# 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**

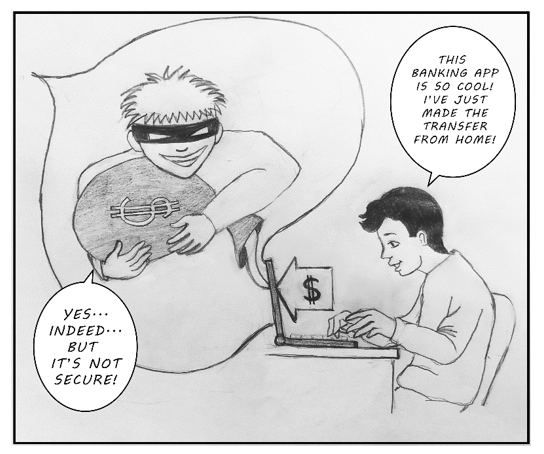
* What Spring Security is and what you can solve by using it
* What security is for a software application
* Why software security is essential and why you should care
* Common vulnerabilities that you’ll encounter at the application level

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](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?**

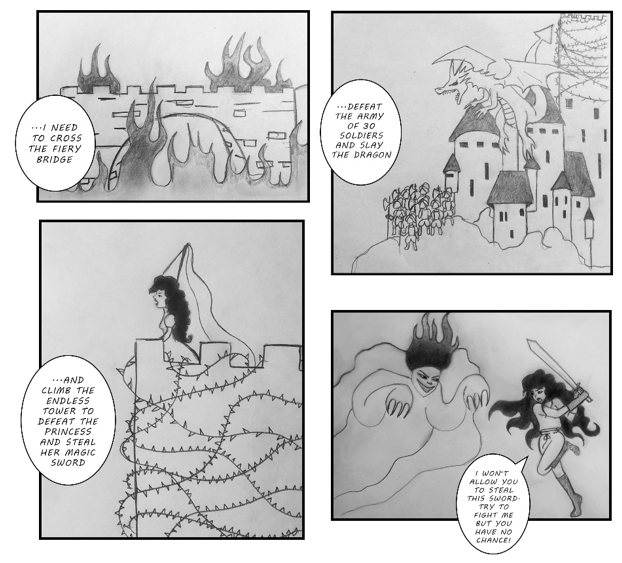
Software systems today manage large amounts of data, out of which a significant part can be considered sensitive, especially given the current General Data Protection Regulations (GDPR) requirements. Any information that you, as a user, consider private is sensitive for your software application. Sensitive 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.

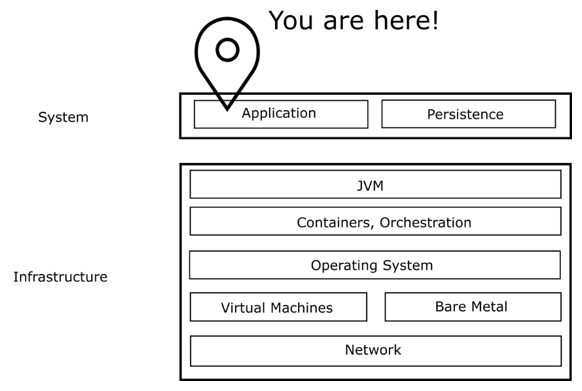
**Figure 1.2 The Dark Wizard (a hacker) has to bypass multiple obstacles (security layers) that steal the Magic Sword (user resources) from the Princess.**



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.

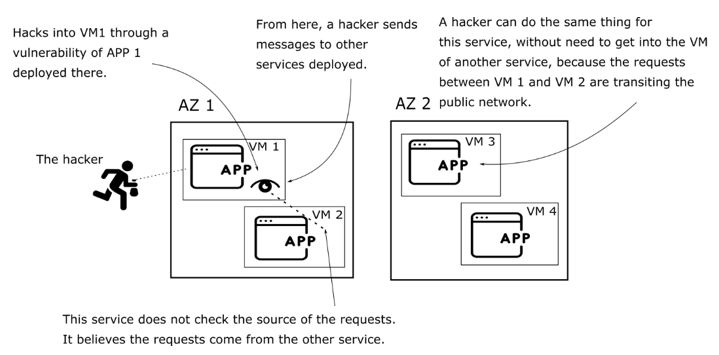
Application-level security (figure 1.3) refers to everything that an application should do to protect the environment it executes in, as well as the data it processes and stores. Mind that this isn’t only about the data affected and used by the application. An application might contain vulnerabilities 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](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**

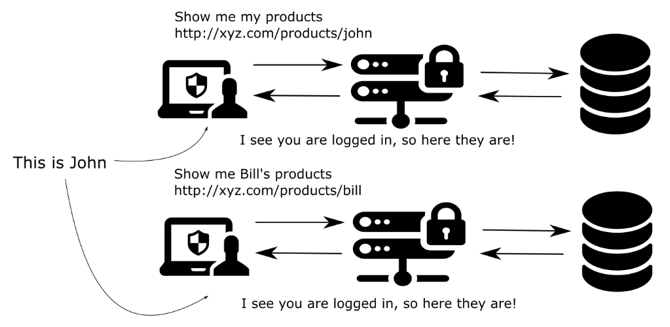
In this book, we discuss authentication and authorization in-depth, and you’ll learn several ways in which you can implement it with Spring Security. Authentication represents the process in which an application identifies someone trying to use it. When someone or something uses the app, we want to find 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.

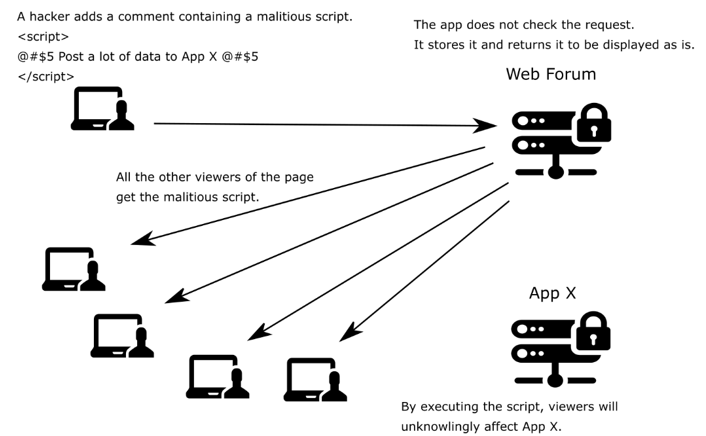
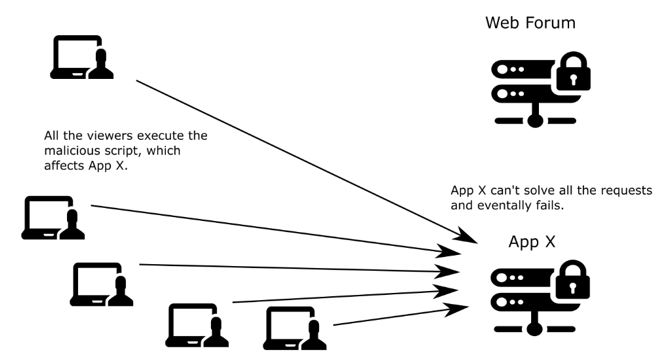
**1.4.3   What is cross-site scripting (XSS)?**

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 1.7 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.**

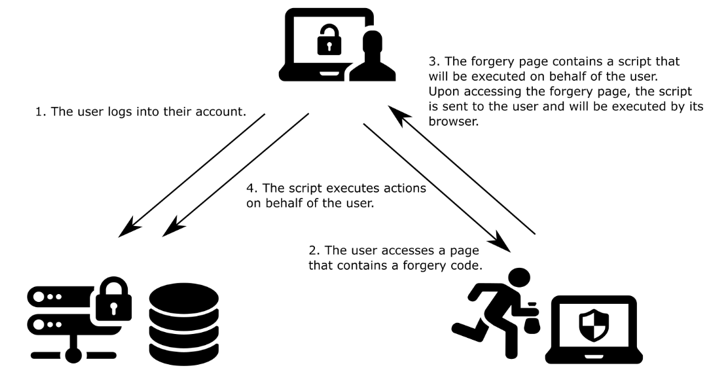
**Figure 1.6 A user posts a comment containing a script on a web forum. The user has defined the script such that it will make requests that try to post or get massive amounts of data from another application (App X)., which represents the victim of the attack. If the Web Forum app allows cross-site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

**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.

[copy](javascript:void(0))

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"

}

[copy](javascript:void(0))

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]

[copy](javascript:void(0))

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 "

}

[copy](javascript:void(0))

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 "

}

[copy](javascript:void(0))

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).

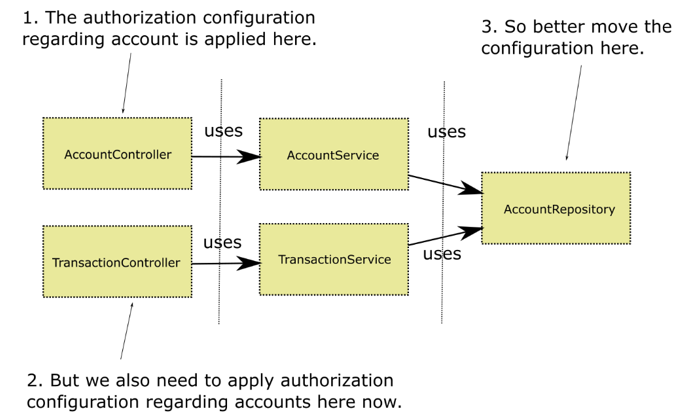
Say you have a web application with a straightforward design. The app has a controller exposing endpoints. The controller directly calls a service that implements some logic, and that uses persisted data managed through a repository (figure 1.9). Imagine a situation where the authorization is done only 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.

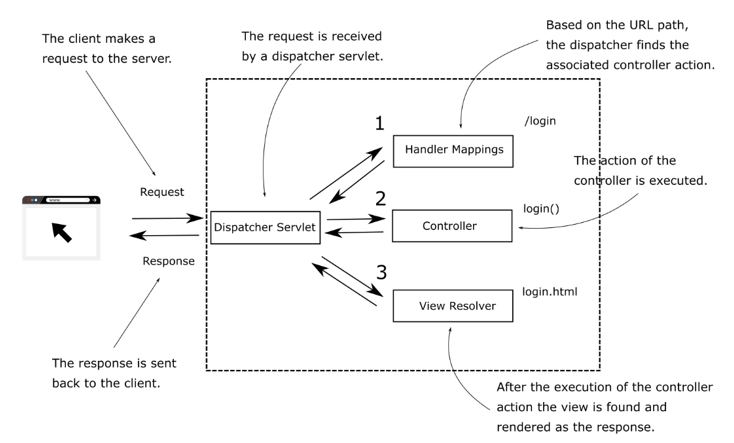
Because the security approach is different depending on the solution we implement, the configuration in Spring Security is also different. In this section, we discuss some examples based on different architectural styles as well as other requirements, how the security approach is affected, and what we 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.**

 https://dpzbhybb2pdcj.cloudfront.net/spilca/v-7/Figures/image011.gif

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.

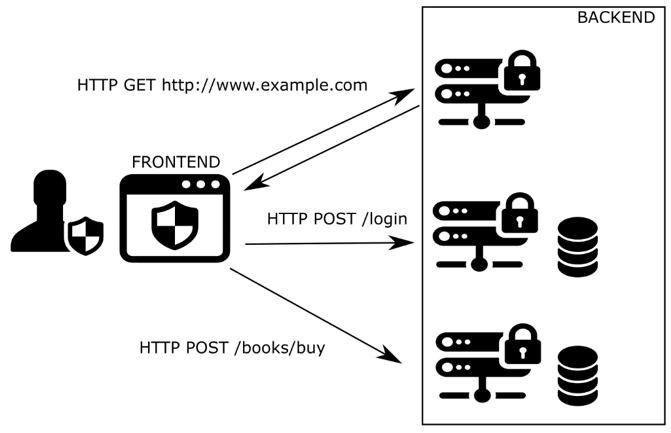
If you’ve authentication and authorization, it also means that you should have some users with valid credentials. Depending on your choice, you could have your application managing the credentials for the users, or you could choose to benefit from another system to do this (for example, you might want 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**

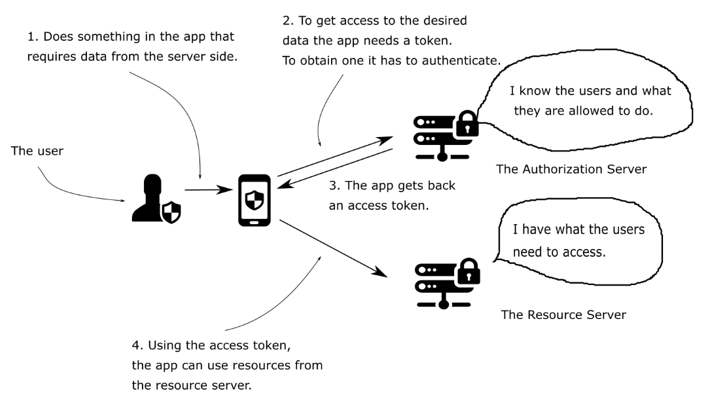
In this section, e discuss a high-level overview of the OAuth2 flow.I’ll focus on the reason for applying OAuth2 and how it relates to what we discussed in section 1.5.2. We’ll discuss this topic in detail in chapters 12 to 15.

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. OAtuh 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**

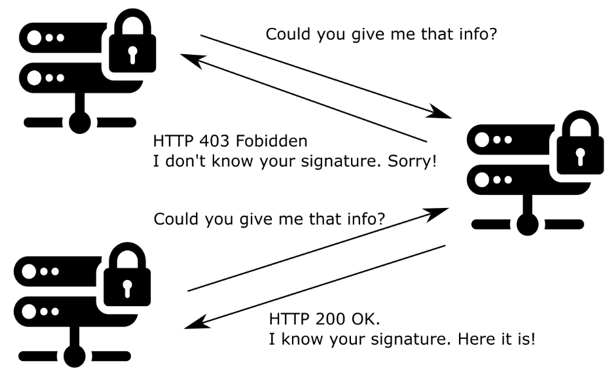
In some cases, you don’t need a username and a password to authenticate and authorize a caller, but you still want to make sure that nobody altered the exchanged messages. You might need this approach when the requests are made between two backend components. Sometimes you’d like to make sure that the 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”.

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 what doesn’t match your implementation. Before this possibility of working, we developers had to write dozens of lines of code, and we would repeat writing them again and again for all the applications we had to develop. This process was less visible in the past, when most architectures were developed monolithically. With 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). It describes writing a web application with Spring 3. You’ll see 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 applications 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 systems.

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 which is 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 in that book accurately describes creating a web app with Spring Boot (<https://livebook.manning.com/book/spring-boot-in-action/chapter-2/>).

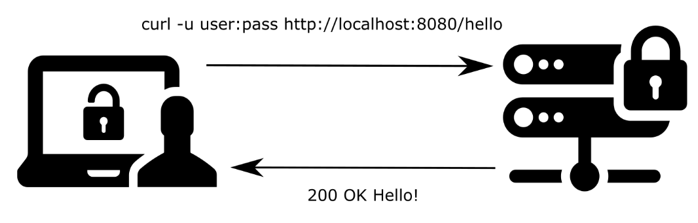
The examples in this book refer to the source code. With each example, I’ll also specify the dependencies that you need to add to your pom.xml file. You can, and I recommend that you do so, download the projects with the book and the available source code. This will help if you get stuck with something, as well as for validating 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**>

[copy](javascript:void(0))

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 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!";

}

}

[copy](javascript:void(0))

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:

1

**Using** generated security password: 93a01cf0-794b-4b98-86ef-54860f36f7f3

[copy](javascript:void(0))

Each time you run the application, it generates a new password. 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 for all the examples, as we consider curl to be the most readable solution. If you prefer, you can use a tool of your choosing. 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 you’ll probably need to install them yourself.

