1

Spring I/O - Spring Security Training

Contents

1.1 Starting your first project

Let's create the first project so that we have something to work on for the first example. This project is a small web application, exposing a REST endpoint. You'll see how, without doing much, Spring Security secures this endpoint using HTTP Basic authentication. HTTP Basic is a way a web app authenticates a user by means of a set of credentials (username and password) that the app gets in the header of the HTTP request.

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

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

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 1.1. This project is enough to learn the first steps when developing an application with Spring Security and Spring Boot. It presents the basics of the Spring Security architecture for authentication and authorization.

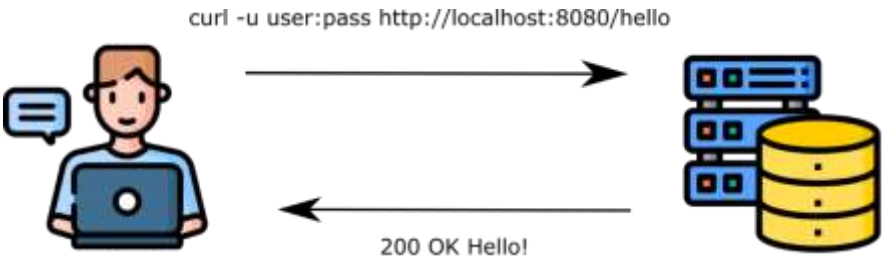

Figure 1.1 Our first application uses HTTP Basic to authenticate and authorize the user against an endpoint. The application exposes a REST endpoint at a defined path (/hello). For a successful call, the response returns an HTTP 200 status message and a body. This example demonstrates how the authentication and authorization configured by default with Spring Security works.

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 1.1. 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 1.1 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. In the following listing, the `HelloController` class defines a REST controller and a REST endpoint for our example.

Listing 1.2 The `HelloController` class and a REST endpoint

```
@RestController
public class HelloController {

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

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

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

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 sends the call, we need to know how to form the correct value for the `Authorization` header. You do this using the Base64 tool in a Linux console. You could also find a web page that encodes strings in Base64, like https://www.base64encode.org. This snippet shows the command in a Linux or a Git Bash console (the -n parameter means no trailing new line should be added):

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

Running this command returns this Base64-encoded string:

```
dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01NDg2MGYzNmY3ZjM=
```

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

```
curl -H "Authorization: Basic dXNlcjo5M2EwMWNmMC03OTRiLTRiOTgtODZlZi01
NDg2MGYzNmY3ZjM="  localhost:8080/hello
```

```
The result of the call is

Hello!
```

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

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

1.2 The big picture of Spring Security class design

Let's 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 section. It would be too much to discuss these all at once, so to minimize your learning efforts in this section, I'll discuss the high-level picture for each component.

You saw earlier 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 1.2, you can see the big picture of the main actors (components) in the Spring Security architecture and the relationships among these. These components have a preconfigured implementation in the first project. Here, I make you aware of what Spring Boot configures in your application in terms of Spring Security. We'll also discuss the relationships among the entities that are part of the authentication flow presented.

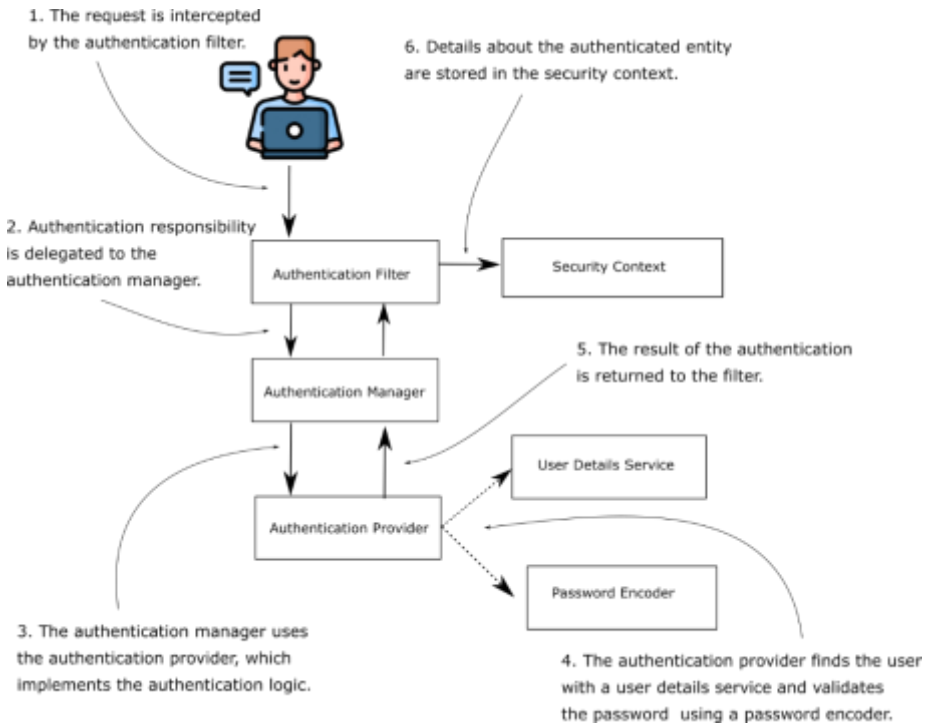

Figure 1.2 The main components acting in the authentication process for Spring Security and the relationships among these. This architecture represents the backbone of implementing authentication with Spring Security.

In figure 1.2, you observe 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 1.2 as well. The authentication provider uses these beans to find users and to check their passwords. Let's start with the way you provide the needed credentials for authentication.

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

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

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

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

Even if it's not as obvious as the `UserDetailsService` object, the `PasswordEncoder` is mandatory for the Basic authentication flow. The simplest implementation manages the passwords in plain text and doesn't encode these.

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.

1.3 Overriding convention 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, the development process is also about how you write configurations to keep your applications highly maintainable. With the projects we'll work on, you'll often find multiple ways to override a configuration. This flexibility can create confusion. I frequently see a mix of different styles of configuring different parts of Spring Security in the same application, which is undesirable. So this flexibility comes with a caution. You need to learn how to choose from these, so this section is also about knowing what your options are.

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

In this section, you'll learn how to configure a `UserDetailsService` and a `PasswordEncoder`. These two components usually take part in authentication, and most applications customize them depending on their requirements. The implementations we use in this section are all provided by Spring Security.

1.4 Customizing user details management

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

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

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

> **NOTE** An `InMemoryUserDetailsManager` implementation isn't meant for production-ready applications, but it's an excellent tool for examples or proof of concepts.

In some cases, all you need is users. You don't need to spend the time implementing this part of the functionality. In our case, we use it to understand how to override the default `UserDetailsService` implementation.

We start by defining a configuration class. Generally, we declare configuration classes in a separate package named config. Listing 1.3 shows the definition for the configuration class.

Listing 1.3 The configuration class for the `UserDetailsService` bean

```
@Configuration       #A
public class ProjectConfig {

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

#A The @Configuration annotation marks the class as a configuration class.
#B The @Bean annotation instructs Spring to add the returned value as a bean in the Spring context.

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 1.2, 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`.

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 1.4, I name the authority `read`, but because we won't use this authority for the moment, this name doesn't really matter.

Listing 1.4 Creating a user with the `User` builder class for `UserDetailsService`

```
@Configuration
```

```
public class ProjectConfig {

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

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

#A Builds the user with a given username, password, and authorities list
#B Adds the user to be managed by UserDetailsService

As presented in listing 1.4, we have to provide a value for the username, one for the password, and at least one authority. But this is still not enough to allow us to call the endpoint. We also need to declare a `PasswordEncoder`.

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

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

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

```
java.lang.IllegalArgumentException:
There is no PasswordEncoder mapped for the id "null"
    at
org.springframework.security.crypto.password
.DelegatingPasswordEncoder$
UnmappedIdPasswordEncoder.matches(DelegatingPasswordEncoder.java:289)
~[spring-security-crypto-6.0.0.jar:6.0.0]
    at org.springframework.security.crypto.password
.DelegatingPasswordEncoder.matches(DelegatingPasswordEncoder.java:237)
~[spring-security-crypto-6.0.0.jar:6.0.0]
```

To solve this problem, we can add a `PasswordEncoder` bean in the context, the same as we did with the `UserDetailsService`. For this bean, we 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 1.5 The full definition of the configuration class

```
@Configuration
public class ProjectConfig {

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

    return new InMemoryUserDetailsManager(user);
  }

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

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

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

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

1.5 Applying authorization at the endpoint level

Before diving into details, you must understand the big picture. And the best way to achieve this is with our first example. With default configuration, all the endpoints assume you have a valid user managed by the application. Also, by default, your app uses HTTP Basic authentication, but you can easily override this configuration.

As you'll learn next, HTTP Basic authentication doesn't fit into most application architectures. Sometimes we'd like to change it to match our application. Similarly, not all endpoints of an application need to be secured, and for those that do, we might need to choose different authentication methods and authorization rules. To customize authentication and authorization, we'll need to define a bean of type `SecurityFilterChain`.

Listing 1.6 Defining a `SecurityFilterChain` bean

```
@Configuration
public class ProjectConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {


    return http.build();
  }

  // Omitted code

}
```

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

Listing 1.7 Using the `HttpSecurity` parameter to alter the configuration

```
@Configuration
public class ProjectConfig {

  @Bean
  SecurityFilterChain configure(HttpSecurity http)
    throws Exception {

    http.httpBasic();    #A
    http.authorizeHttpRequests()
          .anyRequest().authenticated();    #B

    return http.build();
  }

  // Omitted code

}
```

#A App uses HTTP Basic authentication.
#B All the requests require authentication.

The code in listing 1.7 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. 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 1.8 Using `permitAll()` to change the authorization configuration

```
@Configuration
public class ProjectConfig {

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

```
        http.httpBasic();

        http.authorizeHttpRequests()
              .anyRequest().permitAll();     #A

        return http.build();
      }

      // Omitted code
  }
```

#A None of the requests need to be authenticated.

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

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

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.

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

1.6 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.

1.7 Describing users with the UserDetails contract

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

Listing 1.9 The `UserDetails` interface

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

#A These methods return the user credentials.
#B Returns the actions that the app allows the user to do as a collection of GrantedAuthority instances
#C 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.
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.

1.8 Detailing on the GrantedAuthority contract

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

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

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

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

Now, we'll implement the `getAuthority()` method to return the authority's name as a `String`. The `GrantedAuthority` interface has only one abstract method, 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");
```

1.9 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 1.10. 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 1.10 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 1.10 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 1.11 shows the implementation of the `getAuthorities()` method. This method returns a collection with only one implementation of the `GrantedAuthority` interface.

Listing 1.11 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 1.12 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 1.13 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

}
```

1.10 Using a builder to create instances of the UserDetails type

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

The `User` class from the org.springframework.security.core.userdetails package is a simple way to build instances of the `UserDetails` type. Using this class, you can create immutable instances of `UserDetails`. You need to provide at least a username and a password, and the username shouldn't be an empty string. Listing 1.14 demonstrates how to use this builder. Building the user in this way, you don't need to have a custom implementation of the `UserDetails` contract.

Listing 1.14 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 1.15, 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 1.15 Creating the User.UserBuilder instance

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

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

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

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

#A Builds a user with their username
#B The password encoder is only a function that does an encoding.
#C At the end of the build pipeline, calls the build() method

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

You can see with any of the builders defined in listing 1.15 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.

1.11 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 1.16 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 1.17 The `User` class has two responsibilities

```
@Entity
public class User implements UserDetails {

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

  @Override
  public String getUsername() {
```

```
    return this.username;
  }

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

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

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

  // Omitted code

}
```

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

How can we write this code to be cleaner? The root of the muddy aspect of the previous code example is a mix of two responsibilities. While it's true that you need both in the application, in this case, nobody says that you have to put these into the same class. Let's try to separate those by defining a separate class called `SecurityUser`, which decorates the `User` class. As listing 1.18 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 1.18 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 1.19 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 1.19 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.

1.12 Instructing Spring Security on how to manage users

Let's experiment with various ways of implementing the `UserDetailsService` class. You'll understand how user management works by implementing the responsibility described by the `UserDetailsService` contract in our example. After that, you'll find out how the `UserDetailsManager` interface adds more behavior to the contract defined by the `UserDetailsService`. At the end of this section, we'll use the provided implementations of the `UserDetailsManager` interface offered by Spring Security.

We'll write an example project where we'll use one of the best known implementations provided by Spring Security, the `JdbcUserDetailsManager` class. Learning this, you'll know how to tell Spring Security where to find users, which is essential in the authentication flow.

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 1.3). The username is, of course, considered unique. The user returned by this method is an implementation of the `UserDetails` contract. If the username doesn't exist, the method throws a `UsernameNotFoundException`.

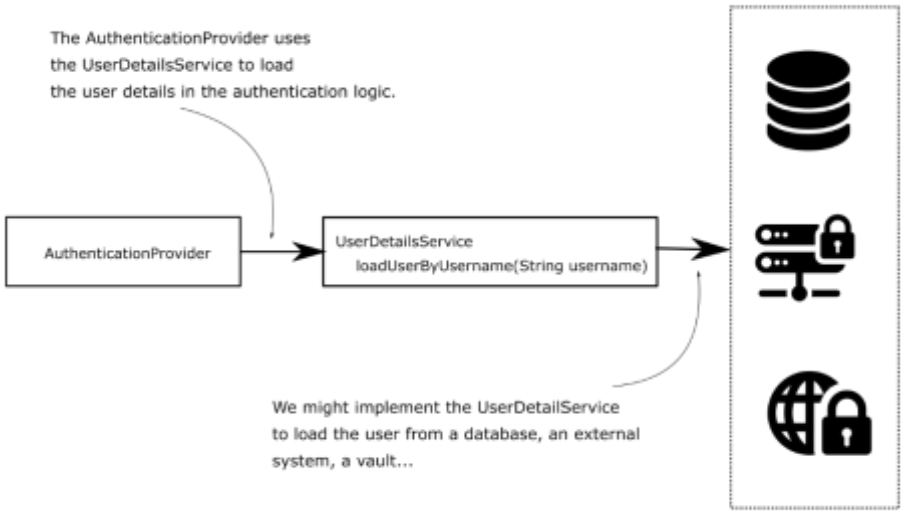

Figure 1.3 The `AuthenticationProvider` is the component that implements the authentication logic and uses the `UserDetailsService` to load details about the user. To find the user by username, it calls the `loadUserByUsername(String username)` method.

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.

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 1.3). 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. Earlier, 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.

Listing 1.20 The implementation of the `UserDetails` interface

```
public class User implements UserDetails {

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

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

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

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

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

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

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

```
  }

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

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

#A The User class is immutable. You give the values for the three attributes when you build the instance, and these values cannot be changed afterward.
#B To make the example simple, a user has only one authority.
#C Returns a list containing only the GrantedAuthority object with the name provided when you built the instance
#D 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 1.21 The implementation of the `UserDetailsService` interface

```
public class InMemoryUserDetailsService implements UserDetailsService {

  private final List<UserDetails> users;     #A

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

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

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

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

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 1.22 `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 1.23 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!
```

1.13 Using password encoders

Different actors manage user representation during the authentication and authorization processes. You 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 1.4 reminds you of where the `PasswordEncoder` fits into the authentication process.

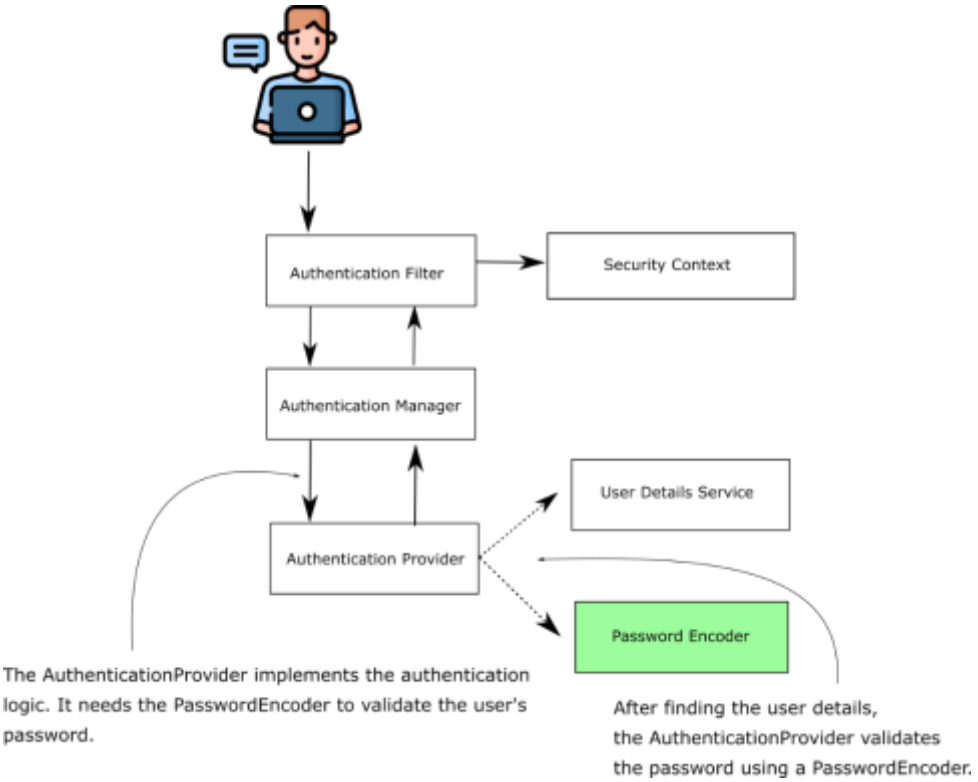

Figure 1.4 The Spring Security authentication process. The `AuthenticationProvider` uses the `PasswordEncoder` to validate the user's password in the authentication process.

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.

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.

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.

Listing 1.24 The simplest implementation of a `PasswordEncoder`

```
public class PlainTextPasswordEncoder
  implements PasswordEncoder {

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

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

```
}
```

#A We don't change the password, just return it as is.
#B 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 1.25 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 1.25, we use a method to hash the string value provided with SHA-512. I omit the implementation of this method in listing 1.25, but you can find it in listing 1.26. 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 1.26 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.

1.14 Choosing from the provided PasswordEncoder implementations

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

- `NoOpPasswordEncoder`—Doesn't encode the password but keeps it in clear-text. We use this implementation only for examples. Because it doesn't hash the password, *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. Preferably, if you find it in existing apps, you should change it with some other, more powerful password encoder.
- `Pbkdf2PasswordEncoder`—Uses the password-based key derivation function 2 (PBKDF2).
- `BCryptPasswordEncoder`—Uses a bcrypt strong hashing function to encode the password.
- `SCryptPasswordEncoder`—Uses an scrypt hashing function to encode the password.

