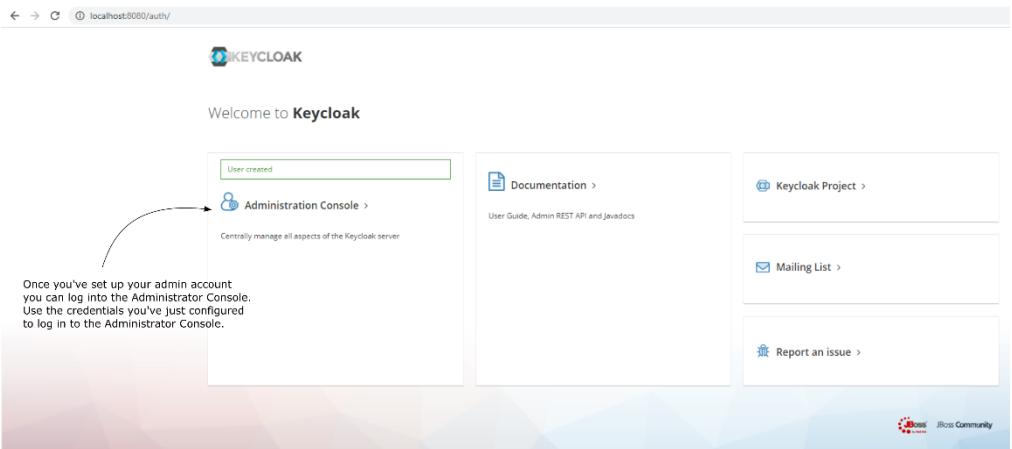
To install Keycloak, you only need to download an archive containing the latest version from the official website: <https://www.keycloak.org/downloads>. Then, you unzip the archive in a folder, and you can already start Keycloak. To start Keycloak, you use the standalone executable file, which you'll find in the bin folder. In case you're using Linux, you need to run standalone.sh. For Windows, you'll run standalone.bat.

Once you've started the Keycloak server, you access it in a browser at <http://localhost:8080>. In this first page, you configure an admin account by entering a username and a password (figure 18.4).



**Figure 18.4** To manage Keycloak, you first need to set up your admin credentials. You do this by accessing Keycloak first time when you start it.

Then you log in with these credentials you've set up to manage Keycloak as presented in figure 18.5.



**Figure 18.5** Once you've set up your admin account, you can log in to the Administration Console. To log in to the Administration console, you use the credentials you've just set up.

When in the Administration Console, you can start configuring the Authorization Server. We'll need to know which OAuth2-related-endpoints Keycloak exposes. You find those endpoints in the General section of the Realm Settings page, which is the first page you land into after logging in the Administration console (figure 18.6).

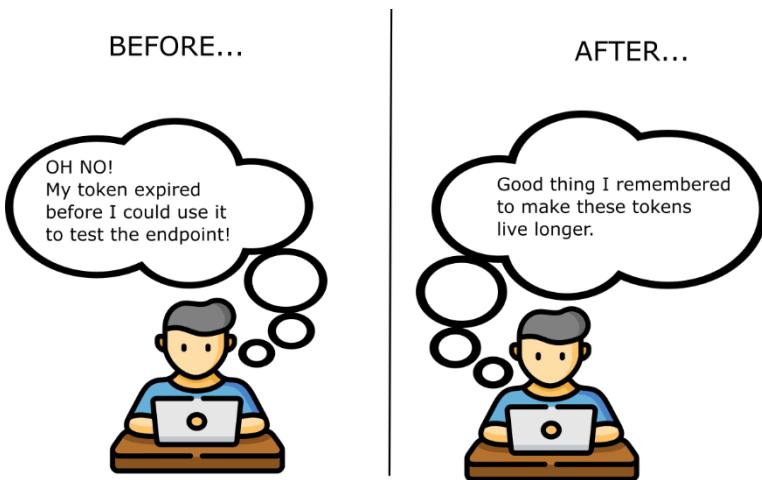
General		Login	Keys	Email	Themes	Cache	Tokens	Client Registration	Security Defenses
Name	master								
Display name	Keycloak								
HTML Display name	<div class="kc-logo-text"><span>Keycloak</span></div>								
Frontend URL									
Enabled	<input checked="" type="checkbox"/> ON	The OpenID Endpoint Configuration link offers you all the needed details regarding the OAuth 2 endpoints exposed by this Authorization Server.							
User-Managed Access	<input type="checkbox"/> OFF								
Endpoints	<a href="#">OpenID Endpoint Configuration</a>	SAML 2.0 Identity Provider Metadata							
	<a href="#">Save</a>	<a href="#">Cancel</a>							

**Figure 18.6** You find the endpoints related to the Authorization Server by clicking the OpenID Endpoint Configuration link. You'll need these endpoints to obtain the access token and for the Resource Server configuration.

In the next code snippet, I have extracted a part of the OAuth 2 configuration that you find by clicking the OpenID Endpoint Configuration link. You observe in this configuration the token endpoint, the authorization endpoint, and the list of supported grant types. These details should be familiar to you as we discussed them in detail in chapters 12 to 15.

```
{  
    "issuer":  
        "http://localhost:8080/auth/realm/master",  
  
    "authorization_endpoint":  
        "http://localhost:8080/auth/realm/master/[CA]  
protocol/openid-connect/auth",  
  
    "token_endpoint":  
        "http://localhost:8080/auth/realm/master/[CA]  
protocol/openid-connect/token",  
  
    "jwks_uri":  
        "http://localhost:8080/auth/realm/master/protocol/[CA]  
openid-connect/certs",  
  
    "grant_types_supported": [  
        "authorization_code",  
        "implicit",  
        "refresh_token",  
        "password",  
        "client_credentials"  
    ],  
    ...  
}
```

You'll find the testing of the app more comfortable if you have short-lived access tokens (figure 18.7). However, in a real-world scenario, remember not to give a very large lifespan for your tokens. For example, in a production system, a token should expire in a few minutes. But for testing, you can leave it active for one day.



**Figure 18.7** To test the application, we'll manually generate the access tokens, which we use to call the endpoints. If you define a short lifespan for the tokens, you'll need to generate the tokens more often, or you'll get annoyed by cases in which the token expires before you use it.

You can change the length of their life from the Tokens tab, as presented in figure 18.8.

The screenshot shows the Keycloak Admin Console interface for managing realms. The left sidebar shows navigation options like Master, Configure, Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication, Groups, Users, Sessions, Events, and Import. The main content area is titled 'Master' and shows the 'Tokens' tab selected. The 'Tokens' tab contains several configuration options:

- Default Signature Algorithm: A dropdown menu.
- Revoke Refresh Token: A switch set to 'OFF'.
- SSO Session Idle: Set to 30 Minutes.
- SSO Session Max: Set to 10 Hours.
- SSO Session Idle Remember Me: Set to 0 Minutes.
- SSO Session Max Remember Me: Set to 0 Minutes.
- Offline Session Idle: Set to 30 Days.
- Offline Session Max Limited: A switch set to 'OFF'.
- Access Token Lifespan: Set to 100 Minutes.
- Access Token Lifespan For Implicit Flow: Set to 15 Minutes.

Annotations explain the purpose of these settings:

- An annotation points to the 'Tokens' tab with the text: "In the Tokens tab you can change the configuration related to tokens issued by the Authorization Server."
- An annotation points to the 'Access Token Lifespan' field with the text: "You configure a longer life time of the access token here."

**Figure 18.8** You'll find testing more comfortable if an issued access token doesn't expire quickly. You can change its lifespan in the Tokens tab.

Now that we've installed Keycloak and set up the admin credentials, we can configure the Authorization Server. First, a list of steps we'll follow to accomplish this configuration.

1. *Register a client for the system* – an OAuth 2 system needs at least a client recognized by the Authorization Server, which will make the authentication requests for the users. In section 18.2.1, you'll learn how to add a new client registration.
2. *Define a client scope* – The client scope identifies the purpose of the client in the system. We'll use the definition of the client scope to customize the access tokens issued by the Authorization Server. In section 18.2.2, you'll learn how to add a client scope, and in section 18.2.4, we'll configure it to customize the access token.
3. *Add users for our application* – To call the endpoints on the Resource Server, we, of course, need users of our application. You'll learn how to add users managed by Keycloak in section 18.2.3.
4. *Define user roles and customizing the access token* – After adding the users, you can already issue access tokens for them. We'll observe that the access tokens don't have all the details we'll need to accomplish our scenario. In section 18.2.4, you'll learn how to configure roles for them and customize the access token to present the details expected by the Resource Server we'll implement using Spring Security.

### 18.2.1 Registering a client for our system

In this section, we discuss registering a client when using Keycloak as an Authorization Server. Like in any other OAuth 2 system, we need to register the client applications at the Authorization Server level. To add a new client, we'll use the Keycloak Administration Console. As presented in figure 18.9, by navigating, you find a list of clients to the Client tab from the left menu, and from here, you can also add a new client registration.

The screenshot shows the Keycloak Administration Console interface. The left sidebar has a 'Clients' tab selected. The main area displays a table of clients with columns: Client ID, Enabled, Base URL, Actions (Edit, Export, Delete). The table contains rows for 'account', 'account-console', 'admin-cli', 'broker', 'master-realm', and 'security-admin-console'. A tooltip is overlaid on the right side of the table, pointing to the 'Create' button in the top right corner of the table header. The tooltip text reads: 'To add a new client you use the Create button in the right upper corner of the clients table.'

Client ID	Enabled	Base URL	Actions
account	True	<a href="http://localhost:8080/auth/realm/master/account/">http://localhost:8080/auth/realm/master/account/</a>	Edit Export Delete
account-console	True	<a href="http://localhost:8080/auth/realm/master/account/">http://localhost:8080/auth/realm/master/account/</a>	Edit Export Delete
admin-cli	True	Not defined	Edit Export Delete
broker	True	Not defined	Edit Export Delete
master-realm	True	Not defined	Edit Export Delete
security-admin-console	True	<a href="http://localhost:8080/auth/admin/master/console/">http://localhost:8080/auth/admin/master/console/</a>	Edit Export Delete

Figure 18.9 To add a new client, you navigate to the clients list using the Clients tab in the left menu. Here you can add a new client registration by clicking the Create button in the right upper corner of the clients table.

I'll add a new client which I'll name `fitnessapp`. This client represents the application allowed to call endpoints from the Resource Server we'll implement later in this chapter, in section 18.3.

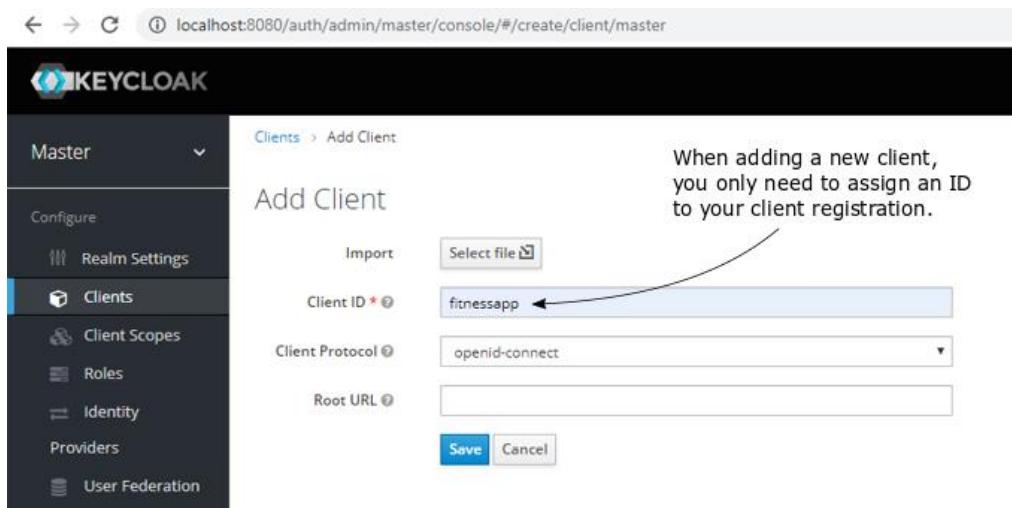


Figure 18.10 When adding the client, you only need to assign it a unique client ID and save. In my case, I named the client `fitnessapp`.

### 18.2.2 Specifying the client scopes

In this section, we'll define a scope for the client we registered in section 18.2.1. The client scope identifies the purpose of this client. We'll also use this client scope in section 18.2.4 to customize the access token issued by Keycloak. To add a scope to the client, we'll use the Keycloak Administration Console. As presented in figure 18.11, you find a list of client scopes navigating to the Client Scopes tab from the left menu. From here, you can also add a new client scope to the list.

You find a list of all the client scopes by navigating to the Client Scopes tab.

You add a new client scope by clicking the Create button.

Name	Protocol	GUI order	Actions
address	openid-connect		Edit Delete
email	openid-connect		Edit Delete
microprofile-jwt	openid-connect		Edit Delete
offline_access	openid-connect		Edit Delete
phone	openid-connect		Edit Delete
profile	openid-connect		Edit Delete
role_list	saml		Edit Delete

**Figure 18.11** For a list of all the client scopes, you navigate to the Client Scopes tab. Here, you add a new client scope by clicking on the Create button on the right upper corner of the client scopes table.

For the app that we're building, I'll add a new client scope named `fitnessapp`. When adding this new scope, also make sure the protocol for which you set the client scope is `openid-connect` (figure 18.12).

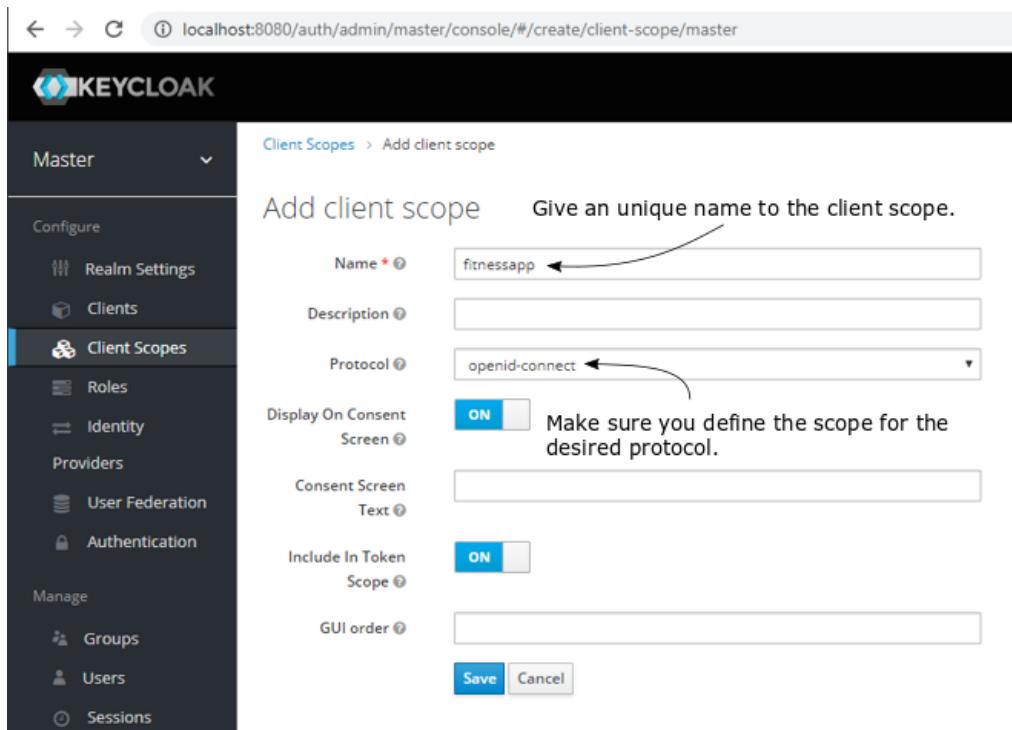


Figure 18.12 When adding a new client scope, give it a unique name, and make sure you define it for the desired protocol. In our case, the protocol we use is openid-connect.

**NOTE** The other protocol you can choose is SAML 2.0. Spring Security used to offer an extension for this protocol that you can still find at <https://projects.spring.io/spring-security-saml/#quick-start>. We don't discuss using SAML 2.0 in this book because it's not actively developed anymore for Spring Security. Also, SAML 2.0 is less frequently encountered than OAuth 2 in applications.

Once you have created the new role, you assign it to your client, as presented in figure 18.13. You get to this screen by navigating to the Client menu of the left and then selecting the Client Scopes tab.

The screenshot shows the Keycloak admin interface for managing clients. On the left, a sidebar menu is open under the 'Master' realm, showing options like 'Configure', 'Clients', 'Client Scopes', 'Roles', 'Identity', 'Providers', 'User Federation', and 'Authentication'. The 'Clients' option is selected. In the main content area, the URL is 'localhost:8080/auth/admin/master/console/#/realms/master/clients/4a698810-3966-449b-9005-1ca1466f440a/client-sco'. The page title is 'Fitnessapp' with a trash icon. The top navigation bar includes 'Settings', 'Roles', 'Client Scopes' (which is highlighted in blue), 'Installation', 'Mappers', 'Scope', 'Revocation', and 'Sessions'. A note on the right says: 'Once you've created the client scope, configure the scope to your client.' Below this, there are two sections: 'Default Client Scopes' and 'Optional Client Scopes'. Each section has a list of available scopes ('address', 'email', 'micropointer-jwt', 'offline\_access', 'phone') and a 'Add selected >' button. To the right of each list is a box labeled 'Assigned [Type] Client Scopes' containing the selected scope ('fitnessapp' for Default, empty for Optional). Below each assigned box is a 'Remove selected' button.

**Figure 18.13** Once you have a client scope, you assign it to a client. In this figure you can see I've already moved the scopes I need into the right box named “Assigned Default Client Scopes”. This way, you can now use the defined scope with the specific client to which you have assigned it.

### 18.2.3 Adding the users and obtaining access tokens

In this section, we create and configure the users for our application. Up to now, we have configured the client and its scope in sections 18.2.1 and 18.2.2. But besides the client app, we need users to authenticate and access the services offered by our Resource Server. We'll configure three users that we need to test our application at the end. I'll name the users Mary, Bill, and Rachel.

If the users aren't displayed up-front, press the View all users button.

To add a new user, click on the Add user button.

By navigating to the Users tab from the left menu you'll find a list of all the existing users.

**Figure 18.14** By navigating to the Users tab, you find a list of all the users. Here you can also add a new user by clicking the Add user button in the right upper corner of the users table.

When adding a new user, give it a unique username and check the box stating the email was verified. Also, make sure the user has no Required User Actions. While a user has Required User Actions pending, you cannot use it for authentication, thus, you cannot obtain an access token for that user.

The screenshot shows the Keycloak administration interface for adding a new user. The left sidebar is titled 'Master' and includes sections for Configure (Realm Settings, Clients, Client Scopes, Roles, Identity, Providers, User Federation, Authentication) and Manage (Groups, Users, Sessions, Events). The 'Users' option under 'Manage' is selected. The main page title is 'Users > Add user'. The 'Add user' form has the following fields:

- ID: A placeholder text area with the instruction "Give the user a unique username."
- Created At: A timestamp field.
- Username **\***: An input field containing "mary".
- Email: An empty input field.
- First Name: An empty input field.
- Last Name: An empty input field.
- User Enabled **ON**: A toggle switch.
- Email Verified **ON**: A toggle switch.
- Required User Actions **Select an action**: A dropdown menu.

Annotations with arrows point to the 'Username' field, the 'Email Verified' switch, and the 'Select an action' dropdown, each with their respective instructions.

Figure 18.15 When adding a new user, give the user a unique username and make sure it has no required actions pending.

Once you've created the users, you should find all of them in the users list, as presented in figure 18.16.

ID	Username	Email	Last Name	First Name	Actions		
ba82df00-cdfe-42...	bill				Edit	Impersonate	Delete
06df3d91-f4c1-4f0...	mary				Edit	Impersonate	Delete
c42b534f-7f08-45...	racel				Edit	Impersonate	Delete
56ebfb7-2f61-4f7...	root				Edit	Impersonate	Delete

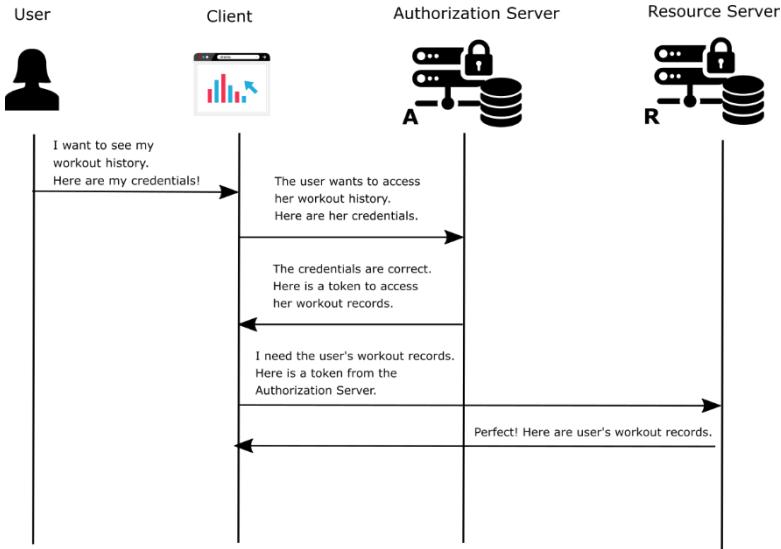
**Figure 18.16** The newly created users appear now in the users list. You can choose them from here for editing or deleting them.