1

curl http://localhost:8080/hello

[copy](javascript:void(0))

The response of the call:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/hello"

}

[copy](javascript:void(0))

**NOTE**

1

curl -u user:93a01cf0-794b-4b98-86ef-54860f36f7f3 http://localhost:8080/hello

[copy](javascript:void(0))

The response of the call:

1

Hello!

[copy](javascript:void(0))

**NOTE**

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 can do this either by using the Base64 tool in a Linux console or find a web page that does this, like [https://www.base64encode.org](https://www.base64encode.org/). This snippet shows the command in a Linux console (or a Git bash console):

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.

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 like the one using the -u option:

1

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

[copy](javascript:void(0))

1

dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01NDg2MGYzNmY3ZjM=

[copy](javascript:void(0))

With this 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.

1

curl -H "Authorization: Basic dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01NDg2MGYzNmY3ZjM=" http://localhost:8080/hello

[copy](javascript:void(0))

The result of the call is:

1

Hello!

[copy](javascript:void(0))

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 want to 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.

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.

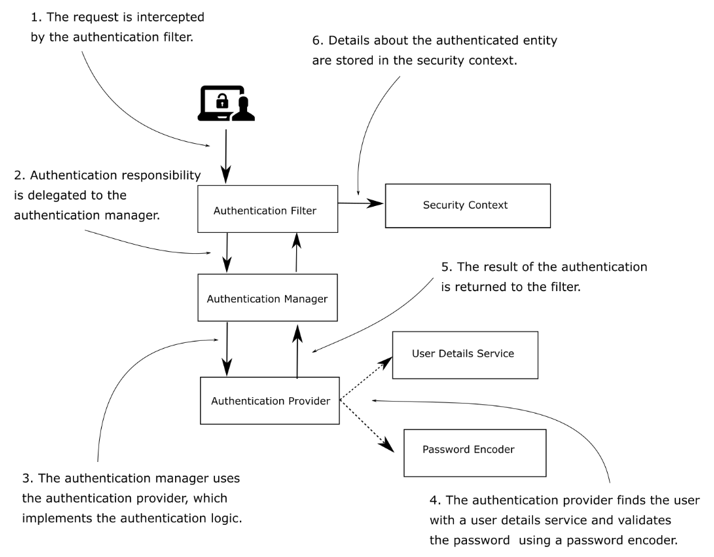
**2.2      Which are the default configurations?**

*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.

**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 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 default random universally unique identifier (UUID) password, generated when the Spring container 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.

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

* And then we have the PasswordEncoder. PasswordEncoder does two things:
* Encode a password

Verify 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 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. Usually, we don’t use HTTP Basic authentication without at least HTTPS for confidentiality. You can read more about 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.

**NOTE**

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.

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.

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 winpty as shown in the next code snippet:

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:

1

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

[copy](javascript:void(0))

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

1

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

[copy](javascript:void(0))

Add a test endpoint to your application and then call it using HTTPS.

1

2

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

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

[copy](javascript:void(0))

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.

1

2

3

server.ssl.key-store-type=PKCS12

server.ssl.key-store=classpath:certificate.p12

server.ssl.key-store-password=12345

**A**

[copy](javascript:void(0))

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**

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. The implementation can store credentials in memory, which can then be used by Spring Security to authenticate a request.

**NOTE**

**2.3.1   Overriding the UserDetailsService component**

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.

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.

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

**NOTE**

#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:

**NOTE**

You don’t have a PasswordEncoder.

**Listing 2.3 The configuration class for the UserDetailsService bean**

@Configuration

**public** **class** **ProjectConfig** {

@Bean

**public** UserDetailsService **userDetailsService**() {

**var** userDetailsService =

**new** InMemoryUserDetailsManager();

**return** userDetailsService;

}

}

**A**

**B**

**C**

[copy](javascript:void(0))

Create at least one user who has a set of credentials (username and password)

Add the user to be managed by our implementation of UserDetailsService

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. 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.
* You don’t have a PasswordEncoder.

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

1. #B Adds the user to be managed by UserDetailsService
2. **Note**
3. 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 you can see 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.

**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")

.password("12345")

.authorities("read")

.build();

userDetailsService.createUser(user);

**return** userDetailsService;

}

}

**A**

**B**

[copy](javascript:void(0))

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:

#B Adds the user to be managed by UserDetailsService

**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.

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.

1

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

[copy](javascript:void(0))

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

1

2

java.lang.IllegalArgumentException: There **is** no PasswordEncoder mapped **for** the id "null"

at org.springframework.security.crypto.password.DelegatingPasswordEncoder$UnmappedIdPasswordEncoder.matches(DelegatingPasswordEncoder.java:244) ~[spring-security-core-5.1.6.RELEASE.jar:5.1.6.RELEASE]

[copy](javascript:void(0))

**NOTE**

1

2

3

4

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

[copy](javascript:void(0))

**NOTE**

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, HTTP Basic authentication is used as the authorization method. You can easily override these configurations.

As you’ll see 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 2.5 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

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

}

**A**

[copy](javascript:void(0))

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

Fkr’z qrt ykr peidotnn rdjw xyr xwn ytoc vhagni rbk rsameneu “yivn” hnc rgo soawsdpr “12345”:

1

2

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

Hello!

[copy](javascript:void(0))

**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**

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.

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 {

// …

}

}

[copy](javascript:void(0))

**2.3.3   Setting the configuration in different ways**

**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()

.anyRequest().authenticated();

}

}

**A**

[copy](javascript:void(0))

#A All the requests require authentication.

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 2.8 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()

.anyRequest().permitAll();

}

}

**A**

[copy](javascript:void(0))

#A None of the requests need to be authenticated.

#B We define a user with all its details.

1

curl http://localhost:8080/hello

[copy](javascript:void(0))

The response body of the call is:

1

Hello!

[copy](javascript:void(0))

**NOTE**

**2.3.3   Setting the configuration in different ways**

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

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.

#A Creates an instance of InMemoryUserDetailsManager()

**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();

**var** user = User.withUsername("john")

.password("12345")

.authorities("read")

.build();

userDetailsService.createUser(user);

auth.userDetailsService(userDetailsService)

.passwordEncoder(NoOpPasswordEncoder.getInstance());

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

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

#B We define a user with all its details.

#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.

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**

#B But the UserDetailsService 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 UserDetailsService 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.

**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();

**var** user = User.withUsername("john")

.password("12345")

.authorities("read")

.build();

userDetailsService.createUser(user);

auth.userDetailsService(userDetailsService)

.passwordEncoder(

NoOpPasswordEncoder.getInstance());

}

@Override

**protected** **void** **configure**(HttpSecurity http) **throws** Exception {

http.httpBasic();

http.authorizeRequests()

.anyRequest().authenticated();

}

}

**A**

**B**

**C**

**D**

**E**

[copy](javascript:void(0))

#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**() {

**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);

}

@Override

**protected** **void** **configure**(HttpSecurity http) **throws** Exception {

http.httpBasic();

http.authorizeRequests()

.anyRequest().authenticated();

}

}

**A**

**B**

[copy](javascript:void(0))

#A The PasswordEncoder is designed as a bean.

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

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.

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());

}

[copy](javascript:void(0))

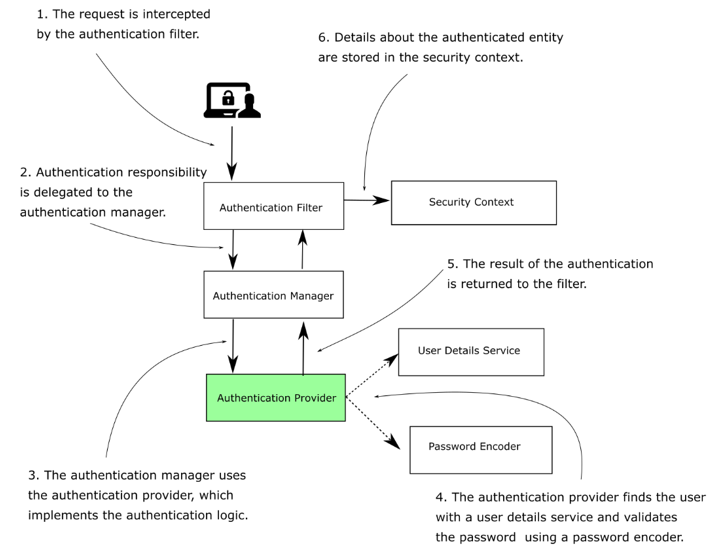
#B This condition would generally call UserDetailsService and PasswordEncoder to test the username and password.

**2.3.4   Overriding the AuthenticationProvider 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.

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

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.

**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

}

}

[copy](javascript:void(0))

In chapter 5, you’ll learn a more detailed description of the AuthenticationProvider and how to override the authentication process. There we’ll also discuss the Authentication interface and its implementations like the UserPasswordAuthenticationToken.

**Listing 2.14 Implementing the authentication logic**

@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[CA]

("Error in authentication!");

}

}

**A**

**B**

[copy](javascript:void(0))

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 will start 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 approach 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.

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.

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 2.15 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);

}

}

[copy](javascript:void(0))

Here the WebAuthorizationConfig class needs to extend WebSecurityConfigurerAdapter and override the configure(HttpSecurity http) 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();

}

}

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

The response body will be:

1

Hello

[copy](javascript:void(0))

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.

**2.3.5   Using multiple configuration classes in your project**

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

**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();

}

}

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

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.

**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 InMemoryUserDetailsManager, JdbcUserDetailsManager, and LdapUserDetailsManager. For when these provided implementations aren’t a good fit for your system, you’ll write a custom implementation.

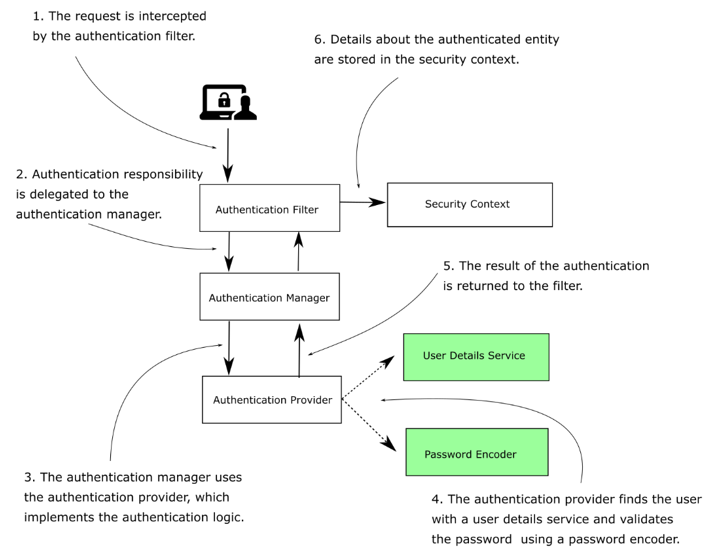
**3.1      Implementing authentication in Spring Security**

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.

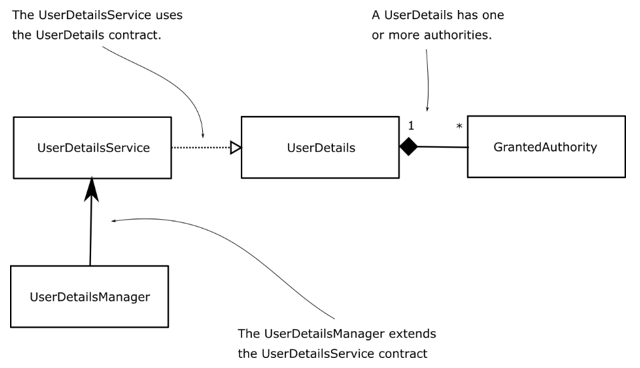
**Figure 3.1 The authentication flow: The AuthenticationFilter intercepts the request and delegates the authentication responsibility to the AuthenticationManager. The AuthenticationManager uses further an authentication provider, which implements the authentication logic. To check the username and the 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 is 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**();

String **getPassword**();

Collection<? extends GrantedAuthority> getAuthorities();

**boolean** **isAccountNonExpired**();

**boolean** **isAccountNonLocked**();

**boolean** **isCredentialsNonExpired**();

**boolean** **isEnabled**();

}

**A**

**B**

**C**

[copy](javascript:void(0))

#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.

1

2

3

**public** **interface** **GrantedAuthority** **extends** **Serializable** {

String **getAuthority**();

}

[copy](javascript:void(0))

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.

1

2

GrantedAuthority g1 = () -> "READ";

GrantedAuthority g2 = **new** SimpleGrantedAuthority("READ");

[copy](javascript:void(0))

**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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

**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();

[copy](javascript:void(0))

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");

UserDetails u1 = builder1

.password("12345")

.authorities("read", "write")

.passwordEncoder(p -> encode(p))

.accountExpired(false)

.disabled(true)

.build();

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

UserDetails u2 = builder2.build();

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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)

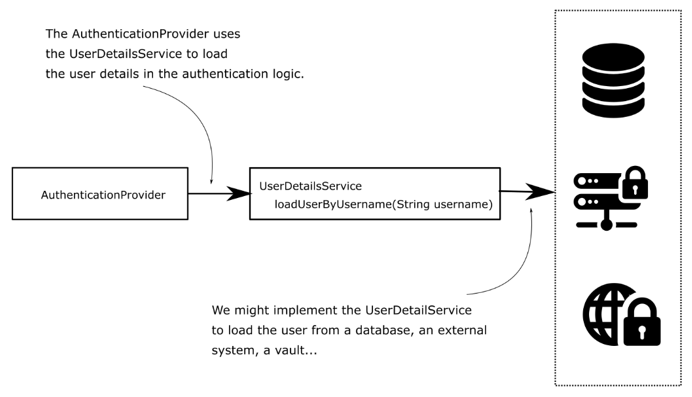
**throws** UsernameNotFoundException;

}

[copy](javascript:void(0))

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;

**private** **final** String password;

**private** **final** String authority;

**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);

}

@Override

**public** String **getPassword**() {

**return** password;

}

@Override

**public** String **getUsername**() {

**return** username;

}

@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**

**B**

**C**

**D**

[copy](javascript:void(0))

#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;

**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))

.findFirst()

.orElseThrow(

() -> **new** UsernameNotFoundException("User not found"));

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#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();

}

}

[copy](javascript:void(0))

Finally, we create a simple endpoint and test the implementation.