For more about hashing and these algorithms, you can find a good discussion in chapter 2 of *Real-World Cryptography* by David Wong (Manning, 2021). Here's the link:

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

Let's take a look at some examples of how to create instances of these types of `PasswordEncoder` implementations. The `NoOpPasswordEncoder` doesn't encode the password. It has an implementation similar to the `PlainTextPasswordEncoder` from our example earlier. 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 document, 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.

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.

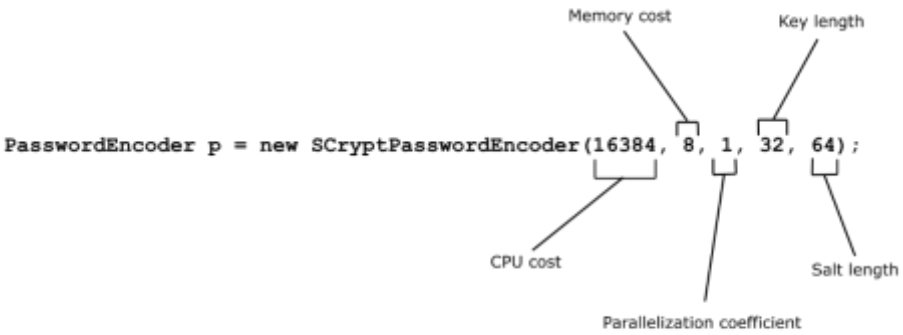

Figure 1.5 The `SCryptPasswordEncoder` constructor takes five parameters and allows you to configure CPU cost, memory cost, key length, and salt length.

1.15 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. If we want to execute certain logic before authentication, we do this by inserting a filter before the authentication filter.

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

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

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

The *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.
- `CorsFilter` takes care of cross-origin resource sharing (CORS) authorization rules.

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 document, 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.

You add a new filter to the chain relative to another one (figure 1.6). Or, you can add a filter either before, after, or at the position of a known one. Each position is, in fact, an index (a number), and you might find it also referred to as "the order."

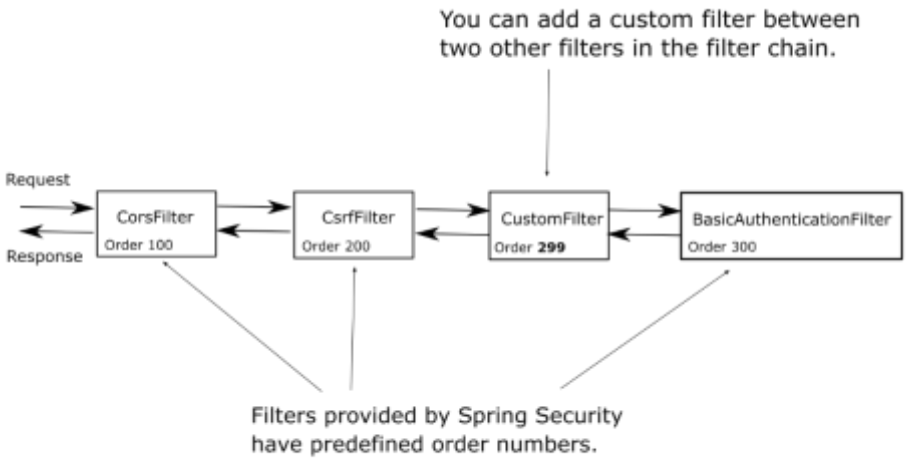

Figure 1.6 Each filter has an order number. This determines the order in which filters are applied to a request. You can add custom filters along with the filters provided by Spring Security.

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

You can add two or more filters in the same position (figure 1.7).

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

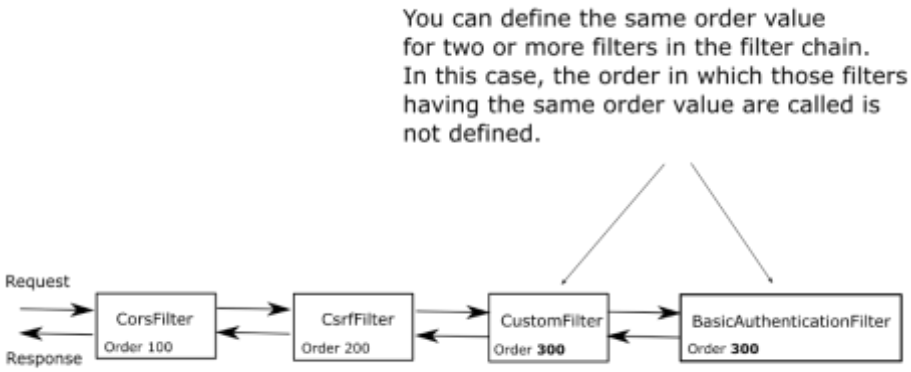

Figure 1.7 You might have multiple filters with the same order value in the chain. In this case, Spring Security doesn't guarantee the order in which they are called.

1.16 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`. 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 1.8:

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.

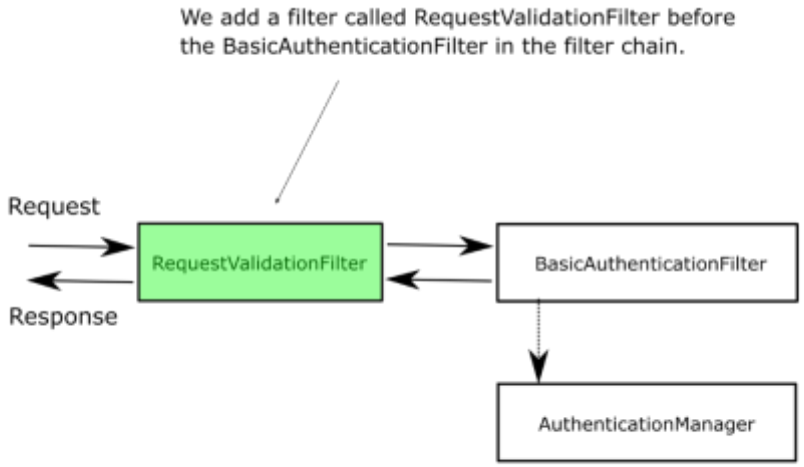

Figure 1.8 For our example, we add a `RequestValidationFilter`, which acts before the authentication filter. The `RequestValidationFilter` ensures that authentication doesn't happen if the validation of the request fails. In our case, the request must have a mandatory header named `Request-Id`.

To accomplish step 1, implementing the filter, we define a custom filter. The next listing shows the implementation.

Listing 1.27 Implementing a custom filter

```
public class RequestValidationFilter
  implements Filter {      #A

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

#A To define a filter, 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 5.7). Listing 1.28 presents the logic.

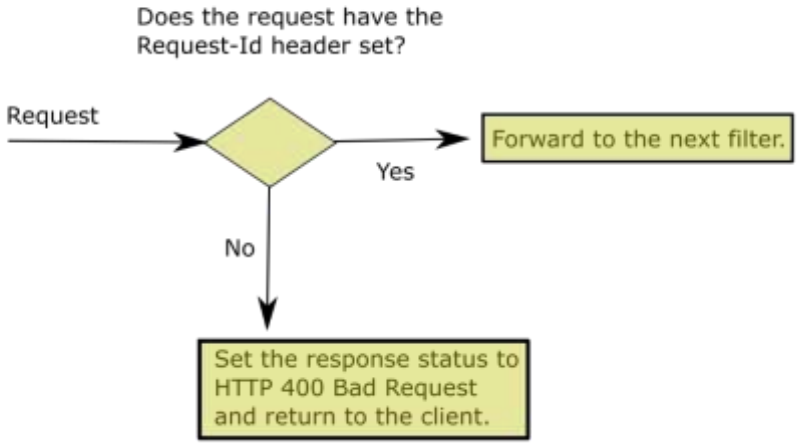

Figure 1.9 The custom filter we add before authentication checks whether the `Request-Id` header exists. If the header exists on the request, the application forwards the request to be authenticated. If the header doesn't exist, the application sets the HTTP status 400 Bad Request and returns to the client.

Listing 1.28 Implementing the logic in the `doFilter()` method

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

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

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

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

  filterChain.doFilter(request, response);    #B

}
```

#A 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.
#B If the header exists, the request is 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`.
- *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.

Listing 1.29 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 1.29 Configuring the custom filter before authentication

```
@Configuration
public class ProjectConfig {

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

    http.addFilterBefore(     #A
            new RequestValidationFilter(),
            BasicAuthenticationFilter.class)
        .authorizeRequests()
           .anyRequest().permitAll();

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

#A 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 1.30 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!
```

1.17 Adding a filter after an existing one in the 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 1.10). 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 1.10). We consider that what bypasses the authentication filter represents a successfully authenticated event and we want to log it. Continuing the example from section before, we also log the request ID received through the HTTP header.

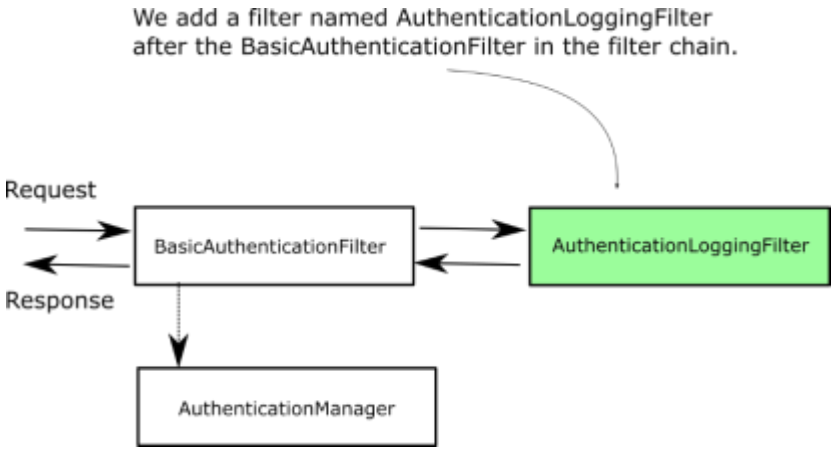

Figure 1.10 We add the `AuthenticationLoggingFilter` after the `BasicAuthenticationFilter` to log the requests that the application authenticates.

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

Listing 1.31 Defining a filter to log requests

```
public class AuthenticationLoggingFilter implements Filter {

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

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

      var httpRequest = (HttpServletRequest) request;

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

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

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

#A Gets the request ID from the request headers
#B Logs the event with the value of the request ID
#C Forwards the request to the next filter in the chain

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 1.32 Adding a custom filter after an existing one in the filter chain

```
@Configuration
public class ProjectConfig {

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

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

    return http.build();
```

```
    }
}
```

#A Adds an instance of AuthenticationLoggingFilter to the filter chain after the authentication filter

Running the application and calling the endpoint, we observe that for every successful call to the endpoint, the application prints a 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
```

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

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

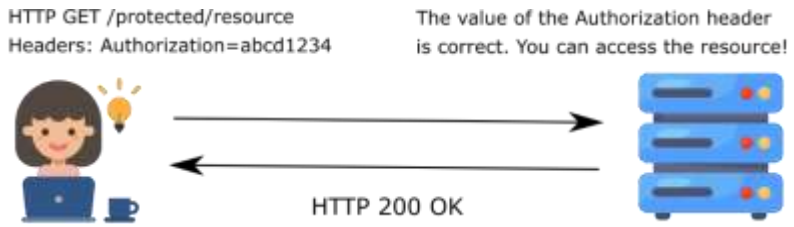

Figure 1.11 The request contains a header with the value of the static key. If this value matches the one known by the application, it accepts the request.

In our second scenario, using symmetric keys to sign and validate requests, both client and server know the value of a key (client and server share the key). The client uses this key to sign a part of the request (for example, to sign the value of specific headers), and the server checks if the signature is valid using the same key (figure 1.12). 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.

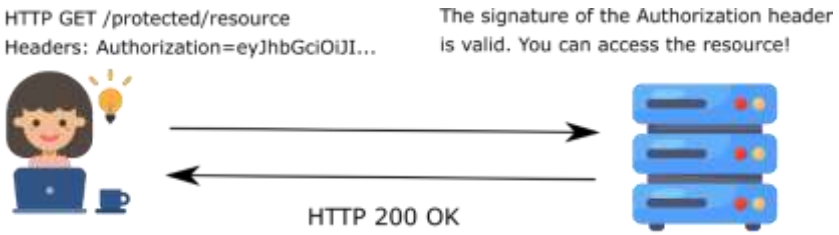

Figure 1.12 The `Authorization` header contains a value signed with a key known by both client and server (or a private key for which the server has the public key). The application checks the signature and, if correct, allows the request.

And finally, for our third scenario, using an OTP in the authentication process, the user receives the OTP via a message or by using an authentication provider app like Google Authenticator (figure 1.13).

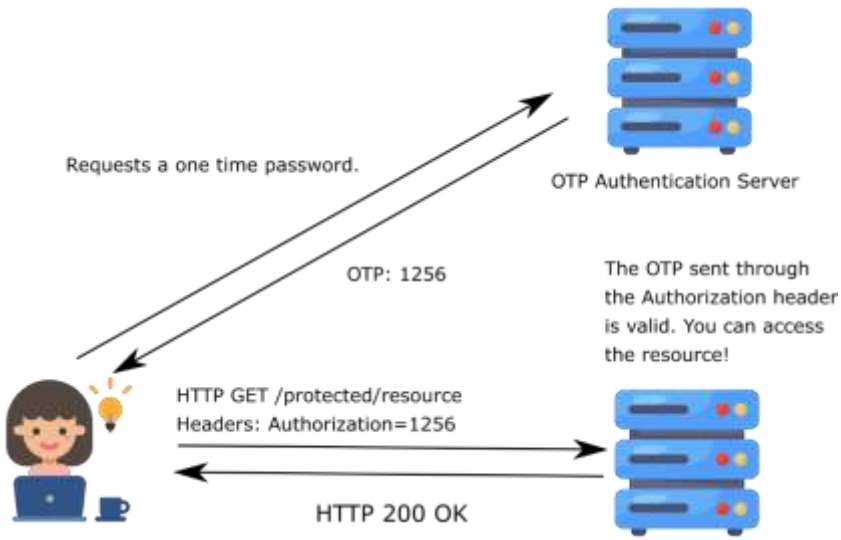

Figure 1.13 To access the resource, the client has to use a one-time password (OTP). The client obtains the OTP from a third-party authentication server. Generally, applications use this approach during login when multifactor authentication is required.

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 1.14.

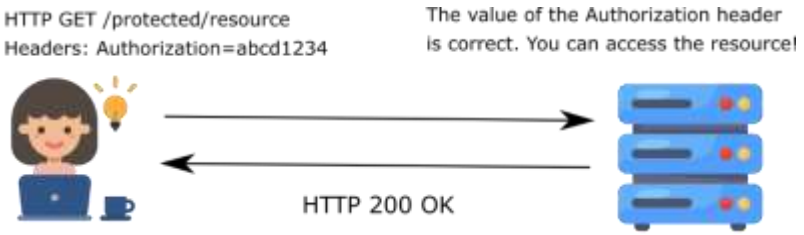

Figure 1.14 The client adds a static key in the `Authorization` header of the HTTP request. The server checks if it knows the key before authorizing the requests.

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 1.33 defines the `StaticKeyAuthenticationFilter` class.

Listing 1.33 The definition of the `StaticKeyAuthenticationFilter` class

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

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

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

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

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

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

```
                                HttpServletResponse.SC_UNAUTHORIZED);
        }
      }
    }
```

#A To allow us to inject values from the properties file, adds an instance of the class in the Spring context
#B Defines the authentication logic by implementing the Filter interface and overriding the doFilter() method
#C Takes the value of the static key from the properties file using the @Value annotation
#D 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 1.15).

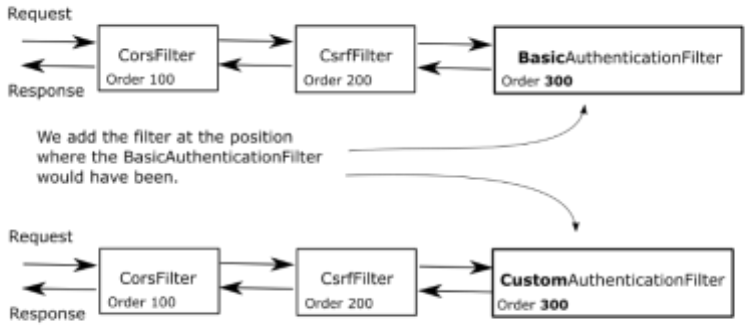

Figure 1.15 We add our custom authentication filter at the location where the class `BasicAuthenticationFilter` would have been if we were using HTTP Basic as an authentication method. This means our custom filter has the same ordering value.

But remember what we discussed earlier. 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 1.34, 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 1.34 Adding the filter in the configuration class

```
@Configuration
public class ProjectConfig {

  private final StaticKeyAuthenticationFilter filter;    #A

  // omitted constructor

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

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

#A Injects the instance of the filter from the Spring context
#B Adds the filter at the position of the basic authentication filter in the filter chain

To test the application, we also need an endpoint. For that, we define a controller, as given in listing 1.34. You should add a value for the static key on the server in the application.properties file, as shown in this code snippet:

```
authorization.key=SD9cICjl1e
```

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

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

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

returns this response body:

```
Hello!
```

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

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

The response status is

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

In this case, because we don't configure a `UserDetailsService`, Spring Boot automatically configures one. 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 })
```

1.19 Restricting access based on authorities and roles

Eralier, 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 this interface is mainly related to the authorization process. We can now return to `GrantedAuthority` to examine its purpose. Figure 1.16 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.

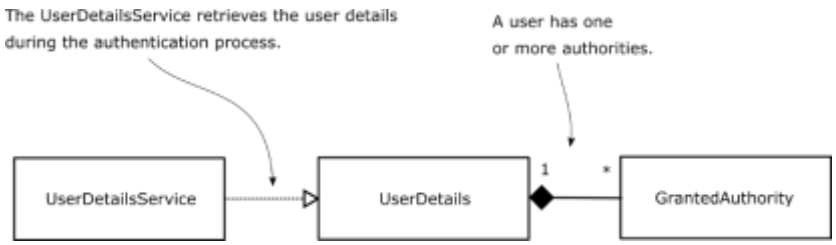

Figure 1.16 A user has one or more authorities (actions that a user can do). During the authentication process, the `UserDetailsService` obtains all the details about the user, including the authorities. The application uses the authorities as represented by the `GrantedAuthority` interface for authorization after it successfully authenticates the user.

Listing 1.35 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 1.35 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 1.16. You can allow a user one or more privileges. The `getAuthorities()` method returns the collection of `GrantedAuthority` instances. In listing 1.36, 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 1.36 The `getAuthorities()` method from the `UserDetails` contract

```
public interface UserDetails extends Serializable {
  Collection<? extends GrantedAuthority> getAuthorities();

  // Omitted code
}
```

1.20 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 1.17). We'll write several examples so that you learn various ways in which you can apply these restrictions with Spring Security.