Of course, the users also need passwords to login. Usually, they'd configure their own passwords, and the administrator shouldn't know their credentials. In our case, we have no other choice but to configure these passwords ourselves for the three users, as presented in figure 18.17. To keep our example simple, I'll configure the password 12345 for all the users. I'll also make sure that the password isn't temporary by unchecking the Temporary switch. If you make the password temporary, Keycloak will automatically add a required action for the user to change the password at their first login. Because of this required action, we wouldn't be able to authenticate with the user.

The screenshot shows the Keycloak Admin Console interface. On the left, there's a sidebar with navigation links like 'Master', 'Configure', 'Realm Settings', 'Clients', 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', 'Authentication', 'Manage Groups', and 'Users'. The 'Users' link is highlighted. In the main area, the URL is 'localhost:8080/auth/admin/master/console/#/realms/master/users/ba82df00-cdfe-42af-88f2-d56bb65993b/user-credentials'. The page title is 'Bill'. Below the title, there are tabs: 'Details', 'Attributes', 'Credentials' (which is selected), 'Role Mappings', 'Groups', 'Consents', and 'Sessions'. Under the 'Credentials' tab, there's a section titled 'Manage Credentials' with a table header: 'Position', 'Type', 'User Label', 'Data', and 'Actions'. Below this, there's a 'Set Password' section. It has three input fields: 'Password' (containing '12345'), 'Password Confirmation' (also containing '12345'), and a 'Temporary' checkbox which is checked and labeled 'OFF'. A note to the right of the checkbox says: 'When setting a password for the user make sure to uncheck the Temporary checkbox.' At the bottom of the 'Set Password' section is a 'Set Password' button.

**Figure 18.17** You can select a user from the list to change or configure its credentials. Before saving the changes, remember to make sure you set the “Temporary” switch to off. If the credentials are temporary, you wouldn’t be able to authenticate with the user upfront.

Having the users configured, you can now obtain an access token from your Authorization Server implemented with Keycloak. The next code snippet shows you how to obtain the token using the password grant type. To make our example simple, we'll use the password grant type. However, as you observed from section 18.2.1, Keycloak also supports the other grant types discussed by us in chapter 12. Figure 18.18 is a refresher for the password grant type we discussed in chapter 12.



**Figure 18.18** When using the password grant type, the user shares their credentials with the client. The client uses the credentials to obtain an access token from the Authorization Server. With the token, the client can then access the user's resources exposed by the Resource Server.

To obtain the access token, you call the `/token` endpoint of the Authorization Server as presented in the next code snippet.

```
curl -XPOST "http://localhost:8080/auth/realms/master/protocol/openid-connect/token" \
-H "Content-Type: application/x-www-form-urlencoded" \
--data-urlencode "grant_type=password" \
--data-urlencode "username=rachel" \
--data-urlencode "password=12345" \
--data-urlencode "scope=fitnessapp" \
--data-urlencode "client_id=fitnessapp"
```

You will receive the access token in the body of the HTTP response, as presented in the next code snippet.

```
{
  "access_token": "eyJhbGciOiJIUzI...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUz... ",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "1f4ddae7-7fe0-407e-8314-a8e7fc34d1b",
  "scope": "fitnessapp"
}
```

In the presented HTTP response, I have truncated the JWT tokens because they are long. The next code snippet presents the decoded JSON body of the JWT access token. Taking a short look at the code snippet, you observe that the token doesn't contain all the details we'd

need to make our application work. The roles and the username are missing. In section 18.2.4, you'll learn how to assign roles to our users and customize the JWT to contain all the data the Resource Server needs.

```
{
  "exp": 1585392296,
  "iat": 1585386296,
  "jti": "01117f5c-360c-40fa-936b-763d446c7873",
  "iss": "http://localhost:8080/auth/realms/master",
  "sub": "c42b534f-7f08-4505-8958-59ea65fb3b47",
  "typ": "Bearer",
  "azp": "fitnessapp",
  "session_state": "fce70fc0-e93c-42aa-8ebc-1aac9a0dba31",
  "acr": "1",
  "scope": "fitnessapp"
}
```

#### 18.2.4 Defining the users' roles

In section 18.2.3, we managed to obtain an access token. Previously we have added a client registration and configured the users to be able to obtain tokens. But still, the token doesn't have all the details our Resource Server needs to apply the authorization rules. To be able to write a complete app for our scenario, we need to add roles for our users.

Adding roles to a user is simple. You can find a list of all the roles and add new roles accessing the Roles tab in the left menu, as presented in figure 18.19. I have created two new roles named `fitnessuser` and `fitnessadmin`.

Role Name	Composite	Description
admin	True	\${role_admin}
create-realm	False	\${role_create-realm}
fitnessadmin	False	
fitnessuser	False	
offline_access	False	\${role_offline-access}
uma_authorization	False	\${role_uma_authorization}

Figure 18.19 By accessing the Roles tab in the left menu, you find all the defined roles, and you can create new ones. You'll assign later these roles to the users.

We now assign these roles to our users. I'll assign the role `fitnessadmin` to Mary, while Bill and Rachel will be regular users having the role `fitnessuser`. Figure 18.20 shows you how to attach the roles to the users.

The screenshot shows the Keycloak admin interface for a user named 'Mary'. The left sidebar has sections for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users). The main area shows 'Users > mary' for 'Mary'. The 'Role Mappings' tab is selected. Under 'Role Mappings', there are four sections: 'Realm Roles' (admin, create-realm, fitnessuser, offline\_access, umaa\_authorization), 'Available Roles' (admin, create-realm, fitnessuser, offline\_access, umaa\_authorization), 'Assigned Roles' (fitnessadmin), and 'Effective Roles' (fitnessadmin). A callout box with an arrow points to the 'Assigned Roles' section, containing the text: 'From the Role Mappings section, you assign the roles to a specific user.'

**Figure 18.20** From the Role Mappings section of the user, you assign the roles to the selected user. These role mappings will appear as the user's authorities in the access token, and you'll use them to implement the authorization configurations.

Unfortunately, by default, these new details won't appear in the access token. We have to customize the token according to the requirements of the application. We customize the token by configuring the client scope we created and assigned to the token in section 18.2.2. We need to add three more details to our tokens:

- **The roles** - used to apply a part of the authorization rules at the endpoint layer according to the scenario.
- **The username** – needed to filter the data when we apply the authorization rules.
- **The audience claim (AUD)** - used by the Resource Server to acknowledge the requests, as you'll learn in section 18.3.

In the next code snippet, you find the fields that will be added to the token once we finish with this setup.

```
{
  // ...
  "authorities": [
    "fitnessuser"
  ],
  "aud": "fitnessapp",
  "user_name": "rachel",
  // ...
}
```

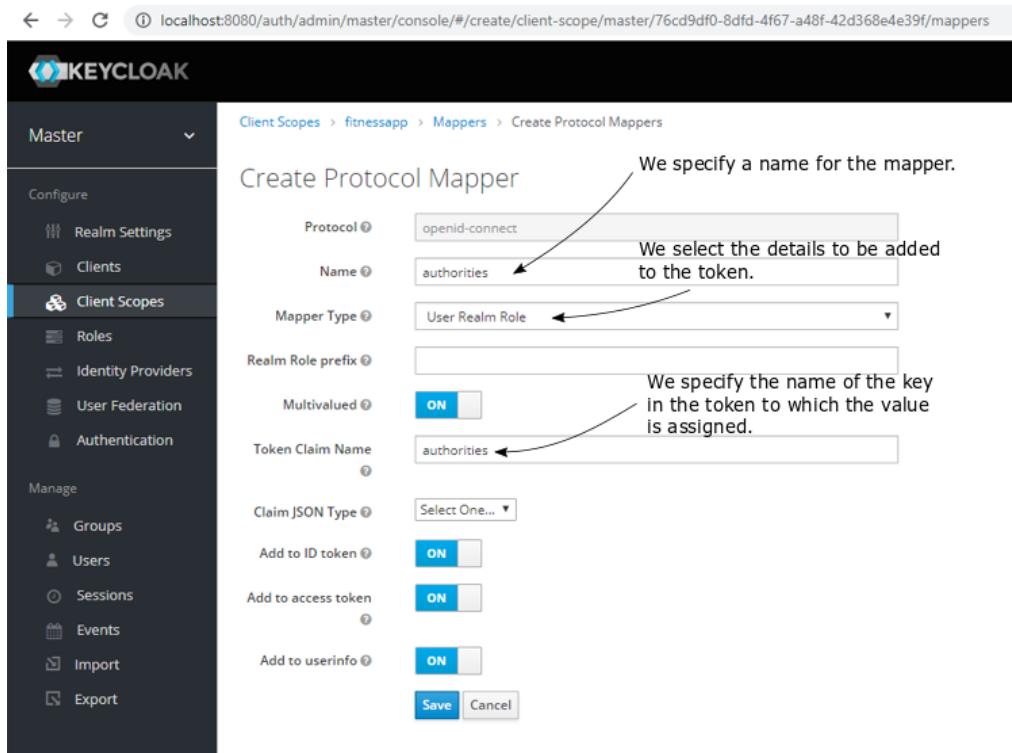
{}

We add custom claims by defining mappers on the client scope, as presented in figure 18.21.

The screenshot shows the Keycloak admin interface at the URL `localhost:8080/auth/admin/master/console/#/realms/master/client-scopes/76cd9df0-8dfd-4f67-a48f-42d368e4e39f/mappers`. The left sidebar is titled 'Master' and includes 'Configure' with sub-options: 'Realm Settings', 'Clients', 'Client Scopes' (which is selected and highlighted in blue), 'Roles', and 'Identity Providers'. The main content area is titled 'Client Scopes > fitnessapp' and shows a sub-section for 'Fitnessapp'. It has tabs for 'Settings', 'Mappers' (which is active and highlighted in blue), and 'Scope'. A note on the right says: 'For a specific client scope you create mappers by clicking on the Create button.' Below this is a search bar and a button labeled 'Create' with an arrow pointing to it. The message 'No mappers available' is displayed below the search bar.

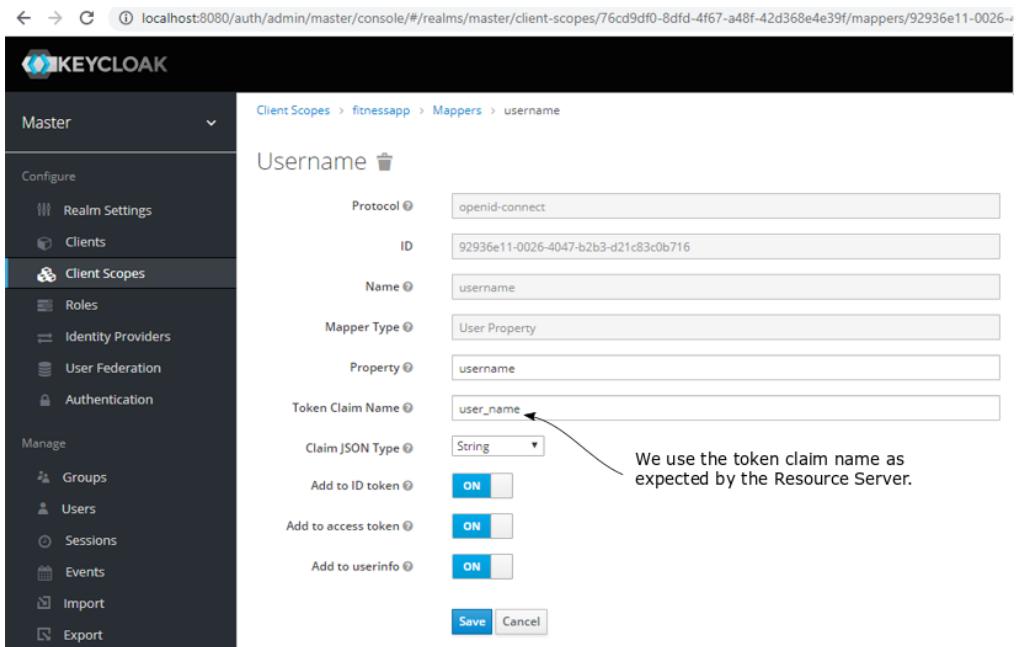
**Figure 18.21** We create mappers for a specific client scope to customize the access token. This way, we'll provide all the details the Resource Server needs to authorize the requests.

Figure 18.22 shows how to create a mapper to add the roles to the token. We will add the roles under the "authorities" key in the token as this is the way the Resource Server expects it.



**Figure 18.22** To add the roles in the access token, we define a mapper. When adding a mapper, we need to provide a name to the mapper. We also specify the details added to the token and the name of the claim identifying the assigned details.

With a similar approach to the one presented in figure 18.22, we also define a mapper to add the username to the token. Figure 18.23 shows how to create the mapper for username.



**Figure 18.23** We create a mapper to add the username to the access token. When adding the username to the access token, we'll choose the name of the claim `user_name`, which is how the Resource Server expects to find it in the token.

Finally, we need to specify the audience. The audience claim (AUD) defines the intended recipient of the access token. We'll set up a value for this claim, and we'll configure the same value for the Resource Server, as you'll learn in section 18.3. Figure 18.24 shows you how to define the mapper, so Keycloak adds the AUD claim to the JWT.

The screenshot shows the Keycloak Admin Console interface. On the left, there's a sidebar with a dark background containing various navigation items like 'Realm Settings', 'Clients', 'Client Scopes', 'Roles', etc. The 'Client Scopes' item is currently selected and highlighted in blue. The main content area has a light background and displays the configuration for a client scope named 'Aud'. The 'Mapper Type' is set to 'Audience', and the 'Included Client Audience' dropdown is populated with 'fitnessapp'. There are two toggle switches at the bottom: 'Add to ID token' is set to 'OFF', and 'Add to access token' is set to 'ON'. At the bottom right of the form are 'Save' and 'Cancel' buttons.

**Figure 18.24** The AUD claim representing the Audience defines the recipient of the access token, which in our case, is the Resource Server. We'll configure the same value at the Resource Server side for the Resource Server to accept the token.

If you obtain an access token again and decode it, you should find the authorities, `user_name`, and `aud` claims in the token's body. With this, we can now use this JWT to authenticate and call endpoints exposed by the Resource Server. Now that we have a fully configured Authorization Server, in section 18.3, we implement the Resource Server according to our scenario, as presented in section 18.1.

```
{
  "exp": 1585395055,
  "iat": 1585389055,
  "jti": "305a8f99-3a83-4c32-b625-5f8fc8c2722c",
  "iss": "http://localhost:8080/auth/realm/master",
  "aud": "fitnessapp",
  "sub": "c42b534f-7f08-4505-8958-59ea65fb3b47",
  "typ": "Bearer",
  "azp": "fitnessapp",
  "session_state": "f88a4f08-6cfa-42b6-9a8d-a2b3ed363bdd",
  "acr": "1",
  "scope": "fitnessapp",
  "user_name": "rachel",
  "authorities": [
    "fitnessuser"
  ]
}
```

```
 ]  
}
```

### 18.3 Implementing the application's Resource Server

In this section, we use Spring Security to implement the Resource Server for our scenario. Up to now, in section 18.2, we have configured Keycloak as the Authorization Server for the system (figure 18.25).

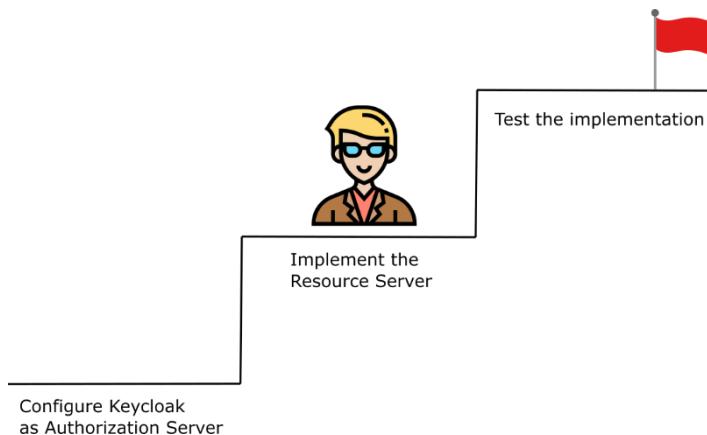
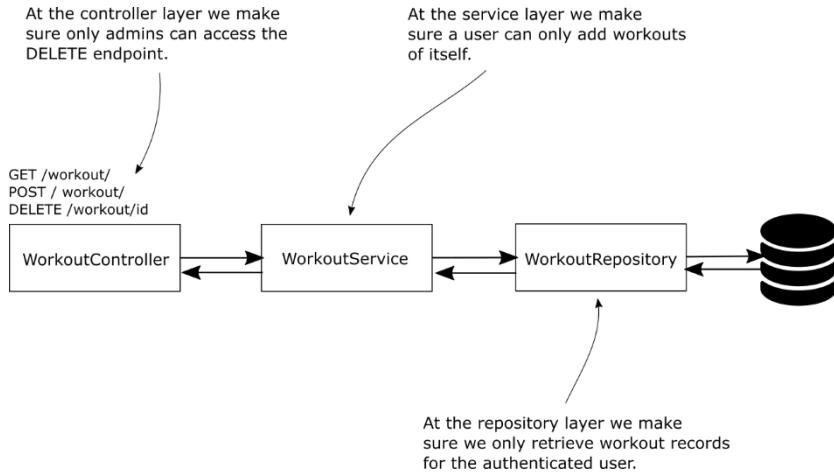


Figure 18.25 Now that we have set the Keycloak Authorization Server, we start the next big step in implementing the hands-on example: Implementing the Resource Server.

To build the Resource Server, I'll create a new project named `ssia-ch18-ex1`. The class design is straightforward, as presented in figure 18.26, based on three layers: a controller, a service, and a repository. We'll implement authorization rules at each of these layers.



**Figure 18.26** The class design for the Resource Server is very simple. We have three layers: the controller, the service, and the repository. Depending on the implemented use-case, we'll configure the authorization rules for one of these layers.

In the `pom.xml` file, I'll add the dependencies as presented in the next code snippet.

```
<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.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-data</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

We'll store the workout details in a database, so I'll also add the `schema.sql` and `data.sql` file to the project. In these files, I'll put the SQL queries to create the database structure and some test data we'll use later when testing the application.

We only need a simple table, so my `schema.sql` file stores only the query to create this table, which looks as presented in the next code snippet.

```
CREATE TABLE IF NOT EXISTS `spring`.`workout` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `user` VARCHAR(45) NULL,
  `start` DATETIME NULL,
  `end` DATETIME NULL,
  `difficulty` INT NULL,
  PRIMARY KEY (`id`));
```

I'll need some records in the workout table to test the application at the end. To add these records, I write some `INSERT` queries in the `data.sql` file, as presented in the next code snippet.

```
INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(1, 'bill', '2020-06-10 15:05:05', '2020-06-10 16:10:07', '3');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(2, 'rachel', '2020-06-10 15:05:10', '2020-06-10 16:10:20', '3');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(3, 'bill', '2020-06-12 12:00:10', '2020-06-12 13:01:10', '4');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(4, 'rachel', '2020-06-12 12:00:05', '2020-06-12 12:00:11', '4');
```

With these four `INSERT` statements, we now have a couple of workout records for user Bill and another two for the user Rachel to use in our tests.

Before starting to write our application logic, we'll also define the `application.properties` file. We already have the Keycloak Authorization Server running on port 8080, so I'll change the port for the Resource Server to 9090. Also, in the `application.properties` file, I'll write the properties needed by Spring Boot to create the data source. The next code snippet shows you the contents of my `application.properties` file.

```
server.port=9090

spring.datasource.url=jdbc:mysql://localhost/spring
    ?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

Now, let's first implement the JPA entity and the Spring Data JPA repository. The JPA entity class named `Workout` is presented in listing 18.1.