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

1

2

3

4

5

6

7

8

@RestController

**public** **class** **HelloController** {

@GetMapping("/hello")

**public** String **hello**() {

**return** "Hello";

}

}

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

The response body is:

1

Hello

[copy](javascript:void(0))

**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);

}

[copy](javascript:void(0))

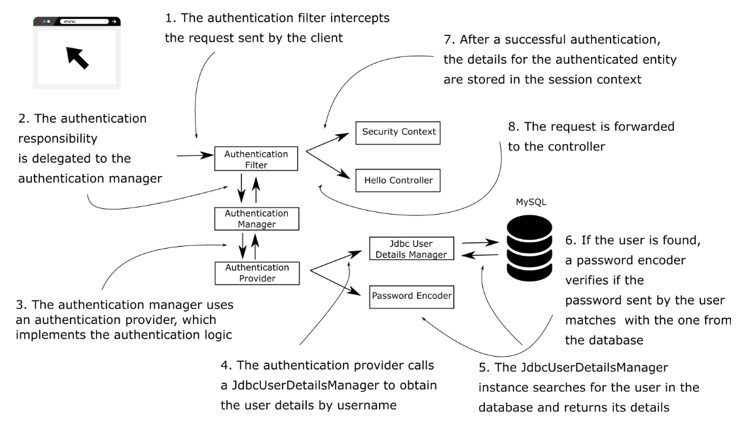
The InMemoryUserDetailsManager object that we have used in chapter 2 is actually a UserDetailsManager. At that time, we have only considered its UserDetailsService characteristics, but now you understand better why we were able to call a createUser() method on the instance.

**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’ 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`));

[copy](javascript:void(0))

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**

1

2

3

4

5

**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`));

[copy](javascript:void(0))

**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:

1

2

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

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

[copy](javascript:void(0))

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**>

[copy](javascript:void(0))

**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

[copy](javascript:void(0))

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

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

@Configuration

**public** **class** **ProjectConfig** {

@Bean

**public** UserDetailsService **userDetailsService**(DataSource dataSource) {

**return** **new** JdbcUserDetailsManager(dataSource);

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

}

[copy](javascript:void(0))

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";

}

}

[copy](javascript:void(0))

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

1

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

[copy](javascript:void(0))

The response of the call is:

1

Hello

[copy](javascript:void(0))

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;

}

[copy](javascript:void(0))

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 UserDetailsManager for LDAP. Even if it is less popular than the JdbcUserDetailsManager, 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 LdapUserDetailsManager. 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

**objectclass**: top

**objectclass**: domain

**objectclass**: extensibleObject

**dc**: springframework

**dn**: ou=groups,dc=springframework,dc=org

**objectclass**: top

**objectclass**: organizationalUnit

**ou**: groups

**dn**: uid=john,ou=groups,dc=springframework,dc=org

**objectclass**: top

**objectclass**: person

**objectclass**: organizationalPerson

**objectclass**: inetOrgPerson

**cn**: John

**sn**: John

**uid**: john

**userPassword**: 12345

**A**

**B**

**C**

[copy](javascript:void(0))

#A The definition of the base entitiy

#B The definition of a group entitiy

#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 to 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**>

[copy](javascript:void(0))

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

[copy](javascript:void(0))

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

**public** UserDetailsService **userDetailsService**() {

**var** cs = **new** DefaultSpringSecurityContextSource(

"ldap://127.0.0.1:33389/dc=springframework,dc=org");

cs.afterPropertiesSet();

**var** manager = **new** LdapUserDetailsManager(cs);

manager.setUsernameMapper(

**new** DefaultLdapUsernameToDnMapper("ou=groups", "uid"));

manager.setGroupSearchBase("ou=groups");

**return** manager;

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

}

**A**

**B**

**C**

**D**

**E**

[copy](javascript:void(0))

#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!";

}

}

[copy](javascript:void(0))

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**

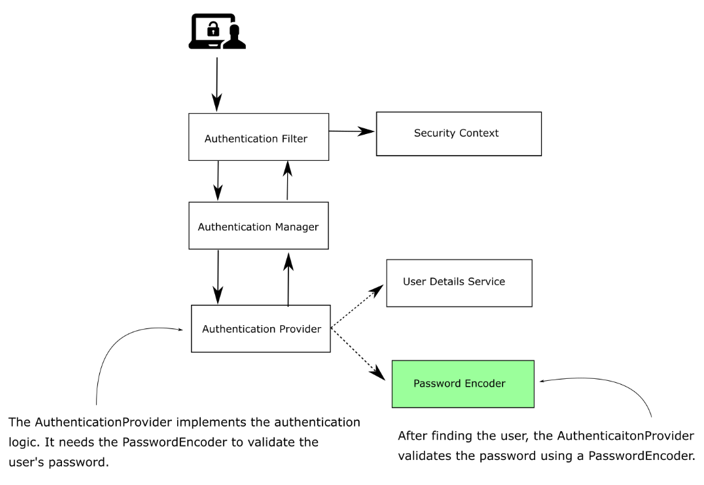
* Implementing and working with the PasswordEncoder.
* Using the tools offered by the Spring Security Crypto Module.

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**;

}

}

[copy](javascript:void(0))

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**

14

**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);

}

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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();

}

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

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.

1

2

PasswordEncoder p = **new** StandardPasswordEncoder();

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

[copy](javascript:void(0))

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:

1

2

3

PasswordEncoder p = **new** Pbkdf2PasswordEncoder();

PasswordEncoder p = **new** Pbkdf2PasswordEncoder("secret");

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

[copy](javascript:void(0))

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 the 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);

[copy](javascript:void(0))

The log rounds (logarithmic rounds) value provided affect the number of iterations used in the hashing operation. The number of iterations used is 2log 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:

1

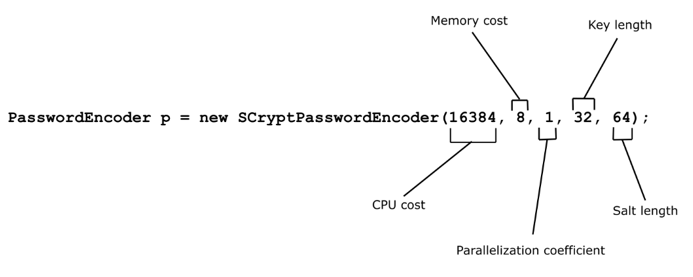
2

PasswordEncoder p = **new** SCryptPasswordEncoder();

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

[copy](javascript:void(0))

**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.

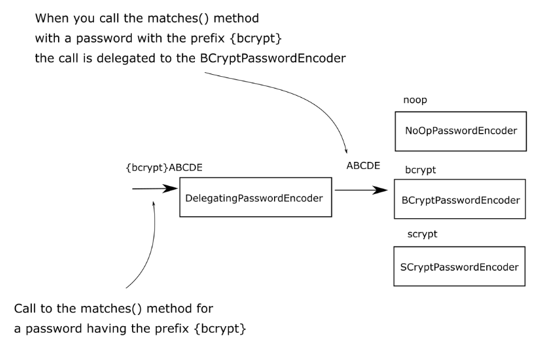
In some applications, you might find it useful to have various password encoders and choose from them depending on some specific configuration. A most common scenario in which I found the DelegatingPasswordEncoder in production applications is when the encoding algorithm is changed, starting with a particular 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.

**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.**

**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.**

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);

}

}

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

**ENCODING VS. ENCRYPTING VS. HASHING**

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

Encoding refers to any transformation of a given input. For example, if we have a function, x, that reverses a string, function x -> y applied to "ABCD" 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) -> 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) -> 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, k1) -> y and decryption (y, k2) -> x, then we say that the encryption is done with asymmetric keys. Then, (k1, k2) is called a key pair. The key used for encryption, k1, is also referred to as the public key, while k2 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 -> y then we should also have a matching function (x,y) -> boolean

Sometimes the hashing function could also use a random value added to the input: (x, k) -> y. We refer to this value as the “salt”. The salt makes the function stronger, enforcing the difficulty of applying a reverse function to obtain the input from the result.

To 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 is 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**();

}

[copy](javascript:void(0))

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.

1

2

**StringKeyGenerator** keyGenerator = KeyGenerators.string();

**String** salt = keyGenerator.generateKey();

[copy](javascript:void(0))

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**();

}

[copy](javascript:void(0))

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 ByteKeyGenerator generates keys of 8 bytes length, but you can specify the length of the keys that will be generating when obtaining the generator instance.

1

2

3

**BytesKeyGenerator** keyGenerator = KeyGenerators.secureRandom();

**byte** [] key = keyGenerator.generateKey();

**int** keyLength = keyGenerator.getKeyLength();

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

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.

1

2

3

**BytesKeyGenerator** keyGenerator = KeyGenerators.shared(16);

**byte** [] key1 = keyGenerator.generateKey();

**byte** [] key2 = keyGenerator.generateKey();

[copy](javascript:void(0))

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);

}

[copy](javascript:void(0))

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);

}

[copy](javascript:void(0))

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);

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

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);

[copy](javascript:void(0))

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);

**String** encrypted = e.encrypt(valueToEncrypt);

**String** decrypted = e.decrypt(encrypted);

**A**

[copy](javascript:void(0))

#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);

**String** encrypted1 = e.encrypt(valueToEncrypt);

**String** encrypted2 = e.encrypt(valueToEncrypt);

**A**

[copy](javascript:void(0))

#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

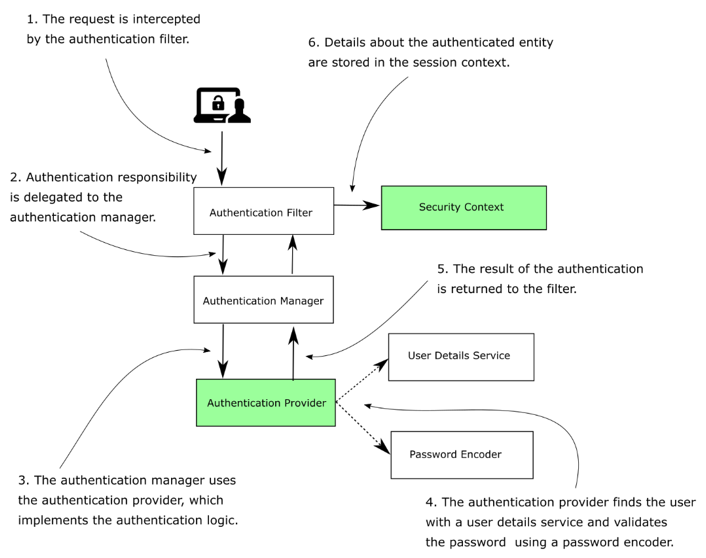
In chapters 3 and 4, we covered a few of the components acting in the authentication flow. We discussed the UserDetails and how to define the prototype to describe a user in Spring Security. We then used the UserDetails in examples where you learned how the UserDetailsService and UserDetailsManager contracts 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.**

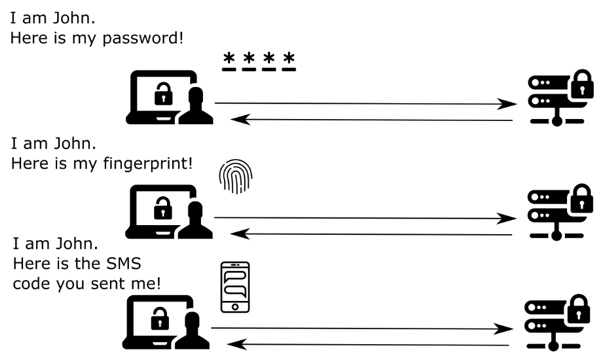


This chapter covers the remaining parts of the authentication flow (the shaded boxes in figure 5.1). Then, in chapters 7 and 8, you’ll learn how authorization works, which is the process that follows authentication in the HTTP request. We have to discuss how to implement the AuthenticationProvider. To 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:

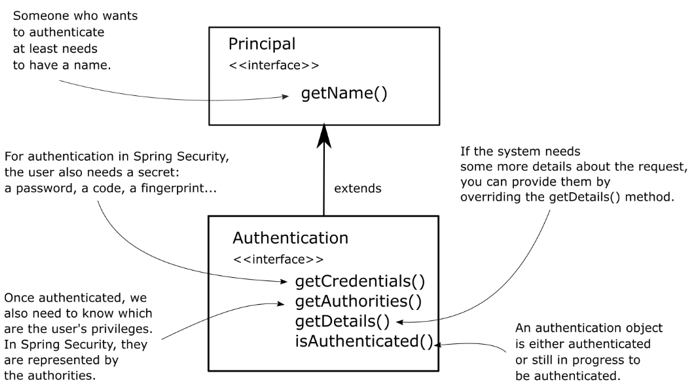
* In section 5.1.1, we analyze how Spring Security represents the authentication event.
* In section 5.1.2, we discuss the AuthenticationProvider contract, which is responsible for the authentication logic.
* In section 5.1.3, you’ll write custom authentication logic by implementing the AuthenticationProvider 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;

}

[copy](javascript:void(0))

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);

}

[copy](javascript:void(0))

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.

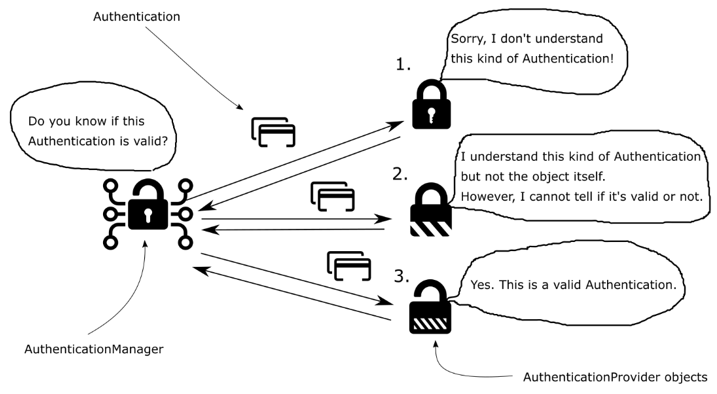
An analogy of how the authentication manager and authentication providers work together to validate or invalidate an authentication request is a more complex door lock. You can open this door lock either by using a card or an old fashioned physical key (figure 5.4). The lock itself is the authentication 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);

}

}

[copy](javascript:void(0))

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());

} **else** {

**throw** **new** BadCredentialsException("Something went wrong!");

}

}

// Omitted code

}

**A**

**B**

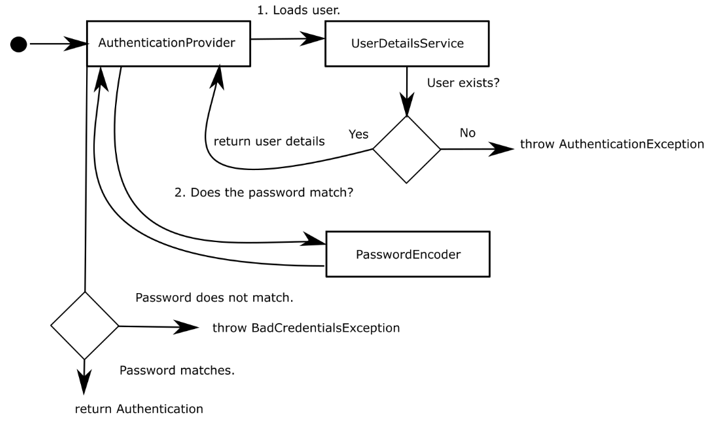
[copy](javascript:void(0))

#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

}

[copy](javascript:void(0))

**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 re-designing 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.

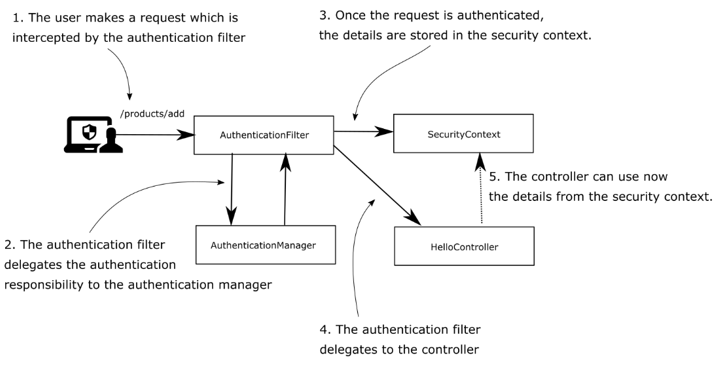
When we rely on functionalities implemented by a framework, we have several benefits. We know they’re tested and, there are fewer changes they include vulnerabilities. Also, a good framework relies on abstractions, which helps you create maintainable applications. When you write your own implementations, 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);

}

[copy](javascript:void(0))

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.