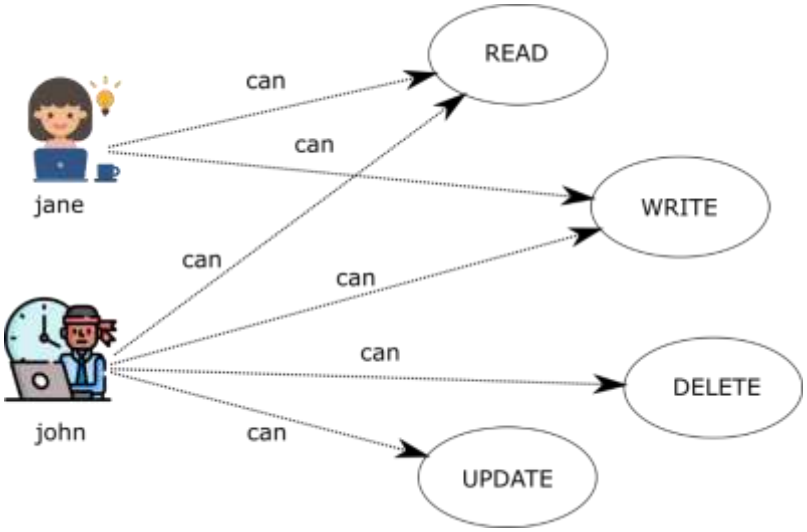

Figure 1.17 Authorities are actions that users can perform in the application. Based on these actions, you implement the authorization rules. Only users having specific authorities can make a particular request to an endpoint. For example, Jane can only read and write to the endpoint, while John can 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. I show you three ways in which you can configure access as mentioned using these methods:

- `hasAuthority()`—Receives as parameters only one authority for which the application configures the restrictions. Only users having that authority can call the endpoint.
- `hasAnyAuthority()`—Can receive more than one authority for which the application configures the restrictions. I remember this method as "has any of the given authorities." The user must have at least one of the specified authorities to make a request.

  I recommend using this method or the `hasAuthority()` method for their simplicity, depending on the number of privileges you assign the users. These are simple to read in configurations and make your code easier to understand.
- `access()`—Offers you unlimited possibilities for configuring access because the application builds the authorization rules based on a custom object named `AuthorizationManager` you implement. You can provide any implementation for the `AuthorizationManager` contract depending on your case. Spring Security provides a few implementations as well. The most common implementation is the

`WebExpressionAuthorizationManager` which helps you apply authorization rules based on Spring Expression Language (SpEL). But using the `access()` method can make the authorization rules more difficult to read and understand. For this reason, I recommend it as the lesser solution and only if you cannot apply the `hasAnyAuthority()` or `hasAuthority()` methods.

The only dependencies needed in your pom.xml file are `spring-boot-starter-web` and `spring-boot-starter-security`. These dependencies are enough to approach all three solutions previously enumerated.

```
<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 1.37 Declaring the `UserDetailsService` and assigning users

```
@Configuration
public class ProjectConfig {

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

    var user1 = User.withUsername("john")       #C
                    .password("12345")
                    .authorities("READ")
                    .build();

    var user2 = User.withUsername("jane")       #D
                    .password("12345")
                    .authorities("WRITE")
                    .build();

    manager.createUser(user1);       #E
    manager.createUser(user2);
```

```
    return manager;
  }

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

#A The UserDetailsService returned by the method is added in to SpringContext.
#B Declares an InMemoryUserDetailsManager that stores a couple of users
#C First user john has the READ authority
#D Second user jane has the WRITE authority
#E The users are added and managed by the UserDetailsService.
#F Don't forget that a PasswordEncoder is also needed.

The next thing we do is add the authorization configuration. To do that, you created a `SecurityFilterChain` bean in the app's context, similar to what you see in the next listing.

Listing 1.38 Making all the endpoints accessible for everyone without authentication

```
@Configuration
public class ProjectConfig {

  // Omitted code

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

    http.httpBasic();

    http.authorizeRequests()
          .anyRequest().permitAll();     #A

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

#A 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 1.39 Restricting access to only users having `WRITE` authority

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

```
    // Omitted code

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

      http.httpBasic();

      http.authorizeRequests()
           .anyRequest()
            .hasAuthority("WRITE");      #A

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

#A 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 1.40 Applying the `hasAnyAuthority()` method

```
@Configuration
public class ProjectConfig {

  // Omitted code

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

    http.httpBasic();

    http.authorizeRequests()
          .anyRequest()
            .hasAnyAuthority("WRITE", "READ");     #A

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

#A 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 an `AuthorizationManager` implementation. You can provide any implementation for this object that can apply any kind of logic that defines the authorization rules. This method is powerful, and it doesn't refer only to authorities. However, this method also makes the code more difficult to read and understand. For this reason, I recommend it as the last option, and only if you can't apply one of the `hasAuthority()` or `hasAnyAuthority()` methods presented earlier in this section.

To make this method easier to understand, I first present it as an alternative to specifying authorities with the `hasAuthority()` and `hasAnyAuthority()` methods. As you learn in this example, you'll use an `AuthorizationManager` implementation where you have to provide a SpEL expression as a parameter. The authorization rule we defined becomes more challenging to read, and this is why I don't recommend this approach for simple rules.

However, the `access()` method has the advantage of allowing you to customize rules through the `AuthorizationManager` implementation you provide as a parameter. And this is really powerful! As with SpEL expressions, you can basically define any condition.

> **NOTE** In most situations, you can implement the required restrictions with the `hasAuthority()` and `hasAnyAuthority()` methods, and I recommend that you use these. Use the `access()` method only if the other two options do not fit and you want to implement more generic authorization rules.

I start with a simple example to match the same requirement as in the previous cases. If you only need to test if the user has specific authorities, the expression you need to use with the `access()` method can be one of the following:

- `hasAuthority('WRITE')`—Stipulates that the user needs the `WRITE` authority to call the endpoint.
- `hasAnyAuthority('READ', 'WRITE')`—Specifies that the user needs one of either the `READ` or `WRITE` authorities. With this expression, you can enumerate all the authorities for which you want to allow access.

Observe that these expressions have the same name as the methods presented earlier in this section. The following listing demonstrates how you can use the `access()` method.

Listing 1.41 Using the `access()` method to configure access to the endpoints

```
@Configuration
public class ProjectConfig {

  // Omitted code

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

    http.httpBasic();

    http.authorizeRequests()
          .anyRequest()
            .access(
               new WebExpressionAuthorizationManager
                        "hasAuthority('WRITE')"));      #A

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

#A Authorizes requests from users with the WRITE authority

The example presented in listing 1.41 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 1.42, you find the `access()` method applied with an expression that's not easy to write otherwise. Precisely, the configuration presented in listing 1.42 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.

It is a hypothetical example, of course, but it's simple enough to be easy to understand and complex enough to prove why the `access()` method is more powerful. To implement this with the `access()` method, you can use an `AuthorizationManager` implementation that takes a SpEL expression. The SpEL expression must reflect the requirement. For example:

```
"hasAuthority('read') and !hasAuthority('delete')"
```

The next listing illustrates how to apply the `access()` method with a more complex expression.

Listing 1.42 Applying the `access()` method with a more complex expression

```
@Configuration
public class ProjectConfig {

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

    var user1 = User.withUsername("john")
            .password("12345")
            .authorities("read")
            .build();

    var user2 = User.withUsername("jane")
            .password("12345")
            .authorities("read", "write", "delete")
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

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

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
```

```
        http.httpBasic();

        String expression =
               "hasAuthority('read') and        #A
                 !hasAuthority('delete')";       #A

        http.authorizeRequests()
             .anyRequest()
             .access(new WebExpressionAuthorizationManager(expression));

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

#A States that the user must have the authority read but not the authority delete

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.

1.21 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 1.18). 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.

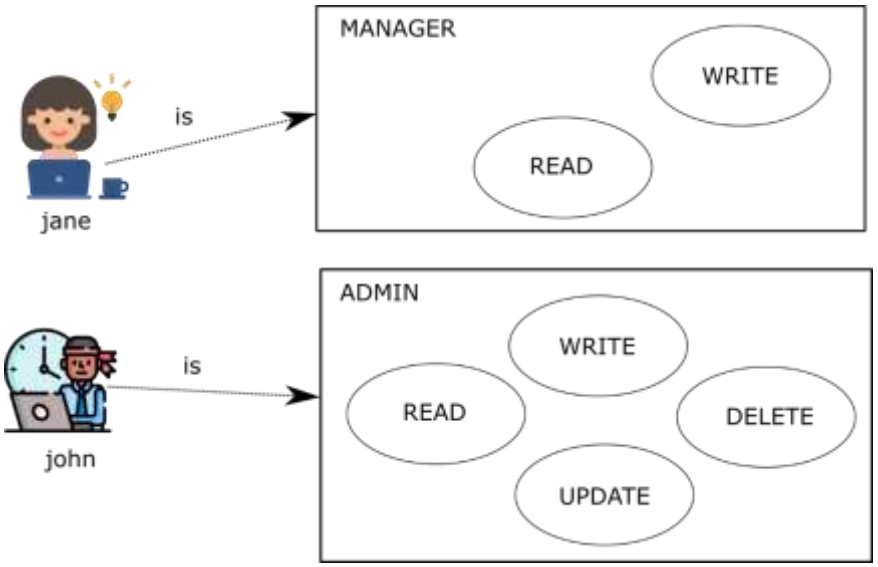

Figure 1.18 Roles are coarse grained. Each user with a specific role can only do the actions granted by that role. When applying this philosophy in authorization, a request is allowed based on the purpose of the user in the system. Only users who have a specific role can call a certain endpoint.

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. In the next listing, take a look at the change I made to the previous example.

Listing 1.43 Setting roles for users

```
@Configuration
public class ProjectConfig {

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

    var user1 = User.withUsername("john")
                    .password("12345")
                    .authorities("ROLE_ADMIN")      #A
                    .build();

    var user2 = User.withUsername("jane")
                    .password("12345")
                    .authorities("ROLE_MANAGER")
                    .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  // Omitted code

}
```

#A Having the ROLE_ prefix, GrantedAuthority now represents a role.

To set constraints for user roles, you can use one of the following methods:

- `hasRole()`—Receives as a parameter the role name for which the application authorizes the request.
- `hasAnyRole()`—Receives as parameters the role names for which the application approves the request.
- `access()`—Uses an `AuthorizationManager` to specify the role or roles for which the application authorizes requests. In terms of roles, you could use `hasRole()` or `hasAnyRole()` as SpEL expressions together with the `WebExpressionAuthorizationManager` implementation.

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 1.44 Configuring the app to accept only requests from admins

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
    http.httpBasic();

    http.authorizeRequests()
         .anyRequest().hasRole("ADMIN");     #A


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

#A 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 1.45 Setting up roles with the `roles()` method

```
@Configuration
public class ProjectConfig {

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

    var user1 = User.withUsername("john")
                    .password("12345")
                    .roles("ADMIN")      #A
                    .build();

    var user2 = User.withUsername("jane")
                    .password("12345")
                    .roles("MANAGER")
                    .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

  // Omitted code
}
```

#A The roles() method specifies the user's roles.

1.22 Using the requestMatchers() method to select endpoints

In this section, you learn how to use the `requestMatchers()` method in general so that in next sections, we can continue describing various approaches you have to select HTTP requests for which you need to apply authorization restrictions. By the end of this section, you'll be able to apply the `requestMatchers()` method in any authorization configurations you might need to write for your application's requirements. Let's start with a straightforward example.

We create an application that exposes two endpoints: /hello and /ciao. We want to make sure that only users having the ADMIN role can call the /hello endpoint. Similarly, we want to make sure that only users having the MANAGER role can call the /ciao endpoint. The following listing provides the definition of the controller class.

Listing 1.46 The definition of the controller class

```
@RestController
public class HelloController {

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

  @GetMapping("/ciao")
  public String ciao() {
    return "Ciao!";
  }
}
```

In the configuration class, we declare an `InMemoryUserDetailsManager` as our `UserDetailsService` instance and add two users with different roles. The user John has the ADMIN role, while Jane has the MANAGER role. To specify that only users having the ADMIN role can call the endpoint /hello when authorizing requests, we use the `requestMatchers()` method. The next listing presents the definition of the configuration class.

Listing 1.47 The definition of the configuration class

```
@Configuration
public class ProjectConfig {

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

    var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")
            .build();

    var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER")
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

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

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
    http.httpBasic();

    http.authorizeHttpRequests()
```

```
            .requestMatchers("/hello").hasRole("ADMIN")       #A
            .requestMatchers("/ciao").hasRole("MANAGER");     #B

        return http.build();
      }

    }
```

#A Only calls the path /hello if the user has the ADMIN role
#B Only calls the path /ciao if the user has the MANAGER role

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 1.48 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 1.49.

> **NOTE** It is good practice to make all your rules explicit. Listing 1.49 clearly and unambiguously indicates the intention to permit requests to endpoints for everyone, except for the endpoints /hello and /ciao.

Listing 1.49 Marking additional requests explicitly as accessible without authentication

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
    http.httpBasic();

    http.authorizeHttpRequests()
           .mvcMatchers("/hello").hasRole("ADMIN")
           .mvcMatchers("/ciao").hasRole("MANAGER")
           .anyRequest().permitAll();      #A

    return http.build();
```

```
    }
}
```

#A The permitAll() method states that all other requests are allowed without authentication.

NOTE When you use matchers to refer to requests, the order of the rules should be from particular to general. This is why the `anyRequest()` method cannot be called before a more specific `requestMatchers()` method.

Unauthenticated vs. failed authentication

If you have designed an endpoint to be accessible to anyone, you can call it without providing a username and a password for authentication. In this case, Spring Security won't do the authentication. If you, however, provide a username and a password, Spring Security evaluates them in the authentication process. If they are wrong (not known by the system), authentication fails, and the response status will be 401 Unauthorized. To be more precise, if you call the /hola endpoint for the configuration presented in listing 8.4, the app returns the body "Hola!" as expected, and the response status is 200 OK. For example,

```
curl http://localhost:8080/hola
```

The response body is

```
Hola!
```

But if you call the endpoint with invalid credentials, the status of the response is 401 Unauthorized. In the next call, I use an invalid password:

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

The response body is

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

This behavior of the framework might look strange, but it makes sense as the framework evaluates any username and password if you provide them in the request.

The authorization filter allows any request to the /hola path. But because the application first executes the authentication logic, the request is never forwarded to the authorization filter. Instead, the authentication filter replies with an HTTP 401 Unauthorized.

In conclusion, any situation in which authentication fails will generate a response with the status 401 Unauthorized, and the application won't forward the call to the endpoint. The `permitAll()` method refers to authorization configuration only, and if authentication fails, the call will not be allowed further.

You could decide, of course, to make all the other endpoints accessible only for authenticated users. To do this, you would change the `permitAll()` method with `authenticated()` as presented in the following listing. Similarly, you could even deny all other requests by using the `denyAll()` method.

Listing 1.50 Making other requests accessible for all authenticated users

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
    http.httpBasic();
```

```
        http.authorizeHttpRequests()
                .requestMatchers("/hello").hasRole("ADMIN")
                .requestMatchers("/ciao").hasRole("MANAGER")
                .anyRequest().authenticated();      #A

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

#A 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 section, we take each type of matcher method and discuss these aspects in detail.

1.23 Selecting requests to apply authorization restrictions

In this section, we deep dive into configuring request matchers. Using the `requestMatchers()` method is a common approach to refer to requests for applying authorization configuration. So I expect you to have many opportunities to use this method to refer to requests in the applications you develop.

This matcher uses the standard ANT syntax for referring to paths. This syntax is the same one you use when writing endpoint mappings with annotations like `@RequestMapping`, `@GetMapping`, `@PostMapping`, and so forth. The two methods you can use to declare MVC matchers are as follows:

- `requestMatchers(HttpMethod method, String... patterns)`—Lets you specify both the HTTP method to which the restrictions apply and the paths. This method is useful if you want to apply different restrictions for different HTTP methods for the same path.
- `requestMatchers(String... patterns)`—Simpler and easier to use if you only need to apply authorization restrictions based on paths. The restrictions can automatically apply to any HTTP method used with the path.

In this section, we approach multiple ways of using `requestMatchers()` methods. To demonstrate this, we start by writing an application that exposes multiple endpoints.

For the first time, we write endpoints that can be called with other HTTP methods besides GET. You might have observed that until now, I've avoided using other HTTP methods. The reason for this is that Spring Security applies, by default, protection against cross-site request forgery (CSRF). In next sections, 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 next sections, 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 1.51, you can see the definitions of these endpoints.

Listing 1.51 Definition of the four endpoints for which we configure authorization

```
@RestController
public class TestController {

  @PostMapping("/a")
  public String postEndpointA() {
    return "Works!";
  }

  @GetMapping("/a")
  public String getEndpointA() {
    return "Works!";
  }

  @GetMapping("/a/b")
  public String getEnpointB() {
    return "Works!";
  }

  @GetMapping("/a/b/c")
  public String getEnpointC() {
    return "Works!";
  }
}
```

We also need a couple of users with different roles. To keep things simple, we continue using an `InMemoryUserDetailsManager`. In the next listing, you can see the definition of the `UserDetailsService` in the configuration class.

Listing 1.52 The definition of the `UserDetailsService`

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

```
  @Bean
  public UserDetailsService userDetailsService() {
    var manager = new InMemoryUserDetailsManager();   #A

    var user1 = User.withUsername("john")
            .password("12345")
            .roles("ADMIN")       #B
            .build();

    var user2 = User.withUsername("jane")
            .password("12345")
            .roles("MANAGER")     #C
            .build();

    manager.createUser(user1);
    manager.createUser(user2);

    return manager;
  }

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

#A Defines an InMemoryUserDetailsManager to store users
#B User john has the ADMIN role
#C User jane has the MANAGER role
#D 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 1.53 Authorization configuration for the first scenario, /a

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
    http.httpBasic();

    http.authorizeHttpRequests()
            .requestMatchers(HttpMethod.GET, "/a")
               .authenticated()     #A
            .requestMatchers(HttpMethod.POST, "/a")
               .permitAll()     #B
            .anyRequest()
               .denyAll();    #C
```

```
      http.csrf().disable();      #D

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

#A For path /a requests called with an HTTP GET method, the app needs to authenticate the user.
#B Permits path /a requests called with an HTTP POST method for anyone
#C Denies any other request to any other path
#D Disables CSRF to enable a call to the /a path using the HTTP POST method

In the next code snippets, we analyze the results on the calls to the endpoints for the configuration presented in listing 8.8. For the call to 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.

Listing 1.54 Changes in the configuration class for multiple paths

```
@Configuration
public class ProjectConfig {

  // Omitted code

  @Bean
  public void configure(HttpSecurity http) throws Exception {
    http.httpBasic();

    http.authorizeHttpRequests()
          .requestMatchers( "/a/b/**")    #A
             .authenticated()
          .anyRequest()
             .permitAll();

    http.csrf().disable();

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

#A 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 pathnames. You can use it as we have done in the last example so that you can match requests with paths having a known prefix. You can also use it in the middle of a path to refer to any number of pathnames or to refer to paths ending in a specific pattern like `/a/**/c`. Therefore, `/a/**/c` would not only match /a/b/c but also /a/b/d/c and a/b/c/d/e/c and so on. If you only want to match one pathname, then you can use a single `*`. For example, `a/*/c` would match a/b/c and a/d/c but not a/b/d/c.