### **Listing 18.1 The `Workout` class**

```
@Entity
public class Workout {
```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private int id;
private String user;
private LocalDateTime start;
private LocalDateTime end;
private int difficulty;

// Omitted getter and setters
}

```

In listing 18.2, you find the Spring Data JPA repository interface for the `Workout` entity. Here, at the repository layer, we'll define a method to retrieve all the workout records for a specific user from the database. As you learned in chapter 17 instead of using `@PostFilter`, we'll choose to apply the constraint directly in the query.

### **Listing 18.2 The `WorkoutRepository` interface**

```

public interface WorkoutRepository
    extends JpaRepository<Workout, Integer> {

    @Query("SELECT w FROM Workout w WHERE[CA]
        w.user = ?#{authentication.name}")      #A
    List<Workout> findAllByUser();
}

```

#A We use SpEL to retrieve the value of the authenticated username from the security context.

If we now have a repository, we could continue with implementing the service class. The methods of the service class will be called directly by the controller. According to the scenario, we need to implement three methods:

- `saveWorkout()` – called to add a new workout record in the database.
- `findWorkouts()` – called to retrieve the workout records for a user.
- `deleteWorkout()` – called to delete a workout record for a given ID.

Listing 18.3 presents the implementation of the `WorkoutService` class.

### **Listing 18.3 The `WorkoutService` class**

```

@Service
public class WorkoutService {

    @Autowired
    private WorkoutRepository workoutRepository;

    @PreAuthorize[CA]      #A
    ("#workout.user == authentication.name")
    public void saveWorkout(Workout workout) {
        workoutRepository.save(workout);
    }

    public List<Workout> findWorkouts() {      #B
        return workoutRepository.findAllByUser();
    }

    public void deleteWorkout(Integer id) {      #C
}

```

```

        workoutRepository.deleteById(id);
    }
}

#A By pre-authorization, we make sure the method isn't called if the workout record doesn't belong to the user.
#B For this method, we have already applied the filtering at the repository layer.
#C We will apply the authorization for this method at the endpoint layer.

```

**NOTE** You may be wondering why I chose to implement the authorization rules precisely like you've seen in the example and not in a different way. Why, for the `deleteWorkout()` use case, did I write the authorization rules at the endpoint level and not at the service layer? For this case, I've chosen to do so to cover more ways for configuring authorization in the example. It'd have been the same thing if I had set the authorization rules for workout deletion at the service layer. Of course, in a more complex application, like in a real-world app, you might have restrictions that force you to choose a specific layer.

The controller class is simple, only defining the endpoints, which further call the service methods. Listing 18.4 presents the implementation of the controller class.

#### Listing 18.4 The `WorkoutController` class

```

@RestController
@RequestMapping("/workout")
public class WorkoutController {

    @Autowired
    private WorkoutService workoutService;

    @PostMapping("/")
    public void add(@RequestBody Workout workout) {
        workoutService.saveWorkout(workout);
    }

    @GetMapping("/")
    public List<Workout> findAll() {
        return workoutService.findWorkouts();
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Integer id) {
        workoutService.deleteWorkout(id);
    }
}

```

The last thing we need to define to have a complete application is the configuration class. We need to choose a way in which the Resource Server validates the tokens issued by the Authorization Server. We discussed three approaches in chapters 14 and 15:

- Using direct call to the Authorization Server
- Using a blackboarding approach
- Using cryptographic signatures

Because we already know the Authorization Server issues JWTs, the most comfortable choice is to rely on the cryptographic signature of the token.

As you know from chapter 15, we need to provide the Resource Server the key to validate the signature. Fortunately, Keycloak also exposes an endpoint where the public keys are exposed. With Keycloak, the endpoint where the public keys are exposed is the following:

<http://localhost:8080/auth/realm/master/protocol/openid-connect/certs>

We'll add this URI, together with the value of the AUD claim we set on the token in the application.properties file, as presented in the next code snippet.

```
server.port=9090

spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always

claim.aud=fitnessapp
jwkSetUri=http://localhost:8080/auth/realm/master/protocol/openid-connect/certs
```

Now we can write the configuration file, which looks as presented by listing 18.5.

#### Listing 18.5 The resource server configuration class

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity      #A
    (prePostEnabled = true)
public class ResourceServerConfig      #B
    extends ResourceServerConfigurerAdapter {

    @Value("${claim.aud}")      #C
    private String claimAud;

    @Value("${jwkSetUri}")      #C
    private String urlJwk;

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());      #D
        resources.resourceId(claimAud);      #D
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwkTokenStore(urlJwk);      #E
    }
}
```

#A Enabling Global Method Security pre/post annotations.

#B Extending ResourceServerConfigurerAdapter to customize the Resource Server configurations.

#C Injecting from the context the keys URI and the AUD claim value.

#D Setting up the token store and the value expected for the AUD claim.

#E Creating the TokenStore bean, which verifies the tokens based on keys found at the provided URI.

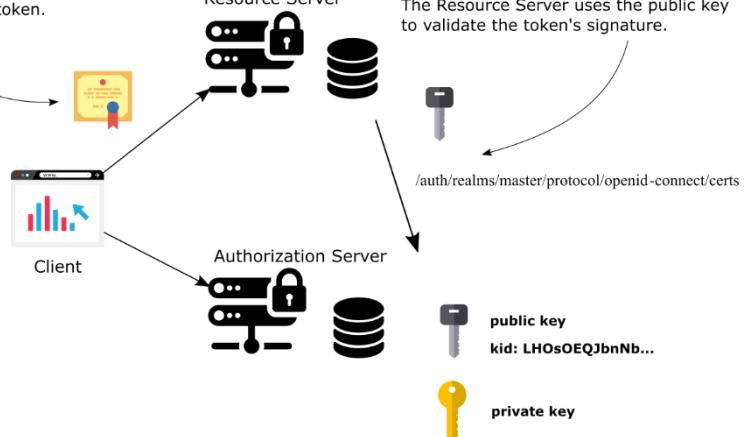
To create an instance of `TokenStore`, we use an implementation named `JwkTokenStore`. This implementation uses an endpoint where multiple keys could be exposed. To validate a

token, the `JwkTokenStore` looks for a specific key whose ID should exist in the header of the provided JWT token.

To be authorized, the client sends a signed token to the Resource Server. The token's header contains the ID of a key pair. The Authorization Server used the private key of the pair to sign the token.

Resource Server

The Resource Server gets the key ID from the token's header. It then calls an endpoint of the Authorization Server to get the public key from the key pair having that ID. The Resource Server uses the public key to validate the token's signature.



**Figure 18.27** The Authorization Server uses a private key to sign the token. When it signs the token, the Authorization Server also adds an ID of the key pair in the token's header. To validate the token, the Resource Server calls an endpoint of the Authorization Server and gets the public key for the ID found in the token's header. The Resource Server uses this public key to validate the token's signature.

**NOTE** Remember, we took the path `/openid-connect/certs` to the endpoint where Keycloak exposed the key at the beginning of the chapter, from Keycloak. You might find other tools to use a different path for this endpoint.

If you call the keys URI, you will see something similar to the next code snippet. In the HTTP response body, you have multiple keys. We call this collection of keys the key set. Each key has multiple attributes, including the value of the key, and a unique ID for each key. The attribute `kid` represents the key ID in the JSON response.

```
{
  "keys": [
    {
      "kid": "LHOsOEQJbnNbUn8PmZXA9TUoP56hY0tc3V0k0kUvj5U",
      "kty": "RSA",
      "alg": "RS256",
      "use": "sig",
      ...
    }
    ...
  ]
}
```

The JWT needs to specify which is the key ID of the key used to sign the token. The Resource Server needs to find the key ID in the JWT header. If you generate a token with our Resource Server, as we've done in section 18.2, and decode the header of the token, you'll see the token contains the key ID as expected. In the next code snippet, you find the decoded header of a token generated with our Keycloak Authorization Server.

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "LHOs0EQJbnNbUn8PmZXA9TUoP56hY0tc3V0k0kUvj5U"
}
```

To complete our configuration class, let's also add the authorization rules for the endpoint level and the `SecurityEvaluationContextExtension`. Our application needs the `SecurityEvaluationContextExtension` to evaluate the SpEL expression we used at the repository layer. The final configuration class looks as presented in listing 18.6.

#### **Listing 18.6 The configuration class**

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${claim.aud}")
    private String claimAud;

    @Value("${jwkSetUri}")
    private String urlJwk;

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
        resources.resourceId(claimAud);
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwkTokenStore(urlJwk);
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()      #A
            .mvcMatchers(HttpMethod.DELETE, "/**")
                .hasAuthority("fitnessadmin")
            .anyRequest().authenticated();
    }

    @Bean    #B
    public SecurityEvaluationContextExtension
        securityEvaluationContextExtension() {
        return new SecurityEvaluationContextExtension();
    }
}
```

```
}
```

#A Applying the authorization rules at the endpoint level.

#B Adding a SecurityEvaluationContextExtension bean to the Spring context.

## Using OAuth 2 web security expressions

In most of the cases, using common expressions to define the authorization rules is enough. Spring Security allows us to easily refer to authorities, roles, and username.

But in OAuth 2 resource servers, we sometimes need to refer to other values specific for this protocol, like the client roles or the scope. While we have these details in the JWT token, we can't access them directly with SpEL expressions to use them quickly in the authorization rules we define.

Fortunately, Spring Security offers us the possibility to enhance the expression by adding conditions related directly to OAuth 2. To use such SpEL expressions, we need to configure a `SecurityExpressionHandler`. The `SecurityExpressionHandler` implementation that allows us to enhance our authorization expression with OAuth 2 specific elements is `OAuth2WebSecurityExpressionHandler`. To configure it, we'd change the configuration class as presented in the next code snippet.

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    // Omitted code

    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
        resources.resourceId(claimAud);
        resources.expressionHandler(handler());
    }

    @Bean
    public SecurityExpressionHandler<FilterInvocation> handler() {
        return new OAuth2WebSecurityExpressionHandler();
    }
}

With such an expression handler configured, you could write an expression like the one presented in the next code snippet.

@PreAuthorize(
    "#workout.user == authentication.name and #oauth2.hasScope('fitnessapp')")
public void saveWorkout(Workout workout) {
    workoutRepository.save(workout);
}

Observe the condition I added to the @PreAuthorize annotation checking for the client scope:
#oauth2.hasScope('fitnessapp'). You can now add such expressions that will be evaluated by the
```

`OAuth2WebSecurityExpressionHandler` we added to our configuration. Same, you could use the `clientHasRole()` method in the expression instead of `hasScope()` to test if the client has a specific role. But mind that you can use client roles with the client credentials grant type.

To avoid mixing this example with the current hands-on project we develop in this chapter, I have separated it into a project named `ssia-ch18-ex2`.

## 18.4 Testing the application

Now that we have a complete system, we can run some tests to prove it works as desired. In this section, we'll run both our Authorization Server and Resource Server and use `curl` to test the implemented behavior.

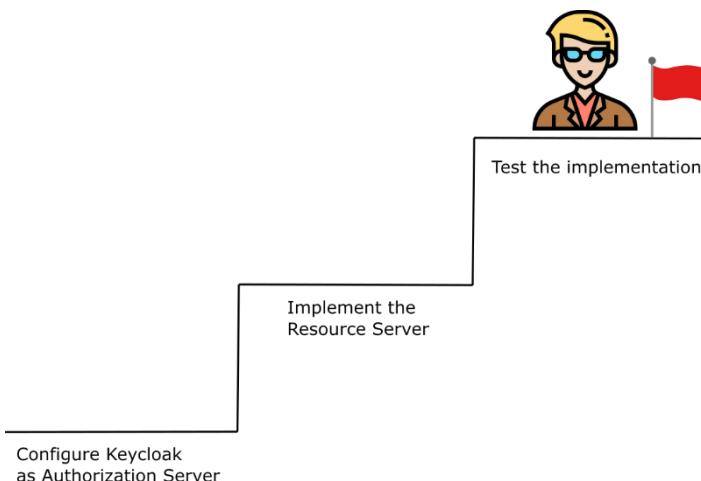


Figure 18.28 You got to the top! This is the last step of implementing the hands-on application of this chapter. In this section, we test the system and prove what we configured and implemented works as desired.

The scenarios we need to test are the following:

- Prove that the client can add a workout only for the authenticated user.
- Prove that the client can only retrieve their own workout records.
- Prove that only the admin users can delete a workout.

In my case, the Keycloak Authorization Server runs on port 8080, and for the Resource Server, I've configured in the `application.properties` file the port 9090. You'll need to make sure you make the calls to the correct component by using the ports you've configured.

Let's take each of the three test scenarios and prove the system is correctly secured.

### 18.4.1 Prove that the authenticated user can only add a record for themselves

According to the scenario, a user can only add a record for themselves. In other words, if I authenticated as Bill, I shouldn't be able to add a workout record for Rachel. To prove this is the behavior, we call the Authorization Server and issue a token for one of the users, say Bill. Then we try to add both a workout record for Bill and a workout record for Rachel. We'll prove that Bill can add a record for himself, but the app doesn't allow him to add a record for Rachel.

To issue a token, we call the Authorization Server as presented in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Among other details, you'll also get an access token for Bill. I've truncated the value of the token in the code snippet to make it shorter. The access token contains all the details needed for authorization, like the username and the authorities we added previously by configuring Keycloak in section 18.1.

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI...",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "0630a3e4-c4fb-499c-946b-294176de57c5",
  "scope": "fitnessapp"
}
```

Having the access token, we can call the endpoint to add a new workout record. We first try to add a workout record for Bill. We expect that adding a workout record for Bill works because the access token we have was generated for Bill. The next code snippet presents the `curl` command you should run to add a new workout for Bill. Running this command, you'll get an HTTP response status for 200 OK, and a new workout record will be added in the database. Of course, as the value of the `Authorization` header, you should add your access token previously generated. I have truncated the value of my token in the next code snippet to make the command shorter and easier to read.

```
curl -v -XPOST 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOi...' \
-H 'Content-Type: application/json' \
--data-raw '{
  "user" : "bill",
  "start" : "2020-06-10T15:05:05",
  "end" : "2020-06-10T16:05:05",
  "difficulty" : 2
}'
```

But if you call the endpoint trying to add a record for Rachel, as presented in the next code snippet, you'll get back an HTTP response status of 403 Forbidden.

```
curl -v -XPOST 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOi...' \
-H 'Content-Type: application/json' \
--data-raw '{
    "user" : "rachel",
    "start" : "2020-06-10T15:05:05",
    "end" : "2020-06-10T16:05:05",
    "difficulty" : 2
}'
```

The response body is:

```
{
    "error": "access_denied",
    "error_description": "Access is denied"
}
```

#### 18.4.2 Prove that a user can only retrieve their records

In this section, we'll prove the second test scenario. Our Resource Server should only return the workout records for the authenticated user. To demonstrate this behavior, we'll generate access tokens for both Bill and Rachel, and we'll call the endpoint to retrieve their workout history. Neither one of them should see records for the other.

To generate an access token for Bill, use `curl` as presented in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Calling the endpoint to retrieve the workout history with the access token generated for Bill, you will only get back in the response Bill's records.

```
curl 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSl...'
```

The response body is:

```
[
    {
        "id": 1,
        "user": "bill",
        "start": "2020-06-10T15:05:05",
        "end": "2020-06-10T16:10:07",
        "difficulty": 3
    },
    ...
]
```

Now generate a token for Rachel and call the same endpoint. To generate an access token for Rachel, run the `curl` command in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=rachel' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Using the access token for Rachel to get the workout history, the application only returns records owned by Rachel.

```
curl 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiS1...'
```

The response body is:

```
[  
  {  
    "id": 2,  
    "user": "rachel",  
    "start": "2020-06-10T15:05:10",  
    "end": "2020-06-10T16:10:20",  
    "difficulty": 3  
  },  
  ...  
]
```

### 18.4.3 Prove that only admins can delete records

The third and last test scenario in which we want to prove the application behaves as desired is that only admin users can delete workout records. To demonstrate this behavior, we'll generate an access token for our admin user, Mary, and an access token for one of the other users who are not admins, let's say, Rachel. Using the access token generated for Mary, we'll be able to delete a workout. But the application will forbid us calling the endpoint to delete a workout record using an access token generated for Rachel.

To generate a token for Rachel, use `curl` as presented in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=rachel' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

If you use this token to delete an existing workout, you'll get back a 403 Forbidden HTTP response status. Of course, the record won't be deleted from the database.

```
curl -XDELETE 'localhost:9090/workout/2' \
--header 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsIn...'
```

Generate a token for Mary and rerun the same call to the endpoint with the new access token. To generate a token for Mary, use the `curl` command presented in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=mary' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Calling the endpoint to delete a workout record with the access token for Mary will return an HTTP status 200 OK, and the workout record will be removed from the database.

```
curl -XDELETE 'localhost:9090/workout/2' \
--header 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsIn...'
```

## 18.5 Summary

- You don't necessarily need to implement your custom Authorization Server. Often in real-world scenarios, we use tools such as Keycloak to implement the Authorization Server.
- Keycloak is an open-source identity and access management solution which offers you great flexibility in dealing with user management and authorization. Often, you'd prefer using such a tool over implementing a custom solution to spare time.
- Having solutions as Keycloak doesn't mean you'll never implement custom solutions for authorization anymore. In real-world scenarios, you'll find situations in which stakeholders of an application you need to build don't consider third-party implementations trustworthy. You have to be prepared to deal with all the possible cases you might encounter.
- You can use Global Method Security in a system implemented over the OAuth 2 framework. In such a system, you implement the Global Method Security restrictions at the Resource Server level, which protects the user resources.
- You can use specific OAuth 2 elements in your SpEL authorization expressions. To write such SpEL expressions, you need to configure an `OAuth2WebSecurityExpressionHandler` to interpret these expressions.

# 19

## *Spring Security for reactive apps*

### This chapter covers

- Using Spring Security with reactive applications.
- Using reactive apps in a system designed with OAuth 2.

Reactive is a different programming paradigm, where we apply a different way of thinking when developing the application than usual. Reactive applications are a powerful way of developing web apps. We've seen them lately used more and more in real-world examples. I would even say that it became a fashion a few years ago when any important conference had at least a few presentations discussing reactive apps. Like any other technology in software development, reactive programming doesn't represent a solution applicable to every situation.

In some cases, a reactive approach is an excellent fit. In other cases, it might only complicate your life. But the reactive approach exists in the end because it addresses some limitations of imperative programming. And you'd like to use reactive programming to avoid such limitations. One of these limitations is referring to executing large tasks that could be somehow fragmented. With an imperative approach, you give the application a task to execute, and the application has the responsibility to solve it. If the task is large, it might take a substantial amount of time for the application to solve it. The client who assigned the task needs to wait for the task to be entirely solved until receiving a response. With reactive programming, you could divide the task so that the app has the opportunity to approach some of the subtasks concurrently. This way, the client who assigned the task will also receive the processed data faster.

In this chapter, we discuss implementing application-level security in reactive applications with Spring Security. Like for any other application, security is an important aspect of the reactive apps as well. But because reactive apps are designed differently, Spring Security has adapted the way we implement features discussed previously in this book.

We'll start with a short overview of implementing reactive apps with Spring framework in section 19.1. Then, we'll apply the security features you learned throughout this book on

security apps. In section 19.2, we discuss user management in reactive apps, and we continue with applying authorization rules in section 19.3. Finally, in section 19.4, you'll learn how to implement reactive applications in a system designed over OAuth 2. You'll learn what changes from the Spring Security perspective when it comes to reactive applications, and of course, you'll learn how to apply them with examples.

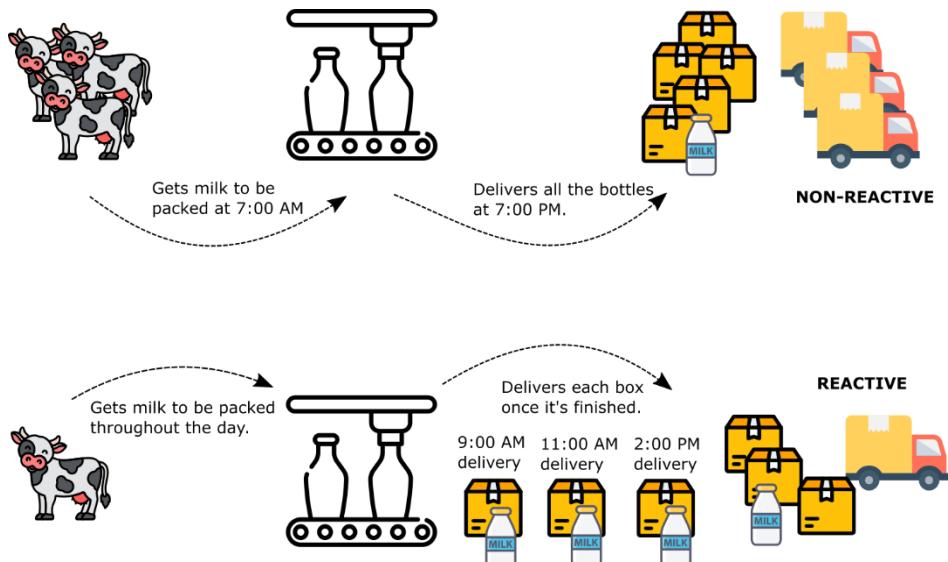
## 19.1 What are reactive apps?