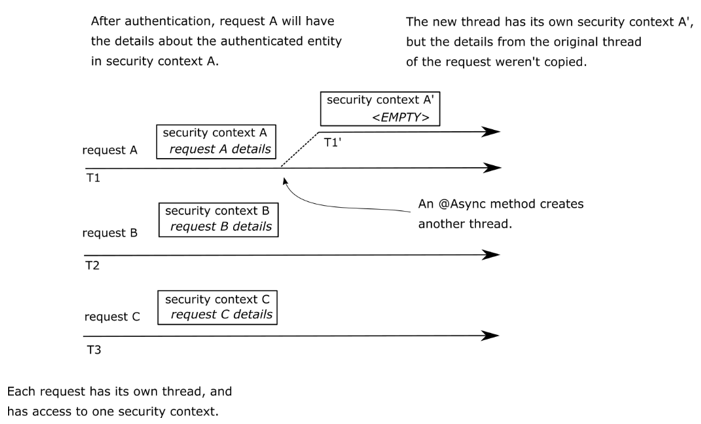
Besides these three strategies for managing the security context provided by Spring Security, in this section, we also discuss what happens when you define your own threads that are not known by Spring. As you will learn, for these cases, you will have to explicitly copy the details from the security 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() + "!";

}

[copy](javascript:void(0))

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**

1

2

3

4

@GetMapping("/hello")

**public** String **hello**(Authentication a) {

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

}

**A**

[copy](javascript:void(0))

#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.

1

2

**curl** -u user:99ff79e3-8ca0-401c-a396-0a8625ab3bad http://localhost:8080/hello

Hello, user!

[copy](javascript:void(0))

**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

**public** **void** **goodbye**() {

SecurityContext context = SecurityContextHolder.getContext();

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

// do something with the username

}

**A**

[copy](javascript:void(0))

#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** {

}

[copy](javascript:void(0))

**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);

}

}

[copy](javascript:void(0))

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);

}

[copy](javascript:void(0))

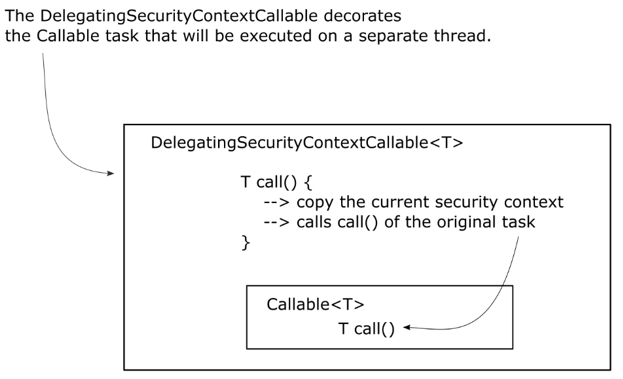
Be also aware that the SecurityContext is not thread-safe. So with this strategy where all the threads of the application can access the SecurityContext object, you will have to take care also of the concurrent access.

**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.

**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();

};

…

}

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

Calling now the endpoint, you will observe that Spring propagated the security context to the thread in which the tasks execute.

1

curl -u user:2eb3f2e8-debd-420c-9680-48159b2ff905[CA] http://localhost:8080/ciao

[copy](javascript:void(0))

The response body for this call is:

1

Ciao, user!

[copy](javascript:void(0))

**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();

}

}

[copy](javascript:void(0))

Call the endpoint to test that the DelegatingSecurityContextExecutorService correctly delegated the security context.

1

curl -u user:5a5124cc-060d-40b1-8aad-753d3da28dca http://localhost:8080/hola

[copy](javascript:void(0))

The response body for this call is:

1

Hola, user!

[copy](javascript:void(0))

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 |
| DelegatingSecurityContextExecutor | Implements the Executor interface and is designed to decorate an Executor object with the capability of forwarding a security context to the threads created by its pool. |
| DelegatingSecurityContextExecutorService | Implements the ExecutorService interface and is designed to decorate an ExecutorService 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();

}

}

[copy](javascript:void(0))

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();

}

[copy](javascript:void(0))

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.

1

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

[copy](javascript:void(0))

The response of the call is:

/

…

< WWW-Authenticate: Basic realm="OTHER"

…

[copy](javascript:void(0))

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());

}

}

[copy](javascript:void(0))

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();

}

[copy](javascript:void(0))

If you make now a call to an endpoint such that the authentication fails, you should find in the response the newly added header.

1

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

[copy](javascript:void(0))

The response of the call is:

…

< HTTP/1.1 401

< Set-Cookie: JSESSIONID=459BAFA7E0E6246A463AD19B07569C7B; Path=/; HttpOnly

< message: Luke, I am your father!

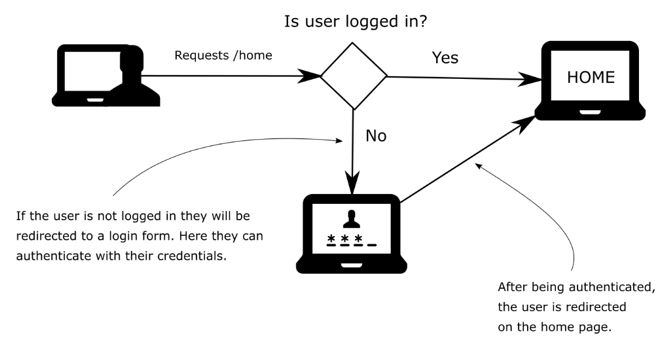
…

[copy](javascript:void(0))

**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();

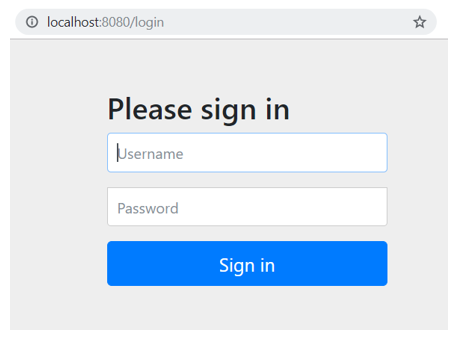
}

}

[copy](javascript:void(0))

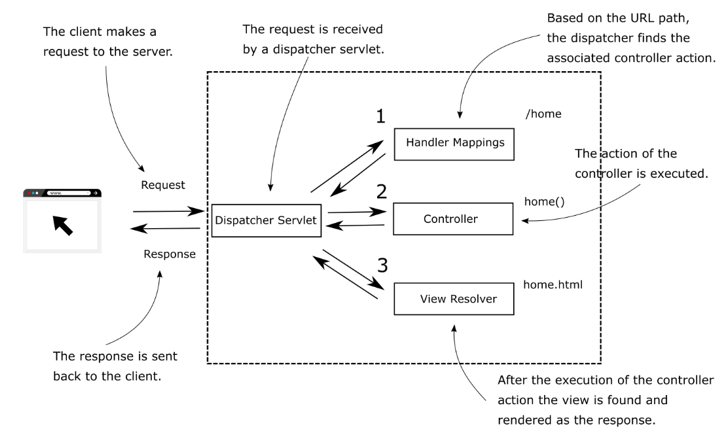
Even with this minimal configuration, Spring Security has already configured a login form as well as a logout page for your project. Starting the application and accessing it with the browser should redirect you to a login page.

**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";

}

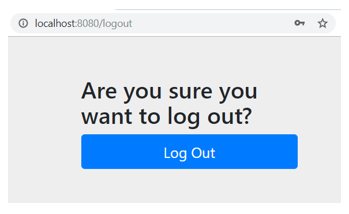
}

[copy](javascript:void(0))

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();

}

[copy](javascript:void(0))

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();

**if** (auth.isPresent()) {

httpServletResponse

.sendRedirect("/home");

} **else** {

httpServletResponse

.sendRedirect("/error");

}

}

}

**A**

**B**

[copy](javascript:void(0))

#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());

}

}

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

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();

}

[copy](javascript:void(0))

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

1

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

[copy](javascript:void(0))

The response of the call is:

1

<**h1**>Welcome</**h1**>

[copy](javascript:void(0))

**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 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
* o DelegatingSecurityContextRunnable,
* o DelegatingSecurityContextCallable, and
* o 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**

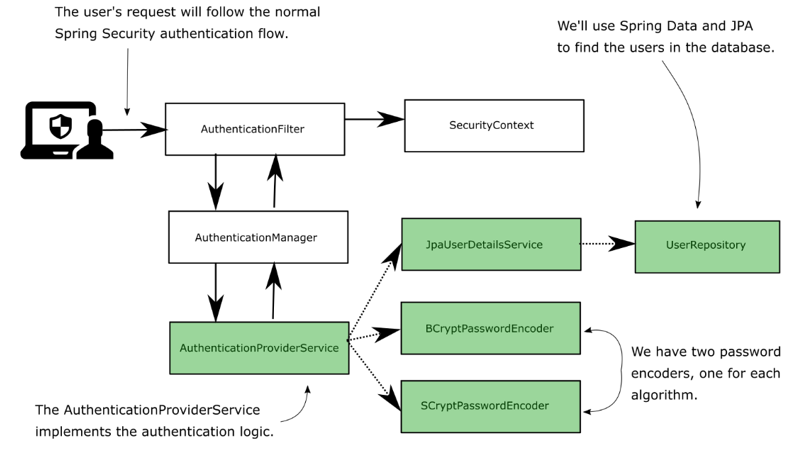
In this section, we implement a small web application where the user, after successful authentication, can see a list of products on the main page. You find the complete implementation with the provided projects, in example ssia-ch6-ex1. The products, as well as the users,  are stored in a database. 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.**

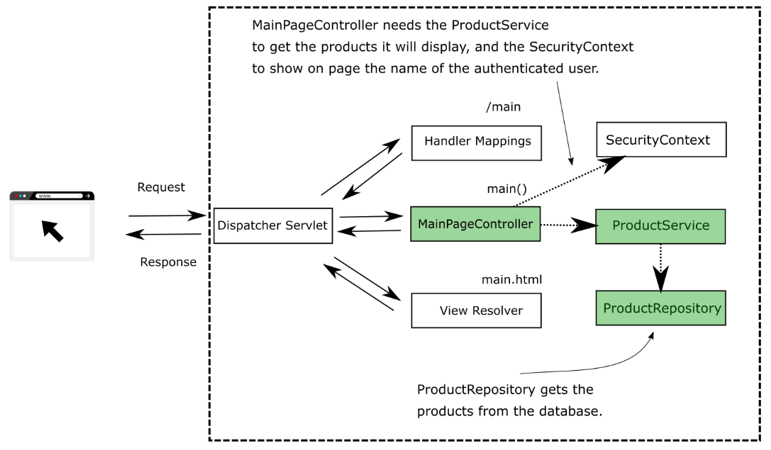


**NOTE**

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).

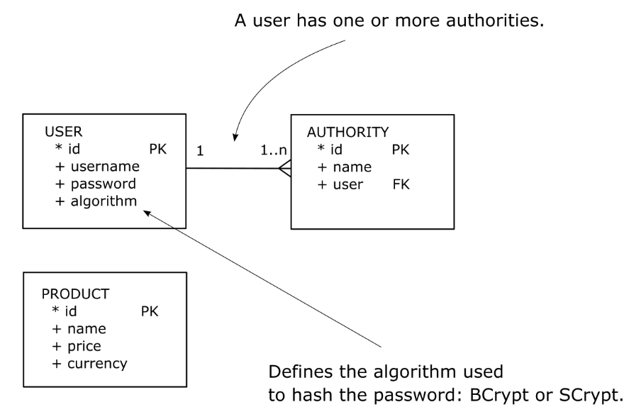
The application will also have a main page that the user can access after a successful login. This page displays details about products stored in the database. In figure 6.2, I have shaded the components that we’ll create differently. We’ll need a MainPageController that defines the action which the 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`));

[copy](javascript:void(0))

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`));

[copy](javascript:void(0))

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`));

[copy](javascript:void(0))

**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.

1

2

3

4

**INSERT** **IGNORE** **INTO** `spring`.`user` (`id`, `username`, `password`, `algorithm`) **VALUES** ('1', 'john', '$2a$10$xn3LI/AjqicFYZFruSwve.681477XaVNaUQbr1gioaWPn4t1KsnmG', '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');

[copy](javascript:void(0))

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

1

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

[copy](javascript:void(0))

**NOTE**

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**>

[copy](javascript:void(0))

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

1

2

3

4

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

[copy](javascript:void(0))

**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 UserDetails 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();

}

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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

1

2

3

**public** **enum** EncryptionAlgorithm {

BCRYPT, SCRYPT

}

[copy](javascript:void(0))

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

}

[copy](javascript:void(0))

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**

1

2

3

4

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

Optional<User> **findUserByUsername**(String username);

}

**A**

[copy](javascript:void(0))

#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;

}

}

[copy](javascript:void(0))

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()))

.collect(Collectors.toList());

}

@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**

**B**

[copy](javascript:void(0))

#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 =

() -> **new** UsernameNotFoundException(

"Problem during authentication!");

User u = userRepository

.findUserByUsername(username)

.orElseThrow(s);

**return** **new** CustomUserDetails(u);

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#A We declare a supplier which created 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

**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**

[copy](javascript:void(0))

#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 =

userDetailsService.loadUserByUsername(username);

**switch** (user.getUser().getAlgorithm()) {

**case** BCRYPT:

**return** checkPassword(user, password, bCryptPasswordEncoder);

**case** SCRYPT:

**return** checkPassword(user, password, sCryptPasswordEncoder);

}

**throw** **new** BadCredentialsException("Bad credentials");

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#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");

}

}

[copy](javascript:void(0))

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

**private** AuthenticationProviderService authenticationProvider;

@Bean

**public** BCryptPasswordEncoder **bCryptPasswordEncoder**() {

**return** **new** BCryptPasswordEncoder();

}

@Bean

**public** SCryptPasswordEncoder **sCryptPasswordEncoder**() {

**return** **new** SCryptPasswordEncoder();

}

@Override

**protected** **void** **configure**(AuthenticationManagerBuilder auth) {

auth.authenticationProvider(authenticationProvider);

}

}

**A**

**B**

[copy](javascript:void(0))

#A We get the instance of AuthenticationProviderService 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();

}

[copy](javascript:void(0))

**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

}

[copy](javascript:void(0))

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

1

2

3

**public** **enum** Currency {

USD, GBP, EUR

}

[copy](javascript:void(0))

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**

1

2

3

**public** **interface** **ProductRepository**

**extends** **JpaRepository**<**Product**, **Integer**> {

}

**A**

[copy](javascript:void(0))

#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();

}

}

[copy](javascript:void(0))

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";

}

}

[copy](javascript:void(0))

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">

<**head**>

<**meta** charset="UTF-8">

<**title**>Products</**title**>

</**head**>

<**body**>

<**h2** th:text="'Hello, ' + ${username} + '!'" />

<**p**><**a** href="/logout">Sign out here</**a**></**p**>

<**h2**>These are all the products:</**h2**>

<**table**>

<**thead**>

<**tr**>

<**th**> Name </**th**>

<**th**> Price </**th**>

</**tr**>

</**thead**>

<**tbody**>

<**tr** th:if="${products.empty}">

<**td** colspan="2"> No Products Available </**td**>

</**tr**>

<**tr** th:each="book : ${products}">

<**td**><**span** th:text="${book.name}"> Name </**span**></**td**>

<**td**><**span** th:text="${book.price}"> Price </**span**></**td**>

</**tr**>

</**tbody**>

</**table**>

</**body**>

</**html**>

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#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](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**

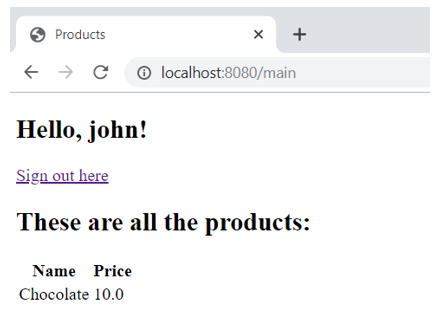
In a real-world application, you should never allow your users to define simple passwords like “12345”. Passwords so simple are easy to guess, and they represent a security risk.