Because you generally use path variables, you can find it useful to apply authorization rules for such requests. You can even apply rules referring to the path variable value. Do you remember the discussion from the previous sections 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. The following listing presents the controller.

Listing 1.55 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 1.56 Configuring the authorization to permit only specific digits

```
@Configuration
public class ProjectConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
```

```
        throws Exception {
        http.httpBasic();

        http.authorizeHttpRequests()
             .requestMatchers
              ("/product/{code:^[0-9]*$}")       #A
                  .permitAll()
             .anyRequest()
                  .denyAll();

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

#A The regex refers to strings of any length, containing any digit.

NOTE When using parameter expressions with a regex, make sure to not have a space between the name of the parameter, the colon (:), and the regex, as displayed in the listing.

Running this example, you can see the result as presented in the following code snippets. The application only accepts the call when the path variable value has only digits. To call the endpoint using the value `1234a`, use

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

The response is

```
{
  "status":401,
  "error":"Unauthorized",
  "message":"Unauthorized",
  "path":"/product/1234a"
}
```

To call the endpoint using the value `12345`, use

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

The response is

```
12345
```

We discussed a lot and included plenty of examples of how to refer to requests using `requestMatchers()` method. Table 1.1 is a refresher for the path expressions you used in this section. You can simply refer to it later when you want to remember any of them.

Table 1.1 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.

1.24 How CSRF protection works in Spring Security

In this section, we discuss how Spring Security implements CSRF protection. I consider it essential first to understand the underlying mechanism of CSRF protection. I encounter many situations in which misunderstanding the way CSRF protection works leads developers to misuse it, either disabling it in scenarios where it should be enabled or the other way around. Like any other feature in a framework, you have to use it correctly to bring value to your applications.

As an example, consider this scenario (figure 1.19): you are at work, where you use a web tool to store and manage your files. With this tool, in a web interface you can add new files, add new versions for your records, and even delete them. You receive an email asking you to open a page for a specific reason (for example, a promotion at your favorite store). You open the page, but the page is blank or it redirects you to a known website (the online shop of your favorite store). You go back to your work but observe that all your files are gone!

What happened? You were logged into your work application so you could manage your files. When you add, change, or delete a file, the web page you interact with calls some endpoints from the server to execute these operations. When you opened the foreign page by clicking the unknown link in the email, that page called your app's backend and executed actions on your behalf (it deleted your files).

It could do that because you logged in previously, so the server trusted that the actions came from you. You might think that someone couldn't trick you so easily into clicking a link from a foreign email or message, but trust me, this happens to a lot of people. Most web app users aren't aware of security risks. So it's wiser if you who know all the tricks, protect your users, and build secure apps rather than rely on your apps' users to protect themselves.

CSRF attacks assume that a user is logged into a web application. They're tricked by the attacker into opening a page that contains scripts that execute actions in the same application the user was working on. Because the user has already logged in (as we've assumed from the beginning), the forgery code can now impersonate the user and do actions on their behalf.

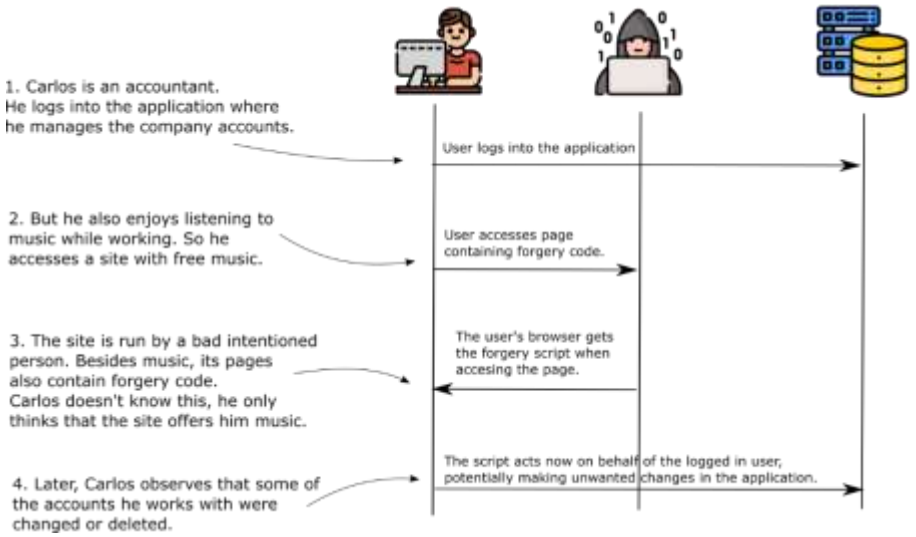

Figure 1.19 After the user logs into their account, they access a page containing forgery code. This code impersonates the user and can execute unwanted actions on behalf of the user.

How do we protect our users from such scenarios? What CSRF protection wants to ensure is that only the frontend of web applications can perform mutating operations (by convention, HTTP methods other than GET, HEAD, TRACE, or OPTIONS). Then, a foreign page, like the one in our example, can't act on behalf of the user.

How can we achieve this? What you know for sure is that before being able to do any action that could change data, a user must send a request using HTTP GET to see the web page at least once. When this happens, the application generates a unique token. The application now accepts only requests for mutating operations (POST, PUT, DELETE, and so forth) that contain this unique value in the header.

The application considers that knowing the value of the token is proof that it is the app itself making the mutating request and not another system. Any page containing mutating calls, like POST, PUT, DELETE, and so on, should receive through the response the CSRF token, and the page must use this token when making mutating calls.

The starting point of CSRF protection is a filter in the filter chain called `CsrfFilter`. The `CsrfFilter` intercepts requests and allows all those that use these HTTP methods: GET, HEAD, TRACE, and OPTIONS. For all other requests, the filter expects to receive a header containing a token. If this header does not exist or contains an incorrect token value, the application rejects the request and sets the status of the response to HTTP 403 Forbidden.

What is this token, and where does it come from? These tokens are nothing more than string values. You have to add the token in the header of the request when you use any method other than GET, HEAD, TRACE, or OPTIONS. If you don't add the header containing the token, the application doesn't accept the request, as presented in figure 1.20.

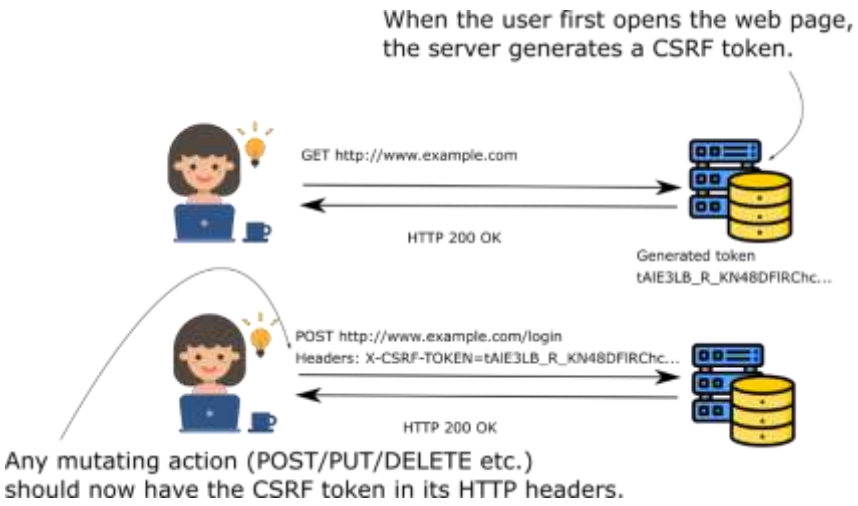

Figure 1.20 To make a POST request, the client needs to add a header containing the CSRF token. The application generates a CSRF token when the page is loaded (via a GET request), and the token is added to all requests that can be made from the loaded page. This way, only the loaded page can make mutating requests.

The `CsrfFilter` (figure 1.21) uses a component named `CsrfTokenRepository` to manage the CSRF token values that generate new tokens, store tokens, and eventually invalidate these. By default, the `CsrfTokenRepository` stores the token on the HTTP session and generates the tokens as random string values. In most cases, this is enough, but 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. Let's create an application that exposes two endpoints. We can call one of these with HTTP GET and the other with HTTP POST.

As you know by now, you are not able to call endpoints with POST directly without disabling CSRF protection. In this example, you learn how to call the POST endpoint without disabling CSRF protection. You need to obtain the CSRF token so that you can use it in the header of the call, which you do with HTTP POST.

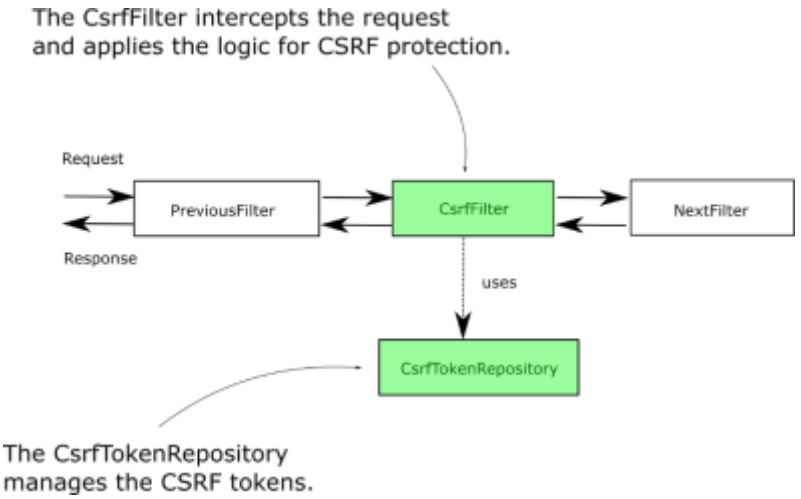

Figure 1.21 The `CsrfFilter` is one of the filters in the filter chain. It receives the request and eventually forwards it to the next filter in the chain. To manage CSRF tokens, `CsrfFilter` uses a `CsrfTokenRepository`.

As you learn with this example, the `CsrfFilter` adds the generated CSRF token to the attribute of the HTTP request named `_csrf` (figure 1.22). 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`. You use this custom filter to print in the console of the application the CSRF token that the app generates when we call the endpoint using HTTP GET. We can then copy the value of the token from the console and use it to make the mutating call with HTTP POST. In the following listing, you can find the definition of the controller class with the two endpoints that we use for a test.

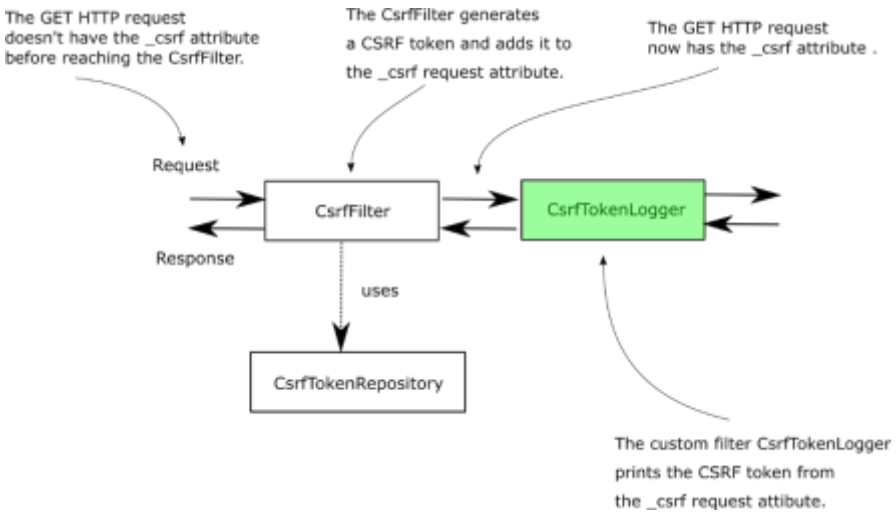

Figure 1.22 Adding the `CsrfTokenLogger` (shaded) after the `CsrfFilter`. This way, the `CsrfTokenLogger` can obtain the value of the token from the `_csrf` attribute of the request where the `CsrfFilter` stores it. The `CsrfTokenLogger` prints the CSRF token in the application console, where we can access it and use it to call an endpoint with the HTTP POST method.

Listing 1.57 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 1.58 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 1.58 The definition of the custom filter class

```
public class CsrfTokenLogger implements Filter {

  private Logger logger =
```

```
            Logger.getLogger(CsrfTokenLogger.class.getName());

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

      CsrfToken o =
       (CsrfToken) request.getAttribute("_csrf");    #A


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

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

#A 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 1.59 Adding the custom filter in the configuration class

```
@Configuration
public class ProjectConfig {

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

    http.addFilterAfter(
            new CsrfTokenLogger(), CsrfFilter.class)
        .authorizeRequests()
            .anyRequest().permitAll();

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

We can now test the endpoints. We begin by calling the endpoint with HTTP GET. Because the default implementation of the `CsrfTokenRepository` interface uses the HTTP session to store the token value on the server side, we also need to remember the session ID. For this reason, I add the `-v` flag to the call so that I can see more details from the response, including the session ID. Calling the endpoint

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

returns this (truncated) response:

```
...
< Set-Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206;
...
Get Hello!
```

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

```
INFO 21412 --- [nio-8080-exec-1] c.l.ssia.filters.CsrfTokenLogger : CSRF
token tAlE3LB_R_KN48DFlRChc…
```

If you call the endpoint using the HTTP POST method without providing the CSRF token, the response status is 403 Forbidden, as this command line shows:

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

The response body is

```
{
    "status":403,
    "error":"Forbidden",
    "message":"Forbidden",
    "path":"/hello"
}
```

But if you provide the correct value for the CSRF token, the call is successful. You also need to specify the session ID (`JSESSIONID`) because the default implementation of `CsrfTokenRepository` stores the value of the CSRF token on the session:

```
curl -X POST   http://localhost:8080/hello
-H 'Cookie: JSESSIONID=21ADA55E10D70BA81C338FFBB06B0206'
-H 'X-CSRF-TOKEN: tAlE3LB_R_KN48DFlRChc…'
```

The response body is

```
Post Hello!
```

1.25 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, and that web app actually used CSRF protection. Did you notice that the login operation in that application used HTTP POST? Then why didn't we need to do anything explicitly about CSRF in that case? The reason why we didn't observe this was because we didn't develop any mutating operation within it ourselves.

For the default form login, Spring Security correctly applies CSRF protection for us. The framework takes care of adding the CSRF token to the login request. Let's now develop a similar application to look closer at how CSRF protection works. As figure 1.23 shows, in this section we

- Build an example of a web application with the login form
- Look at how the default implementation of the login uses CSRF tokens

- Implement an HTTP POST call from the main page

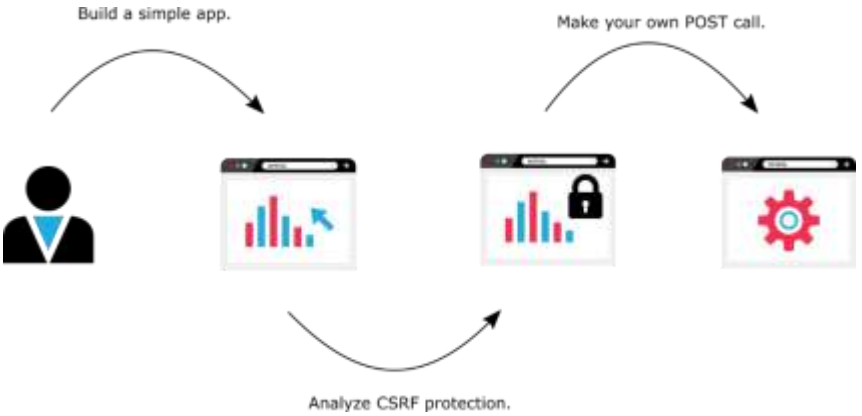

Figure 1.23 The plan. In this section, we start by building and analyzing a simple app to understand how Spring Security applies CSRF protection, and then we write our own POST call.

In this example application, you'll notice that the HTTP POST call won't work until we correctly use the CSRF tokens, and you'll learn how to apply the CSRF tokens in a form on such a web page. To implement this application, we start by creating a new Spring Boot project. 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 1.60 The definition of the configuration class

```
public class ProjectConfig {

  @Bean     #A
  public UserDetailsService uds() {
    var uds = new InMemoryUserDetailsManager();