In this section, we shortly discuss reactive apps. The whole chapter is about applying security for reactive apps. So with this section, I want to make sure you remember the essentials of reactive apps before going deeper with the Spring Security configurations. The topic of reactive applications is big, so in this section, I'll only review the main aspects of them as a refresher. If you aren't aware yet of how reactive applications work, or you need to understand them in detail, I recommend you first read chapter 10 of *Spring in Action* by Craig Walls (Manning, 2018) <https://livebook.manning.com/book/spring-in-action-fifth-edition/chapter-10/>.

When we implement apps, we use two fashions of implementing the functionalities:

- **Imperative** - your application processes a bulk of data all at once. For example, a client application calls an endpoint exposed by the server and sends all the data that needs to be processed to the backend side. Say you implement a functionality where the user uploads files. If the user selects a number of files, and all of them are received by the backend application all at once to be processed - you're working with an imperative approach.
- **Reactive** – your application receives and processes the data in fragments. Not all the data has to be fully available from the beginning to be processed. The backend receives and processes the data as it gets the data. Say the user selects some files, and those files have to be uploaded and processed by the backend. The backend won't wait to receive all the files at once and process them. The backend might receive the files one by one and process each while waiting for more files to come.

Figure 19.1 presents an analogy for the two programming approaches. Imagine a factory bottling milk. If the factory gets all the milk in the morning and, once it finishes the bottling, it delivers the milk, we say it's non-reactive (imperative). If the factory gets the milk throughout the day and once it finishes bottling enough milk for an order, it delivers the order, then we say it's reactive. Clearly, in this case, for the milk factory, it's more advantageous to use a reactive approach rather than a non-reactive one.



**Figure 19.1 Non-Reactive vs. Reactive:** In a non-reactive approach, the milk factory gets all the milk to be packed in the morning and delivers all the boxes in the evening. In a reactive approach, while the milk is brought to the factory, it's packed and then delivered. For this scenario, a reactive approach is better as it allows the milk to be collected throughout the day and be delivered sooner to the clients.

For implementing reactive apps, the Reactive Streams specification (<http://www.reactive-streams.org/>) provides a standard way for asynchronous streams processing. One of the implementations of this specification is Project Reactor, which also makes the foundations of the Spring framework's reactive programming model. Project Reactor provides a functional API from composing Reactive Streams.

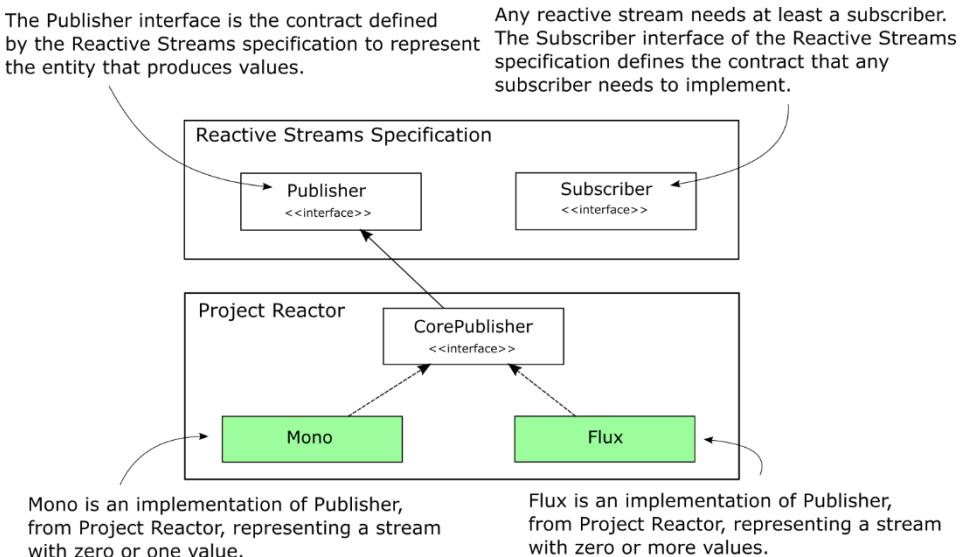
To get a more hands-on feeling, let's start a simple implementation of a reactive app. We'll continue further with this same application in section 19.2 when discussing user management in reactive apps. I'll create a new project named `ssia-ch19-ex1` and develop a reactive web application that exposes a demo endpoint. In the `pom.xml` file, we'll need to add the reactive web dependency as presented in the next code snippet. This dependency houses project Reactor and will enable us to use its related classes and interfaces in our project.

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

Now we define a simple `HelloController` to hold the definition of our demo endpoint. Listing 19.1 shows the definition of the `HelloController` class. In the definition of the endpoint, you observe I used as returned type a `Mono`. `Mono` is one of the essential concepts defined by Reactor implementation. When working with Reactor, you often use `Mono` and `Flux`, which both define publishers – sources of data.

In the Reactive Streams specification, a publisher is described by the `Publisher` interface, which describes one of the essential contracts used with reactive streams. The other contract is the `Subscriber`. The `Subscriber` describes the component consuming the data.

When designing an endpoint that returns something, the endpoint becomes a publisher, so it has to return a `Publisher` implementation. If using Project Reactor, this will be a `Mono` or a `Flux`. `Mono` is a publisher for a single value, while `Flux` is a publisher of multiple values. Figure 19.2 describes the discussed components and the relationships among them.



**Figure 19.2** In a reactive stream, a publisher produces values, and a subscriber consumes these values. Contracts defined by the Reactive Streams specification describe publishers and subscribers. Project Reactor implements the Reactive Streams specification and implements the Publisher and Subscriber contracts. Shaded differently are the components we use in the examples of this chapter.

To make my explanation even more precise, I'll go back to the milk factory analogy. The milk factory is a reactive backend implementation that exposes an endpoint to receive the milk to be processed. This endpoint produces something – bottled milk – so this endpoint needs to return a `Publisher`. If we exercise our imagination and assume for a request more bottles of milk are produced, then the milk factory would need to return a `Flux` (which is Project Reactor's publisher implementation that deals with zero or more produced values).

#### **Listing 19.1** The definition of the `HelloController` class

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello!");
    }
}
```

```
}
```

#### #A Creating a returning a Mono stream source with one the value on the stream.

You can now start and test the application. The first thing you'll observe by looking in the app's terminal is that Spring Boot doesn't configure a Tomcat server anymore. You used to see Spring Boot configuring a Tomcat by default in a web application, and you could observe this aspect in any of the previous examples we developed in this book. Instead, now Spring Boot will configure Netty as the webserver. Netty is the default reactive web server implementation auto-configured in a Spring Boot project.

The second thing you'll observe when calling the endpoint is that it doesn't behave differently from an endpoint developed with a non-reactive approach. You'll still find in the HTTP response body the `Hello!` message that the endpoint returns in its defined `Mono` stream. The app's behavior when calling the endpoint is presented in the next code snippet.

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

The response body is:

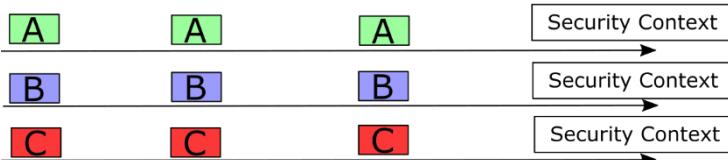
```
Hello!
```

But why is the reactive approach different in terms of Spring Security? Behind the scenes, a reactive implementation uses multiple threads to solve the tasks on the stream. In other words, it changes the philosophy of one-thread-per-request, which we were used to for a web application designed with an imperative approach. And from here, some differences:

- The `SecurityContext` implementation, which we discussed in chapter 5, can't work the same way anymore in reactive applications. Remember, the `SecurityContext` is based on a `ThreadLocal`, and now we have more than one thread per request (figure 19.3).
- Because of the `SecurityContext`, any authorization configuration is now affected. Remember, the authorization rules generally rely on the `Authentication` instance stored in the `SecurityContext`. So now, the security configurations applied at the endpoint layer as well as Global Method Security functionality are affected (figure 19.3).

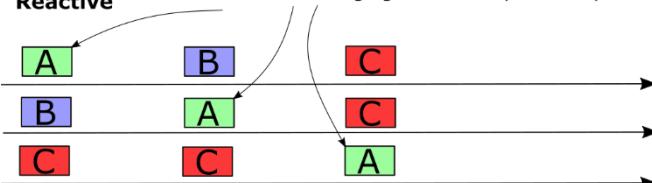
In a non-reactive web app, a thread is allocated for each request. So we always know within the same thread, we always work with tasks for the same request. For this reason, the app manages the SecurityContext per thread.

### Non-Reactive



In a reactive app, a task from a request can be managed by multiple different threads. So managing the SecurityContext per thread is no longer an option.

### Reactive



**Figure 19.3** In the figure, each arrow represents the timeline of a different thread. The processed tasks from requests A, B, and C are represented with squares. Because in a reactive app, tasks from one request might be handled on multiple threads, the Authentication details cannot be stored at the thread level anymore.

- The `UserDetailsService`, which we learned in chapter 2 as being the component responsible for retrieving the user details, is a source of data. So the user details service also needs to support a reactive approach.

Fortunately, Spring Security offers support for reactive apps and covers all the cases in which you can't use the implementations for non-reactive applications anymore. We'll continue in this chapter by discussing the way you implement security configurations with Spring Security for reactive apps. We start in section 19.2 with implementing the user management and continue in section 19.3 with applying endpoint authorization rules and the way security context works in reactive apps. We then continue our discussion with Reactive Method Security, which replaces what we've learned in imperative applications as being Global Method Security.

## 19.2 User management in reactive apps

Often in applications, the way a user authenticates is based on a pair of username and password credentials. This approach is basic, and we discussed it, starting with the most straightforward application we implemented in chapter 2. But with reactive apps, the implementation of the component taking care of the user management changes as well. In this section, we discuss implementing user management in a reactive application.

We'll continue the implementation of the `ssia-ch19-ex1` application we started in section 19.1, and we'll add a `ReactiveUserDetailsService` to the context of the application. We want to make sure the `/hello` endpoint can be called only by an authenticated user. As its name

suggests, the `ReactiveUserDetailsService` contract defines the user details service for a reactive app.

The definition of the contract is as simple as for the one for `UserDetailsService`. The `ReactiveUserDetailsService` defines one method used by Spring Security to retrieve a user by its user name. The big difference is that the method described by the `ReactiveUserDetailsService` returns a `Mono<UserDetails>` and not directly the `UserDetails` as it happened for `UserDetailsService`. The next code snippet shows the definition of the `ReactiveUserDetailsService` interface.

```
public interface ReactiveUserDetailsService {
    Mono<UserDetails> findByUsername(String username);
}
```

Like in the case of the `UserDetailsService`, you can write a custom implementation of the `ReactiveUserDetailsService` to give Spring Security a way to obtain the details of the users. To simplify this demonstration, I'll use an implementation provided by Spring Security. The `MapReactiveUserDetailsService` implementation stores the user details in memory (same as `InMemoryUserDetailsManager`, which you learned in chapter 2 does).

I'll change the `pom.xml` file of the `ssia-ch19-ex1` and add the Spring Security dependency as presented in the next code snippet.

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

I'll then create a configuration class and add a `ReactiveUserDetailsService` and a `PasswordEncoder` to the Spring Security context. I'll name the configuration class `ProjectConfig`. You find the definition of this class in listing 19.2. Using a `ReactiveUserDetailsService`, I'll define one user with its username `john`, the password `12345`, and an authority I'll name `read`. As you observe, it's very similar to working with a `UserDetailsService`. The main difference in the implementation of the `ReactiveUserDetailsService` is that the method returns a reactive Publisher containing the `UserDetails` instead of the `UserDetails` instance itself. Spring Security takes the rest of the duty for integration.

### **Listing 19.2 The ProjectConfig class**

```
@Configuration
public class ProjectConfig {

    @Bean      #A
    public ReactiveUserDetailsService userDetailsService() {
        var u = User.withUsername("john")      #B
            .password("12345")
            .authorities("read")
```

```

    .build();

    var uds = new MapReactiveUserDetailsService(u);      #C
    return uds;
}

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

#A Adding a ReactiveUserDetailsService to the Spring context.
#B Creating a new user with its username, password and authorities.
#C Creating a MapReactiveUserDetailsService to manage the UserDetails instances.
#D Adding a PasswordEncoder to the Spring context.

```

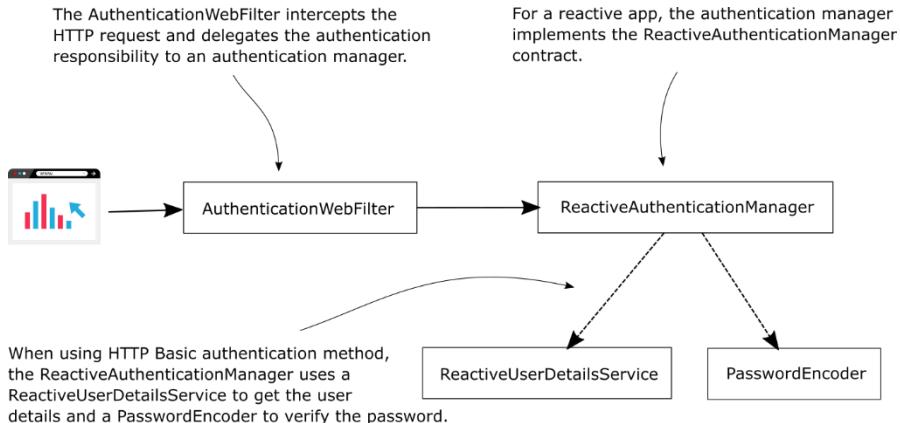
Starting and testing the application now, you'll observe you can call the endpoint only when you authenticate using the proper credentials. In our case, we can only use `john` with its password `12345`, as it's the only user record we've added. The following code snippet shows you the behavior of the app when calling the endpoint with valid credentials.

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

The response body is:

```
Hello!
```

Figure 19.2 explains the architecture we used in this application. Behind the scenes, an `AuthenticationWebFilter` intercepts the HTTP request. This filter delegates the authentication responsibility to an authentication manager. The authentication manager implements the `ReactiveAuthenticationManager` contract. Unlike non-reactive apps, we don't have authentication providers. The `ReactiveAuthenticationManager` directly implements the authentication logic. If you'd like to create your own custom authentication logic, implement the `ReactiveAuthenticationManager` interface. This architecture for reactive apps is not much different from the one we already discussed throughout this book for non-reactive applications. As presented in figure 19.4, if the authentication involves user credentials, then a `ReactiveUserDetailsService` is used to obtain the user details and a `PasswordEncoder` to verify the password.



**Figure 19.4 An AuthenticationWebFilter intercepts the request and delegates the authentication responsibility to a ReactiveAuthenticationManager. If the authentication logic involves users and passwords, the ReactiveAuthenticationManager will use a ReactiveUserDetailsService to find the user details and a PasswordEncoder to verify the password.**

Moreover, the framework still knows to inject an authentication when you request this. You request the `Authentication` details by merely adding a `Mono<Authentication>` as a parameter to the method in the controller class. Listing 19.3 presents the changes done to the controller class. Again, the significant change is that you use reactive publishers – observe we need to use `Mono<Authentication>` instead of plain `Authentication` as we were used to in non-reactive apps.

### Listing 19.3 The HelloController class

```

@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello(Mono<Authentication> auth) {      #A
        Mono<String> message =      #B
            auth.map(a -> "Hello " + a.getName());

        return message;
    }
}
  
```

#A We request the framework to provide us the authentication object.

#B We return the name of the principal in the response.

Rerunning the application and calling the endpoint, you'll observe the behavior is as presented in the next code snippet.

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

The response body is:

```
Hello john
```

And now, your question is probably: Where did the `Authentication` object come from? Being that this is a reactive app, we can't afford using a `ThreadLocal` anymore as the framework as designed to manage the `SecurityContext`. Correct! The app can't use anymore a `ThreadLocal`, but Spring Security offers us a different implementation of context holder for reactive apps. This implementation is named `ReactiveSecurityContextHolder`, and we use it to work with the `SecurityContext` in a reactive application. So we still have the `SecurityContext`, just that now is managed differently. Figure 19.5 describes the end of the authentication process, once the `ReactiveAuthenticationManager` successfully authenticated the request.

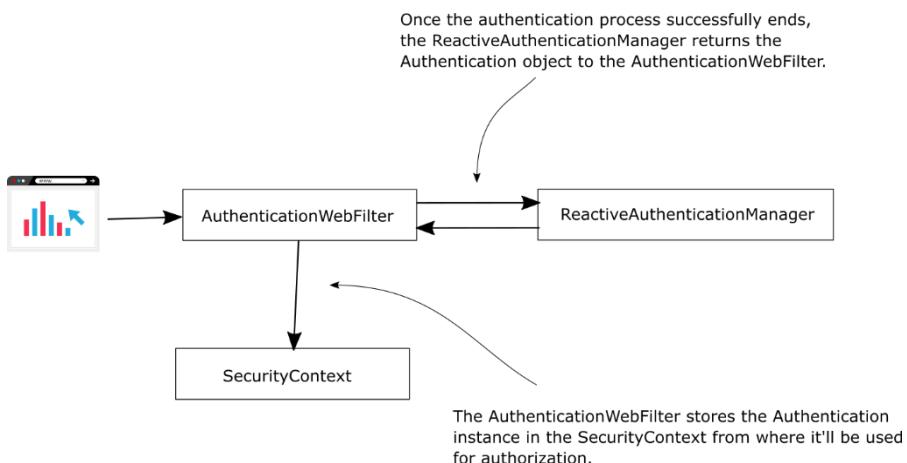


Figure 19.5 Once the `ReactiveAuthenticationManager` successfully authenticated the request, it returns the `Authentication` object to the filter. The filter stores the `Authentication` instance in the `SecurityContext`.

Listing 19.4 shows you how to rewrite the controller class if you wished to get the authentication details directly from the security context. This approach is an alternative to allowing the framework to inject it through the method's parameter. You find this change implemented in project `ssia-ch19-ex2`.

#### Listing 19.4 Working with a `ReactiveSecurityContextHolder`

```

@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        Mono<String> message =
            ReactiveSecurityContextHolder.getContext()      #A
                .map(context -> context.getAuthentication())  #B
                .map(auth -> "Hello " + auth.getName());     #C
        return message;
    }
}
    
```

#A From the `ReactiveSecurityContextHolder` we take a `Mono<SecurityContext>`.

#B We map the `SecurityContext` to the `Authentication` object.

#C We map the `Authentication` object to the message we return.

If you rerun the application and test again the endpoint, you'll observe it behaves the same as in the previous examples of this section.

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

The response body is:

```
Hello john
```

Now that you know Spring Security offers us an implementation to properly manage the `SecurityContext` in a reactive environment, you know this is how your app applies the authorization rules. And these details that you've just learned opens the path to configuring the authorization rules, which we'll discuss in section 19.3.

## 19.3 Configuring authorization rules in reactive apps

In this section, we discuss configuring authorization rules. As you already know from the previous chapters, authorization follows authentication. We discussed in section 19.1 and 19.2 how Spring Security manages the users and the `SecurityContext` in reactive apps. But once the app finished the authentication and stored the details of the authenticated request in the `SecurityContext`, it's time for authorization.

Like for any other application, you'll most probably need to configure authorization rules when developing reactive applications as well. To teach you how to set authorization rules in reactive apps, we'll discuss first the way you make configurations at the endpoint layer in section 19.3.1. Once we finish discussing the authorization configuration at the endpoint layer, you'll learn how to apply it at any other layer of your application using method security in section 19.3.2.

### 19.3.1 Applying authorization at the endpoint layer in reactive apps

In this section, we discuss configuring authorization at the endpoint layer in reactive apps. Setting the authorization rules of the endpoint layer is the most common approach for configuring authorization in a web app. You've already discovered this yourself while working on the previous examples of this book. Authorization configuration at the endpoint layer is essential – you use it in almost every app. Thus, you need to know how to apply it for reactive applications as well.

You learned from previous chapters to set the authorization rules by overriding the `configure(HttpSecurity http)` method of the `WebSecurityConfigurerAdapter` class. This approach doesn't work anymore in reactive apps. To teach you how to configure the authorization rules for the endpoint layer properly, we'll start working on a new project, which I'll name `ssia-ch19-ex3`.

In reactive apps, Spring Security uses a contract named `SecurityWebFilterChain` to apply the configurations we used to do by overriding one of the `configure()` methods of the `WebSecurityConfigurerAdapter` class as discussed in previous chapters. With reactive apps,

we'll add a bean of type `SecurityWebFilterChain` in the Spring context. To teach you how to do this, let's implement a basic application having two endpoints that we'll secure independently.

In the `pom.xml` file of our newly created `ssia-ch19-ex3` project, I'll add the dependencies for reactive web apps and, of course, Spring Security.

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

I'll create a controller class to define the two endpoints for which I'll configure the authorization rules. The endpoints I'll define are accessible at paths `/hello` and `/ciao`. To call the `/hello` endpoint, someone needs to authenticate, while `/ciao` can be called without the need for authentication. Listing 19.5 presents the definition of the controller.

#### **Listing 19.5 The HelloController class defines the endpoints to secure**

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello(Mono<Authentication> auth) {
        Mono<String> message = auth.map(a -> "Hello " + a.getName());
        return message;
    }

    @GetMapping("/ciao")
    public Mono<String> ciao() {
        return Mono.just("Ciao!");
    }
}
```

In the configuration class (listing 19.6), I make sure to declare a `ReactiveUserDetailsService` and a `PasswordEncoder` to define a user as you learned in section 19.2.