**Figure 6.4 The login form of the application.**



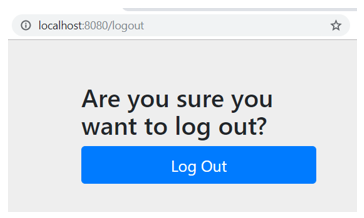
Once logged in, the application redirects you to the main page (figure 6.5). Here, the username taken from the security context appears on the page, together with the list of the products from the database.

**Figure 6.5 The main page of the application**



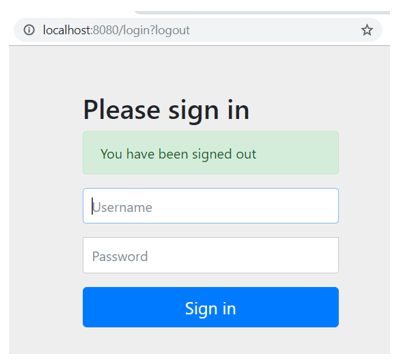
When you click the “Sign out here” link, the application redirects you to the standard sign out confirmation page (figure 6.6) as it was auto defined by Spring Security because we are using the Form Login authentication method.

**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.

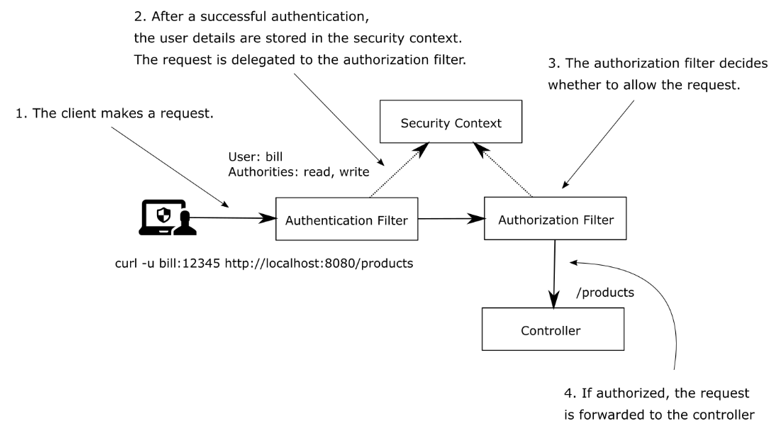
In this chapter, we discuss authorization. Authorization is the process during which the system decides if an identified client has permission to access the requested resource.

**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.

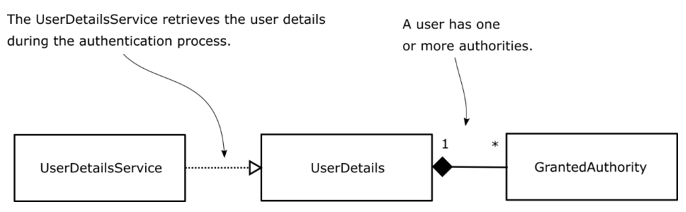
In chapter 8, we continue with selecting the endpoints to which we apply the authorization rules.

**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**();

}

[copy](javascript:void(0))

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

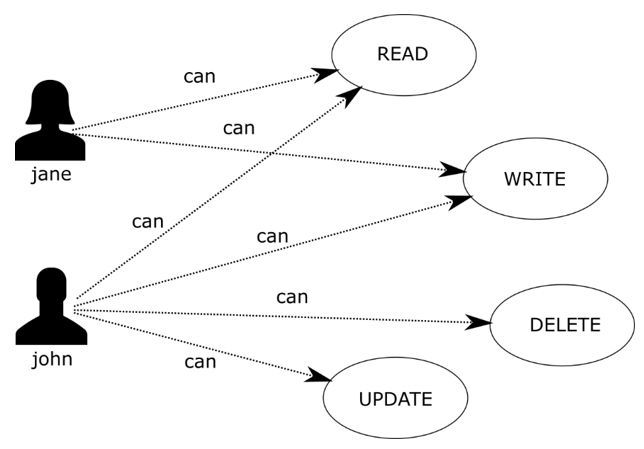
}

[copy](javascript:void(0))

**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**>

[copy](javascript:void(0))

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!";

}

}

[copy](javascript:void(0))

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

**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("WRITE")

.build();

manager.createUser(user1);

manager.createUser(user2);

**return** manager;

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

}

**A**

**B**

**C**

**D**

**E**

**F**

[copy](javascript:void(0))

#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**

[copy](javascript:void(0))

#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**

[copy](javascript:void(0))

#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”:

1

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The response body of the call is:

1

Hello

[copy](javascript:void(0))

Calling the endpoint with user “john”:

1

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

[copy](javascript:void(0))

The response body of the call is:

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

}

[copy](javascript:void(0))

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**

[copy](javascript:void(0))

#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.

1

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

[copy](javascript:void(0))

The response body is:

1

2

3

Hello!

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

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**

[copy](javascript:void(0))

#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]

!hasAuthority('delete')";

http.authorizeRequests()

.anyRequest()

.access(expression);

}

}

**A**

[copy](javascript:void(0))

#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”:

1

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

[copy](javascript:void(0))

The body of the response is:

1

Hello!

[copy](javascript:void(0))

Calling the endpoint with the user “jane”:

1

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The body of the response is:

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

}

[copy](javascript:void(0))

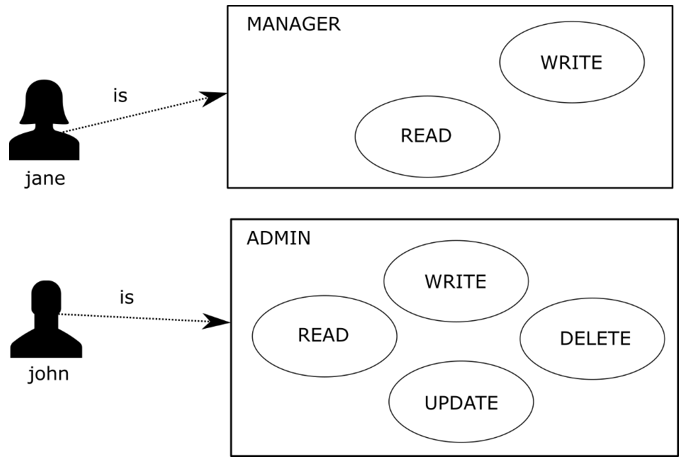
The user “john” has only the “read” authority and can call the endpoint successfully. But “jane” also has the “delete” authority and is not authorized to call the endpoint. The HTTP status for its call will be 403 Forbidden.

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")

.build();

**var** user2 = User.withUsername("jane")

.password("12345")

.authorities("ROLE\_MANAGER")

.build();

manager.createUser(user1);

manager.createUser(user2);

**return** manager;

}

// Omitted code

}

**A**

[copy](javascript:void(0))

#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**

[copy](javascript:void(0))

#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”:

1

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

[copy](javascript:void(0))

The response body is:

1

Hello

[copy](javascript:void(0))

Calling the endpoint with user “jane”:

1

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The response body is:

1

2

3

4

5

6

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

}

[copy](javascript:void(0))

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")

.build();

**var** user2 = User.withUsername("jane")

.password("12345")

.roles("MANAGER")

.build();

manager.createUser(user1);

manager.createUser(user2);

**return** manager;

}

// Omitted code

}

**A**

[copy](javascript:void(0))

#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:

1

**T**(**java**.time.LocalTime).now().isAfter(**T**(**java**.time.LocalTime).of(12, 0))

[copy](javascript:void(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**

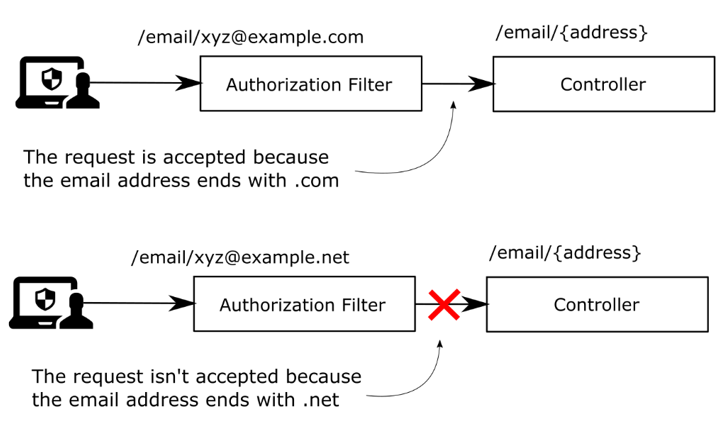
[copy](javascript:void(0))

#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.

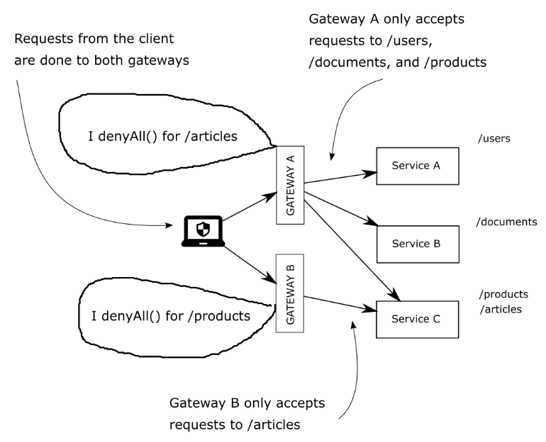
Let’s assume that you have an endpoint receiving as path variable an email address. What you want is to allow the requests that have the value of the variable addresses ending in .com. You don’t want the application to accept any other format for the email address. You’ll learn in the next session how 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!";

}

}

[copy](javascript:void(0))

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")

.mvcMatchers("/ciao").hasRole("MANAGER");

}

}

**A**

**B**

[copy](javascript:void(0))

#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”:

1

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

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

Calling the endpoint /hello with user “jane”:

1

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The response body is:

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

}

[copy](javascript:void(0))

Calling the endpoint /ciao with user “jane”:

1

curl -u jane:12345 http://localhost:8080/ciao

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

Calling the endpoint /ciao with user “john”:

1

curl -u john:12345 http://localhost:8080/ciao

[copy](javascript:void(0))

The response body is:

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/ciao"

}

[copy](javascript:void(0))

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!";

}

}

[copy](javascript:void(0))

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:

1

curl http://localhost:8080/hola

[copy](javascript:void(0))

The response body is:

1

Hola!

[copy](javascript:void(0))

Calling the endpoint /hola with user “john”:

1

curl -u john:12345 http://localhost:8080/hola

[copy](javascript:void(0))

The response body is:

1

Hola!

[copy](javascript:void(0))

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**

[copy](javascript:void(0))

#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.

1

curl http://localhost:8080/hola

[copy](javascript:void(0))

The response body is:

1

Hola!

[copy](javascript:void(0))

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.

1

curl -u bill:abcde http://localhost:8080/hola

[copy](javascript:void(0))

The response body is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/hola"

}

[copy](javascript:void(0))

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**

[copy](javascript:void(0))

#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:

1

**http**.csrf().disable();

[copy](javascript:void(0))

**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 **getEnpointB**() {

**return** "Works!";

}

@GetMapping("/a/b/c")

**public** String **getEnpointC**() {

**return** "Works!";

}

}

[copy](javascript:void(0))

We will also need a couple of users with different roles. To keep things simple, we will continue using an InMemoryUserDetailsManager. 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** 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();

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#A We define an InMemoryUserDetailsManager 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()

.mvcMatchers(HttpMethod.POST, "/a")

.permitAll()

.anyRequest()

.denyAll();

http.csrf().disable();

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#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:

1

curl -XPOST http://localhost:8080/a

[copy](javascript:void(0))

The response body is:

1

Works!

[copy](javascript:void(0))

When calling the path /a using HTTP GET and without authenticating:

1

curl -XGET http://localhost:8080/a

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/a"

}

[copy](javascript:void(0))

If you want to change the response to a successful one, you need to authenticate with a valid user. For the following call:

1

curl -u john:12345 -XGET http://localhost:8080/a

[copy](javascript:void(0))

The response body is:

1

Works!

[copy](javascript:void(0))

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.

1

curl -u john:12345 -XGET http://localhost:8080/a/b

[copy](javascript:void(0))

The response is:

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/a/b"

}

[copy](javascript:void(0))

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/\*\*")

.authenticated()

.anyRequest()

.permitAll();

http.csrf().disable();

}

}

**A**

[copy](javascript:void(0))

#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:

1

curl http://localhost:8080/a

[copy](javascript:void(0))

The response body is:

1

Works!

[copy](javascript:void(0))

Calling the /a/b path without authenticating:

1

curl http://localhost:8080/a/b

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/a/b"

}

[copy](javascript:void(0))

Calling the /a/b/c path without authenticating:

1

curl http://localhost:8080/a/b/c

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/a/b/c"

}

[copy](javascript:void(0))

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;

}

}

[copy](javascript:void(0))

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]

("/product/{code:^[0-9]\*$}")

.permitAll()

.anyRequest()

.denyAll();

}

}

**A**

[copy](javascript:void(0))

#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:

1

curl http://localhost:8080/product/1234a

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/product/1234a"

}

[copy](javascript:void(0))

Calling the endpoint using the value 12345:

1

curl http://localhost:8080/product/12345

[copy](javascript:void(0))

The response is:

1

12345

[copy](javascript:void(0))

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(HttpMethod 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(HttpMethod method), which is the equivalent of antMatchers(httpMethod, “/\*\*”). 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!";

}

}

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

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:

1

curl http://localhost:8080/hello

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/hello"

}

[copy](javascript:void(0))

Calling the /hello endpoint using the /hello/ path (with one more / at the end), unauthenticated:

1

curl http://localhost:8080/hello/

[copy](javascript:void(0))

The response is:

1

2

3

4

5

6

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/hello"

}

[copy](javascript:void(0))

Calling the /hello endpoint authenticating as “jane”:

1

curl -u jane:12345 http://localhost:8080/hello

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

Calling the /hello endpoint using the /hello/ path (with one more / at the end), authenticating as “jane”:

1

curl -u jane:12345 http://localhost:8080/hello/

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

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();

}

}

[copy](javascript:void(0))

In the next code snippets, you find the results for calling the endpoint with the /hello and /hello/ paths.