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

```
                .password("12345")
                .authorities("READ")
                .build();

    uds.createUser(u1);

    return uds;
  }

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

  @Bean      #C
  public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {

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

    http.formLogin()
        .defaultSuccessUrl("/main", true);

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

#A Adds a UserDetailsService bean managing one user to test the application
#B Adds a PasswordEncoder
#C Creates a bean of type SecurityFilterChain 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 the first execution of the application, we only focus on how the login page uses the CSRF tokens. The following listing presents the `MainController` class, which serves the main page.

Listing 1.61 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 1.24 shows how the login form sends the CSRF token through hidden input.

But what about developing our own endpoints that use POST, PUT, or DELETE as HTTP methods? For these, we have to take care of sending the value of the CSRF token if CSRF protection is enabled. To test this, let's add an endpoint using HTTP POST to our application. We call this endpoint from the main page, and we create a second controller for this, called `ProductController`. Within this controller, we define an endpoint, /product/add, that uses HTTP POST. Further, we use a form on the main page to call this endpoint. The next listing defines the `ProductController` class.

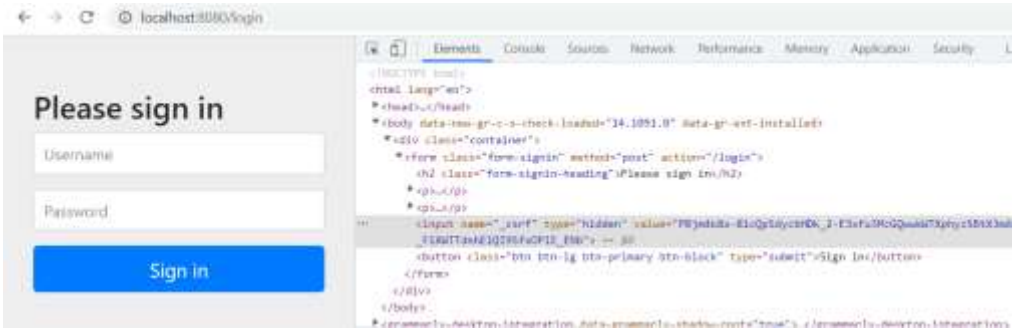

Figure 1.24 The default form login uses a hidden input to send the CSRF token in the request. This is why the login request that uses an HTTP POST method works with CSRF protection enabled.

Listing 1.62 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 1.63 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 1.25). The reason for the HTTP 403 Forbidden status is the absence of the CSRF token.

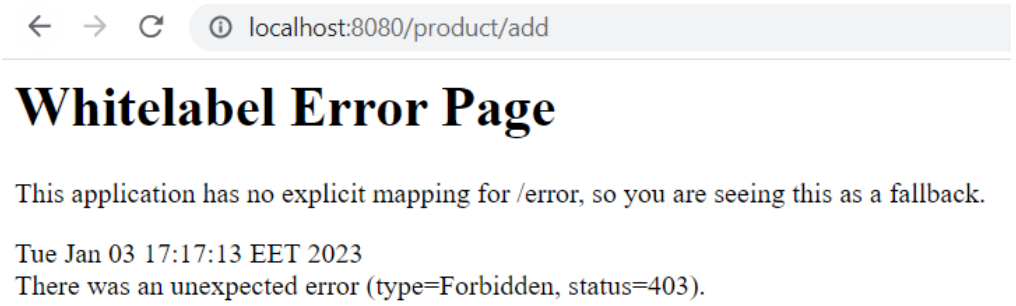

Figure 1.25 Without sending the CSRF token, the server won't accept the request done with the HTTP POST method. The application redirects the user to a default error page, which confirms that the status on the response is HTTP 403 Forbidden.

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 1.64 Adding the CSRF token to the request done through the form

```
<form action="/product/add" method="post">
   <span>Name:</span>
   <span><input type="text" name="name" /></span>
   <span><button type="submit">Add</button></span>

   <input type="hidden"       #A
          th:name="${_csrf.parameterName}"   #B
          th:value="${_csrf.token}" />       #B
</form>
```

#A Uses hidden input to add the request to the CSRF token
#B The "th" prefix enables Thymeleaf to print the token value.

After rerunning the application, you can test the form again. This time the server accepts the request, and the application prints the log line in the console, proving that the execution succeeds. Also, if you inspect the form, you can find the hidden input with the value of the CSRF token (figure 1.26).

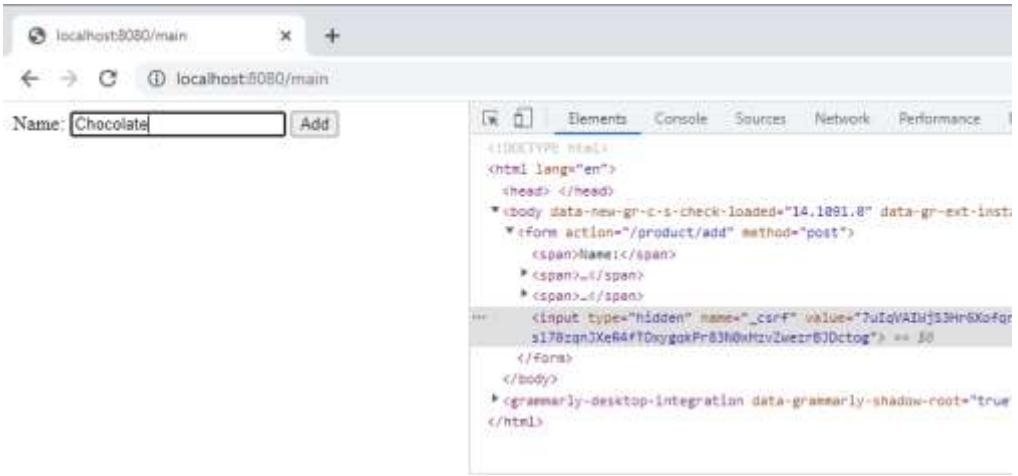

Figure 1.26 The form defined on the main page now sends the value for the CSRF token in the request. This way, the server allows the request and executes the controller action. In the source code for the page, you can now find the hidden input used by the form to send the CSRF token in the request.

After submitting the form, you should find in the application console a line similar to this one:

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

Of course, for any action or asynchronous JavaScript request your page uses to call a mutating action, you need to send a valid CSRF token. This is the most common way used by an application to make sure the request doesn't come from a third party. A third-party request could try to impersonate the user to execute actions on their behalf.

CSRF tokens work well in an architecture where the same server is responsible for both the frontend and the backend, mainly for its simplicity. But CSRF tokens don't work well when the client is independent of the backend solution it consumes. This scenario happens when you have a mobile application as a client or a web frontend developed independently. A web client developed with a framework like Angular, ReactJS, or Vue.js is ubiquitous in web application architectures, and this is why you need to know how to implement the security approach for these cases as well.

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.

Let's discuss cross-origin resource sharing (CORS) and how to apply it with Spring Security. First, what is CORS and why should you care? The necessity for CORS came from web applications. By default, browsers don't allow requests made for any domain other than the one from which the site is loaded. For example, if you access the site from example.com, the browser won't let the site make requests to api.example.com. Figure 1.27 shows this concept.

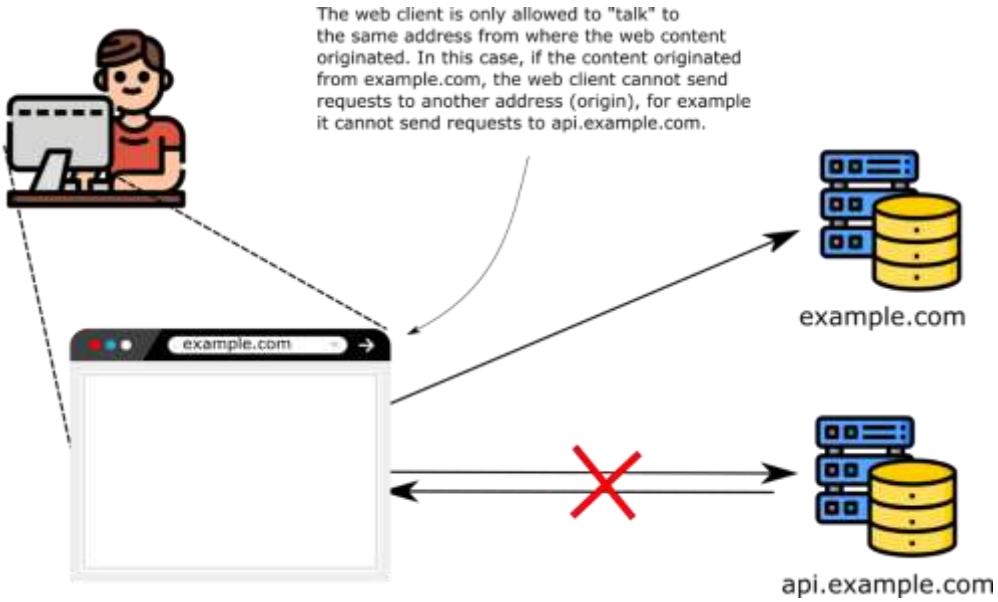

Figure 1.27 Cross-origin resource sharing (CORS). When accessed from example.com, the website cannot make requests to api.example.com because they would be cross-domain requests.

We can briefly say that an app uses the CORS mechanism to relax this strict policy and allow requests made between different origins in some conditions. You need to know this because it's likely you will have to apply it to your applications, especially nowadays where the frontend and backend are separate applications. It is common that a frontend application is developed using a framework like Angular, ReactJS, or Vue and hosted at a domain like example.com, but it calls endpoints on the backend hosted at another domain like api.example.com.

For this part of the training, 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.

1.26 How does CORS work?

In this section, we discuss how CORS applies to web applications. If you are the owner of example.com, for example, and for some reason the developers from example.org decide to

call your REST endpoints (api.example.com) from their website, they won't be able to. The same situation can happen if a domain loads your application using an `iframe`, for example (see figure 1.28).

> **NOTE** An `iframe` is an HTML element that you use to embed content generated by a web page into another web page (for example, to integrate the content from example.org inside a page from example.com).

Any situation in which an application makes calls between two different domains is prohibited. But, of course, you can find cases in which you need to make such calls. In these situations, CORS allows you to specify from which domain your application allows requests and what details can be shared. The CORS mechanism works based on HTTP headers (figure 1.28). The most important are

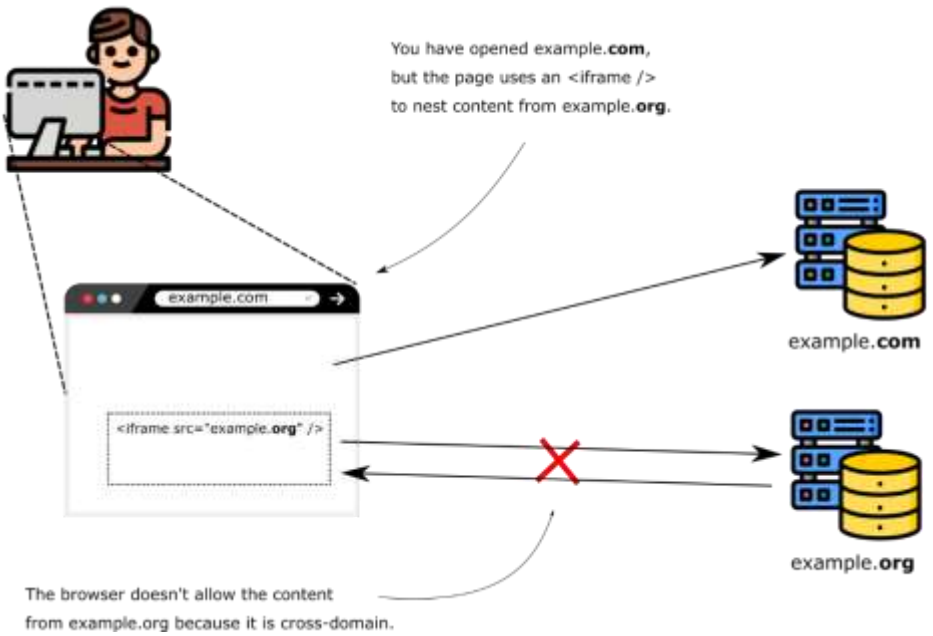

Figure 1.28 Even if the example.org page is loaded in an `iframe` from the example.com domain, the calls from the content loaded in example.org won't load. Even if the application makes a request, the browser won't accept the response.

- `Access-Control-Allow-Origin`—Specifies the foreign domains (origins) that can access resources on your domain.
- `Access-Control-Allow-Methods`—Lets us refer only to some HTTP methods in situations in which we want to allow access to a different domain, but only to specific

HTTP methods. You use this if you're going to enable example.com to call some endpoint, but only with HTTP GET, for example.

- `Access-Control-Allow-Headers`—Adds limitations to which headers you can use in a specific request. For example, you don't want the client to be able to send a specific header for a given request.

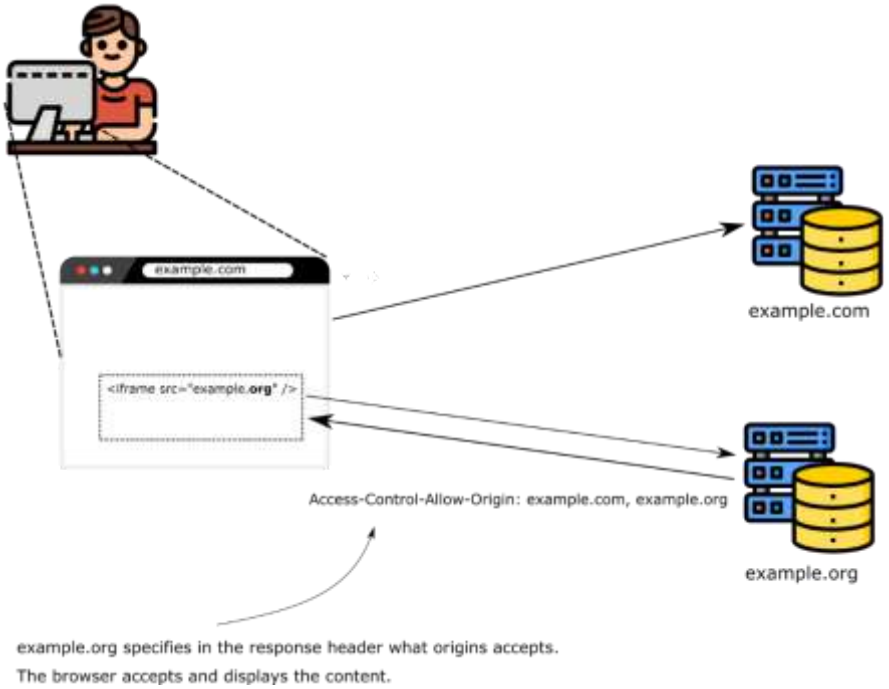

Figure 1.29 Enabling cross-origin requests. The example.org server adds the `Access-Control-Allow-Origin` header to specify the origins of the request for which the browser should accept the response. If the domain from where the call was made is enumerated in the origins, the browser accepts the response.

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.

```
<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 1.65 The definition of the controller class

```
@Controller
public class MainController {

  private Logger logger =      #A
    Logger.getLogger(MainController.class.getName());

  @GetMapping("/")   #B
  public String main() {
    return "main.html";
  }

  @PostMapping("/test")
  @ResponseBody
  public String test() {    #C
    logger.info("Test method called");
    return "HELLO";
  }
}
```

#A Uses a logger to observe when the test() method is called
#B Defines a main.html page that makes the request to the /test endpoint
#C 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 1.66 The definition of the configuration class

```
@Configuration
public class ProjectConfig {

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

    http.csrf().disable();
```

```
      http.authorizeHttpRequests()
        .anyRequest().permitAll();

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

Of course, we also need to define the main.html file in the resources/templates folder of the project. The main.html file contains the JavaScript code that calls the /test endpoint. To simulate the cross-origin call, we can access the page in a browser using the domain localhost. From the JavaScript code, we make the call using the IP address 127.0.0.1. Even if localhost and 127.0.0.1 refer to the same host, the browser sees these as different strings and considers these different domains. The next listing defines the main.html page.

Listing 1.67 The main.html page

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <script>
      const http = new XMLHttpRequest();
      const url='http://127.0.0.1:8080/test';    #A
      http.open("POST", url);
      http.send();

      http.onreadystatechange = (e) => {
        document      #B
          .getElementById("output")
          .innerHTML = http.responseText;
      }
    </script>
  </head>
  <body>
    <div id="output"></div>
  </body>
</html>
```

#A Calls the endpoint using 127.0.0.1 as host to simulate the cross-origin call
#B Sets the response body to the output div in the page body

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 1.30 presents this request flow.

When the browser omits to make the preflight request if the HTTP method is GET, POST, or OPTIONS, it only has some basic headers as described in the official documentation at https://fetch.spec.whatwg.org/#http-cors-protocol

In our example, the browser makes the request, but we don't accept the response if the origin is not specified in the response. 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.

1.27 Applying CORS policies with the @CrossOrigin annotation

In this section, we discuss how to configure CORS to allow requests from different domains using the `@CrossOrigin` annotation. You can place the `@CrossOrigin` annotation directly above the method that defines the endpoint and configure it using the allowed origins and methods. As you learn in this section, the advantage of using the `@CrossOrigin` annotation is that it makes it easy to configure CORS for each endpoint.

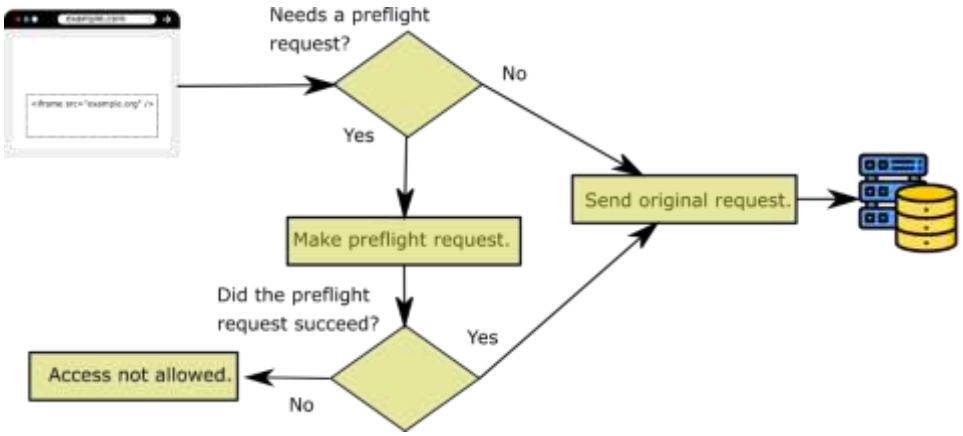

Figure 1.30 For simple requests, the browser sends the original request directly to the server. The browser rejects the response if the server doesn't allow the origin. In some cases, the browser sends a preflight request to test if the server accepts the origin. If the preflight request succeeds, the browser sends the original request.

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 1.68 Making localhost an allowed origin

```
@PostMapping("/test")
@ResponseBody
@CrossOrigin("http://localhost:8080")      #A
public String test() {
  logger.info("Test method called");
  return "HELLO";
}
```

#A Allows the localhost origin for cross-origin requests

You can rerun and test the application. This should now display on the page the string returned by the /test endpoint: HELLO.

The value parameter of `@CrossOrigin` receives an array to let you define multiple origins; for example, `@CrossOrigin({"example.com", "example.org"})`. You can also set the allowed headers and methods using the `allowedHeaders` attribute and the `methods` attribute of the annotation. For both origins and headers, you can use the asterisk (*) to represent all headers or all origins. But I recommend you exercise caution with this approach. It's always better to filter the origins and headers that you want to allow and never allow any domain to implement code that accesses your applications' resources.

By allowing all origins, you expose the application to cross-site scripting (XSS) requests, which eventually can lead to DDoS attacks. I personally avoid allowing all origins even in test

environments. I know that applications sometimes happen to run on wrongly defined infrastructures that use the same data centers for both test and production. It is wiser to treat all layers on which security applies independently 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.