#### **Listing 19.6 The configuration class declares components for the user management**

```
@Configuration
public class ProjectConfig {

    @Bean
    public ReactiveUserDetailsService userDetailsService() {
        var u = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

        var uds = new MapReactiveUserDetailsService(u);
        return uds;
    }
}
```

```

        return uds;
    }

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

    // ...
}

```

In listing 19.7, I'll work in the same configuration class we declared in listing 19.6, but I'll omit the declaration of the `ReactiveUserDetailsService` and the `PasswordEncoder` to allow you to focus on the authorization configuration we discuss. In listing 19.7, you observe I add a bean of type `SecurityWebFilterChain` to the Spring context. The method receives as a parameter an object of type `ServerHttpSecurity`, which will be injected by Spring. This `ServerHttpSecurity` enables us to build a `SecurityWebFilterChain` instance. `ServerHttpSecurity` provides methods for configuration similar to the ones you're used to when configuring authorization for non-reactive apps.

### Listing 19.7 Configuring endpoint authorization for reactive apps

```

@Configuration
public class ProjectConfig {

    // Omitted code

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http) {

        return http.authorizeExchange()      #A
            .pathMatchers(HttpMethod.GET, "/hello")
            .authenticated()      #B
            .anyExchange()        #C
            .permitAll()          #D
            .and().httpBasic()    #E
            .and().build();       #F
    }
}

```

#A We begin the endpoint authorization configuration.

#B We select the requests for which we apply the authorization rules.

#C We configure the selected requests to only be accessible when authenticated.

#D We refer to any other request.

#E We allow the requests to be called without needing authentication.

#F We build the `SecurityWebFilterChain` object to be returned.

We started the authorization configuration with the `authorizeExchange()` method. We call the `authorizeExchange()` method similar to the way we were calling the `authorizeRequests()` method when configuring endpoint authorization for non-reactive apps. Then we continued by using the `pathMatchers()` method. You can consider the `pathMatchers()` method as the equivalent of using `mvcMatchers()` when configuring endpoint authorization for non-reactive apps. Like for non-reactive apps, once we use the matcher

method to group the requests to which we apply the authorization rule, we then specify what the authorization rule is. In our example, we called the `authenticated()` method, which states that only authenticated requests are accepted. You used a method named `authenticated()` also when configuring endpoint authorization for non-reactive apps. The methods for reactive apps were named the same to make them intuitive. Similarly to the `authenticated()` method, you could also call:

- `permitAll()`, to configure the app to allow the requests without authentication.
- `denyAll()`, to deny all the requests.
- `hasRole()` and `hasAnyRole()` to apply rules based on roles.
- `hasAuthority()` and `hasAnyAuthority()` to apply rules based on authorities.

It looks like something's missing, isn't it? Do we also have an `access()` method as we had for configuring authorization rules in non-reactive apps? Yes. But it's a bit different, so we'll work on a separate example to prove it.

Another similarity in namings is the `anyExchange()` that takes the role of what used to be `anyRequest()` in case of non-reactive apps.

**NOTE** Why exchange? Why is it called `anyExchange()`, and they didn't keep the same name for the method `anyRequest()`? Why `authorizeExchange()` and not `authorizeRequests()`? This simply comes from the reactive apps' terminology. We generally refer to the communication between two components in a reactive fashion as exchanging data. This way, we enforce the image of the data being sent as segmented in a continuous stream and not as a big bunch in one request.

We also have to specify the authentication method, like any other related configurations. We do this also by the use of the same `ServerHttpSecurity` instance, using methods with the same name and in the same fashion you learned to use for non-reactive apps: `httpBasic()`, `formLogin()`, `csrf()`, `cors()`, adding filters and customizing the filter chain and so on. In the end, we call the `build()` method to create the `SecurityWebFilterChain` instance, which we finally return to be added in the Spring context.

I've told you earlier in this section that you can also use an `access()` method in the endpoint authorization configuration of reactive apps as you could for non-reactive apps. But like I said when discussing the configuration of non-reactive apps in chapters 7 and 8, use the `access()` method only when you can't apply your configuration otherwise. The `access()` method offers you great flexibility, but also makes your app's configuration more difficult to be read. You'd always prefer the simpler solution over the more complex one.

But you'll find situations in which you need this flexibility. You'll need to apply a more complex authorization rule, and then, only using the `hasAuthority()` or `hasRole()`, and its companion methods isn't enough. For this reason, I'll also teach you how to use the `access()` method. I have created a new project named `ssia-ch19-ex4` for this example. In listing 19.8, you see how I build the `SecurityWebFilterChain` object to allow access to the `/hello` path only if the user has the role `ADMIN`. Also, access can be done only before noon. For all the other endpoints, I completely restricted the access.

**Listing 19.8 Using the `access()` method when implementing configuration rules**

```

@Configuration
public class ProjectConfig {

    // Omitted code

    @Bean
    public SecurityWebFilterChain
        securityWebFilterChain(ServerHttpSecurity http) {

        return http.authorizeExchange()
            .anyExchange()      #A
                .access(this::getAuthorizationDecisionMono)
            .and().httpBasic()
            .and().build();
    }

    private Mono<AuthorizationDecision>
        getAuthorizationDecisionMono(      #B
            Mono<Authentication> a,
            AuthorizationContext c) {

        String path = getRequestPath(c);      #C

        boolean restrictedTime =      #D
            LocalTime.now().isAfter(LocalTime.NOON);

        if(path.equals("/hello")) {      #E
            return a.map(isAdmin())
                .map(auth -> auth && !restrictedTime)
                .map(AuthorizationDecision::new);
        }

        return Mono.just(new AuthorizationDecision(false));
    }

    // Omitted code
}

```

#A For any request, we apply a custom authorization rule.

#B The method defining the custom authorization rule receives the Authentication and the request context as parameters.

#C From the context, we obtain the path of the request.

#D We define the restricted time.

#E For the /hello path, we apply the custom authorization rule.

It might look difficult, but it's not that complicated. When you use the `access()` method, you provide a function receiving all the details possible about the request: the `Authentication` object and the `AuthorizationContext`. Using the `Authentication` object, you have the details of the authenticated user: `username`, `roles` or `authorities`, and other custom details depending on how you implemented the authentication logic. The `AuthorizationContext` provides you the information on the request: the path, headers, query params, cookies, and so on.

The method should return an object of type `AuthorizationDecision`. As you guessed, this `AuthorizationDecision` object is the answer that tells the app if the request is or not allowed.

When you create an instance with `new AuthorizationDecision(true)`, it means you allow the request. If you create it with `new AuthorizationDecision(false)`, it means you disallow the request.

In listing 19.9, you find the two methods I have omitted in listing 19.8 for your convenience: `getRequestPath()` and `isAdmin()`. By omitting them, I let you focus on the logic used by the `access()` method. As you observe, the methods are simple. The `isAdmin()` returns a function that returns true for an `Authentication` instance having the `ROLE_ADMIN`. The `getRequestPath()` method simply returns the path of the request.

#### **Listing 19.9 The definition of the `getRequestPath()` and `isAdmin()` methods**

```
@Configuration
public class ProjectConfig {

    // Omitted code

    private String getRequestPath(AuthorizationContext c) {
        return c.getExchange()
            .getRequest()
            .getPath()
            .toString();
    }

    private Function<Authentication, Boolean> isAdmin() {
        return p ->
            p.getAuthorities().stream()
                .anyMatch(e -> e.getAuthority().equals("ROLE_ADMIN"));
    }
}
```

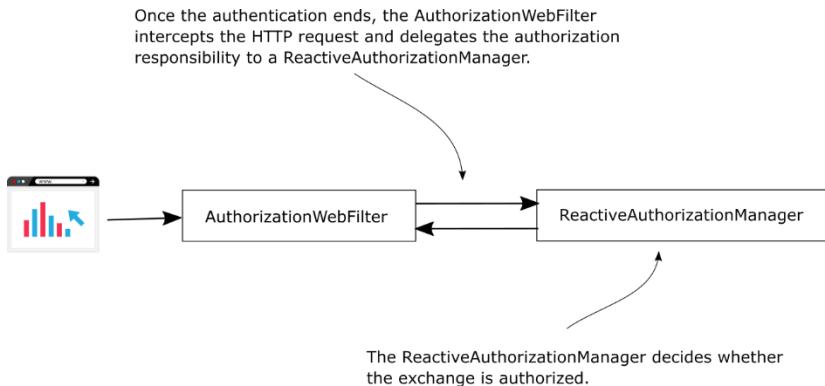
Running the application and calling the endpoint will either result in a response status 403 Forbidden if any of the authorization rules we applied isn't fulfilled or will simply display a message in the HTTP response body as presented in the next code snippet.

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

The response body is:

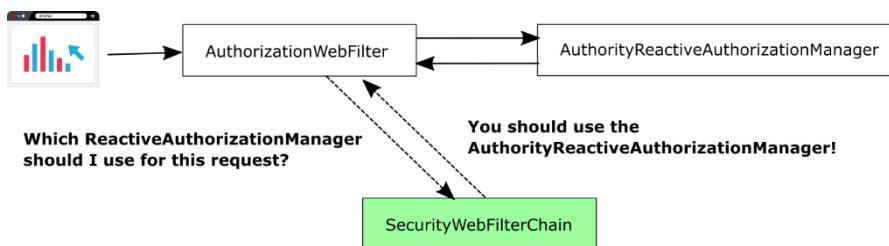
```
Hello john
```

What happened behind the scenes in the examples of this section? When the authentication ended, another filter intercepted the request. This filter, named the `AuthorizationWebFilter`, delegates the authorization responsibility to a `ReactiveAuthorizationManager` (figure 19.6).



**Figure 19.6** After the authentication process successfully ends, another filter named AuthorizationWebFilter intercepts the request. This filter delegates the authorization responsibility to a ReactiveAuthorizationManager.

Wait! Does this mean we only have a ReactiveAuthorizationManager? How does this component know how to authorize a request based on the configurations we made? No, there are actually multiple implementations of the ReactiveAuthorizationManager. The AuthorizationWebFilter uses the SecurityWebFilterChain bean we added to the Spring context. With the SecurityWebFilterChain, the filter decides to which ReactiveAuthorizationManager implementation to delegate the authorization responsibility (figure 19.7).



**Figure 19.7** The AuthorizationFilter uses the SecurityWebFilterChain bean we added to the context (shaded differently) to know which ReactiveAuthorizationManager to use.

### 19.3.2 Using method security in reactive apps

In this section, we discuss applying authorization rules for all the layers of the reactive apps. For non-reactive apps, we used Global Method Security and, in chapters 16 and 17, you learned different approaches to apply authorization rules at the method level. Being able to apply authorization rules not only at the endpoint layer offers you great flexibility and enables you to apply authorization also for non-web applications. To teach you how to use method security for reactive apps, we'll work on a separate example, which I'll name ssia-ch19-ex5.

Instead of Global Method Security, when working with non-reactive apps, we call Reactive Method Security the approach in which we apply authorization rules directly at the method level. Unfortunately, Reactive Method Security isn't a mature implementation yet and only enables us to use the `@PreAuthorize` and `@PostAuthorize` annotations. For the usage of `@PreFilter` and `@PostFilter` annotations, an issue was added for the Spring Security team back in 2018, but it didn't yet get implemented (<https://github.com/spring-projects/spring-security/issues/5249>).

For our example, we'll use `@PreAuthorize` to validate that a user has a specific role to call a test endpoint. To keep the example simple, I'll use the `@PreAuthorize` annotation directly over the method defining the endpoint. But you can use it the same way we discussed in chapter 16 for non-reactive apps. So you can actually use it on any other component's method in your reactive application. Listing 19.10 gives you the definition of the controller class. Observe that I used `@PreAuthorize` similar to what you've already learned in chapter 16. Using Spring Expression Language (SpEL), I declare that only an admin can call the annotated method.

#### **Listing 19.10 the definition of the controller class**

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    @PreAuthorize("hasRole('ADMIN')")      #A
    public Mono<String> hello() {
        return Mono.just("Hello");
    }
}
```

#A Using `@PreAuthorize` to restrict access to the method.

In listing 19.11, you find the configuration class in which I used the `@EnableReactiveMethodSecurity` annotation to enable the Reactive Method Security feature. Similar to the Global Method Security, we need to explicitly use an annotation to enable the Reactive Method Security feature. Besides this annotation, in the configuration class, you also find the usual user management definition.

#### **Listing 19.11 The configuration class**

```
@Configuration
@EnableReactiveMethodSecurity      #A
public class ProjectConfig {

    @Bean
    public ReactiveUserDetailsService userDetailsService() {
        var u1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")
            .build();

        var u2 = User.withUsername("bill")
            .password("12345")
            .roles("REGULAR_USER")
    }
}
```

```

    .build();

    var uds = new MapReactiveUserDetailsService(u1, u2);

    return uds;
}

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

}

```

#### #A Enabling the reactive method security feature.

You can now start the application and test the behavior of the endpoint by calling it with each of the users. You should observe that only John can call the endpoint because we defined him as an admin. Bill is just a regular user, so if we try to call the endpoint authenticating as Bill, we'll get back a response having the status HTTP 403 Forbidden.

Calling the `/hello` endpoint authenticating with user "john":

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

The response body is:

```
Hello
```

Calling the `/hello` endpoint authenticating with user "bill":

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

The response body is:

```
Denied
```

Behind the scenes, this functionality works the same as for non-reactive apps. In chapters 16 and 17, you learned that an aspect intercepts the call to the method and implements the authorization. If the call doesn't fulfill the pre-authorization rules you specified, the aspect doesn't delegate the call to the method (figure 19.8).

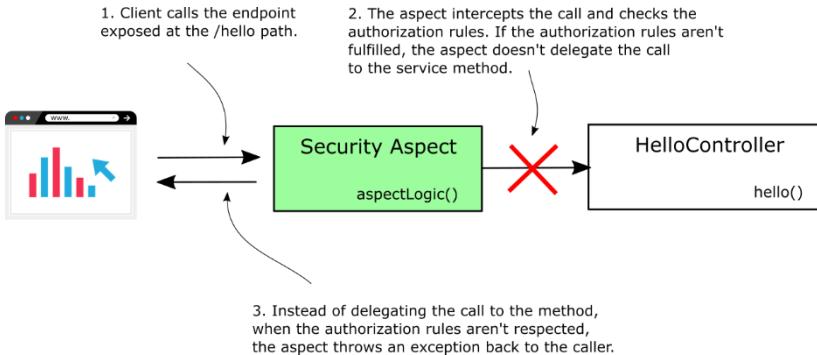


Figure 19.8 When using method security, an aspect intercepts the call to a protected method. If the call doesn't fulfill the pre-authorization rules, the aspect doesn't delegate the call to the method.

## 19.4 Reactive apps and OAuth 2

You're probably wondering by now if we could use reactive applications in a system designed over the OAuth 2 framework. In this section, we discuss implementing a resource server as a reactive app. You'll learn how to configure your reactive application to rely on an authentication approach implemented over OAuth 2. Because using OAuth 2 is so common nowadays, you, of course, have chances to encounter requirements where your resource server application needs to be designed reactive. I'll create a new project `ssia-ch19-ex6`, and we'll implement a reactive resource server application.

You need to add the dependencies in `pom.xml`, as presented in the next code snippet.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
  
```

We, of course, need an endpoint to test the application in the end. So we'll need to add a controller class as presented in the next code snippet.

```

@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello, World!");
    }
}
  
```

```

        return Mono.just("Hello!");
    }
}

```

And now, the most important part of the example, the security configuration. For this example, I'll configure the Resource Server to use the public key exposed by the Authorization Server for the token signature validation. This approach is the same we used in chapter 18 when we used Keycloak as our Authorization Server. I'll actually use the same configured server for this example. You can choose to do the same, or you can implement a custom Authorization Server, as we discussed in chapter 13.

To configure the authentication method, we use the `SecurityWebFilterChain`, as you learned in section 19.3. But instead of using `httpBasic()`, I'll call the `oauth2ResourceServer()` method. Then, by calling the `jwt()` method, I define the kind of token I use and using a `Customizer` object, I specify the way the token signature will be validated. In listing 19.12, you find the definition of the configuration class.

### **Listing 19.12 The configuration class**

```

@Configuration
public class ProjectConfig {

    @Value("${jwk.endpoint}")
    private String jwkEndpoint;

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http) {

        return http.authorizeExchange()
            .anyExchange().authenticated()
            .and().oauth2ResourceServer()      #A
            .jwt(jwtSpec -> {             #B
                jwtSpec.jwkSetUri(jwkEndpoint);
            })
            .and().build();
    }
}

```

#A Configuring the Resource Server authentication method.

#B Specifying the way the token is validated.

In the same way, I could've configured the public key instead of specifying an URI where the public key is exposed. The only change was to call the `publicKey()` method of the `jwtSpec` instance and provide a valid public key as a parameter. You can basically use any of the approaches we discussed in chapters 14 and 15, where we analyzed in detail approaches for the Resource Server to validate the access token.

I'll change the `application.properties` file to add the value for the key set URI as well as change the server port to 9090. This way, we allow Keycloak to run on 8080. In the next code snippet, you find the contents of the `application.properties` file.

```

server.port=9090
jwk.endpoint=http://localhost:8080/auth/realm/master/protocol/openid-connect/certs

```

Let's run and prove the app has the expected behavior. I'll generate an access token using the locally installed Keycloak server, as presented in the next code snippet.

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'client_id=fitnessapp' \
--data-urlencode 'scope=fitnessapp'
```

In the HTTP response body, you receive the access token.

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5c...",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "610f49d7-78d2-4532-8b13-285f64642caa",
  "scope": "fitnessapp"
}
```

Using the access token, I'll call the `/hello` endpoint of our application.

```
curl -H 'Authorization: Bearer
        eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJMSE9zT0VRSmJuTmJVbjhQbVpYQTlUVW9Q
        NTZoWU90YzNWT2swa1V2ajVVIn...' \
'http://localhost:9090/hello'
```

The response body is:

```
Hello!
```

## 19.5 Summary

- Reactive applications have a different style in which they process the data and exchange messages with other components. Reactive apps might be a better choice in specific situations, like cases in which we can split the data into separate smaller segments for processing and exchanging it.
- Like any other application, you also need to protect reactive apps by using security configuration. Spring Security offers an excellent set of tools you can use to apply security configurations for reactive apps as well as for the non-reactive ones.
- To implement user management in reactive apps with Spring Security, we use the `ReactiveUserDetailsService` contract. This component has the same purpose as `UserDetailsService` has for non-reactive apps: it tells the app how to get the user details.
- To implement the endpoint authorization rules for a reactive web application, you need to create an instance of type `SecurityWebFilterChain` and add it to the Spring context. You create the `SecurityWebFilterChain` instance by using the `ServerHttpSecurity` builder.
- Generally, the names of the methods you use to define the authorization configurations

are the same as for the methods you used for non-reactive apps. However, you'll find small differences in namings that are related to the reactive terminology. For example, instead of using `authorizeRequests()`, the name of its counterpart for reactive apps is `authorizeExchange()`.

- Spring Security also provides a way to define authorization rules at the method level for reactive apps. We call this approach Reactive Method Security, and it offers us great flexibility in applying the authorization rules at any layer of the reactive app. It is similar to what we have called Global Method Security for non-reactive apps.
- Reactive Method Security isn't, however, an implementation as mature as the Global Method Security for non-reactive apps. You can use the `@PreAuthorize` and the `@PostAuthorize` annotations already, but the functionality for `@PreFilter` and `@PostFilter` still waits to be implemented.