Calling the /hello endpoint unauthenticated:

1

curl http://localhost:8080/hello

[copy](javascript:void(0))

The response is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":"/hello"

}

[copy](javascript:void(0))

Calling the /hello endpoint unauthenticated, but using the path /hello/ (with one more / at the end):

1

curl http://localhost:8080/hello/

[copy](javascript:void(0))

The response is:

1

Hello!

[copy](javascript:void(0))

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 andteaching 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.”

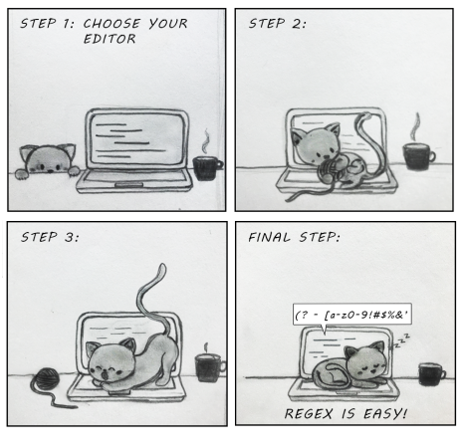
I realized then that the attendees might not be aware of what they were using. And I was right. They were indeed working with ANT matchers, but didn’t understand these configurations and were, I believe, most probably using them mechanically. Copy-paste programming is a risky approach, unfortunately 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.

**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/**](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;

}

}

[copy](javascript:void(0))

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)}")

.permitAll()

.anyRequest()

.denyAll();

[copy](javascript:void(0))

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:

1

**curl** http://localhost:8080/email/jane@example.com

[copy](javascript:void(0))

The response body is:

1

Allowed **for** email jane@example.com

[copy](javascript:void(0))

Calling the endpoint for email jane@example.net:

1

**curl** http://localhost:8080/email/jane@example.net

[copy](javascript:void(0))

The response body is:

{

"status":401,

"error":"Unauthorized",

"message":"Unauthorized",

"path":/email/jane@example.net

}

[copy](javascript:void(0))

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()

.regexMatchers(".\*/[us|uk|ca]+/[en|fr].\*")

.authenticated()

.anyRequest()

.hasAuthority("premium");

}

}

**A**

**B**

[copy](javascript:void(0))

#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:

1

curl -u john:12345 http://localhost:8080/video/us/en

[copy](javascript:void(0))

The response body is:

1

Video allowed **for** us en

[copy](javascript:void(0))

Calling the /video endpoint and authenticating with user “john” for region FR and language French:

1

curl -u john:12345 http://localhost:8080/video/fr/fr

[copy](javascript:void(0))

The response body is:

1

2

3

4

5

6

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/video/fr/fr"

}

[copy](javascript:void(0))

Having premium authority, the user “jane” makes both calls with success.

1

curl -u jane:12345 http://localhost:8080/video/us/en

[copy](javascript:void(0))

The response body is:

1

2

3

Video allowed **for** us en

curl -u jane:12345 http://localhost:8080/video/fr/fr

[copy](javascript:void(0))

The response body is:

1

Video allowed **for** fr fr

[copy](javascript:void(0))

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?

1

(?:[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+(?:\.[a-z0-9!#$%&'\*+/=?^\_`{|}~-]+)*\*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])\**")@(?:(?:[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]\*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]\*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])

[copy](javascript:void(0))

**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.

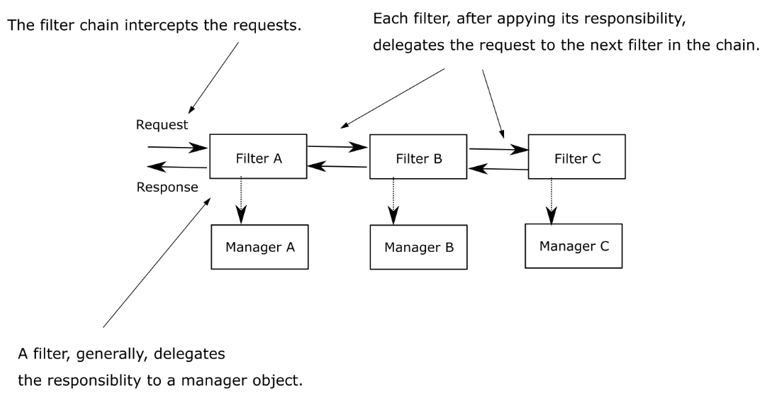
**9 Implementing filters**

**This chapter covers**

* Working with the filter chain.
* Defining custom filters.
* Using classes provided by Spring Security that implement the Filter interface.

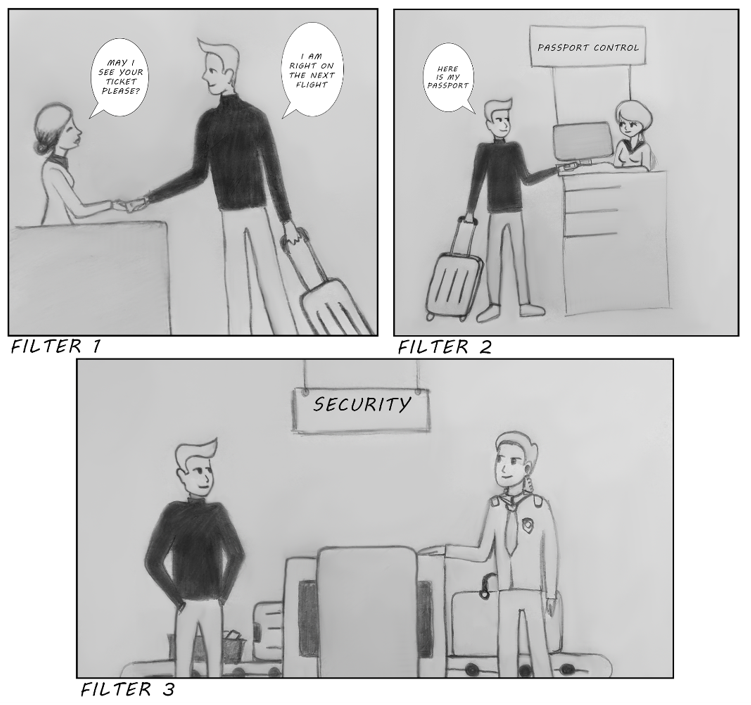
In Spring Security, the HTTP filters delegate the different responsibilities that apply to an HTTP request. In chapters 3 through 5, where we discussed the HTTP Basic authentication and authorization architecture, I’ve often referred to filters. You learned that there is a component we named authentication 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.

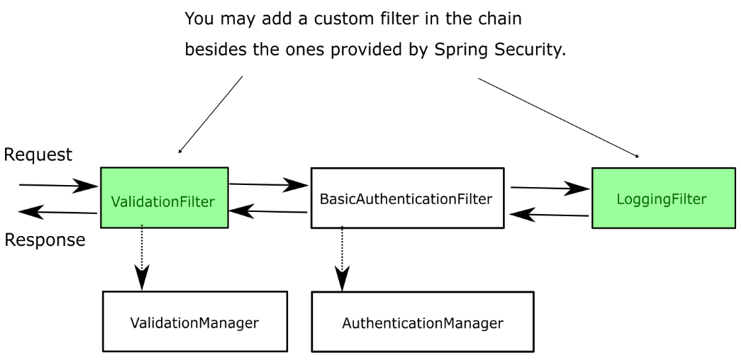
**Figure 9.2 At the airport, you go through a filter chain to eventually board the aircraft. In the same way, Spring Security has a filter chain that acts on the HTTP requests received by the application.**



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:

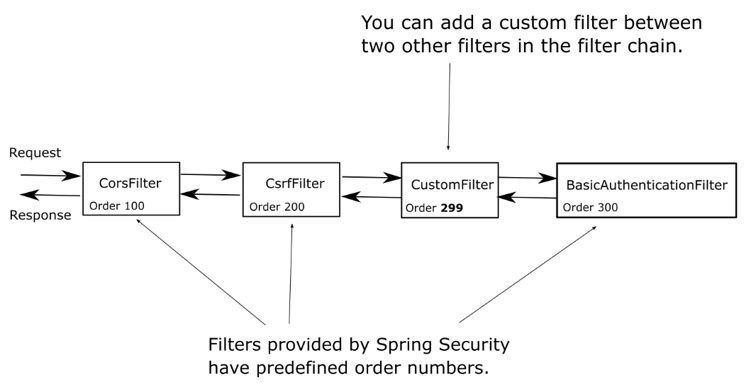
* BasicAuthenticationFilter, which takes care of the HTTP Basic authentication if present.
* CsrfFilter, that takes care of the CSRF protection, which we’ll discuss in chapter 10.
* CorsFilter, that takes care of Cross-Origin Resource Sharing authorization rules, which we’ll also discuss in chapter 10.

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.**



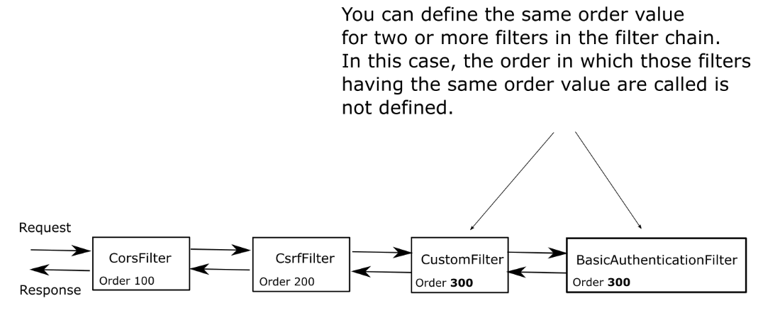
That makes it possible to add two or more filters in the same position.

**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.

**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 and existing one in the filter. You can then adapt this example to any similar requirement you’ll find in a production application.

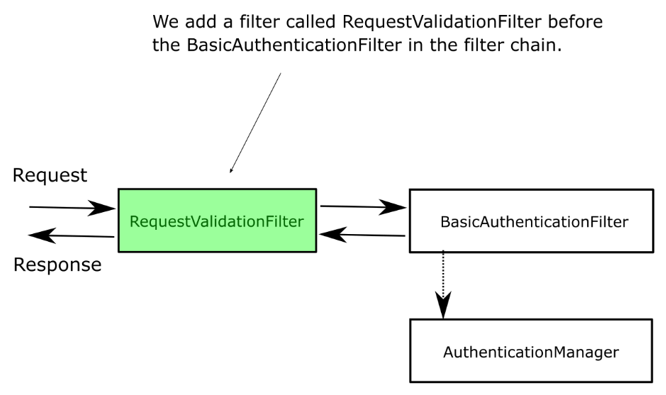
For our first custom filter implementation, let’s consider a trivial scenario. We want to make sure, before starting the authentication process, that any request has a header called Request-Id (project ssia-ch9-ex1). We assume our application uses this header for tracking the requests, and this header 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.

**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** {

@Override

**public** **void** **doFilter**(

ServletRequest servletRequest,

ServletResponse servletResponse,

FilterChain filterChain)

**throws** IOException, ServletException {

//…

}

}

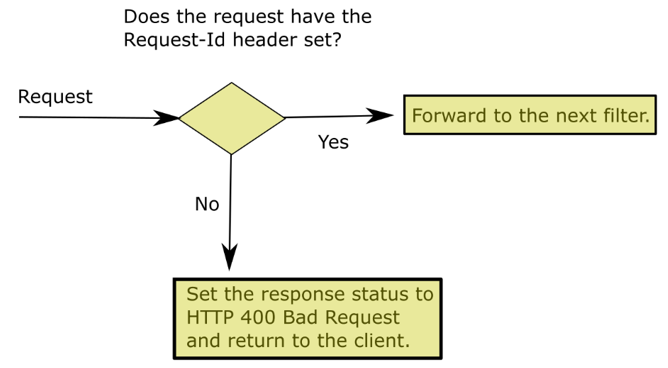
**A**

[copy](javascript:void(0))

#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(HttpServletResponse.SC\_BAD\_REQUEST);

**return**;

}

filterChain.doFilter(request, response);

}

**A**

**B**

[copy](javascript:void(0))

#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(

**new** RequestValidationFilter(),

BasicAuthenticationFilter.class)

.authorizeRequests()

.anyRequest().permitAll();

}

}

**A**

[copy](javascript:void(0))

#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!";

}

}

[copy](javascript:void(0))

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:

1

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

[copy](javascript:void(0))

This call will generate the following (truncated) response:

1

2

3

…

< HTTP/1.1 400

…

[copy](javascript:void(0))

Calling the endpoint and providing the Request-Id header:

1

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

[copy](javascript:void(0))

This call will generate the following (truncated) response:

1

Hello!

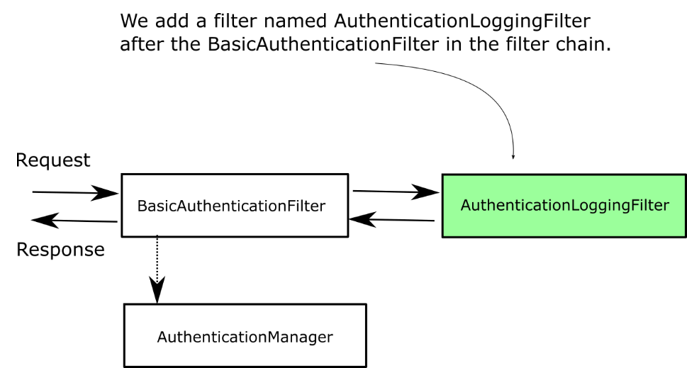
[copy](javascript:void(0))

**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.

**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");

logger.info("Successfully authenticated

request with id " + requestId);

filterChain.doFilter(request, response);

}

}

**A**

**B**

**C**

[copy](javascript:void(0))

#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(

**new** AuthenticationLoggingFilter(),

BasicAuthenticationFilter.class)

.authorizeRequests()

.anyRequest().permitAll();

}

}

**A**

[copy](javascript:void(0))

#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:

1

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

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

In the console, you’ll see a line similar to the one in the next code snippet:

1

**INFO** 5876 **---** [nio-8080-exec-2] **c**.l.s.f.AuthenticationLoggingFilter: **Successfully** **authenticated** **request** **with** **id** 12345

[copy](javascript:void(0))

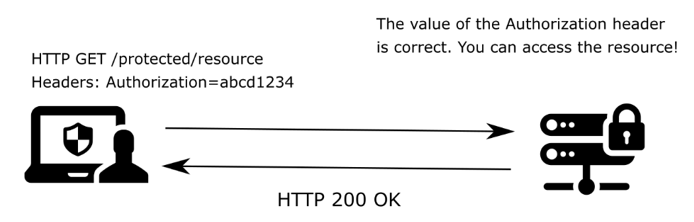
**9.4      Adding a filter at the location of another in the chain**

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.