1.28 Applying CORS using a CorsConfigurer

Although using the `@CrossOrigin` annotation is easy 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 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 1.69 Defining CORS configurations centralized in the configuration class

```
@Configuration
public class ProjectConfig {

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

     http.cors(c -> {     #A
      CorsConfigurationSource source = request -> {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowedOrigins(
            List.of("example.com", "example.org"));
        config.setAllowedMethods(
            List.of("GET", "POST", "PUT", "DELETE"));
        config.setAllowedHeaders(List.of("*"));
        return config;
      };
      c.configurationSource(source);
    });

    http.csrf().disable();

    http.authorizeHttpRequests()
         .anyRequest().permitAll();

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

#A 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 using the `SecurityFilterChain` bean directly. I strongly recommend to separate this code in a different class in your applications. In real-world applications, you could have much longer code, so it becomes difficult to read if not separated by the configuration class.

1.29 Enabling method security

In this section, you learn how to enable authorization at the method level and the different options that Spring Security offers to apply various authorization rules. This approach provides you with greater flexibility in applying authorization. It's an essential skill that allows you to solve situations in which authorization simply cannot be configured just at the endpoint level.

By default, method security is disabled, so if you want to use this functionality, you first need to enable it. Also, method security offers multiple approaches for applying authorization. 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)..

One of the approaches for configuring authorization rules you use with method security is *call authorization*. The call authorization approach refers to applying authorization rules that decide if a method can be called, or that allow the method to be called and then decide if the caller can access the value returned by the method. Often we need to decide if someone can access a piece of logic depending on either the provided parameters or its result. So let's discuss call authorization and then apply it to some examples.

How does method security work? What's the mechanism behind applying the authorization rules? When we enable method security in our application, we actually enable a Spring aspect. This aspect intercepts the calls to the method for which we apply authorization rules and, based on these authorization rules, decides whether to forward the call to the intercepted method (figure 1.31).

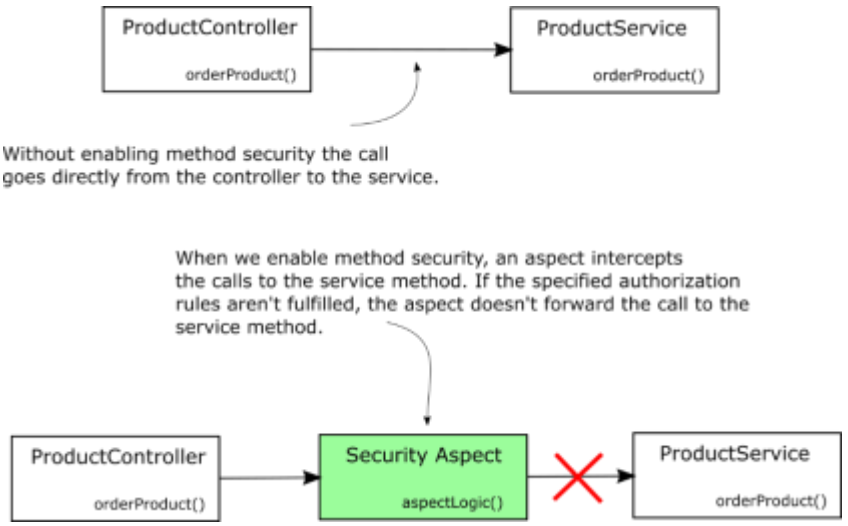

Figure 1.31 When we enable global method security, an aspect intercepts the call to the protected method. If the given authorization rules aren't respected, the aspect doesn't delegate the call to the protected method.

Plenty of implementations in Spring framework rely on aspect-oriented programming (AOP). Method security is just one of the many components in Spring applications relying on aspects. If you need a refresher on aspects and AOP, I recommend you read chapter 6 of *Spring Start Here (Manning, 2021)* , a book I wrote. Briefly, we classify the call authorization as

- *Preauthorization*—The framework checks the authorization rules before the method call.
- *Postauthorization*—The framework checks the authorization rules after the method executes.

Let's take both approaches, detail them, and implement them with some examples.

USING PREAUTHORIZATION TO SECURE ACCESS TO METHODS

Say we have a method `findDocumentsByUser(String username)` that returns to the caller documents for a specific user. The caller provides through the method's parameters the user's name for which the method retrieves the documents. Assume you need to make sure that the authenticated user can only obtain their own documents. Can we apply a rule to this method such that only the method calls that receive the username of the authenticated user as a parameter are allowed? Yes! This is something we do with preauthorization.

When we apply authorization rules that completely forbid anyone to call a method in specific situations, we call this *preauthorization* (figure 1.32). This approach implies that the framework verifies the authorization conditions before executing the method. If the caller doesn't have the permissions according to the authorization rules that we define, the framework doesn't

delegate the call to the method. Instead, the framework throws an exception named `AccessDeniedException`. This is by far the most often used approach to global method security.

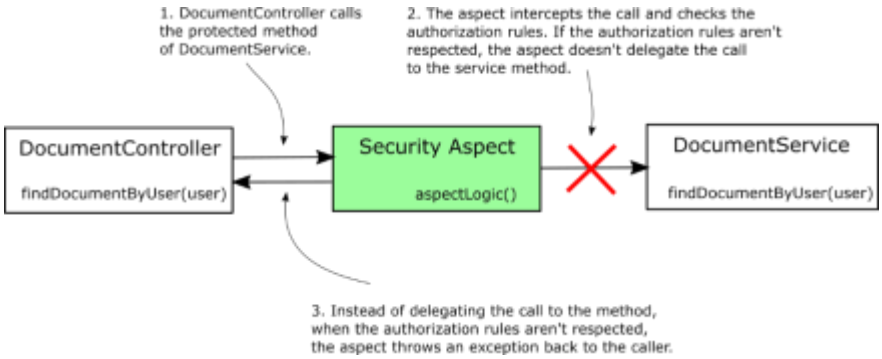

Figure 1.32 With preauthorization, the authorization rules are verified before delegating the method call further. The framework won't delegate the call if the authorization rules aren't respected, and instead, throws an exception to the method caller.

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 1.33). 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.

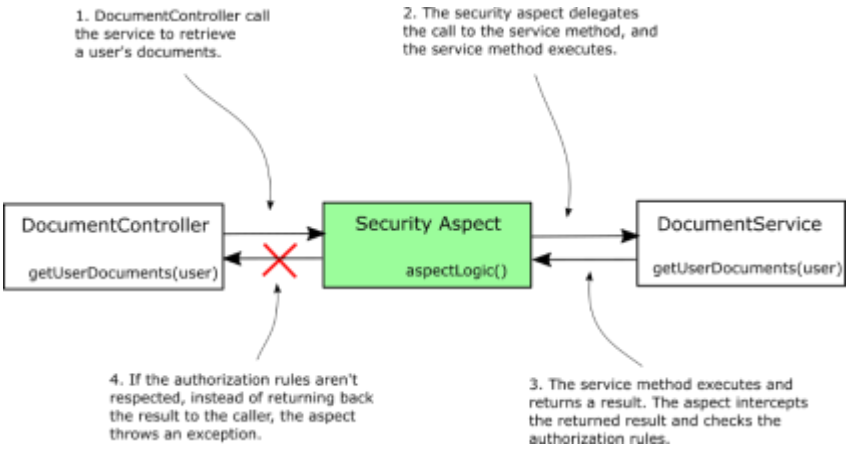

Figure 1.33 With postauthorization, the aspect delegates the call to the protected method. After the protected method finishes execution, the aspect checks the authorization rules. If the rules aren't respected, instead of returning the result to the caller, the aspect throws an exception.

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.

1.30 Enabling method security in your project

In this section, we work on a project to apply the preauthorization and postauthorization features offered by method security. Method security isn't enabled by default in a Spring Security project. To use it, you need to first enable it. However, enabling this functionality is straightforward. You do this by simply using the `@EnableMethodSecurity` annotation on the configuration class.

For this project, I wrote a `ProjectConfig` configuration class, as presented in listing 1.70. On the configuration class, we add the `@EnableMethodSecurity` annotation. Method security offers us three approaches to define the authorization rules that we discuss in this part of the training:

- The pre-/postauthorization annotations (enabled by default)
- The JSR 250 annotation, `@RolesAllowed`
- The `@Secured` annotation

Because in almost all cases, pre-/postauthorization annotations are the only approach used, we discuss this approach here. This approach is pre-enabled once you add the `@EnableMethodSecurity` annotation.

Listing 1.70 Enabling method security

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {
}
```

You can use global method security with any authentication approach, from HTTP Basic authentication to OAuth 2. To keep it simple and allow you to focus on new details, we provide method security with HTTP Basic authentication. For this reason, the pom.xml file for the projects in this example only needs the web and Spring Security dependencies, as the next code snippet presents:

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

> **NOTE** In the previous Spring Security versions, we used the `@EnableGlobalMethodSecurity` annotation, and the pre and post-authorization wasn't enabled by default. If you need to work with method authorization and a previous Spring Security version (older than 6), you will find useful to read chapter 16 from the first edition of Spring Security in Action.

1.31 Applying preauthorization rules

In this section, we implement an example of preauthorization. As we discussed earlier, preauthorization implies defining authorization rules that Spring Security applies before calling a specific method. If the rules aren't respected, the framework doesn't call the method.

The application we implement in this section has a simple scenario. It exposes an endpoint, `/hello`, which returns the string `"Hello, "` followed by a name. To obtain the name, the controller calls a service method (figure 1.34). This method applies a preauthorization rule to verify the user has write authority.

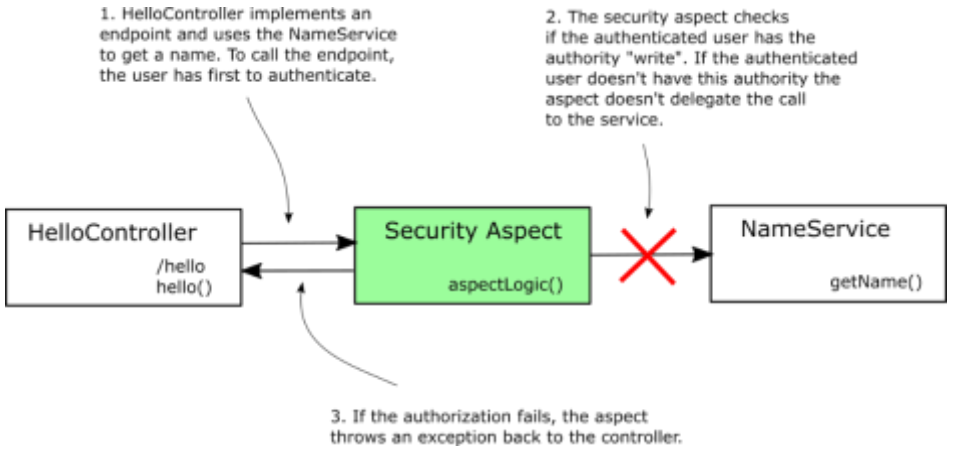

Figure 1.34 To call the `getName()` method of `NameService`, the authenticated user needs to have write authority. If the user doesn't have this authority, the framework won't allow the call and throws an exception.

I added a `UserDetailsService` and a `PasswordEncoder` to make sure I have some users to authenticate. To validate our solution, we need two users: one user with write authority and another that doesn't have write authority. We prove that the first user can successfully call the endpoint, while for the second user, the app throws an authorization exception when trying to call the method. The following listing shows the complete definition of the configuration class, which defines the `UserDetailsService` and the `PasswordEncoder`.

Listing 1.71 The configuration class for `UserDetailsService` and `PasswordEncoder`

```
@Configuration
@EnableMethodSecurity    #A
public class ProjectConfig {

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

    var u1 = User.withUsername("natalie")
              .password("12345")
              .authorities("read")
              .build();

    var u2 = User.withUsername("emma")
              .password("12345")
              .authorities("write")
              .build();

    service.createUser(u1);
    service.createUser(u2);
```

```
    return service;
  }

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

#A Enables method security for pre-/postauthorization
#B Adds a UserDetailsService to the Spring context with two users for testing
#C 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 earlier, where we discussed applying authorization at the endpoint level. The following listing defines the service class, which provides the value for the name.

Listing 1.71 The service class defines the preauthorization rule on the method

```
@Service
public class NameService {

  @PreAuthorize("hasAuthority('write')")      #A
  public String getName() {
    return "Fantastico";
  }
}
```

#A 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 1.72 The controller class implementing the endpoint and using the service

```
@RestController
public class HelloController {

  private final NameService nameService;     #A

  // omitted constructor

  @GetMapping("/hello")
  public String hello() {
    return "Hello, " + nameService.getName();      #B
  }
}
```

#A Injects the service from the context
#B 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 earlier 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 1.35).

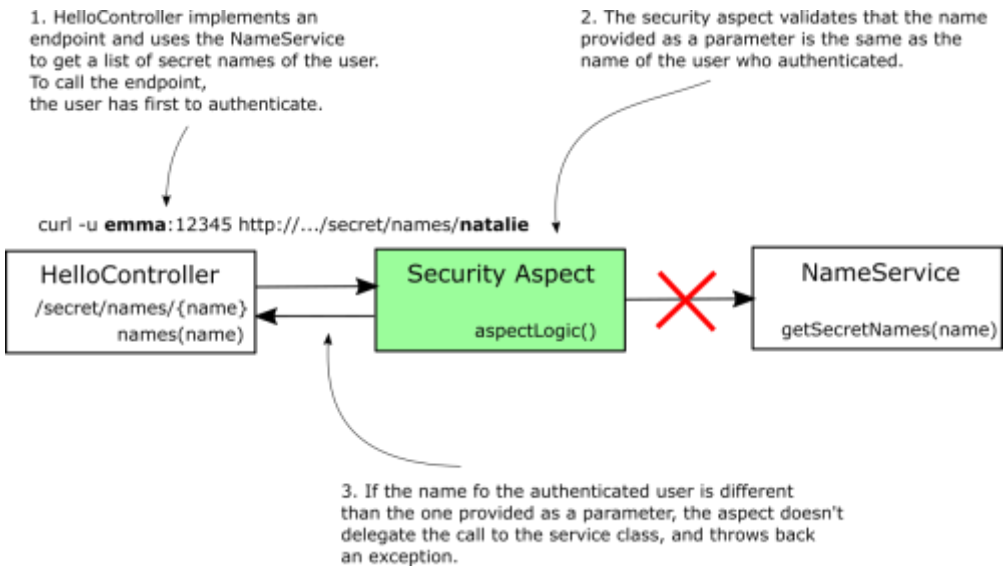

Figure 1.35 When implementing preauthorization, we can use the values of the method parameters in the authorization rules. In our example, only the authenticated user can retrieve information about their secret names.

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 1.74 The controller class defining an endpoint for testing

```
@RestController
public class HelloController {


  private final NameService nameService;     #A

  // omitted constructor

  @GetMapping("/secret/names/{name}")     #B
  public List<String> names(@PathVariable String name) {
      return nameService.getSecretNames(name);      #C
  }
}
```

#A From the context, injects an instance of the service class that defines the protected method
#B Defines an endpoint that takes a value from a path variable
#C 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 1.74. 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 1.74 The `NameService` class defines the protected method

```
@Service
public class NameService {

  private Map<String, List<String>> secretNames =
    Map.of(
     "natalie", List.of("Energico", "Perfecto"),
     "emma", List.of("Fantastico"));

  @PreAuthorize    #A
    ("#name == authentication.principal.username")
  public List<String> getSecretNames(String name) {
    return secretNames.get(name);
  }
}
```

#A 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 method security to any layer of your application. In the examples presented here, you find the authorization rules applied for methods of the service classes. But you can apply authorization rules with method security in any part of your application: controllers, repositories, managers, proxies, and so on.

1.32 Applying postauthorization rules

Now say you want to allow a call to a method, but in certain circumstances, you want to make sure the caller doesn't receive the returned value. When we want to apply an authorization rule that is verified after the call of a method, we use postauthorization. It may sound a little bit awkward at the beginning: why would someone be able to execute the code but not get the result? Well, it's not about the method itself, but imagine this method retrieves some data from a data source, say a web service or a database. The conditions you need to add for authorization depend on the received data. So you allow the method to execute, but you validate what it returns and, if it doesn't meet the criteria, you don't let the caller access the return value.

To apply postauthorization rules with Spring Security, we use the `@PostAuthorize` annotation, which is similar to `@PreAuthorize`. 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 1.36).

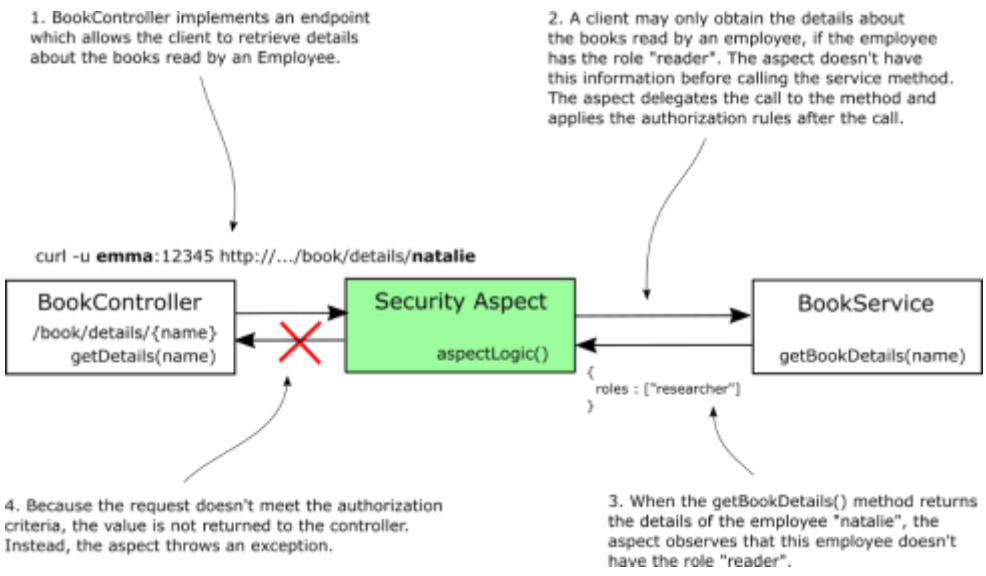

Figure 1.36 With postauthorization, we don't protect the method from being called, but we protect the

returned value from being exposed if the defined authorization rules aren't respected.

The scenario for our example 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, 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 1.75 Enabling method security and defining users

```
@Configuration
@EnableMethodSecurity
public class ProjectConfig {

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

    var u1 = User.withUsername("natalie")
                 .password("12345")
                 .authorities("read")
                 .build();

    var u2 = User.withUsername("emma")
                 .password("12345")
                 .authorities("write")
                 .build();

    service.createUser(u1);
    service.createUser(u2);

    return service;
  }

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

We also need to declare a class to represent the `Employee` object with its name, book list, and roles list. The following listing defines the `Employee` class.