# 20

## *Spring Security testing*

### This chapter covers

- Testing integration with Spring Security configurations for endpoints.
- Defining mock users for tests.
- Testing integration with Spring Security for method level security.

The legend says that writing unit and integration tests started with a short verse:

“99 little bugs in the code,  
99 little bugs in the code.  
Track one down, patch it around,  
113 little bugs in the code.”

**Anonymous**

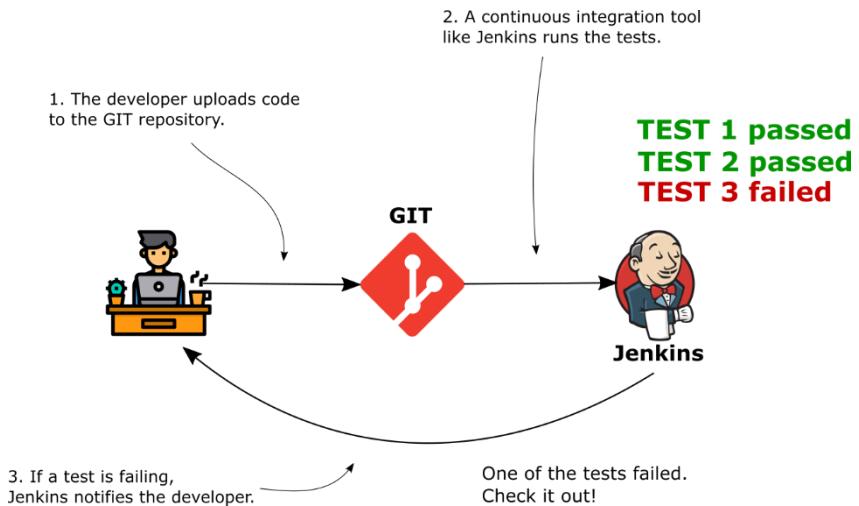
---

With time, software became more complex, and teams became larger. Knowing all the functionalities implemented by others over time became impossible. So, developers needed a way to make sure they don't break existing functionalities while correcting bugs or implementing new features.

While developing applications, we continuously write tests to validate that the functionalities we implement work as desired. The main reason why we write unit and integration tests is to make sure we don't break anything when working on new changes. Testing that we don't break existing functionalities while changing code for fixing a bug or implementing new features is also called *regression testing*.

Nowadays, when a developer finishes a change, they upload the changes to a server used by the team to manage the code versioning. This action automatically triggers a continuous integration tool that runs all the existing tests. If any of the changes broke an existing

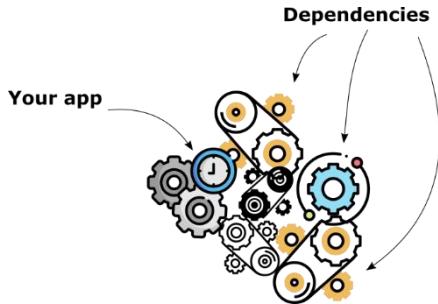
functionality, some tests would fail, and the continuous integration tool notifies the team (figure 20.1). This way, it's less probable to deliver changes that affect the existing features.



**Figure 20.1** Testing is part of the development process. Anytime a developer uploads code, the tests are run. If any test is failing, a continuous integration tool notifies the developer.

**NOTE** Of course, by using Jenkins in this example here, I say neither that this is the only continuous integration tool used and neither that it's the best one. You have many alternatives here to choose from, and you may encounter in the real world like Bamboo, GitLabCI, CircleCI, and so on.

When testing applications, you need to remember that it's not only the code of your app which you need to test. You need to make sure you test as well the integrations with the frameworks and libraries you use (figure 20.2). Sometime in the future, you'll maybe upgrade that framework or library to a new version. When changing the version of a dependency, you want to make sure your app still integrates fine with the new version of that dependency. If your app doesn't integrate the same, you want to easily find where you need to make changes to correct the integration problems.



**Figure 20.2** The functionality of an application relies on many dependencies. Whenever you upgrade or change a dependency, you might affect existing functionality. Having integration tests with the dependencies helps you to discover fast if a change in any dependency affected the existing functionality of your application.

So that's why you need to know what we'll discuss in this chapter: how to test your app integration with Spring Security. Spring Security, like the Spring framework ecosystem in general, evolves quickly. You'll probably upgrade your app to new versions, and you certainly want to be aware if upgrading to a specific version develops vulnerabilities, errors, or incompatibilities in your application. Remember what we discussed right from the first chapter: you need to consider security from the very first design for the app, and you need to take it seriously. Implementing tests for any of your security configurations should be a mandatory task and should be defined as part of your definition-of-done (you shouldn't consider a task finished if security tests aren't ready).

In this chapter, we'll discuss several practices for testing an app integration with Spring Security. We'll go back to some of the examples we worked on in the previous chapters of this book, and you'll learn how to write the integration tests for the implemented functionality.

Testing, in general, is a whole epic. Learning this subject in detail will bring you many benefits. In this chapter, we focus on testing the integration between an application and Spring Security. Before starting our examples, I'd like to recommend a few resources that helped me deeply understand this subject. So if you need to understand the subject more in detail, or even as a refresher, you can read these books. I am positive you'll find them great.

- JUnit in Action, Third Edition by Catalin Tudose et. al. (Manning, 2020)
- Unit Testing Principles, Practices, and Patterns by Vladimir Khorikov (Manning, 2020)
- Testing Java Microservices by Alex Soto Bueno et. al. (Manning, 2018)

Our adventure in writing tests for the security implementations starts with testing the authorization configurations.

1. In section 20.1, you'll learn how to skip authentication and define mock users to test authorization configuration at the endpoint level.
2. We'll then learn in section 20.2 how to test the authorization configurations with users from a `UserDetailsService`.
3. In section 20.3, we'll discuss how to set up the full `SecurityContext` in case you need to use specific implementations of the `Authentication` object.

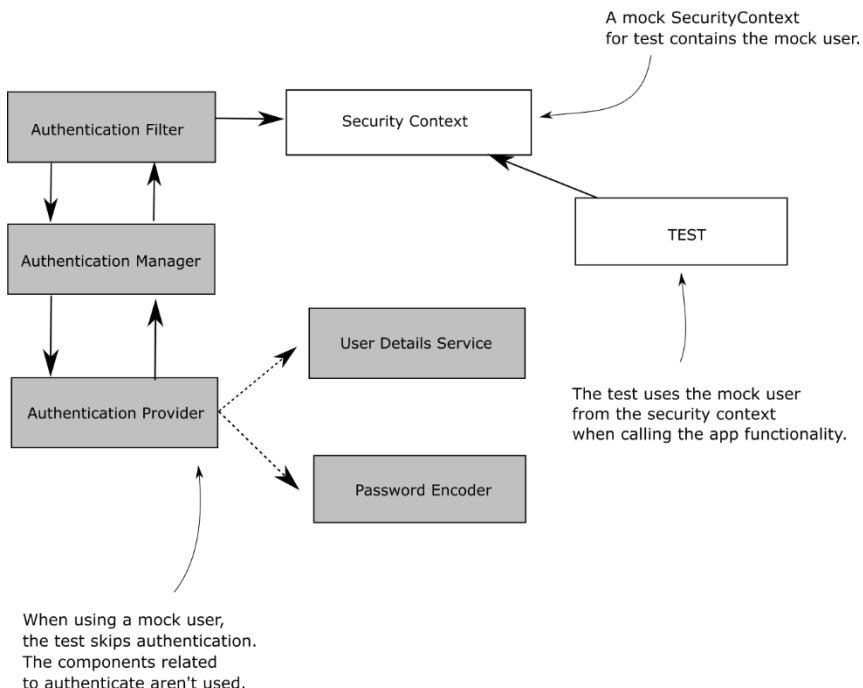
4. In section 20.4, you'll apply the approaches you learned in section 20.1, 20.2, and 20.3 to test authorization configuration on method security.

Once we completed discussing testing authorization, you'll learn in section 20.5 how to test the authentication flow. Then, in sections 20.6 and 20.7, we'll discuss testing other security configurations like the Cross-Site Request Forgery (CSRF) protection or Cross-Origin Resource Sharing (CORS) policies.

We'll end the chapter discussing integration tests of Spring Security with functionalities of reactive applications, in section 20.8.

## 20.1 Using mock users for test

In this section, we discuss using mock users to test the authorization configuration. This approach is the most straightforward but also the most used for testing authorization configurations. When using a mock user, the test completely skips the authentication (figure 20.3). The mock user is valid only for the test execution, and you can configure for this user any characteristics you need to validate a specific scenario. You could, for example, give specific roles to the user, ADMIN or MANAGER, or different authorities to validate that the app behaves as expected in these conditions.



**Figure 20.3** The components shaded differently from the Spring Security authentication flow are skipped when executing a test. The test directly uses a mock SecurityContext, which contains the mock user you define to call the tested functionality.

**NOTE** It's important to know which components from the framework are involved in the integration test. This way, you know what part of the integration you covered with the test. For example, using a mock user can only be used to cover authorization. In section 20.5, you'll learn how to deal with authentication. I've seen in my experience developers getting confused on this aspect. They thought they were also covering, for example, a custom implementation of an `AuthenticationProvider` when working with a mock user – which is not the case. Make sure you correctly understand what you test.

To prove how to write such a test, let's go back to the simplest example we've worked on in this book: `ssia-ch2-ex1`. This project exposes an endpoint for the path `/hello` with only the default Spring Security configuration. What do we expect to happen?

1. When calling the endpoint without a user, the HTTP response status should be 401 Unauthorized.
2. When calling the endpoint having an authenticated user, the HTTP response status should be 200 OK, and the response body should be "Hello!".

Let's test these two scenarios!

We'll need a couple of dependencies in the `pom.xml` file to write the tests. The next code snippet shows you the classes we'll use throughout the examples of this chapter. You should make sure you have them in your `pom.xml` file before starting to write the tests.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

**NOTE** In the examples of the book we use JUnit 5 for writing the tests. But don't be discouraged if you still work with JUnit 4. From the Spring Security integration point of view, the annotations and the rest of the classes you'll learn work the same. Chapter 4 of *JUnit in Action* by Catalin Tudose et. al. (Manning, 2020), dedicated to migrating from JUnit 4 to JUnit 5, contains some interesting tables that show the correspondence between classes and annotations of versions 4 and 5.

<https://livebook.manning.com/book/junit-in-action-third-edition/chapter-4>

In the `test` folder of the Spring Boot maven project, we'll add a class named `MainTests`. I'll write this class as part of the main package of the application. In my case, the name of the main package is `com.laurentiuspilca.ssia`. In listing 20.1, you find the definition of the

empty class for the tests. We use the `@SpringBootTest` annotation, which represents a convenient way to manage the Spring context for our tests suite.

#### **Listing 20.1 A class for writing the tests**

```
@SpringBootTest      #A
public class MainTests {
}
```

#A Making Spring Boot responsible for managing the Spring context for the tests.

A convenient way to implement a test for the behavior of an endpoint is using `MockMvc`. In a Spring Boot application, `MockMvc` utility for testing endpoint calls can be autoconfigured only by adding an annotation over the class as presented in listing 20.2.

#### **Listing 20.2 Adding `MockMvc` for implementing the test scenarios**

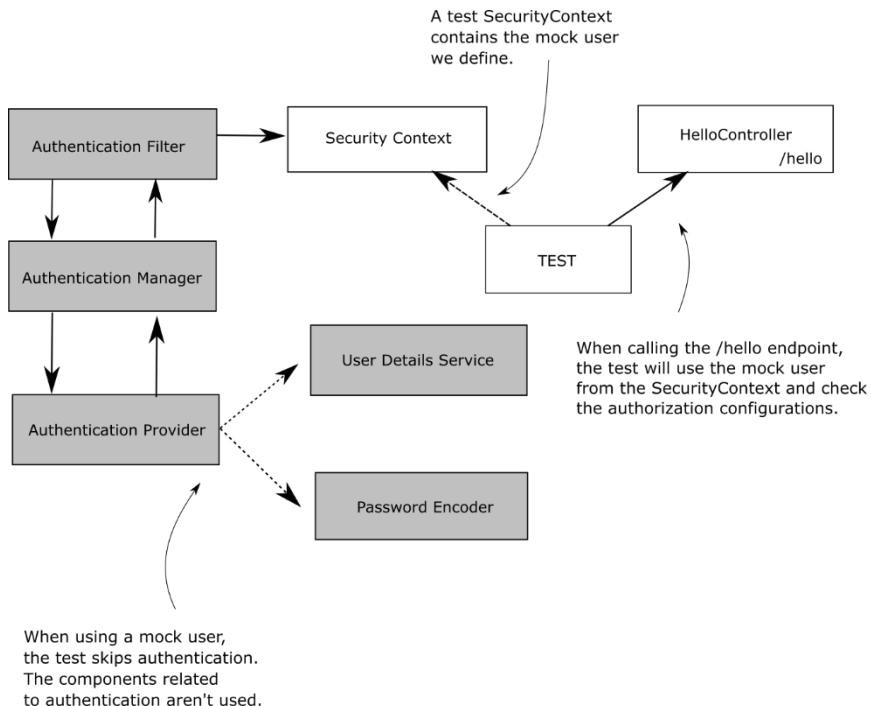
```
@SpringBootTest
@AutoConfigureMockMvc      #A
public class MainTests {
    @Autowired
    private MockMvc mvc;      #B
}
```

#A Enabling Spring Boot to autoconfigure `MockMvc`. As a consequence, an object of type `MockMvc` will be added to the Spring context.

#B Injecting the `MockMvc` object that we'll use to test the endpoint.

Now that we have a tool we can use to test the endpoint's behavior, let's get started with the first scenario. When calling the `/hello` endpoint without an authenticated user, the HTTP response status should be 401 Unauthorized.

You can visualize the relationship between the components in figure 20.4. The test calls the endpoint but uses a mock `SecurityContext`. We decide what we add to this `SecurityContext`. For this test, we need to check that if we don't add a user (which represents the situation in which someone calls the endpoint without authenticating), the app rejects the call with an HTTP response having the status 401 Unauthorized. When we add a user to the `SecurityContext`, the app accepts the call, and the HTTP response status will be 200 OK.



**Figure 20.4** When running the test, the authentication is skipped. The test uses a mock SecurityContext and calls the /hello endpoint exposed by HelloController. We add a mock user in the test SecurityContext to verify the behavior is correct according to the authorization rules. If we don't define a user, we expect the app doesn't authorize the call, while when we define a user, we expect the call is successful.

In listing 20.3, you find the implementation of this scenario.

#### Listing 20.3 Testing that you can't call the endpoint without an authenticated user

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void helloUnauthenticated() throws Exception {
        mvc.perform(get("/hello")) #A
            .andExpect(status().isUnauthorized());
    }
}
```

#A When performing a GET request for the /hello path, we expect to get back a response with status unauthorized.

Mind that I statically imported the methods `get()` and `status()`. You find the method `get()` (and similar methods related to the request which I used in the further examples of this chapter) in class `org.springframework.test.web.servlet.request.MockMvcRequestBuilders`. Also, you find the method `status()` (and similar methods related to the result of the call, which I used in the next examples of this chapter) in class `org.springframework.test.web.servlet.result.MockMvcResultMatchers`.

You can run the tests now and see the status in your IDE. Usually, in any IDE, to run the tests, you can right-click on the tests class and then select Run. The IDE displays a successful test with green and a failing one with another color (usually red or yellow).

**NOTE** In the projects provided with the book, above each method implementing a test, I've also used the `@DisplayName` annotation. This annotation allows us to have a longer, more detailed description of the test scenario. To occupy less space and allow you to focus on the functionality of the tests we discuss, I took the `@DisplayName` annotation out from the listings in the book.

To test the second scenario, we'll need to use a mock user. To validate the behavior of calling the `/hello` endpoint with an authenticated user, we'll use the `@WithMockUser` annotation. By adding this annotation above the test method, we instruct Spring to set up a `SecurityContext`, which contains a `UserDetails` implementation instance. It's basically skipping authentication. Now, calling the endpoint behaves like the user defined with the `@WithMockUser` annotation has successfully authenticated.

With this very simple example, we don't care about the details of the mock user like its username, roles, or authorities. So we simply add the `@WithMockUser` annotation, which will provide us some defaults for the mock user's attributes. Later in this chapter, you'll learn to configure the user's attributes for test scenarios in which their values are important.

In listing 20.4, you find the implementation for the second test scenario.

#### Listing 20.4 Using `@WithMockUser` to define a mock authenticated user

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    // Omitted code

    @Test
    @WithMockUser    #A
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))    #B
            .andExpect(content().string("Hello!"))
            .andExpect(status().isOk());
    }
}
```

#A We specify the method should be called with a mock authenticated user.

#B In this case, when performing a GET request for the /hello path, we expect the response status to be OK.

Run this test now and observe it's successful. But in some situations, to implement the test, we need to use a specific name or give the user specific roles or authorities. Say we want to test the endpoints we defined for example `ssia-ch5-ex2`. For this example, the endpoints return a body depending on the authenticated user's name. So, to write the test, we need to give the user a known username. Listing 20.5 shows you how to configure the details of the mock user by writing a test for the `/hello` endpoint of the `ssia-ch5-ex2` project.

### **Listing 20.5 Configuring details for the mock user**

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    // Omitted code

    @Test
    @WithMockUser(username = "mary")      #A
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(content().string("Hello, mary!"))
            .andExpect(status().isOk());
    }
}
```

#A Setting up a username for the mock user.

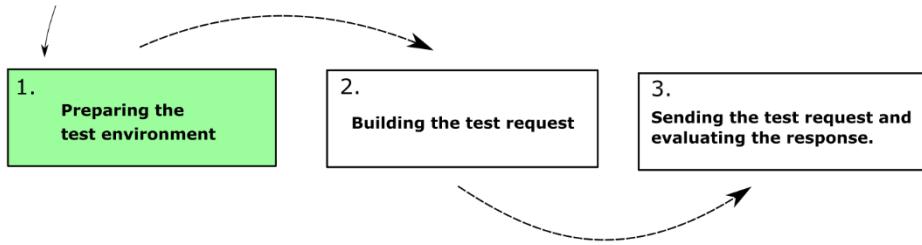
Like setting up the username, you could set the authorities and roles for testing authorization rules.

An alternative approach to creating a mock user is using a `RequestPostProcessor`. We can provide a `RequestPostProcessor` for using the `with()` method as presented in listing 20.6. The class `SecurityMockMvcRequestPostProcessors` provided by Spring Security offers us a bunch of implementations for `RequestPostProcessor`, which helps us cover various test scenarios.

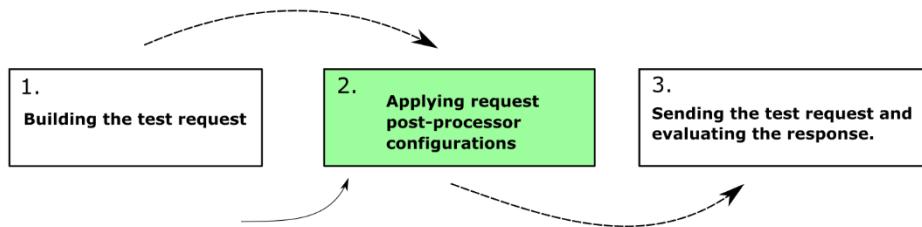
In figure 20.5, you find a comparison between how using annotations to define the test security environment differs from using a `RequestPostProcessor`. The framework interprets annotations like `@WithMockUser` before it executes the test method. This way, the test method creates the test request and executes it in an already configured security environment. When using a `RequestPostProcessor`, the framework first calls the test method and builds the test request. The framework then applies the `RequestPostProcessor`, which alters the request or the environment in which it's executed before sending it. So, in this case, the framework configures the test dependencies like the mock users and the `SecurityContext` after building the test request.

### Configuring test security environment through annotations

An aspect intercepts the test method and configures the SecurityContext and the mock users according to the annotations you specified.



### Using a RequestPostProcessor to define the security environment



After building the test request, a RequestPostProcessor creates the SecurityContext and the mock users according to the configurations you specified.

**Figure 20.5 The difference between using annotations and a RequestPostProcessor to create the test security environment:** When using annotations, first, the framework sets up the test security environment. When using a RequestPostProcessor, the test request is created and then changed to define other constraints, like the test security environment. In the figure, I shaded differently the point where the framework applies the test security environment.

In this chapter, we'll also discuss the most used implementations for RequestPostProcessor. The method user() of class SecurityMockMvcRequestPostProcessors returns a RequestPostProcessor we can use here as an alternative to the @WithMockUser annotation.

#### Listing 20.6 Using a RequestPostProcessor to define a mock user

```

@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    // Omitted code

    @Test
    public void helloAuthenticatedWithUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(user("mary")))
    }
}
  
```