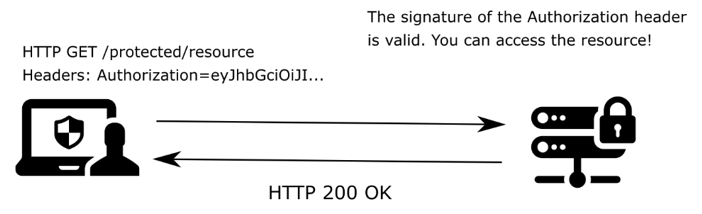
Identification based on a static key for authentication (figure 9.9): In this case, the client has to send a string, which is always the same. The application stores these values somewhere (most probably in a database or a secret vault) and identifies the client by making the request based on this value, 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.**



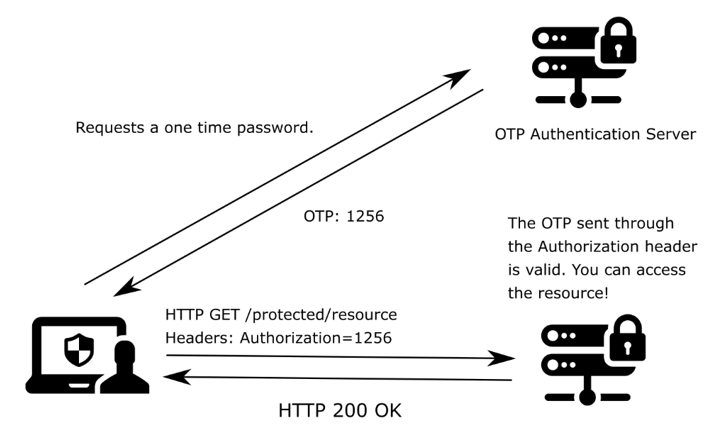
Using symmetric keys to sign and validate the requests (figure 9.10): In this case, both the client and the server know the value of a key (the client and the server share the key). The client uses this key to sign a part of the request (for example, to sign the value of specific headers), and the server 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.**



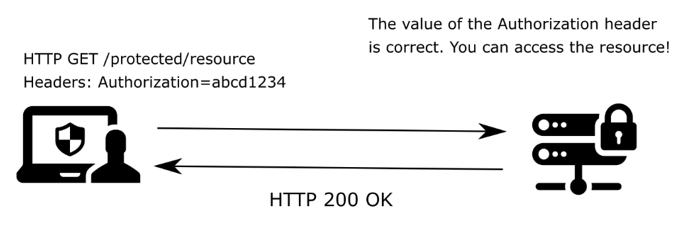
Use a one time password (OTP) received by the user via a message or by using an authentication provider app like Google Authenticator (figure 9.11).

**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

**public** **class** **StaticKeyAuthenticationFilter**

**implements** **Filter** {

@Value("${authorization.key}")

**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 =

httpRequest.getHeader("Authorization");

**if** (authorizationKey.equals(authentication)) {

filterChain.doFilter(request, response);

} **else** {

httpResponse.setStatus(

HttpServletResponse.SC\_UNAUTHORIZED);

}

}

}

**A**

**B**

**C**

**D**

[copy](javascript:void(0))

#A To allow us to inject values from the property file, the class is added as a component in the Spring context.

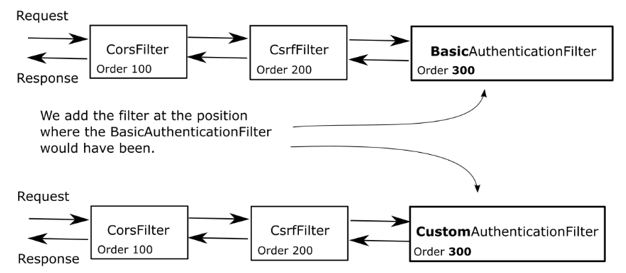
#B The class implements the Filter interface and overrides the doFilter() method to define the authentication logic.

#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

**private** StaticKeyAuthenticationFilter filter;

@Override

**protected** **void** **configure**(HttpSecurity http) **throws** Exception {

http.addFilterAt(filter,

BasicAuthenticationFilter.class)

.authorizeRequests()

.anyRequest().permitAll();

}

}

**A**

**B**

[copy](javascript:void(0))

#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.

1

authorization.key=SD9cICjl1e

[copy](javascript:void(0))

**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!

1

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

[copy](javascript:void(0))

The response body is:

1

Hello!

[copy](javascript:void(0))

If the Authorization header is missing or is incorrect, the response status will be HTTP 401 Unauthorized.

1

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

[copy](javascript:void(0))

The response status is:

…

< HTTP/1.1 401

…

[copy](javascript:void(0))

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 UserDetailService, you can use the exclude attribute of the @SpringBootApplication annotation on the main class to disable this auto-configuration:

1

2

@**SpringBootApplication**(**exclude** =

{**UserDetailsServiceAutoConfiguration**.class })

[copy](javascript:void(0))

**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** {

**private** **final** Logger logger =

Logger.getLogger(

AuthenticationLoggingFilter.class.getName());

@Override

**protected** **void** **doFilterInternal**(

HttpServletRequest request,

HttpServletResponse response,

FilterChain filterChain) **throws**

ServletException, IOException {

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

logger.info("Successfully authenticated request with id " +

requestId);

filterChain.doFilter(request, response);

}

}

**A**

**B**

**C**

[copy](javascript:void(0))

#A Instead of implementing the Filter interface, we extend the OncePerRequestFilter class.

#B We override the doFilterInternal() method which replaces the purpose of the doFiler() method of the Filter interface.

#C The OncePerRequestFilter only supports HTTP filters. This is why the paramters 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.

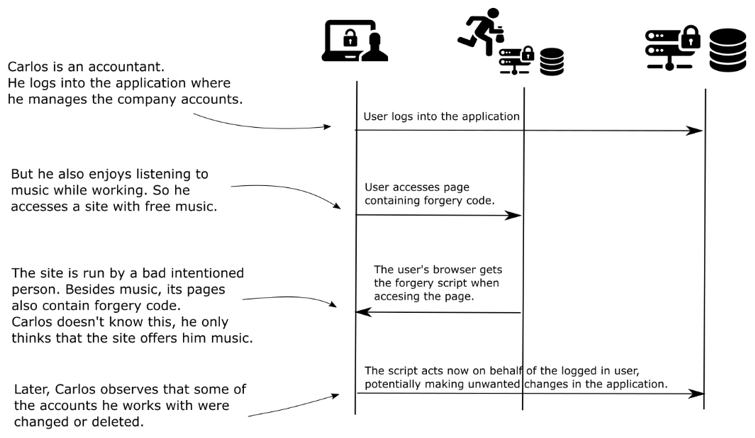
**10.1.1    How CSRF protection works in Spring Security**

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):

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.
2. 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.
3. 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.

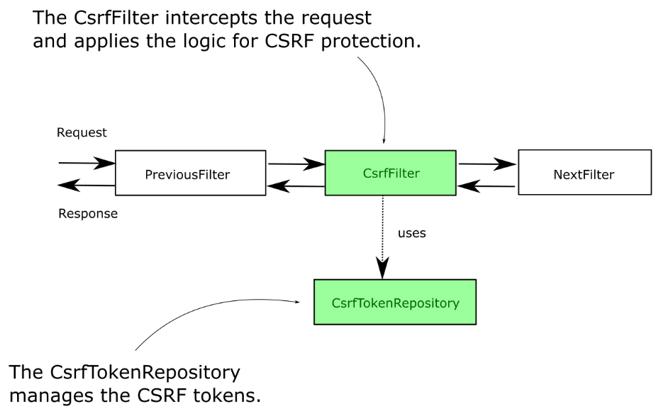
The starting point of the CSRF protection is a filter in the filter chain. This filter, which implements the CSRF protection logic, is called CsrfFilter. The CsrfFilter intercepts the requests and allows all those that use these HTTP methods: GET, HEAD, TRACE, and OPTIONS. For all the other requests, 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.**



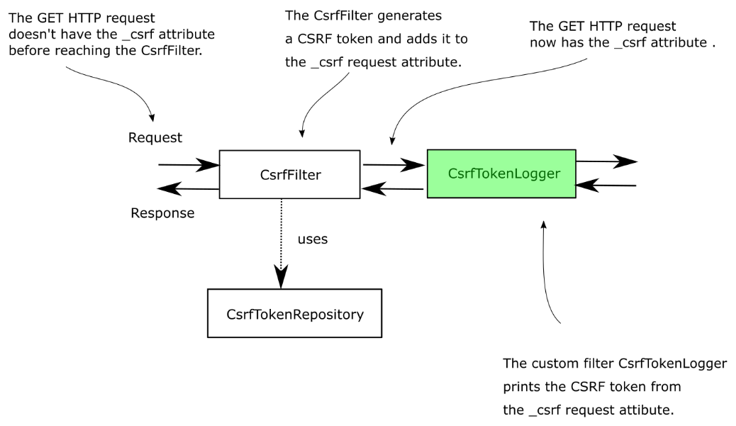
The CsrfFilter uses a component named CsrfTokenRepository to manage the CSRF token values: generate new tokens, store tokens, and eventually invalidate them. By default, the CsrfTokenRepository stores the token on the HTTP session, and the tokens are generated as random UUIDs. In most cases, this is 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.

**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!";

}

}

[copy](javascript:void(0))

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");

CsrfToken token = (CsrfToken) o;

logger.info("CSRF token " + token.getToken());

filterChain.doFilter(request, response);

}

}

**A**

[copy](javascript:void(0))

#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();

}

}

[copy](javascript:void(0))

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:

1

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

[copy](javascript:void(0))

The (truncated) response is:

1

2

3

4

…

< Set-Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206;

…

Get Hello!

[copy](javascript:void(0))

Following the request, in the application console you can find a log line that contains the CSRF token:

1

**INFO** 21412 **---** [nio-8080-exec-1] **c**.l.ssia.filters.CsrfTokenLogger : **CSRF** **token** **c5f0b3fa-2cae-4ca8-b1e6-6d09894603df**

[copy](javascript:void(0))

**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:

1

curl -XPOST http://localhost:8080/hello

[copy](javascript:void(0))

The response body is :

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

}

[copy](javascript:void(0))

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.

1

2

3

**curl** -X POST http://localhost:8080/hello

-H 'Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206'

-H 'X-CSRF-TOKEN: 1127bfda-57b1-43f0-bce5-bacd7d94694e'

[copy](javascript:void(0))

The response body is:

1

**Post** Hello!

[copy](javascript:void(0))

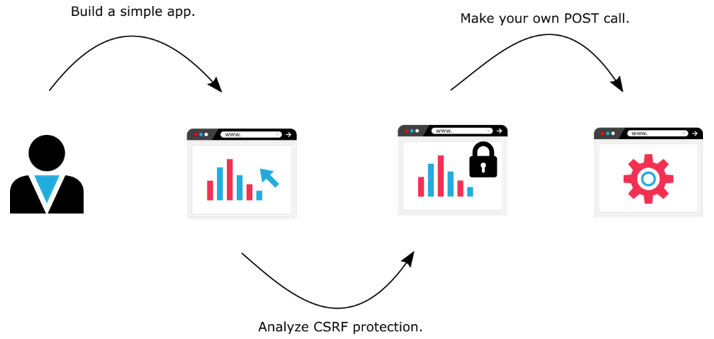
**10.1.2    Using CSRF protection in practical scenarios**

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**>

[copy](javascript:void(0))

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**

31

**public** **class** **ProjectConfig**

**extends** **WebSecurityConfigurerAdapter** {

@Bean

**public** UserDetailsService **uds**() {

**var** uds = **new** InMemoryUserDetailsManager();

**var** u1 = User.withUsername("mary")

.password("12345")

.authorities("READ")

.build();

uds.createUser(u1);

**return** uds;

}

@Bean

**public** PasswordEncoder **passwordEncoder**() {

**return** NoOpPasswordEncoder.getInstance();

}

@Override

**protected** **void** **configure**(HttpSecurity http) **throws** Exception {

http.authorizeRequests()

.anyRequest().authenticated();

http.formLogin()

.defaultSuccessUrl("/main", **true**);

}

}

**A**

**B**

**C**

[copy](javascript:void(0))

#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";

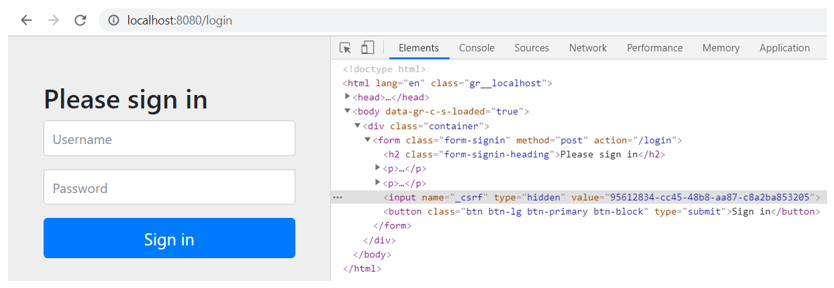
}

}

[copy](javascript:void(0))

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";

}

}

[copy](javascript:void(0))

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**>

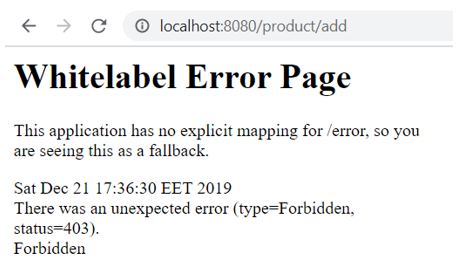
<**span**><**button** type="submit">Add</**button**></**span**>

</**form**>

[copy](javascript:void(0))

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"

th:name="${\_csrf.parameterName}"

th:value="${\_csrf.token}" />

</**form**>

**A**

**B**

[copy](javascript:void(0))

#A We use a hidden input to add to the request the CSRF token.

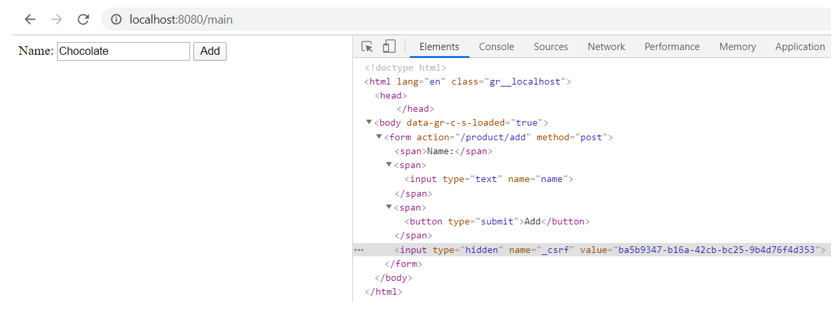
#B The “th” prefix we use is interpreted by Thymeleaf to print the token value.

**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.

1

**INFO** 20892 **---** [nio-8080-exec-7] **c**.l.s.controllers.ProductController : **Adding** **product** **Chocolate**

[copy](javascript:void(0))

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**>

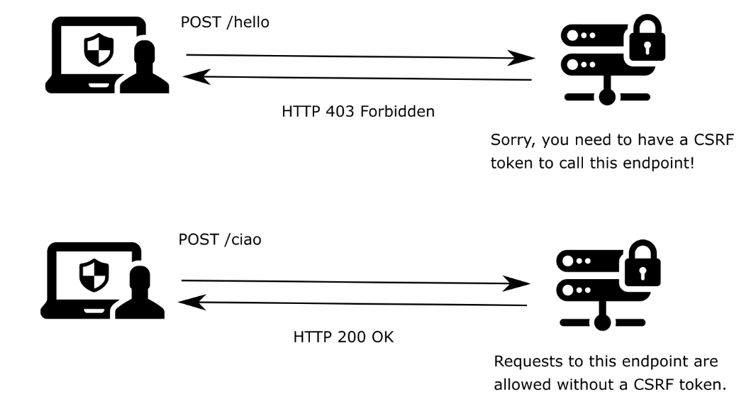
<**artifactId**>spring-boot-starter-web</**artifactId**>

</**dependency**>

[copy](javascript:void(0))

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")

**public** String **postHello**() {

**return** "Post Hello!";

}

@PostMapping("/ciao")

**public** String **postCiao**() {

**return** "Post Ciao";

}

}

**A**

**B**

[copy](javascript:void(0))

#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 -> {

c.ignoringAntMatchers("/ciao");

});

http.authorizeRequests()

.anyRequest().permitAll();

}

}

**A**

[copy](javascript:void(0))

#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().