Listing 1.76 The definition of the `Employee` class

```
public class Employee {

  private String name;
  private List<String> books;
```

```
  private List<String> roles;

  // Omitted constructor, getters, and setters
}
```

We probably get our employee details from a database. To keep our example shorter, I use a `Map` with a couple of records that we consider as our data source. In listing 1.77, you find the definition of the `BookService` class. The `BookService` class also contains the method for which we apply the authorization rules. Observe that the expression we use with the `@PostAuthorize` annotation refers to the value returned by the method `returnObject`. The postauthorization expression can use the value returned by the method, which is available after the method executes.

Listing 1.77 The `BookService` class defining the authorized method

```
@Service
public class BookService {

  private Map<String, Employee> records =
    Map.of("emma",
           new Employee("Emma Thompson",
               List.of("Karamazov Brothers"),
               List.of("accountant", "reader")),
           "natalie",
           new Employee("Natalie Parker",
               List.of("Beautiful Paris"),
               List.of("researcher"))
      );
  @PostAuthorize     #A
  ("returnObject.roles.contains('reader')")
  public Employee getBookDetails(String name) {
      return records.get(name);
  }
}
```

#A 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 1.78 The controller class implementing the endpoint

```
@RestController
public class BookController {

  private final BookService bookService;

  // omitted constructor

  @GetMapping("/book/details/{name}")
  public Employee getDetails(@PathVariable String name) {
    return bookService.getBookDetails(name);
  }
}
```

You can now start the application and call the endpoint to observe the app's behavior. In the next code snippets, you find examples of calling the endpoint. Any of the users can access the details of Emma because the returned list of roles contains the string "`reader`", but no user can obtain the details for Natalie. Calling the endpoint to get the details for Emma and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
  "name":"Emma Thompson",
  "books":["Karamazov Brothers"],
  "roles":["accountant","reader"]
}
```

Calling the endpoint to get the details for Emma and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/emma
```

The response body is

```
{
  "name":"Emma Thompson",
  "books":["Karamazov Brothers"],
  "roles":["accountant","reader"]
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Emma, we use this command:

```
curl -u emma:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/book/details/natalie"
}
```

Calling the endpoint to get the details for Natalie and authenticating with user Natalie, we use this command:

```
curl -u natalie:12345 http://localhost:8080/book/details/natalie
```

The response body is

```
{
  "status":403,
  "error":"Forbidden",
  "message":"Forbidden",
  "path":"/book/details/natalie"
}
```

NOTE You can use both `@PreAuthorize` and `@PostAuthorize` on the same method if your requirements need to have both preauthorization and postauthorization.

1.33 Implementing a basic authentication using JWTs

In this section, we implement a basic OAuth 2 authorization server using the Spring Security authorization server framework. We'll go through all the main components you need to plug into the configuration to make this work and discuss them individually. Then, we'll test the app using the most essential two OAuth 2 grant types: the authorization code grant type and the client credentials grant type.

The main components you need to set up for your authorization server to work properly are:

1. *The configuration filter for protocol endpoints* - helps you define configurations specific to the authorization server capabilities, including various customizations.
2. *The authentication configuration filter* – similar to any web application secured with Spring Security, you'll use this filter to define the authentication and authorization configurations as well as configurations to any other security mechanisms such as CORS and CSRF.
3. *The user details management components* – same as for any authentication process implemented with Spring Security- are established through a `UserDetailsService` bean and a `PasswordEncoder`.
4. *The client details management* – the authorization server uses a component named `RegisteredClientRepository` to manage the client credentials and other details.
5. *The key-pairs (used to sign and validate tokens) management* – when using non-opaque tokens, the authorization server uses a private key to sign the tokens. The authorization server also offers access to a public key that the resource server can use to validate the tokens. The authorization server manages the private-public key pairs through a "key source" component.
6. *The general app settings* – a component named `AuthorizationServerSettings` helps you configure generic customizations such as the endpoints the app exposes.

Figure 1.37 offers a visual perspective of the components we need to plug in and configure for a minimal authorization server app to work.

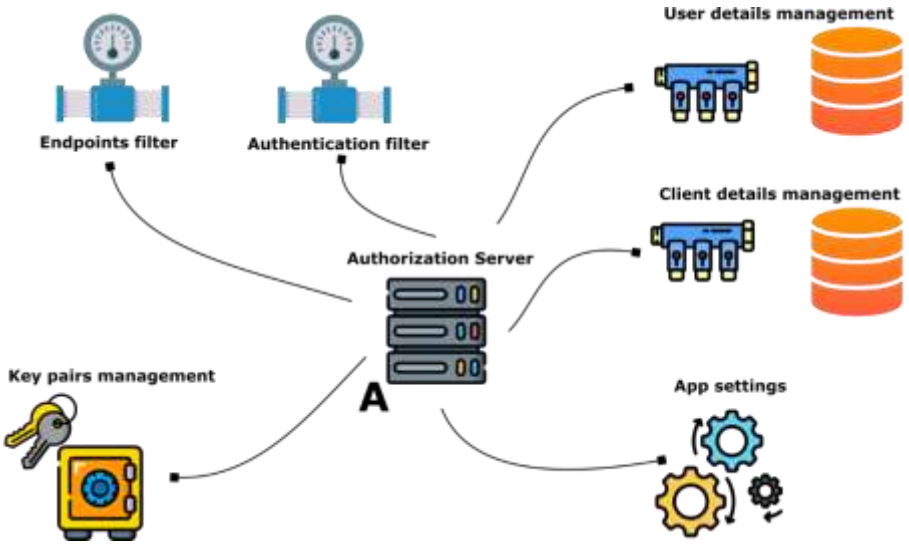

Figure 1.37 The components we need to configure and plug in for an authorization server implemented with Spring Security to work.

To begin, we have to add the needed dependencies to our project. In the next code snippet, you find the dependencies you need to add to your pom.xml project.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-oauth2-authorization-server</artifactId>
   <version>1.0.1</version>
</dependency>
```

We write the configurations in standard Spring configuration classes like in the next code snippet.

```
@Configuration
public class SecurityConfig {


}
```

Remember that like for any other Spring app, the beans can be defined in multiple configuration classes, or they can be (depending on the case) defined using stereotype

annotations. If you need a refresher on managing the Spring context, I recommend you read the first part of Spring Start Here (Manning, 2021), a book I wrote.

Take a look at listing 1.79, which presents the configuration filter for protocol endpoints. The `applyDefaultSecurity()` is a utility method we use to define a minimal set of configurations you can override lately if needed. After calling this method, the listing shows how to enable the OpenID Connect protocol using the `oidc()` method of the `OAuth2AuthorizationServerConfigurer` configurer object.

Also, the filter in listing 1.79 specifies the authentication page to which the app needs to redirect the user when asked to log in. We need this configuration since we expect to enable the authorization code grant type for our example, which implies that the users must authenticate. The default path in a Spring web app is /login, so unless we configure a custom one, this is the one we'll use for the authorization server configuration.

Listing 1.79 Implementing the filter for configuring the protocol endpoints

```
@Bean
@Order(1)
public SecurityFilterChain asFilterChain(HttpSecurity http)
  throws Exception {

    OAuth2AuthorizationServerConfiguration     #A
       .applyDefaultSecurity(http);     #A

    http.getConfigurer(OAuth2AuthorizationServerConfigurer.class)     #B
        .oidc(Customizer.withDefaults());    #B

    http.exceptionHandling((e) ->
      e.authenticationEntryPoint(
          new LoginUrlAuthenticationEntryPoint("/login"))     #C
    );

    return http.build();
  }
```

#A Calling the utility method to apply default configurations for the authorization server endpoints.
#B Enabling the OpenID Connect protocol.
#C Specifying the authentication page for users.

Listing 1.79 configures authentication and authorization. These configurations work similarly to any web app. In listing 1.80, I set up the minimum configurations:

1. Enabling the form login authentication so the app gives the user a simple login page to authenticate.
2. Specify that the app only allows access to authenticated users to any endpoints.

Other configurations you could write here, besides authentication and authorization, could be for specific protection mechanisms such as CSRF or CORS.

Observe also the `@Order` annotation I used in both listings 1.79 and 1.80. This annotation is needed since we have multiple `SecurityFilterChain` instances configured in the app context, and we need to provide the order in which they take priority in configuration.

Listing 1.80 Implementing the filter for authorization configuration

```
@Bean
@Order(2)      #A
public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http)
  throws Exception {

  http.formLogin();     #B

  http.authorizeHttpRequests()        #C
        .anyRequest().authenticated();       #C

  return http.build();
}
```

#A We set the filter to be interpreted after the protocol endpoints one.
#B We enable the form login authentication method.
#C We configure all endpoints to require authentication.

If you expect clients will use the authorization server you build for grant types that imply user authentication (such as the authorization code grant type), then your server needs to manage user details! All you need is a `UserDetailsService` and a `PasswordEncoder` implementation.

Listing 1.81 presents a definition for these two components. In this example, we use an in-memory implementation for the `UserDetailsService`, but remember you learned how to write a custom implementation for it earlier. In most cases, same as for other web apps, you'll keep such details stored in a database. Therefore, you'll need to write a custom implementation for the `UserDetailsService` contract.

Also, remember that earlier we discussed that the `NoOpPasswordEncoder` is something you should only use with learning samples. The `NoOpPasswordEncoder` doesn't transform the passwords anyhow, leaving them in clear text and at the disposal of anyone who can access them, which is not good. You should always use a password encoder with a strong hash function such as BCrypt.

Listing 1.81 Defining the user details management

```
@Bean
public UserDetailsService userDetailsService() {
  UserDetails userDetails = User.withUsername("bill")
        .password("password")
        .roles("USER")
        .build();
```

```
    return new InMemoryUserDetailsManager(userDetails);
  }

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

The authorization server needs a `RegisteredClientRepository` component to manage the clients' details. The `RegisteredClientRepository` interface works similarly to the `UserDetailsService` one, but it's designed to retrieve client details. Similarly, the framework provides the `RegisteredClient` object, whose purpose is to describe a client app the authorization server knows.

To make an analogy to what you learned earier, we can say that `RegisteredClient` is for clients what `UserDetails` is for users. Similarly, `RegisteredClientRepository` works for client details like a `UserDetailsService` works for the users' details (figure 1.38).

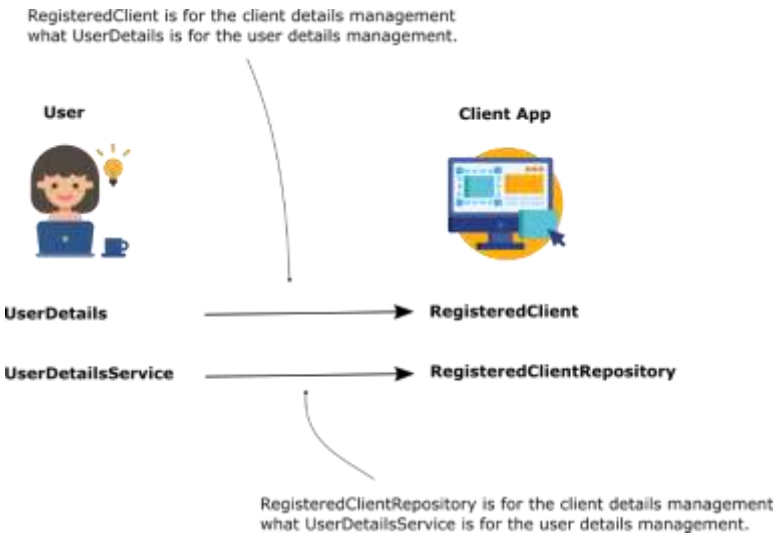

Figure 1.38 To manage the clients' details we use a RegisteredClientRepository implementation. The RegisteredClientRepository uses RegisterdClient objects to represent the clients' details.

In this example, we'll use an in-memory implementation to allow you to focus on the overall implementation of the authorization server. Still, in a real-world app, you'd most likely need to provide an implementation to this interface to grab the data from a database. For this to work, you implement the `RegisteredClientRepository` interface similarly to how you learned to implement the `UserDetailsService` interface.

Listing 1.82 shows the definition of the in-memory `RegisteredClientRepository` bean. The method creates one `RegisteredClient` instance with the needed details and stores it in-memory to be used during authentication by the authorization server.

Listing 1.82 Implementing client details management

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =       #A
    RegisteredClient      #A
        .withId(UUID.randomUUID().toString())       #A
        .clientId("client")      #A
        .clientSecret("secret")       #A
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)         #A
        .authorizationGrantType(      #A
           AuthorizationGrantType.AUTHORIZATION_CODE)        #A
        .redirectUri("https://www.manning.com/authorized")       #A
        .scope(OidcScopes.OPENID)       #A
        .build();       #A

  return new InMemoryRegisteredClientRepository(registeredClient);      #B
}
```

#A Creating a RegisteredClient instance.
#B Adding it to be managed by the in-memory RegisteredClientRepository implementation.

The details we specified when creating the `RegisteredClient` instance are the following:

- *A unique internal ID* – value that uniquily identifies the client and has a purpose only in the internal app processes.
- *A client ID* - external client identifier similar to what a username is for the user.
- *A client secret* – similar to what a password is for a user.
- *The client authentication method* – tells how the authorization server expects the client to authenticate when sending requests for access tokens.
- *Authorization grant type* – A grant type allowed by the authorization server for this client. A client might use multiple grant types.
- *Redirect URI* – One of the URI addresses the authorization server allows the client to request a redirect for providing the authorization code in case of the authorization code grant type.
- *A scope* – Defines a purpose for the request of an access token. The scope can be used later in authorization rules.

In this example, the client only uses the authorization code grant type. But you can have clients that use multiple grant types. In case you want a client to be able to use multiple grant types, you need to specify them as presented in the next code snippet. The client defined in

the next code snippet can use any of the authorization code, client credentials, or the refresh token grant types.

```
RegisteredClient registeredClient =
    RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
        .clientSecret("secret")
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(
           AuthorizationGrantType.AUTHORIZATION_CODE)
        .authorizationGrantType(
           AuthorizationGrantType.CLIENT_CREDENTIALS)
        .authorizationGrantType(
           AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("https://www.manning.com/authorized")
        .scope(OidcScopes.OPENID)
        .build();
```

Similarly, by repeatedly calling the `redirectUri()` method, you can specify multiple allowed redirect URIs. In a similar way, a client might also have access to multiple scopes as well.

Normally, the app would keep all these details in a database from where your `RegisteredClientRepository` custom implementation would retrieve them.

Besides having user and client details, you must configure key pair management if the authorization server uses non-opaque tokens. For non-opaque tokens, the authorization server uses private keys to sign the tokens and provides the clients with public keys they can use to validate the tokens' authenticity.

The `JWKSource` is the object providing keys management for the Spring Security authorization server. Listing 1.83 shows how to configure a `JWKSource` in the app's context. For this example, I create a key pair programmatically and add it to the set of keys the authorization server can use. In a real-world app, the app would read the keys from a location where they're safely stored (such as a vault configured in the environment).

Configuring an environment that perfectly resembles a real-world system would be too complex, and I prefer you focus on the authorization server implementation. However, remember that in a real app, it doesn't make sense to generate new keys every time the app restarts (like it happens in our case). If that happens for a real app, every time a new deployment occurs, the tokens that were already issued would not work anymore (since they can't be validated anymore with the existing keys).

So, for our example, generating the keys programmatically works and we'll help us demonstrate how the authorization server works. In a real-world app, you must keep the keys secured somewhere and read them from the given location.

Listing 1.83 Implementing the key pair set management

```
@Bean
```

```
public JWKSource<SecurityContext> jwkSource()
  throws NoSuchAlgorithmException {

  KeyPairGenerator keyPairGenerator =     #A
    KeyPairGenerator.getInstance("RSA");    #A

    keyPairGenerator.initialize(2048);    #A
    KeyPair keyPair = keyPairGenerator.generateKeyPair();     #A

    RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();     #A
    RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();    #A

    RSAKey rsaKey = new RSAKey.Builder(publicKey)    #A
        .privateKey(privateKey)     #A
        .keyID(UUID.randomUUID().toString())     #A
        .build();     #A

    JWKSet jwkSet = new JWKSet(rsaKey);     #B
    return new ImmutableJWKSet<>(jwkSet);     #C
}
```

#A Generating a public-private key pair programmatically using the RSA cryptographic algorithm.
#B Adding the key pair to the set the authorization server uses to sign the issued tokens.
#C Wrapping the key set into a JWKSource implementation and returning it to be added to the Spring context.

Finally, the last component we need to add to our minimal configuration is an `AuthorizationServerSettings` object (listing 1.84). This object allows you to customize all the endpoints paths that the authorization server exposes. If you simply create the object as shown in listing 14.6, the endpoints paths will get some defaults that we'll analyze later in this section.

Listing 1.84 Configuring the authorization server generic settings

```
@Bean
public AuthorizationServerSettings authorizationServerSettings() {
  return AuthorizationServerSettings.builder().build();
}
```

Now, we can start the app and test if it works.

1.34 Running the authorization code grant type

In this section, we test the authorization server we implemented in section 1.32. We expect that by using the registered client details, we'd be able to follow the authorization code flow and get an access token. We'll follow the next steps:

1. Check the endpoints that the authorization server exposes.
2. Use the authorization endpoint to get an authorization code.
3. Use the authorization code to get an access token.

First step is to find out the endpoints' paths the authorization server exposes. Since we didn't configure custom ones, we must use the defaults. But which are the defaults? You can

call the OpenID configuration endpoint in the next snippet to discover these details. This request uses the HTTP GET method, and no authentication is required.

http://localhost:8080/.well-known/openid-configuration

When calling the OpenID configuration endpoint you should get a response that looks like the one presented in listing 1.85.

Listing 1.85 The response of the OpenID configuration request

```
{
    "issuer": "http://localhost:8080",
    "authorization_endpoint":
       "http://localhost:8080/oauth2/authorize",     #A
    "token_endpoint": "http://localhost:8080/oauth2/token",     #B
    "token_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "jwks_uri": "http://localhost:8080/oauth2/jwks",    #C
    "userinfo_endpoint": "http://localhost:8080/userinfo",
    "response_types_supported": [
        "code"
    ],
    "grant_types_supported": [
        "authorization_code",
        "client_credentials",
        "refresh_token"
    ],
    "revocation_endpoint": "http://localhost:8080/oauth2/revoke",
    "revocation_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "introspection_endpoint":
      "http://localhost:8080/oauth2/introspect",    #D
    "introspection_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "client_secret_post",
        "client_secret_jwt",
        "private_key_jwt"
    ],
    "subject_types_supported": [
        "public"
    ],
    "id_token_signing_alg_values_supported": [
        "RS256"
    ],
    "scopes_supported": [
        "openid"
```

```
        ]
}
```

#A The authorization endpoint to which a client will redirect the user to authenticate.
#B The token endpoint the client will call to request an access token.
#C The key set endpoint a resource server will call to get the public keys it can use to validate tokens.
#D The introspection endpoint a resource server can call to validate opaque tokens.