```

        .andExpect(content().string("Hello!"))
        .andExpect(status().isOk());
    }
}

```

#A Call the /hello endpoint using a mock user with username mary

As you observed in this section, writing tests for the authorization configurations is fun and simple! Most of the tests you'd write for the Spring Security integration with functionalities of your application are for authorization configurations. Maybe you wonder now why didn't we also test the authentication. In section 20.5, we'll discuss testing authentication. But in general, it makes sense to test authorization and authentication separately. Usually, an app has one way to authenticate the users but might expose dozens of endpoints for which authorization is configured differently. That's why, usually, you'll test the authentication separately with a hand of tests and then implement individually tests for each authorization configuration for the endpoints. It'd be a loss of execution time to repeat authentication for each endpoint tested as long as the logic doesn't change.

## 20.2 Testing with users from a UserDetailsService

In this section, we discuss obtaining the user details for tests from a `UserDetailsService`. This approach is an alternative to creating a mock user. The difference is that, instead of creating a fake user, this time, the user needs to be obtained from a given `UserDetailsService`. You will use this approach if you'd like to also test the integration with the data source from where your app loads the user details (figure 20.6).

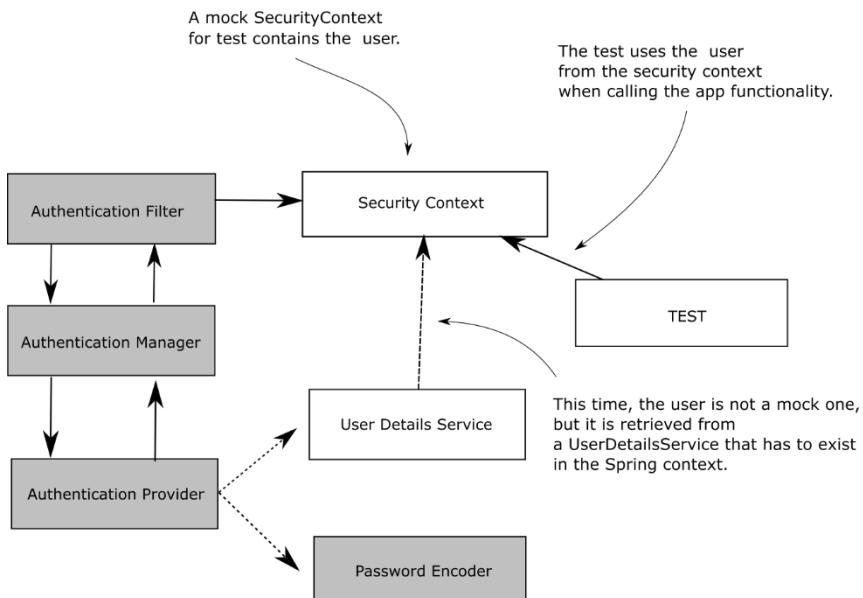


Figure 20.6 Instead of creating a mock user for the test, when creating the `SecurityContext` used by the test,

the user details are taken from a `UserDetailsService`. This way, you can test the authorization using real users taken from a data source. Components shaded differently are skipped from the flow execution during the test.

To demonstrate this approach, let's open project `ssia-ch2-ex2` and implement the tests for the endpoint exposed at the `/hello` path. We'll use the `UserDetailsService` bean that the project already adds to the context. Mind that with this approach, we need to have a `UserDetailsService` bean in the context. To specify the user we authenticate from this `UserDetailsService`, we'll annotate the test method with `@WithUserDetails`. With the `@WithUserDetails` annotation, you specify the username for the user to be found. In listing 20.7, you find the implementation of the test for the `/hello` endpoint using the `@WithUserDetails` annotation to define the authenticated user.

#### **Listing 20.7 Defining the authenticated user with the `@WithUserDetails` annotation**

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

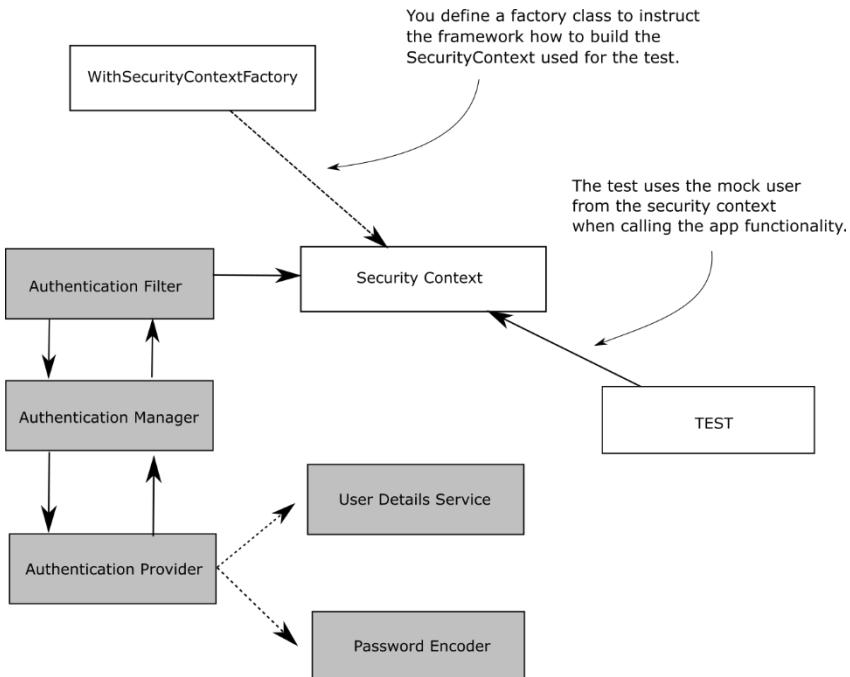
    @Test
    @WithUserDetails("john")      #A
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(status().isOk());
    }
}
```

#A Loading user John using the `UserDetailsService` for running the test scenario.

### **20.3 Using custom Authentication objects for testing**

Generally, when using a mock user for a test, you don't care which class the framework used to create the `Authentication` instances in the `SecurityContext`. But say you have some logic in the controller that depends on the type of the object. Could you somehow instruct the framework to create the `Authentication` object for the test using a specific type? The answer is yes, and this is what we're discussing in this section.

The logic behind this approach is also very simple. We define a factory class responsible for building the `SecurityContext`. This way, we have full control over how the `SecurityContext` for the test is built, including what's inside it (figure 20.7). So we can choose to have a custom `Authentication` object, for example.



**Figure 20.7** To obtain full control of how the `SecurityContext` for the test is defined, we build a factory class that instructs the test on how to build the `SecurityContext`. This way, we gain great flexibility, and we can choose details like the kind of object used as Authentication. The components skipped from the flow during the test are shaded differently.

Let's open project `ssia-ch2-ex5` and write a test in which we configure ourselves the mock `SecurityContext` and instruct the framework on how to create the `Authentication` object. An interesting aspect to remember about this example is that we used it to prove the implementation of a custom `AuthenticationProvider`. The custom `AuthenticationProvider` we implemented only authenticates a user named "john". However, as in the other two previous approaches we discussed in section 20.1 and 20.2, the current approach you learn also skips authentication. For this reason, you'll see at the end of the example we can actually give any name to our mock user.

We have three steps we follow to achieve this behavior (figure 20.8):

1. Write an annotation which we'll use over the test, similarly to the way we're using `@WithMockUser` or `@WithUserDetails`.
2. Write a class that implements the `WithSecurityContextFactory` interface. This class implements the `createSecurityContext()` method, which returns the mock `SecurityContext` the framework will use for the test.
3. Link the custom annotation created in step 1 with the factory class created in step 2 via the `@WithSecurityContext` annotation.

### Step 1

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
public @interface WithCustomUser {

    String username();
}
```

We define a custom annotation we'll use for the test methods for which we need to customize the test SecurityContext.

### Step 2

```
WithSecurityContextFactory

public class CustomSecurityContextFactory
    implements WithSecurityContextFactory<WithCustomUser> {

    @Override
    public SecurityContext createSecurityContext() {
        ...
    }
}
```

We implement a factory class in which we build the custom test SecurityContext. The framework uses this factory class to build the SecurityContext when it finds a test method annotated with the custom annotation we created in Step 1.

### Step 3

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext
(factory = CustomSecurityContextFactory.class)
public @interface WithCustomUser {

    String username();
}
```

We link the custom annotation created in step 1 to the factory class created in step 2. This way, the framework knows which factory class to use to create the test SecurityContext for a test method annotated with our custom annotation.

Figure 20.8 To enable the test to use a custom SecurityContext, you need to follow three steps.

#### STEP 1 – DEFINING A CUSTOM ANNOTATION

In listing 20.8, you find the definition of the custom annotation we define for the test. I'll name this annotation `@WithCustomUser`. As properties of the annotation, you can define whatever details you'll need to create the mock Authentication object. I have added only the `username` here for my demonstration. Also, don't forget to use the annotation `@Retention(RetentionPolicy.RUNTIME)` to set the retention policy to runtime. Spring needs to read this annotation using Java reflection at runtime. To allow Spring to read this annotation at runtime, you need to change its retention policy to runtime.

#### Listing 20.8 Defining the `@WithCustomUser` annotation

```
@Retention(RetentionPolicy.RUNTIME)
public @interface WithCustomUser {

    String username();
}
```

## STEP 2 – CREATING A FACTORY CLASS FOR THE MOCK SECURITYCONTEXT

The second step consists in implementing the code that builds the `SecurityContext`, which the framework will use for the test execution. Here's where we decide what kind of Authentication do we use for the test. Listing 20.9, demonstrates the implementation of the factory class.

### **Listing 20.9 The implementation of a factory for the `SecurityContext`**

```
public class CustomSecurityContextFactory #A
    implements WithSecurityContextFactory<WithCustomUser> {

    @Override #B
    public SecurityContext createSecurityContext(
        WithCustomUser withCustomUser) {
        SecurityContext context = #C
            SecurityContextHolder.createEmptyContext();

        var a = new UsernamePasswordAuthenticationToken(
            withCustomUser.username(), null, null); #D

        context.setAuthentication(a); #E

        return context;
    }
}
```

#A Implementing the `WithSecurityContextFactory` annotation and specifying the custom annotation we'll use for the tests.

#B We implement the `createSecurityContext()` method to define how the `SecurityContext` for the test is created.

#C We build an empty security context.

#D We create an authentication instance.

#E We add the mock Authentication to the `SecurityContext`.

## STEP 3 – LINK THE CUSTOM ANNOTATION TO THE FACTORY CLASS

Using the `@WithSecurityContext` annotation, we now link the custom annotation we created in step 1 to the factory class for the `SecurityContext` we implemented in step 2. Listing 20.10 presents the change I've made to our `@WithCustomUser` annotation to link it to the `SecurityContext` factory class.

### **Listing 20.10 Linking the custom annotation to the `SecurityContext` factory class**

```
@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = CustomSecurityContextFactory.class)
public @interface WithCustomUser {

    String username();
}
```

With all this setup complete, we can write a test to use the custom way we've defined for building the `SecurityContext`.

### **Listing 20.11 Writing a test that uses the custom `SecurityContext`**

```

@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    @WithCustomUser(username = "mary")      #A
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(status().isOk());
    }
}

```

#### #A Executing the test with a user having the username “mary”

Running the test, you’ll observe a successful result. You might think, “Wait! In this example, we implemented a custom `AuthenticationProvider`, which only authenticates a user john. How could the test be successful with the username mary?”. Like in the case of `@WithMockUser` and `@WithUserDetails`, with this method, we skip the authentication logic. So you could use it only to test what’s related to authorization and onwards.

## 20.4 Testing method security

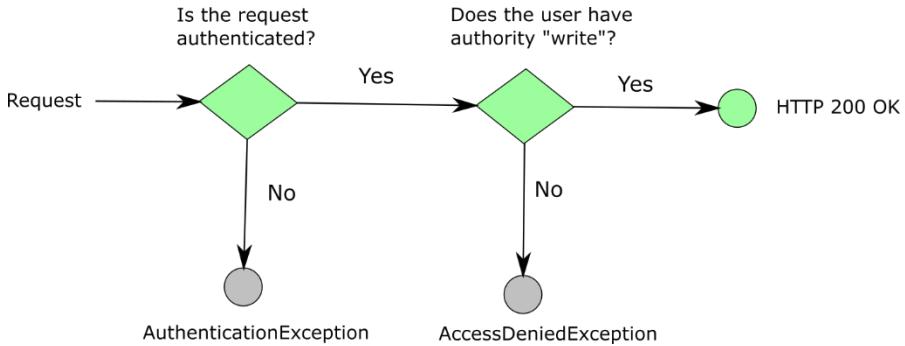
In this section, we discuss testing method security. All the tests we’ve written until now in this chapter refer to endpoints. But what if your application doesn’t have endpoints? If it’s not a web app, it doesn’t have endpoints at all. But you might have used Spring Security through Global Method Security, as we discussed in chapters 16 and 17. You still need to test your security configurations in such scenarios.

Fortunately, you do this by using the same approaches we discussed in section 20.1, 20.2, and 20.3. You can still use `@WithMockUser`, `@WithUserDetails`, or a custom annotation to define your own `SecurityContext`. Just that, instead of using `MockMvc`, you directly inject from the context the class defining the method you need to test.

Let’s open project `ssia-ch16-ex1` and implement the tests for the `getName()` method in the `NameService` class. We protected the `getName()` method using the `@PreAuthorize` annotation. In listing 20.12, you find the implementation of the test class with its three tests:

1. Calling the method without an authenticated user, the method should throw `AuthenticationException`.
2. Calling the method with an authenticated user that has an authority different than the expected one (write), the method should throw `AccessDeniedException`.
3. Calling the method with an authenticated user that has the expected authority returns the expected result.

Figure 20.9 represents graphically the three scenarios we test.



**Figure 20.9** The tested scenarios: If the HTTP request is not authenticated, the expected result is an `AuthenticationException`. If the HTTP request is authenticated but the user doesn't have the expected authority, the expected result is an `AccessDeniedException`. If the authenticated user has the expected authority, the call is successful.

#### **Listing 20.12 Implementation of the three test scenarios for the `getName()` method**

```

@SpringBootTest
class MainTests {

    @Autowired
    private NameService nameService;

    @Test
    void testNameServiceWithNoUser() {
        assertThrows(AuthenticationException.class,
                    () -> nameService.getName());
    }

    @Test
    @WithMockUser(authorities = "read")
    void testNameServiceWithUserButWrongAuthority() {
        assertThrows(AccessDeniedException.class,
                    () -> nameService.getName());
    }

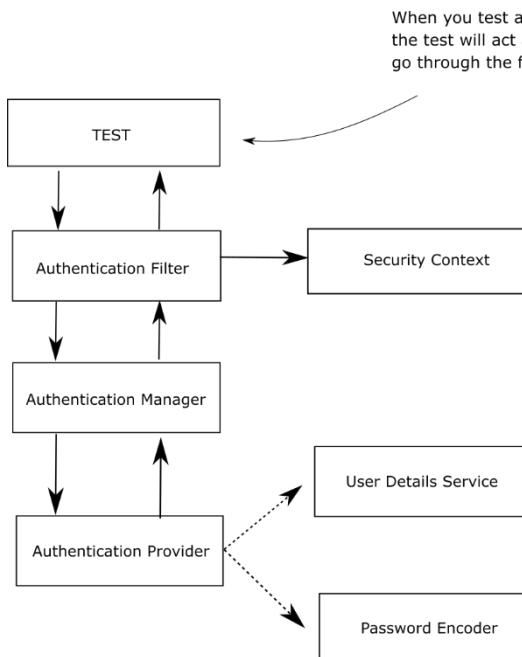
    @Test
    @WithMockUser(authorities = "write")
    void testNameServiceWithUserButCorrectAuthority() {
        var result = nameService.getName();

        assertEquals("Fantastico", result);
    }
}
  
```

You observe we don't configure `MockMvc` anymore because we don't need to call any endpoint. Instead, we directly injected the `NameService` instance to call the tested method. We use the `@WithMockUser` annotation as we discussed it in section 20.1. Similarly, you could have used the `@WithUserDetails` as we discussed in section 20.2 or design a custom way to build the `SecurityContext`, as discussed in section 20.3.

## 20.5 Testing authentication

In this section, we discuss testing authentication. Previously in this chapter, you learned how to define mock users and test the authorization configurations. But what about authentication? Could we also test the authentication logic? You'd need to do this if, for example, you have custom logic implemented for your authentication, and you want to make sure the entire flow works. When testing authentication, the test implementation requests work like normal client requests, as presented in figure 20.10.



When you test authentication,  
the test will act as a client and  
go through the full flow.

**Figure 20.10** When testing authentication, the test will act as a client and go through the full Spring Security flow we discussed throughout the book. This way, you'd also be able to test, for example, your custom `AuthenticationProvider` objects.

For example, going back to project `ssia-ch2-ex5`, could we prove that the custom authentication provider we implemented works correctly and secure it with tests? In this project, we implemented a custom `AuthenticationProvider`, and we'd like to make sure that we secure this custom authentication logic as well with tests. Yes, we can test the authentication logic as well. The logic we implemented is straightforward. Only one set of credentials is accepted: username "john" and the password "12345". We need to prove that using valid credentials, the call is successful, while using some other credentials, the HTTP response status is 401 Unauthorized. Let's open again project `ssia-ch2-ex5` and implement a couple of tests to validate that authentication behaves correctly.

### **Listing 20.13 Using the httpBasic() RequestPostProcessor to test the HTTP Basic authentication**

```

@SpringBootTest
@AutoConfigureMockMvc
public class AuthenticationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void helloAuthenticatingWithValidUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(httpBasic("john", "12345"))      #A
                .andExpect(status().isOk());
    }

    @Test
    public void helloAuthenticatingWithInvalidUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(httpBasic("mary", "12345"))      #B
                .andExpect(status().isUnauthorized());
    }
}

```

#A Authenticating with correct credentials.

#B Authenticating with wrong credentials.

Using the `httpBasic()` request post processor, we instruct the test to execute the authentication. This way, we validate the behavior of the endpoint when authenticating using valid or wrong credentials.

You can use the same approach to test the authentication with a form login. Let's open project `ssia-ch5-ex4`, where we used form login for authentication, and write some tests to prove the authentication works correctly. The scenarios we test are the following:

1. Test the app behavior when authenticating with a wrong set of credentials.
2. Test the app behavior when authenticating with a valid set of credentials, but the user doesn't have a valid authority according to the implementation we wrote in the `AuthenticationSuccessHandler`.
3. Test the app behavior when authenticating with a valid set of credentials, and a user that has a valid authority according to the implementation we wrote in the `AuthenticationSuccessHandler`.

In listing 20.14, you find the implementation for the first scenario. If we authenticate using invalid credentials, the app doesn't authenticate the user and adds the header "failed" to the HTTP response. We customized the app adding the "failed" header with an `AuthenticationFailureHandler` when discussing authentication back in chapter 5.

### **Listing 20.14 Testing form login unsuccessful authentication**

```

@SpringBootTest
@AutoConfigureMockMvc

```

```
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void loggingInWithWrongUser() throws Exception {
        mvc.perform(formLogin()      #A
                    .user("joey").password("12345"))
                    .andExpect(header().exists("failed"))
                    .andExpect(unauthenticated());
    }
}
```

#A Authenticating using form login with an invalid set of credentials.

Back in chapter 5, we customized the authentication logic using an `AuthenticationSuccessHandler`. In our implementation, if the user has the authority `read`, the app redirects them to the page `/home`. Otherwise, the app redirects the user to the page `/error`. In listing 20.15, you find the implementation of these two scenarios.