1

2

3

HandlerMappingIntrospector i = **new** HandlerMappingIntrospector();

MvcRequestMatcher r = **new** MvcRequestMatcher(i, "/ciao");

c.ignoringRequestMatchers(r);

[copy](javascript:void(0))

You can similarly use a regex matcher as presented in the next code snippet.

1

2

3

4

String pattern = ".\*[0-9].\*";

String httpMethod = HttpMethod.POST.name();

RegexRequestMatcher r = **new** RegexRequestMatcher(pattern, httpMethod);

c.ignoringRequestMatchers(r);

[copy](javascript:void(0))

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**();

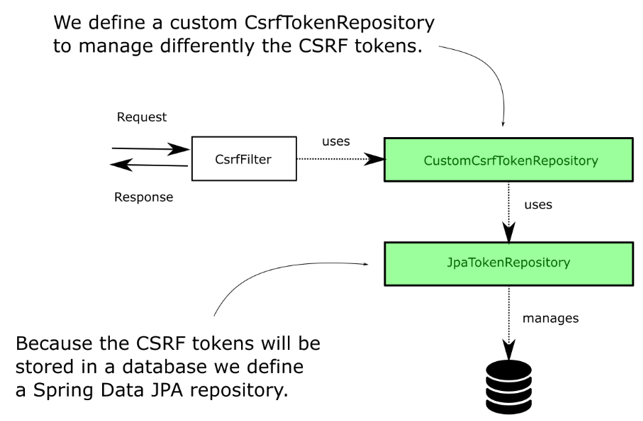
}

[copy](javascript:void(0))

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.

**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**>

[copy](javascript:void(0))

In the application.properties file of the application, we need to add the properties need for the database connection.

1

2

3

4

spring.datasource.url=jdbc:mysql://localhost/spring

spring.datasource.username=root

spring.datasource.password=

spring.datasource.initialization-mode=always

[copy](javascript:void(0))

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.

1

2

3

4

5

**CREATE** **TABLE** **IF** **NOT** **EXISTS** `spring`.`token` (

`id` INT **NOT** NULL AUTO\_INCREMENT,

`identifier` VARCHAR(45) NULL,

`token` TEXT NULL,

PRIMARY **KEY** (`id`));

[copy](javascript:void(0))

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;

**private** String token;

// Omitted code

}

**A**

**B**

[copy](javascript:void(0))

#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**

1

2

3

4

5

**public** **interface** **JpaTokenRepository**

**extends** **JpaRepository**<**Token**, **Integer**> {

Optional<Token> **findTokenByIdentifier**(String identifier);

}

[copy](javascript:void(0))

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) {

// …

}

}

[copy](javascript:void(0))

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);

}

[copy](javascript:void(0))

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()) {

Token token = existingToken.**get**();

token.setToken(csrfToken.getToken());

} **else** {

Token token = **new** Token();

token.setToken(csrfToken.getToken());

token.setIdentifier(identifier);

jpaTokenRepository.save(token);

}

}

**B**

**C**

[copy](javascript:void(0))

#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 recored 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",

"\_csrf",

token.getToken());

}

**return** null;

}

[copy](javascript:void(0))

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

**public** CsrfTokenRepository **customTokenRepository**() {

**return** **new** CustomCsrfTokenRepository();

}

@Override

**protected** **void** **configure**(HttpSecurity http)

**throws** Exception {

http.csrf(c -> {

c.csrfTokenRepository(customTokenRepository());

c.ignoringAntMatchers("/ciao");

});

http.authorizeRequests()

.anyRequest().permitAll();

}

}

**A**

**B**

[copy](javascript:void(0))

#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.

1

2

3

4

@GetMapping("/hello")

**public** String **getHello**() {

**return** "Get Hello!";

}

[copy](javascript:void(0))

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.

1

2

**curl** -H "X-IDENTIFIER:12345" http://localhost:8080/hello

Get Hello!

[copy](javascript:void(0))

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.

1

2

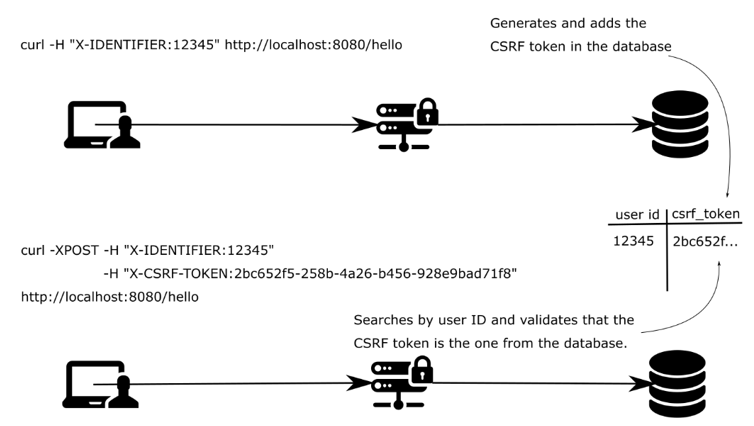
**curl** -XPOST -H "X-IDENTIFIER:12345" -H "X-CSRF-TOKEN:2bc652f5-258b-4a26-b456-928e9bad71f8" http://localhost:8080/hello

Post Hello!

[copy](javascript:void(0))

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.

1

curl -XPOST http://localhost:8080/hello

[copy](javascript:void(0))

The response body is :

{

"status":403,

"error":"Forbidden",

"message":"Forbidden",

"path":"/hello"

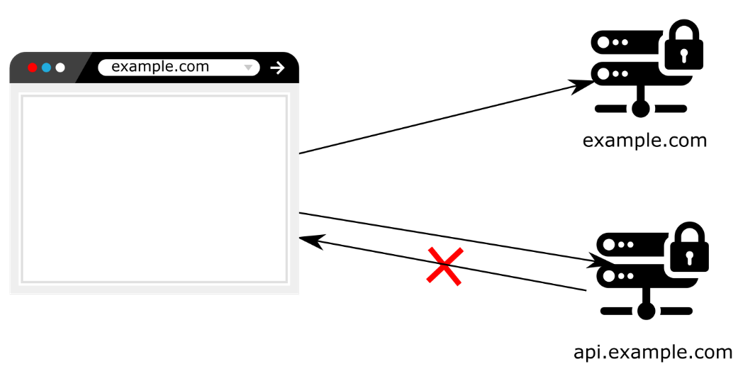
}

[copy](javascript:void(0))

**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](http://www.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.**

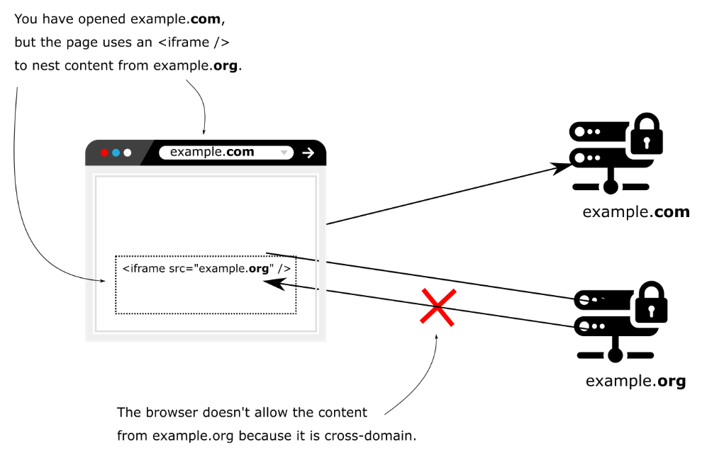


We can briefly say that CORS is a mechanism used by a browser to relax this strict policy and allow requests made between different origins in some conditions. You need to learn this because there’s a high chance that you’ll have to apply it with applications, especially nowadays, where usually the frontend 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.**



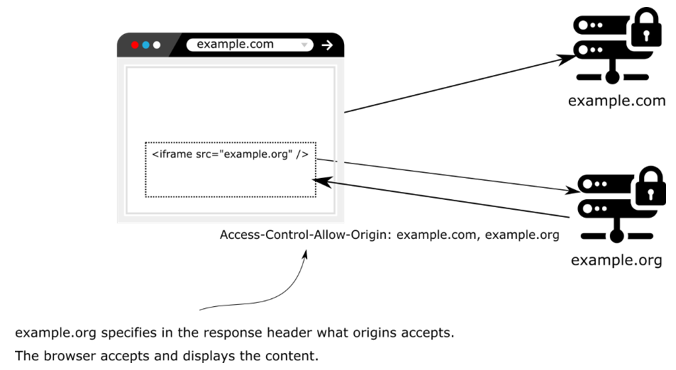
**NOTE**

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.

**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**>

[copy](javascript:void(0))

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 =

Logger.getLogger(MainController.class.getName());

@GetMapping("/")

**public** String **main**() {

**return** "main.html";

}

@PostMapping("/test")

@ResponseBody

**public** String **test**() {

logger.info("Test method called");

**return** "HELLO";

}

}

**A**

**B**

**C**

[copy](javascript:void(0))

#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();

}

}

[copy](javascript:void(0))

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';

http.open("POST", url);

http.send();

http.onreadystatechange = (e) => {

document

.getElementById("output")

.innerHTML = http.responseText;

}

</**script**>

</**head**>

<**body**>

<**div** id="output"></**div**>

</**body**>

</**html**>

**A**

**B**

[copy](javascript:void(0))

#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.

1

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.

[copy](javascript:void(0))

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.

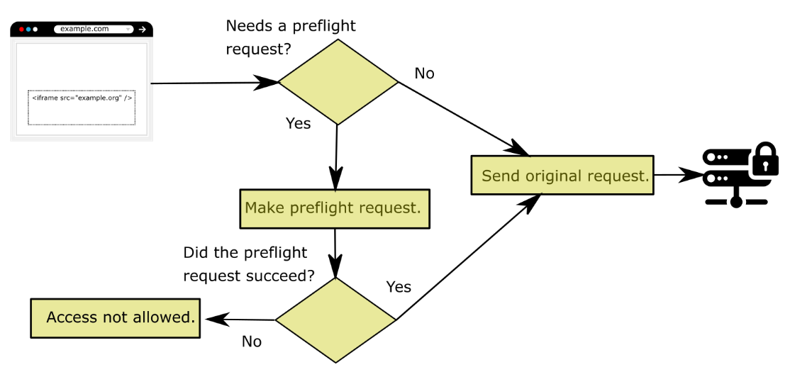
1

**INFO** 25020 **---** [nio-8080-exec-2] **c**.l.s.controllers.MainController : **Test** **method** **called**

[copy](javascript:void(0))

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")

**public** String **test**() {

logger.info("Test method called");

**return** "HELLO";

}

**A**

[copy](javascript:void(0))

#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 -> {

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**

[copy](javascript:void(0))

#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.

# Appendix 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).

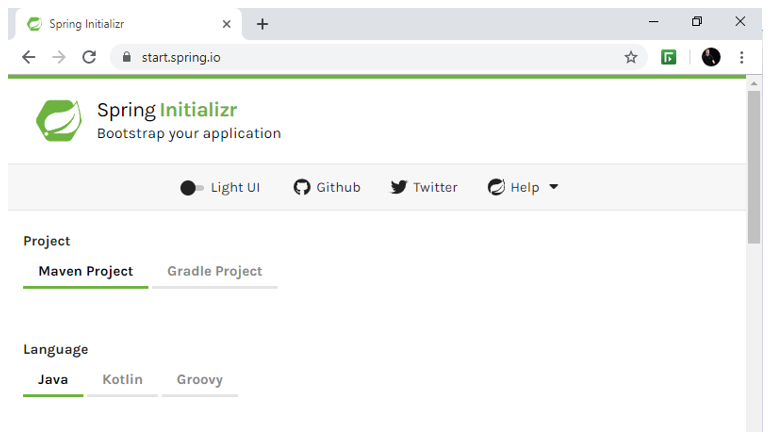
In this appendix, I present two easy options for creating your Spring projects. After creating your project, you can choose (at any time) to add other dependencies (by changing the pom.xml file in the case of Maven projects). Both options create projects with predefined Maven parents (if that’s what 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

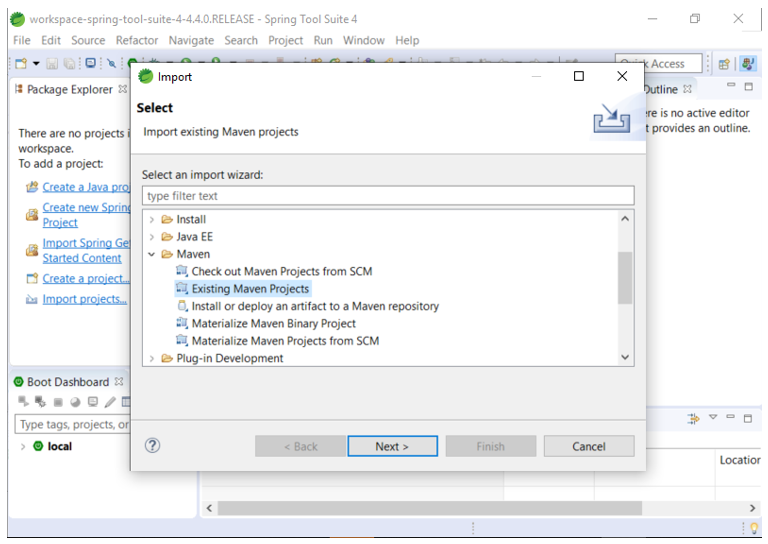
The most direct version of creating a Spring Boot project is by using the start.spring.io generator. From the web page, <https://start.spring.io/>, you can select all the options and dependencies you need, and then download the Maven or Gradle project of choice as a zip archive (figure A.1).

##### 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.

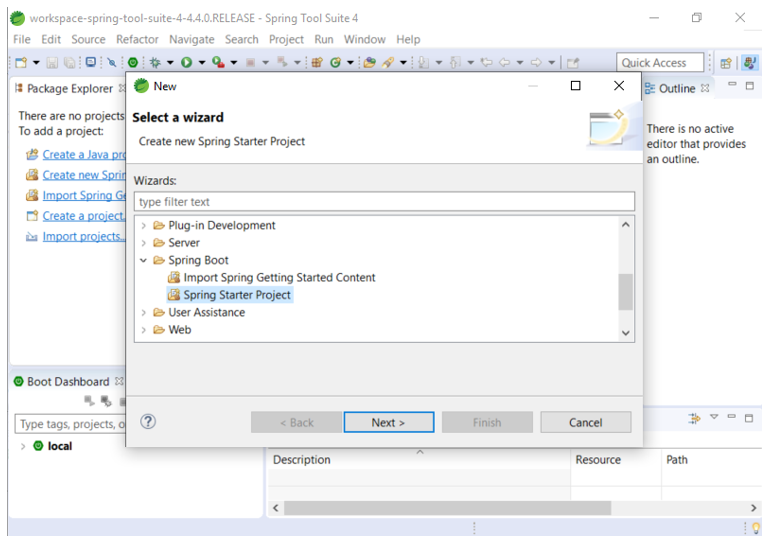
##### Figure A.2 An existing Maven project can be opened from any programming environment. Once you’ve created your project with start.spring.io and downloaded it, unzip it and open it as a Maven project from your IDE.



## 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.