Look at figure 1.39 to remember the authorization code flow. We'll use this now to demonstrate that the authorization server we built works fine.

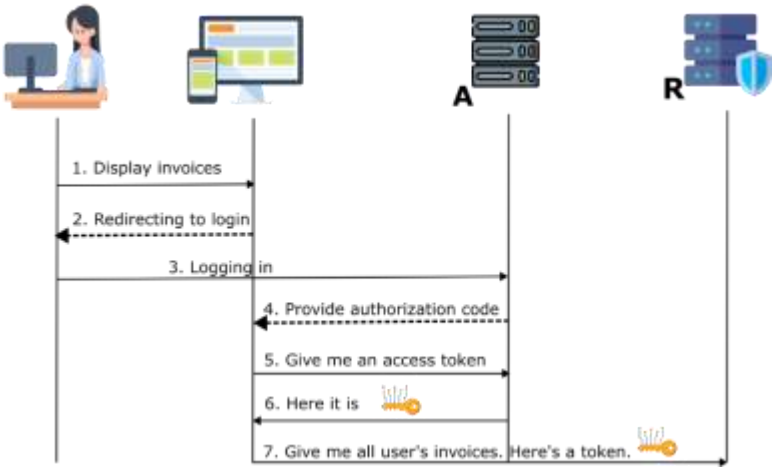

Figure 1.39 The authorization code grant type. After the user successfully authenticates, the client gets an authorization code. The client uses the authorization code to get an access token to help it access resources that the resource server protects.

Since we don't have a client for our example, we need to act like one. Now that you know the authorization endpoint, you can put it in a browser address to simulate how the client would redirect the user to it. The next snippet shows you the authorization request.

```
http://localhost:8080/oauth2/authorize?response_type=code&client_id=client&
scope=openid&redirect_uri=https://www.manning.com/authorized&code_challenge
=QYPAZ5NU8yvtlQ9erXrUYR-T5AGCjCF47vN-KsaI2A8&code_challenge_method=S256
```

For the authorization request, you observe I added a few parameters:

- `response_type=code` – this request parameter specifies to the authorization server that the client wants to use the authorization code grant type. Remember that a client might have configured multiple grant types. It needs to tell the authorization server

which wants it wants to use.

- `client_id=client` – the client identifier is like the "username" for the user. It uniquily identifies the client in the system.
- `scope=openid` – Specifies which scope the client wants to be granted with this authentication attempt.
- `redirect_uri`=https://www.manning.com/authorized – Specifies the URI to which the authorization server will redirect after a successful authentication. This URI must be one of those previously configured for the current client.
- `code_challenge=QYPAZ5NU8yvtlQ…` - If using the authorization code enhanced with PKCE, you must provide the code challenge with the authorization request. When requesting the token, the client must send the verifier pair to prove they are the same application that initially sent this request. The PKCE flow is enabled by default.
- `code_challenge_method=S256` – This request parameter specifies which is the hashing method that has been used to create the challenge from the verifier. In this case, S256 means SHA-256 was used as a hash function.

I recommend you use the authorization code grant type with PKCE, but if you really need to disable the PKCE enhancement of the flow, you can do that as presented in the next code snippet. Observe the `clientSettings()` method that takes a `ClientSettings` instance where you can specify you disable the proof key for code exchange. In this example, we'll demonstrate the authorization code with PCKE, which is the default and recommended way.

```
RegisteredClient registeredClient = RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
        // …
        .clientSettings(ClientSettings.builder()
            .requireProofKey(false)
            .build())
        .build();
```

By sending the authorization request through the browser's address bar, we simulate step 2 from figure 1.39. The authorization server will redirect us to its login page, and we can authenticate using the user's name and password. This is step 3 from figure 1.39. Figure 1.40 shows the login page the authorization server presents to the user.

Figure 1.40 The login page that the authorization server presents to the user in response to the authorization request.

For our implementation, we only have one user. Their credentials are username "bill" and the password "password". Once the user fills in the correct credentials and selects the "Sign in" button, the authorization server redirects the user to the requested redirect URI and provides an authorization code – step 4 in figure 1.39.

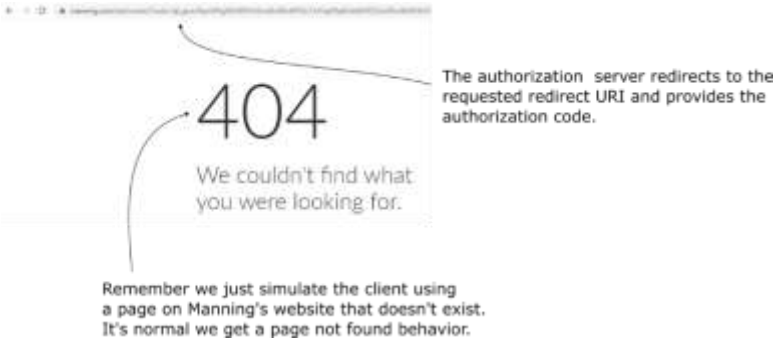

Figure 1.41 After successful authentication, the authorization server redirects the user to the requested redirect URI and provides an authorization code. The client can use the authorization code to get an access token.

Once the client has the authorization code, it can request an access token. The client can request the access token using the token endpoint. Next snippet shows a cURL request of a token. The request uses the HTTP POST method. Because we specified that HTTP Basic authentication is requested when registering the client, the token request needs authentication with HTTP Basic with the client ID and secret.

```
curl -X POST 'http://localhost:8080/oauth2/token?
client_id=client&
redirect_uri=https://www.manning.com/authorized&
```

```
grant_type=authorization_code&
code=ao2oz47zdM0D5gbAqtZVBmdoUWbZI_GXt-
jfGKetWCxaeD1ES9JLx2FtgQZAPMqCZixH4vbhzvuvx2qEJrDnjvOnnojkmYBc8OUMxr0wHkCtE
-NgF1zQ50txKTvPFjTv&
code_verifier=qPsH306-ZDDaOE8DFzVn05TkN3ZZoVmI_6x4LsVglQI' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The request parameters we used are:

- *client_id=client* – needed to identify the client
- *redirect_uri= https://www.manning.com/authorized* – the redirect URI through which the authorization server provided the authorization code after the successful user authentication.
- *grant_type=authorization_code* – which flow the client uses to request the access token
- *code=ao2oz47zdM0D5…*– the value of the authorization code the authorization server provided to the client
- *code_verifier=qPsH306-ZDD…* – the verifier based on which the challenge that the client sent at authorization was created.

NOTE Pay very much attention to all the details. If any of the values doesn't properly match what the app knows or what was sent in the authorization request, the token request won't succeed.

The next snippet shows the response body of the token request. Now the client has an access token it can use to send requests to the resource server.

```
{
    "access_token": "eyJraWQiOiI4ODlhNGFmO…",
    "scope": "openid",
    "id_token": "eyJraWQiOiI4ODlhNGFmOS1…",
    "token_type": "Bearer",
    "expires_in": 299
}
```

Since we enabled the OpenID Connect protocol so we don't only rely on OAuth 2, an ID token is also present in the token response. If the client had been registered using the refresh token grant type, a refresh token would have also been generated and sent through the response.

Generating the code verifier and challenge

In the example we worked on in this section, I used the authorization code with PKCE. In the authorization and token requests, I used a challenge and a verifier value I had previously generated. I didn't pay too much attention to these values since they are the client's business and not something the authorization or resource server generates. In a

real-world app your JavaScript or mobile app will have to generate both these values when using them in the OAuth 2 flow.

But in case you wonder, I will explain how I generated these two values in this sidebar.

The code verifier is a random 32 bytes piece of data. To make it easy to transfer through a HTTP request, this data needs to be Base64 encoded using an URL encoder and without padding. The next code snippet shows you how to do that in Java.

```
SecureRandom secureRandom = new SecureRandom();
byte [] code = new byte[32];
secureRandom.nextBytes(code);
String codeVerifier = Base64.getUrlEncoder()
        .withoutPadding()
        .encodeToString(code);
```

Once you have the code verifier, you use a hash function to generate the challenge. The next code snippet shows you how to create the challenge using the SHA-256 hash function. Same as for the verifier, you need to use Base64 to change the byte array into a String value making it easier to transfer through the HTTP request.

```
MessageDigest messageDigest = MessageDigest.getInstance("SHA-256");

byte [] digested = messageDigest.digest(verifier.getBytes());
String codeChallenge = Base64.getUrlEncoder()
          .withoutPadding()
          .encodeToString(digested);
```

Now you have a verifier and a challenge. You can use them in the authorization and token requests as discussed in this section.

1.35 Running the client credentials grant type

In this section, we'll try out the client credentials grant type using the authorization server we implemented in section 1.33. Rememeber that the client grant type is a flow that allows the client to get an access token without a user's authentication or consent. Preferably, you shouldn't have a client being able to use both a user-dependent grant type (such as the authorization code) and a client-independent one (such as the client credentials).

The authorization implementation might fail to make a difference between an access token obtained through the authorization code grant type and one that the client obtained through the client credential grant type. So it's best to use different registration for such cases and preferably distinguish the token usage through different scopes.

Listing 1.86 shows a registered client able to use the client credentials grant type. Observe I have also configured a different scope. In this case, "CUSTOM" is just a name I chose, you can choose any name for the scopes. The name you choose should generally make the purpose of the scope easier to understand. For example, if this app needs to use the client credentials grant type to get a token to check the resource server's liveness state, then, maybe it's better to name the scope "LIVENESS" so it makes things obvious.

Listing 1.86 Configuring a registered client for the client credentials grant type

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =
    RegisteredClient.withId(UUID.randomUUID().toString())
      .clientId("client")
      .clientSecret("secret")
      .clientAuthenticationMethod(
         ClientAuthenticationMethod.CLIENT_SECRET_BASIC)    #A
      .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
      .scope("CUSTOM")    #B
      .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}
```

#A Allowing the registered client to use the client credentials grant type.
#B Configuring a scope to match the purpose for the access token request.

Figure 1.42 reminds you the client credentials flow. To get an access token, the client simply sends a request and authenticates using their credentials (the client ID and secret).

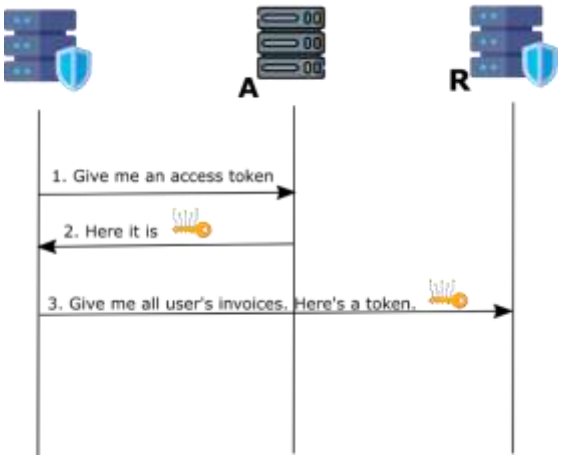

Figure 1.42 The client credentials grant type. An app can get an access token by only authenticating with its client credentials

The following snippet shows a cURL token request. If you compare it with the request we used in the earlier section, when we ran the authorization code grant type, you'll observe this one is simpler. The client only needs to mention they use the client credentials grant type and the scope in which they request a token. The client uses its credentials with HTTP Basic on the request to authenticate.

```
curl -X POST 'http://localhost:8080/oauth2/token?
grant_type=client_credentials&
scope=CUSTOM' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows the HTTP response body containing the requested access token.

```
{
    "access_token": "eyJraWQiOiI4N2E3YjJiNS…",
    "scope": "CUSTOM",
    "token_type": "Bearer",
    "expires_in": 300
}
```

1.36 Using opaque tokens and introspection

By now, we have demonstrated the authorization code grant type and the client credentials grant typ. With both, we managed to configure clients that can get non-opaque access tokens. However, you can also easily configure the clients to use opaque tokens. In this section, I'll show you how to configure the registered clients to get opaque tokens and how the authorization server help with validating the opaque tokens.

Listing 1.87 shows how to configure a registered client to use opaque tokens. Remember that opaque tokes can be used with any grant type. In this section, I'll use the client credentials grant type to keep things simple and allow you to focus on the discussed subject. You can as well generate opaque tokens with the authorization code grant type.

Listing 1.87 Configuring clients to use opaque tokens

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
  RegisteredClient registeredClient =
    RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("client")
        .clientSecret("secret")
        .clientAuthenticationMethod(
           ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
        .tokenSettings(TokenSettings.builder()
           .accessTokenFormat(OAuth2TokenFormat.REFERENCE)      #A
            .build())
        .scope("CUSTOM")
        .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
  }
```

#A Configuring the client to use opaque access tokens

If you request an access token, you'll get an opaque token. This token is shorter and doesn't contain data. The next snippet is a cURL request for an access token.

```
curl -X POST 'http://localhost:8080/oauth2/token?
```

```
grant_type=client_credentials&
scope=CUSTOM' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows the response similar to what you got in return when we were expecting non-opaque tokens. The only difference is the token itself which is no longer a JWT token, but an opaque one.

```
{
    "access_token": "iED8-...",
    "scope": "CUSTOM",
    "token_type": "Bearer",
    "expires_in": 299
}
```

The next snippet shows an example of a full opaque token. Observe is much shorter and doesn't have the same structure as a JWT (the three parts separated by dots).

```
iED8-
aUd5QLTfihDOTGUhKgKwzhJFzYWnGdpNT2UZWO3VVDqtMONNdozq1r9r7RiP0aNWgJipcEu5Hec
AJ75VyNJyNuj-kaJvjpWL5Ns7Ndb7Uh6DI6M1wMuUcUDEjJP
```

Since an opaque token doesn't contain data, how can someone validate it and get more details about the client (and potentially user) for whom the authorization server generated it? The easiest way (and most used) is directly asking the authorization server. The authorization server exposes an endpoint where one can send a request with the token. The authorization server replies with the needed details about the token. This process is called introspection (figure 1.43).

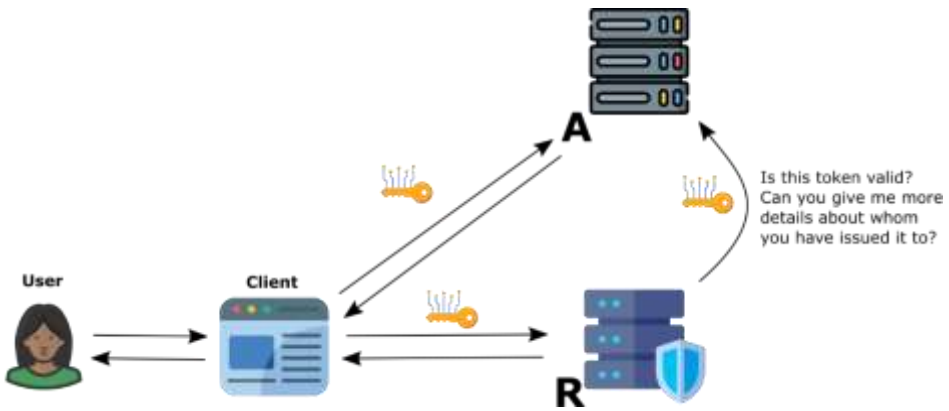

Figure 1.43 Token introspection. When using opaque tokens, the resource server needs to send requests to the authorization server to discover if the token is valid and more details about to whom it was issued.

The next snippet shows the cURL call to the introspection endpoint the authorization server exposes. The client must use HTTP Basic to authenticate using their credentials when sending the request. The client sends the token as a request parameter and receives details about the token in response.

```
curl -X POST 'http://localhost:8080/oauth2/introspect?
token=iED8-…' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

The next snippet shows an example of a response to the introspection request for a valid token. When the token is valid, its status appears as "active" and the response provides all the details the authorization server has about the token.

```
{
    "active": true,
    "sub": "client",
    "aud": [
        "client"
    ],
    "nbf": 1682941720,
    "scope": "CUSTOM",
    "iss": "http://localhost:8080",
    "exp": 1682942020,
    "iat": 1682941720,
    "jti": "ff14b844-1627-4567-8657-bba04cac0370",
    "client_id": "client",
    "token_type": "Bearer"
}
```

If the token doesn't exist or has already expired, then its active status is false, as shown in the next snippet.

```
{
    "active": false,

}
```

The default active time for a token is 300 seconds. In examples, you'll prefer to make the token life longer. Otherwise, you won't have enough time to use the token for tests, which can become frustrating. Listing 1.44 shows you how to change the token time to live. I prefer to make it very large for example purposes (like 12 hours in this case), but remember never to configure it this large for a real-world app. In a real app, you'd usually go with a time to live from 10 to 30 minutes max.

Listing 1.44 Changing the access token time to live

```
RegisteredClient registeredClient = RegisteredClient
        .withId(UUID.randomUUID().toString())
        .clientId("client")
         // …
        .authorizationGrantType(AuthorizationGrantType.CLIENT_CREDENTIALS)
```

```
        .tokenSettings(TokenSettings.builder()
            .accessTokenFormat(OAuth2TokenFormat.REFERENCE)
            .accessTokenTimeToLive(Duration.ofHours(12))     #A
            .build())
        .scope("CUSTOM")
        .build();
```

#A Setting 12 hours as the access token time to live.

1.37 Revoking tokens

Suppose you discover a token has been stolen. How could you make a token invalid for use? Token revocation is a way to invalidate a token the authorization server previously issued. Normally, an access token lifespan is short, so stealing a token still makes it difficult for one to use it. But sometimes you want to be extra cautious.

The following snippet shows a cURL command you can use to send a request to the token revocation endpoint the authorization server exposes. The revocation feature is active by default in a Spring Security authorization server. The request only requires the token that you want to revoke and HTTP Basic authentication with the client credentials. Once you send the request, the token cannot be used anymore.

```
curl -X POST 'http://localhost:8080/oauth2/revoke?
token=N7BruErWm-44-…' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

If you use the introspection endpoint with a token you have revoked, you should observe the token is no longer active after revocation (even if its time to live hasn't expired yet).

```
curl -X POST 'http://localhost:8080/oauth2/introspect?
token=N7BruErWm-44-…' \
--header 'Authorization: Basic Y2xpZW50OnNlY3JldA=='
```

Using token revocation makes sense sometimes, but it's not something you'd always desire. Remember that if you want to use the revocation feature, this also implies you need to use introspection (even for non-opaque tokens) with every call to validate that the token is still active. Using introspection so often might have a big impact on the performance. You should always ask yourself: Do I really need this extra protection layer?

Sometimes, hiding the key under the rug is enough, other times you need advanced, complex, and expensive alarm systems. What you use depends on what you protect.