#### **Listing 20.15 Testing the app behavior when authenticating with existing users**

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    // Omitted code

    @Test
    public void loggingInWithWrongAuthority() throws Exception {
        mvc.perform(formLogin()
                    .user("mary").password("12345")
                    )
                    .andExpect(redirectedUrl("/error"))      #A
                    .andExpect(status().isFound())
                    .andExpect(authenticated());
    }

    @Test
    public void loggingInWithCorrectAuthority() throws Exception {
        mvc.perform(formLogin()
                    .user("bill").password("12345")
                    )
                    .andExpect(redirectedUrl("/home"))      #B
                    .andExpect(status().isFound())
                    .andExpect(authenticated());
    }
}
```

#A When authenticating with a user that doesn't have the `read` authority, the app redirects the user to path `/error`  
#B When authenticating with a user that has the `read` authority, the app redirects the user to path `/home`

## 20.6 Testing CSRF configurations

In this section, we discuss testing the Cross-Site Request Forgery (CSRF) protection configuration for your application. When an app presents a CSRF vulnerability, an attacker can fool the user to take actions they don't want once they're logged into the application. As we discussed in chapter 10, Spring Security uses CSRF tokens to mitigate these vulnerabilities. This way, for any mutating operation (POST, PUT, DELETE), the request needs to have a valid CSRF token in its headers.

Of course, at some point, you'll need to test more than HTTP GET requests. Depending on how you implemented your application, as we discussed in chapter 10, you might need to test also the CSRF protection. You need to make sure it works as expected and protects the endpoint that implements mutating actions.

Fortunately, Spring Security provides an easy approach to test the CSRF protection using a `RequestPostProcessor`. Let's open project `ssia-ch10-ex1` and test that CSRF protection is enabled for endpoint `/hello` when called with HTTP POST. We want to test the following scenarios:

1. When calling the `/hello` path using HTTP POST, if we don't use a CSRF token, the HTTP response status is 403 Forbidden.
2. When calling the `/hello` path using HTTP POST, if we send a CSRF token, the HTTP response status is 200 OK.

Listing 20.16 shows you the implementation of these two scenarios. Observe how we can send a CSRF token in the response simply by using the `csrf()` `RequestPostProcessor`.

### Listing 20.16 Implementing the CSRF protection test scenarios

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testHelloPOST() throws Exception {
        mvc.perform(post("/hello"))      #A
            .andExpect(status().isForbidden());
    }

    @Test
    public void testHelloPOSTWithCSRF() throws Exception {
        mvc.perform(post("/hello").with(csrf()))    #B
            .andExpect(status().isOk());
    }
}
```

#A When calling the endpoint without a CSRF token, the HTTP response status is 403 Forbidden.

#B When calling the endpoint with a CSRF token, the HTTP response status is 200 OK.

## 20.7 Testing CORS configurations

In this section, we discuss testing Cross-Origin Resource Sharing (CORS) configurations. As you've learned in chapter 10, if a browser loads a web app from one origin (say `example.com`), the browser won't allow the app to use an HTTP response that comes from a different origin (say `example.org`). We use CORS policies to relax these restrictions. This way, we might configure our application to work with multiple origins.

Of course, like for any other security configurations, you'll need to test as well the CORS policies. In chapter 10, you learned that CORS is about specific headers on the response whose values define whether the HTTP response is accepted. Two of these headers related to CORS specifications are the `Access-Control-Allow-Origin` and the `Access-Control-Allow-Methods`. We used these headers in chapter 10 to configure multiple origins for our app.

All we need to do when writing tests for the CORS policies is to make sure that these headers (and maybe other CORS related headers depending on the complexity of our configurations), exist and have the correct values. To do this validation, we can act precisely as the browser does when making a preflight request. We make a request using the HTTP OPTIONS method requesting the value for the CORS headers. Let's open project `ssia-ch10-ex4` and write a test to validate the values for the CORS headers. Listing 20.17 shows the definition of the test.

### Listing 20.17 Test implementation for CORS policies

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testCORSForTestEndpoint() throws Exception {
        mvc.perform(options("/test")      #A
                    .header("Access-Control-Request-Method", "POST")
                    .header("Origin", "http://www.example.com"))
            #B
            .andExpect(header().exists("Access-Control-Allow-Origin"))
            .andExpect(header().string("Access-Control-Allow-Origin", "*"))
            .andExpect(header().exists("Access-Control-Allow-Methods"))
            .andExpect(header().string("Access-Control-Allow-Methods", "POST"))
            .andExpect(status().isOk());
    }
}
```

#A We perform an HTTP OPTIONS request on the endpoint requesting the value for the CORS headers.  
#B We validate the values for the header are according to the configuration we made in the app.

## 20.8 Testing reactive Spring Security implementations

In this section, we discuss testing the integration of Spring Security with functionalities developed within a reactive app. You won't be surprised to find out that Spring Security

provides support for testing security configurations also for reactive applications. Like in the case of non-reactive applications, security for reactive apps is a crucial aspect. So testing the security configurations is also essential. To show you how to implement the tests for your security configurations, we'll go back to the examples we worked on in chapter 19.

With Spring Security for reactive applications, you need to know two approaches for writing your tests:

1. Using mock users with `@WithMockUser` annotations.
2. Using a `WebTestClientConfigurer`.

Using the `@WithMockUser` annotation is straightforward because it works the same as for non-reactive apps, as we discussed in section 20.1. The definition of the test is different because, being a reactive app, we can't use `MockMvc` anymore. But this change isn't related to Spring Security. We'll be able to use something similar when testing reactive apps, a tool named `WebTestClient`. In listing 20.18, you find the implementation of a simple test making use of a mock user to verify the behavior of a reactive endpoint.

#### **Listing 20.18 Using the `@WithMockUser` when testing reactive implementations**

```
@SpringBootTest
@AutoConfigureWebTestClient      #A
class MainTests {

    @Autowired      #B
    private WebTestClient client;

    @Test
    @WithMockUser    #C
    void testCallHelloWithValidUser() {
        client.get()      #D
            .uri("/hello")
            .exchange()
            .expectStatus().isOk();
    }
}
```

#A We request Spring Boot to autoconfigure the `WebTestClient` we'll use for the tests.

#B We inject the `WebTestClient` instance configured by Spring Boot from Spring context.

#C We use the `@WithMockUser` annotation to define a mock user for the test.

#D We make the exchange and validate the result.

As you observe, using the `@WithMockUser` annotation is pretty much the same as for non-reactive apps. The framework creates a `SecurityContext` with the mock user. The application skips the authentication process and uses the mock user from the test's `SecurityContext` to validate the authorization rules.

The second approach you could use is a `WebTestClientConfigurer`. This approach is very similar to using the `RequestPostProcessor` in case of a non-reactive app. In the case of a reactive app, for the `TestWebClient` we use, we set a `WebTestClientConfigurer`, which helps us mutating the test context. For example, we can define the mock user or send a CSRF token to test the CSRF protection as we did for non-reactive apps in section 20.6.

Listing 20.19 shows you how to use a `WebTestClientConfigurer`.

**Listing 20.19 Using a WebTestClientConfigurer to define a mock user for the call**

```

@SpringBootTest
@AutoConfigureWebTestClient
class MainTests {

    @Autowired
    private WebTestClient client;

    // Omitted code

    @Test
    void testCallHelloWithValidUserWithMockUser() {
        client.mutateWith(mockUser())      #A
            .get()
            .uri("/hello")
            .exchange()
            .expectStatus().isOk();
    }
}

```

#A Before executing the GET request, we mutate the call to use a mock user.

Assuming you were testing the CSRF protection on a POST call, you'd have written something similar to what's presented by the next code snippet:

```

client.mutateWith(csrf())
    .post()
    .uri("/hello")
    .exchange()
    .expectStatus().isOk();

```

**Mocking dependencies**

It often happens that our functionalities rely on external dependencies. Security-related implementations also rely sometimes on external dependencies. Some examples are databases we use to store users' credentials, authentication keys, or tokens. External applications also represent dependencies. Like in the case of an OAuth 2 system, where the Resource Server needs to call the token introspection endpoint of an Authorization Server to get details about an opaque token.

When we deal with such cases, we usually create mocks for dependencies. For example, instead of finding the user from a database, you'd mock the repository and make its methods return what you consider appropriate for the test scenarios you implement.

In the projects we worked on within this book, you find some examples where I've mocked dependencies. You might be interested in taking a look at the following:

- 1) In project `ssia-ch6-ex1`, I mocked the repository to enable me to test the authentication flow. This way, I don't need to rely on a real database to get the users, but I still manage to test the authentication flow with all its components integrated.
- 2) In project `ssia-ch11-ex1-s2`, I mocked the proxy to test the two authentication steps without needing to rely on the application implemented in project `ssia-ch11-ex1-s1`.
- 3) In project `ssia-ch14-ex1-rs`, I used a tool named `WireMockServer` to mock the Authorization Server's token introspection endpoints.

Different testing frameworks offer us different solutions for creating mocks or stubs to fake the dependencies on which our functionalities rely. Even if this is not directly related to Spring Security, I wanted to make you aware of the subject and its importance. Here are a few resources where you could continue with studying this subject:

Chapter 8 of JUnit in Action by Catalin Tudose et. al. (Manning, 2020)

<https://livebook.manning.com/book/junit-in-action-third-edition/chapter-8>

Chapters 5 and 9 of Unit Testing Principles, Practices, and Patterns by Vladimir Khorikov (Manning, 2020)

<https://livebook.manning.com/book/unit-testing/chapter-5>

<https://livebook.manning.com/book/unit-testing/chapter-9>

## 20.9 Summary

- Writing tests is a best practice. You write tests to make sure your new implementations or fixes don't break existing functionalities.
- You need not only to test your code, but also the integration with libraries and frameworks you use.
- Spring Security offers excellent support for implementing tests for your security configurations.
- You can test the authorization directly by using mock users. We write separate tests for authorization without authentication because, in general, you have fewer ways to authenticate the users. It saves execution time to test the authentication in separate tests (which are less) and the test individually the authorization configuration for your endpoints and methods.
- To test the security configurations for endpoints in non-reactive apps, Spring Security offers excellent support for writing your tests with `MockMvc`.
- To test the security configurations for endpoints in reactive apps, Spring Security offers excellent support for writing your tests with `WebTestClient`.
- You can write tests directly for methods for which you wrote security configurations using method security.

# A

## *Creating the Spring Boot project*

This appendix presents a couple of options to create a Spring Boot project. The examples I show in this book use Spring Boot. Even though I assume that you have some basic experience with Spring Boot, this appendix serves as a reminder of what your options are to create the projects. For more details about Spring Boot and creating Spring Boot projects, I recommend the fun and easy to read book *Spring Boot in Action* by Craig Walls (Manning, 2015).

In this appendix, I present two easy options for creating your Spring projects. After creating your project, you can choose (at any time) to add other dependencies (by changing the `pom.xml` file in the case of Maven projects). Both options create projects with predefined Maven parents (if that's what you chose), some dependencies, a main class, and usually a demo unit test.

You could do this manually as well, by creating an empty Maven project, adding the parent and dependencies, and then creating a main class with the `@SpringBootApplication` annotation. Although, if you choose to do so manually, you'll probably lose more time for each project than with one of the presented options. Even so, you can run the projects I provide with this book with the IDE of your choice and I don't encourage you to change the way you are used to for running your Spring projects.

### **A.1 Creating a project from start.spring.io**

The most direct version of creating a Spring Boot project is by using the `start.spring.io` generator. From the web page, <https://start.spring.io/>, you can select all the options and dependencies you need, and then download the Maven or Gradle project of choice as a zip archive (figure A.1).

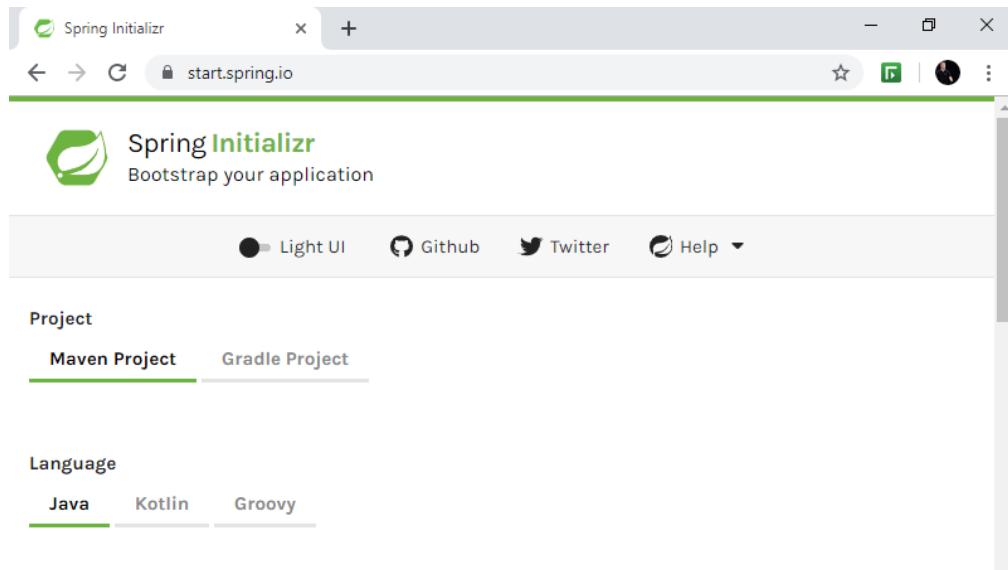


Figure A.1 A partial view of the Spring Initializr page. It offers a light UI that you can use to create a Spring Boot project. In the Initializr, you select the build tool (Maven or Gradle), the language to use (Java, Kotlin, or Groovy), the Spring Boot version and dependencies to download the project as a zip file.

After downloading the project, unzip it and open it as a standard Maven or Gradle project in the IDE of your choice. You choose whether you want to use Maven or Gradle at the beginning of creating the project in the Initializr. Figure A.2 shows the Import dialog for a Maven project.

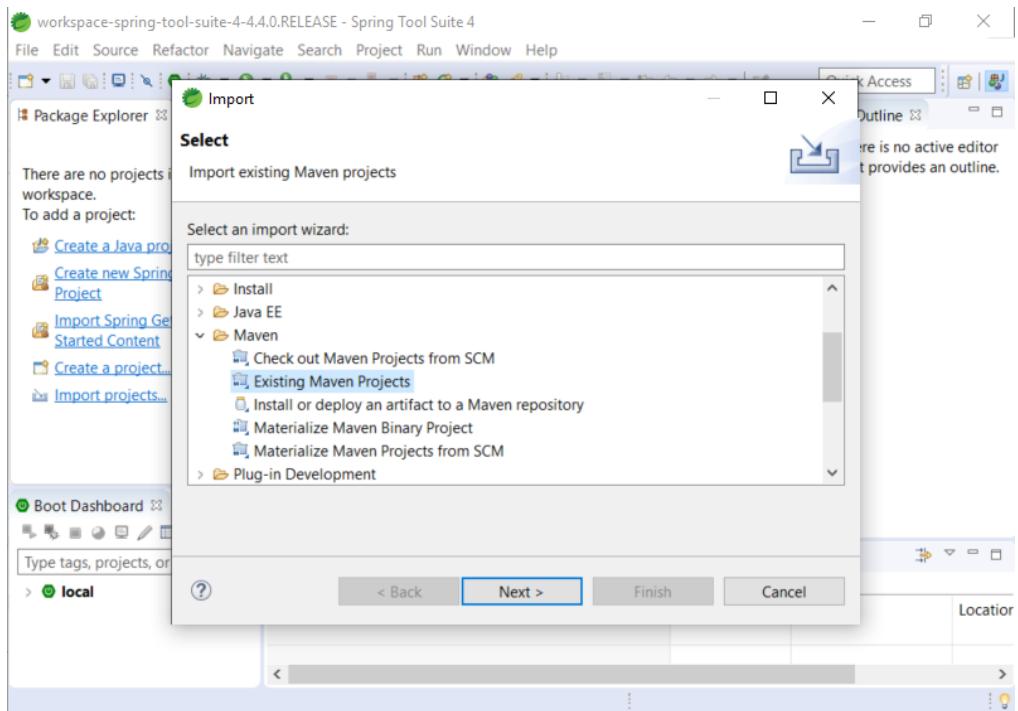


Figure A.2 An existing Maven project can be opened from any programming environment. Once you've created your project with start.spring.io and downloaded it, unzip it and open it as a Maven project from your IDE.

## A.2 Creating a project with the Spring Tool Suite (STS)

The first option presented in section A.1 lets you easily create a Spring project and then import it anywhere. But a lot of IDEs allow you to do this from your development environment. The development environment usually calls the `start.spring.io` web service for you and obtains the project archive. In most development environments, you need to select a new project and then the Spring Starter Project options, like those you see in figure A.3.

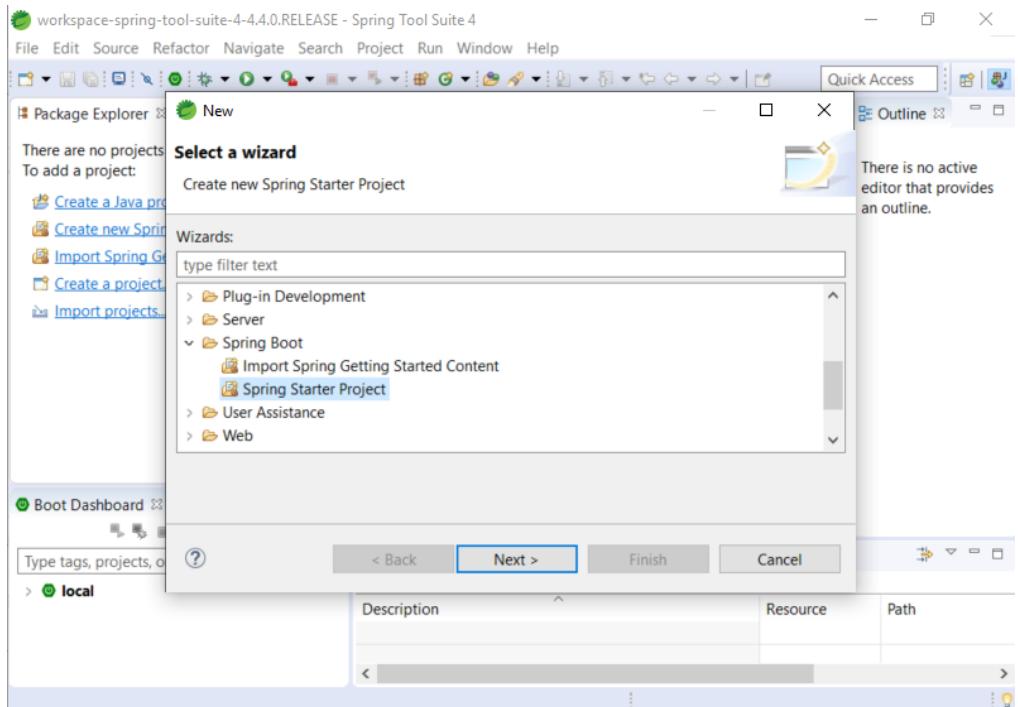


Figure A.3 Some IDEs let you directly create Spring Boot projects. They call start.spring.io in the background and then download, unzip, and import the project for you.

After selecting a new project, you just fill in the same options as in the `start.spring.io` web application in figure A.1: language, version, group, artifact name, dependencies, and so forth. Then your project is created (figure A.4).

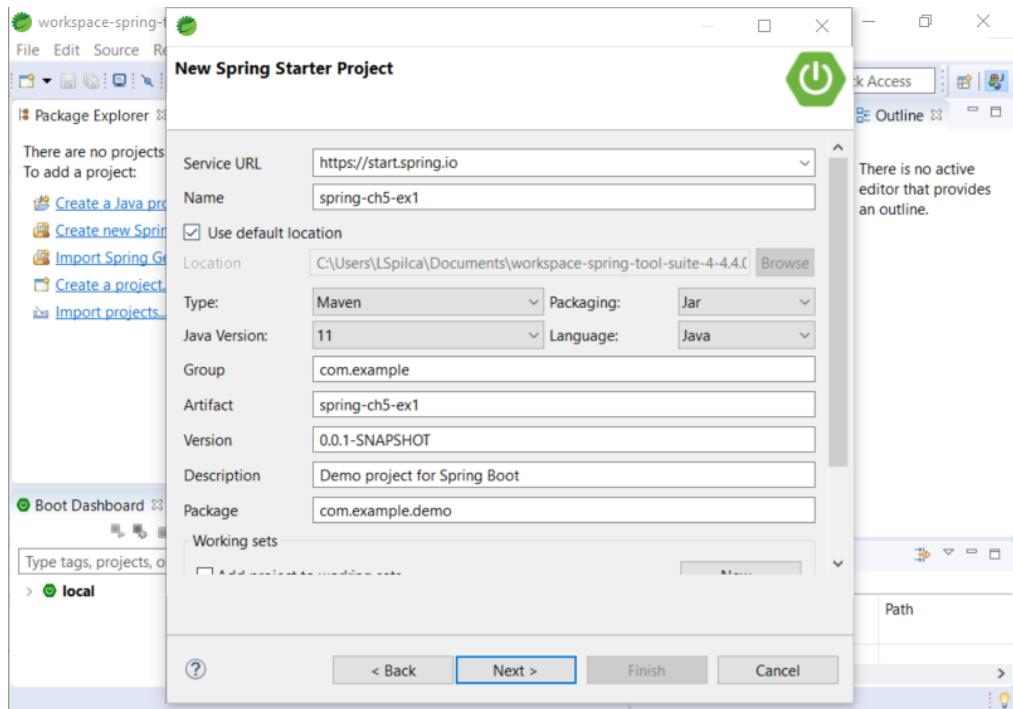


Figure A.4 When creating the Spring Boot project directly from the IDE, you need to choose the same options as on the start.spring.io page. The development environment asks you about the build tool option, preferred language, version, and dependencies among other things.